klib

librería del núcleo

índice

1.- Introducción

Algo importante que hay que saber es que el núcleo (también llamado kernel) de Minix interacciona directamente con el hardware del sistema. Esta interacción comprende básicamente el acceso a memoria principal, el acceso a los puertos de entrada/salida y el manejo de las interrupciones.

Para poder llevar a cabo estos diferentes tipos de interacciones con el hardware se han desarrollado una serie de funciones, escritas en su mayoría en lenguaje ensamblador, siendo las restantes escritas en lenguaje de alto nivel C.

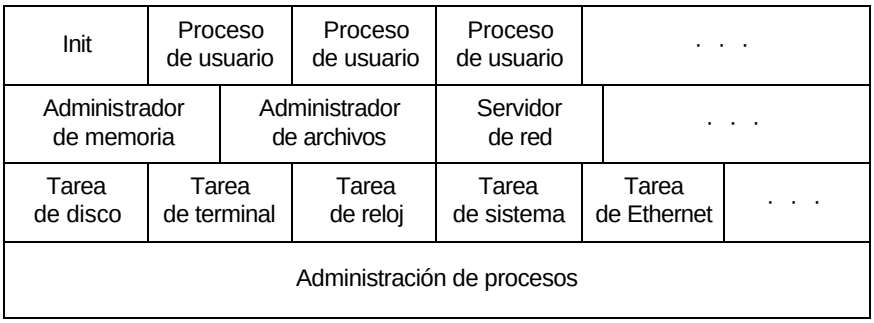
Este conjunto de funciones conforman una librería denominada KLIB. Las funciones de esta librería se reparten entre dos ficheros llamados klib.s y misc.c, en el primero se incluyen aquellas escritas en ensamblador y en el segundo aquellas escritas en lenguaje C.

Algunos de nuestros lectores se estarán preguntando el porqué del uso del lenguaje ensamblador. Pues bien, la utilización de lenguaje de bajo nivel para la realización de esta librería se debe principalmente a dos razones:

1. Optimización: las rutinas implementadas proporcionan operaciones que forman parte del núcleo del sistema operativo y son, por tanto, utilizadas muy frecuentemente. Por esta razón es de vital importancia la eficiencia con que se ejecutan estas funciones, eficiencia que será maximizada con el uso del ensamblador. Ejemplos pueden ser las rutinas de copia de mensajes o las rutinas de acceso a memoria.

2. Imposibilidad de utilizar lenguajes de alto nivel: no debido a que no existan lenguajes de alto nivel que proporcionen acceso a recursos de bajo nivel (memoria, puertos o interrupciones), sino debido a que Minix no fue escrito para estos compiladores.

El klib se encuentra en el kernel, en la capa inferior, interaccionando directamente con el hardware. Ofrece sus servicios a los drivers y a los otros módulos del kernel.



2.- La arquitectura del microprocesador Intel 80386

Una muy buena idea antes de ver el código ensamblador que en este trabajo se expone sería leer esta pequeña introducción a la arquitectura del microprocesador Intel 80386, sobre todo si al programar en ensamblador te quedaste con lo más básico, es decir con el procesador Intel 8086.

La diferencia principal entre la arquitectura del 80386 y aquellas de gama inferior, que nos ha llevado a elaborar este pequeño tratado, es que el Intel 80386 incluye por primera vez registros de 32 bits frente a los registros de 16 bits que existían con anterioridad. Además el bus de datos también aumenta de 16 hasta 32 bits (esto ocurre en la versión 80386DX pero no en la versión 80386SX).

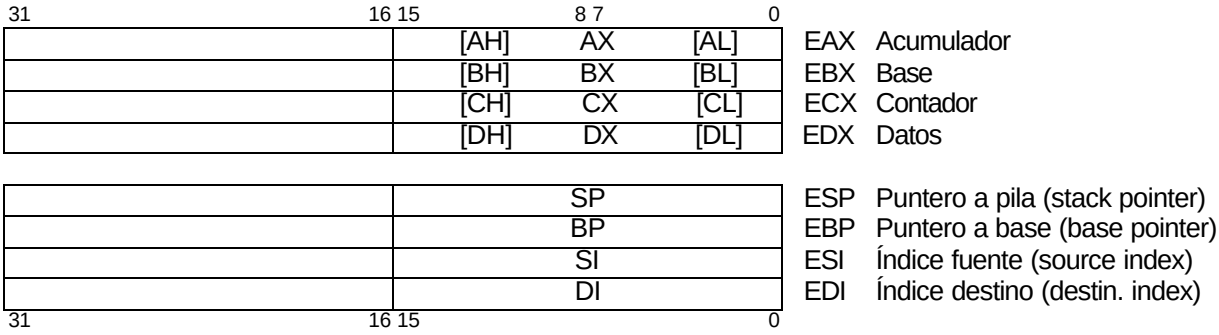
Teniendo en cuenta lo dicho, ahora las instrucciones del lenguaje ensamblador soportan operandos de 8, 16 y además de 32 bits.

Veamos, pues, las características de este microprocesador ya casi caído en el olvido.

2.1.- Registros de propósito general

Recordemos que los registros del 8086 eran de 16 bits y se llamaban AX, BX, etc. Estos registros podían dividirse en dos subregistros de 8 bits, como por ejemplo el AX que se dividía en AH y AL. Pues con el 80386 simplemente se extienden los registros a 32 bits, pudiendo todavía accederse a los 16 bits inferiores y dentro de estos a los 8 bits superiores o inferiores.

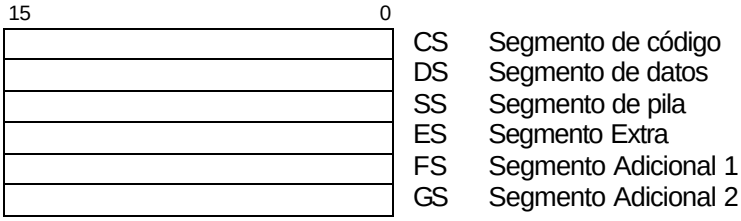
Véase la siguiente figura:



2.2.- Registros de segmento

Los registros de segmento se han quedado en el tamaño de 16 bits. Ahora, junto a aquellos que conocíamos aparecen dos nuevos, FS y GS, que se usan con el mismo fin que ES.

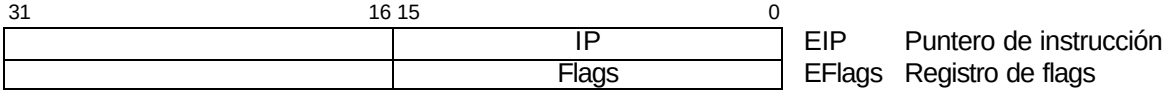
Véase la siguiente figura:



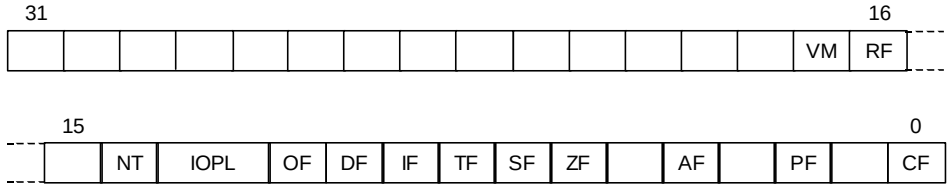
2.3.- Flags y puntero de instrucción

En el 80386 tanto el registro de flags como el puntero de instrucción se extienden hasta los 32 bits, pudiéndose todavía acceder a los registros anteriores, con la mitad de tamaño, en los 16 bits inferiores.

Véase la siguiente figura:



Dentro del registro de flags se han añadido nuevos bits de flag. En la siguiente figura podemos ver en qué posición, dentro del registro de flags, se ubican los diferentes bits:



En la tabla que a continuación les ofrecemos aparecen los nombres completos para cada bit. Seguro que más de un nombre les es familiar, sin embargo hay algunos que necesitan una descripción. Esta aparece después de la tabla.

Bit	Flag	Nombre	Bit	Flag	Nombre
0	CF	Carry Flag	10	DF	Direction Flag
2	PF	Parity Flag	11	OF	Overflow Flag
4	AF	Auxiliary Carry Flag	12/13	IOPL	Input/Output Privilege Level Flag
6	ZF	Zero Flag	14	NT	Nested Task Flag
7	SF	Sign Flag	16	VM	Virtual 8086 Mode Flag
8	TF	Trap Flag	17	RF	Resume Flag
9	IF	Interrupt Enable Flag			

Bits que se usan sólo cuando el microprocesador se encuentra en modo protegido:

IOPL: garantiza que una instrucción realice sólo operaciones permitidas.
NT: indica si la ejecución de la tarea actual está anidada dentro de otra tarea.
RF: usado con el registro de depuración, hace que se ignoren los fallos de depuración de la siguiente instrucción.
VM: el microprocesador cambia al modo de operación virtual del 8086.

2.4.- Registros de control

Son tres registros de 32 bits llamados CR0, CR2 y CR3 que guardan información acerca del estado de la máquina.

El registro CR0 es también conocido como MSW (palabra de estado de la máquina), y contiene los siguientes bits: ET (processor extension type), TS (task switched), EM (emulate coprocessor), MP (monitor coprocessor), y PE (protection enable).

2.5.- Registros de dirección del sistema

El microprocesador 80386 contiene cuatro registros de propósito específico que se usan para referenciar tablas o segmentos soportados por el modo protegido 80286/80386.

En la siguiente tabla se describen:

Nombre de la tabla	Registro asociado	Tamaño del registro
Tabla de Descriptores Globales (GDT)	GDTR	32 (base) +16 (limite) bits
Tabla de Descriptores de Interrupción (IDT)	IDTR	32 (base) +16 (limite) bits
Tabla de Descriptores Locales (LDT)	LDTR	16 (selector) bits
Tabla de Segmentos del Estado de la Tarea (TSS)	TR	16 (selector) bits

2.6.- Registros de testeo y depuración

En el 80386 existen ocho registros de depuración de 32 bits, y dos registros más, también de 32 bits, para el control del testeo de la RAM y de la CAM (contenido de la memoria direccionable) dentro de la TLB.

En la tabla que sigue especificamos el nombre de dichos registros y su utilidad:

Registro	Utilidad
DR0	Dirección 0 de breakpoint lineal
DR1	Dirección 1 de breakpoint lineal
DR2	Dirección 2 de breakpoint lineal
DR3	Dirección 3 de breakpoint lineal
DR4	Reservado por Intel
DR5	Reservado por Intel
DR6	Estado del breakpoint
DR7	Control del breakpoint
TR6	Control del testeo
TR7	Estado del testeo

3.- El acceso a los puertos de E/S

Para poder entender mejor las funciones que tratan con los puertos de E/S del ordenador hemos elaborado un pequeño tratado acerca del tema, y que a continuación exponemos.

Antes de empezar, debemos recordar que en la arquitectura Intel x86 el procesador proporciona un espacio de direcciones de E/S, para los puertos, que se encuentra separado del espacio de direcciones de la memoria principal.

La comunicación entre un programa y el hardware, es decir los chips de soporte y las tarjetas de ampliación, se efectuará mediante los llamados puertos. Como puerto uno tiene que imaginarse una entrada o salida de datos de 8 o 16 bits de ancho, que se identificará por su dirección entre 0 y 65535. Éste autoriza el acceso a los diferentes registros de la correspondiente unidad de hardware y por regla general un grupo de puertos queda cubierto por un dispositivo.

Para la comunicación con los puertos, la CPU utiliza el bus de datos y el bus de direcciones y se comporta de forma muy similar que en el acceso a la memoria.

- En primer lugar ésta manda una señal mediante una conexión especial de bus, para que todas las unidades que estén recibiendo el bus sepan que ahora no se va a dirigir a un punto de memoria sino a un puerto.

- Seguidamente coloca la dirección del puerto en los 16 bits inferiores del bus de direcciones y espera que uno de los “escuchas” del bus declare su competencia.

- Una vez esto ha sucedido, la CPU manda los datos que deben transmitirse por medio del bus de datos.

En dirección contraria , el puerto actuará de la misma forma al leer los datos, el único requisito es que el contenido del puerto lo mande la tarjeta de ampliación a la CPU. Pero sólo mandará los datos en caso de que le sea requerido y no cuando la tarjeta crea que tiene algo importante para mandar.

Una última cosa por comentar es que hoy en día las direcciones de puerto de los elementos más importantes del PC están estandarizadas, es decir en todos los ordenadores dichas direcciones son idénticas.

4.- Las interrupciones

Un tema relacionado con muchas de las funciones ya no sólo de la librería del kernel sino del entero sistema operativo es el de las interrupciones. Téngase en cuenta que todos los sistemas operativos están guiados por interrupciones. Estas se han visto de manera más o menos detallada en otros trabajos de la asignatura, pero aún así no está de más ver cuáles son y cómo funcionan una vez más, esperando esclarecer mejor, si cabe, este tema tan escabroso.

Hay cuatro categorías principales de interrupciones:

1) En primer lugar, hay interrupciones generadas por la circuitería del ordenador en respuesta a algún acontecimiento, tal como la pulsación de una tecla. Estas interrupciones están manejadas por el chip controlador de interrupciones (el 8259, también conocido como PIC), que las prioriza antes de enviarlas a la CPU para que actúe. A estas interrupciones se les suele llamar *interrupciones hardware*.
2) En segundo lugar, hay interrupciones que son generadas por la CPU como resultado de algún suceso inusual producido por el programa como, por ejemplo, una división por cero. A estas interrupciones solemos referirnos como *excepciones*.
3) En tercer lugar, hay interrupciones generadas deliberadamente por los programas para invocar, por ejemplo, las llamadas al sistema. Estas interrupciones son llamadas *interrupciones software.*
4) Por último, hay también un tipo especial de interrupción, llamada *interrupción no enmascarable* (NMI), que se utiliza para solicitar la atención inmediata de la CPU. A menudo indica que se ha producido una emergencia, como, por ejemplo, una caída de voltaje, o un error de paridad de memoria. Cuando se envía una NMI la CPU la atiende antes que al resto de las interrupciones, por supuesto.

Los programas en ensamblador pueden desactivar, o lo que es lo mismo enmascarar, las interrupciones hardware. Por esta razón, también se las llama enmascarables; el resto de las interrupciones que interceptan errores especiales, como la división por cero, no se pueden enmascarar. Se pueden aducir dos razones para desactivar las interrupciones hardware:

a) Cuando se necesita ejecutar un fragmento de código especialmente crítico antes de que suceda ninguna otra tarea en el ordenador, interesa que todas las interrupciones queden bloqueadas. Por ejemplo, cuando se quiere hacer algún cambio en la tabla de vectores de interrupción.
b) A veces interesa enmascarar ciertas interrupciones hardware cuando éstas pueden interferir en alguna actividad cuya dependencia temporal sea crítica. Por ejemplo, si se está ejecutando una rutina de E/S cuya temporización tiene que estar exquisitamente controlada, uno no puede permitirse el lujo de esperar "aparcado" mientras se ejecuta una lenta interrupción de disco.

En el primer caso, se ha de tener en cuenta que en último término, la ejecución de todas las interrupciones descansa sobre el flag de interrupción (bit 9) del registro de estado (flags de estado) del procesador. Cuando este bit toma el valor 0, acepta cualquier solicitud de interrupción que permita el registro de máscaras de interrupción. Cuando es uno, no se permiten interrupciones hardware. Para hacer que este flag tome el valor cero, desactivando así las interrupciones, se utiliza la instrucción CLI. Para volver a poner el flag a uno, autorizando de nuevo las interrupciones, se utiliza la instrucción STI.

Para gestionar las interrupciones hardware, todos los PC's utilizan el llamado chip controlador de interrupciones programable (PIC) Intel 8259. Existe la posibilidad de que aparezca más de una solicitud de interrupción simultáneamente, por lo que el chip establece una *jerarquía de prioridades*. Existen ocho niveles de prioridad, excepto en el AT, que tiene dieciséis, a los que se refieren con las abreviaturas IRQ0 a IRQ7 (IRQ0 a IRQ16 en AT), que corresponden a las siglas inglesas de solicitud de interrupción (interrupt request). La mayor prioridad se consigue en el nivel 0. En el caso del AT, los ocho niveles de prioridad extra se gestionan con un segundo chip 8259 denominado *slave*, en contraposición al primer chip 8259 denominado *master*; esta segunda serie de niveles tiene una prioridad comprendida entre IRQ2 e IRQ3.

En la siguiente tabla se muestra como se asignan los niveles de interrupción a los distintos periféricos:

¡Error! Marcador no definido.

PC	Solo AT	Descripción
IRQ0		Temporizador
IRQ1		Teclado
IRQ2		Canal de E/S
	IRQ8	Reloj de tiempo Real
	IRQ9	software redirigido a IRQ2
	IRQ10	Reservado
	IRQ11	Reservado
	IRQ12	Reservado
	IRQ13	Coprocesador matemático
	IRQ14	Controlador de disco duro
IRQ3	IRQ15	Reservado COM1 y COM3 (COM2 y COM4 en el AT)
IRQ4		COM2 y COM4 (COM1 y COM3 en el AT)
IRQ5		Disco duro (LPT2 en el AT)
IRQ6		Controlador de disquete
IRQ7		LPT1

El 8259 posee tres registros de un byte que controlan y gestionan las ocho (o dieciséis) líneas de interrupción hardware, pero a nosotros sólo nos interesa el registro de máscara de interrupciones (IMR). El 8259 emplea este registro para averiguar si una interrupción de un determinado nivel está permitida en cualquier momento. Para desactivar interrupciones hardware concretas, se envía un patrón de bits al puerto 21h (INT_CTLMASK), que es la dirección del registro de máscaras de interrupción (IMR). El registro de máscara del segundo chip 8259 del AT se sitúa en el puerto A1h (INT2_CTLMASK). En este patrón de bits, se ponen a uno los bits que corresponden a los números de interrupción que se desea enmascarar.

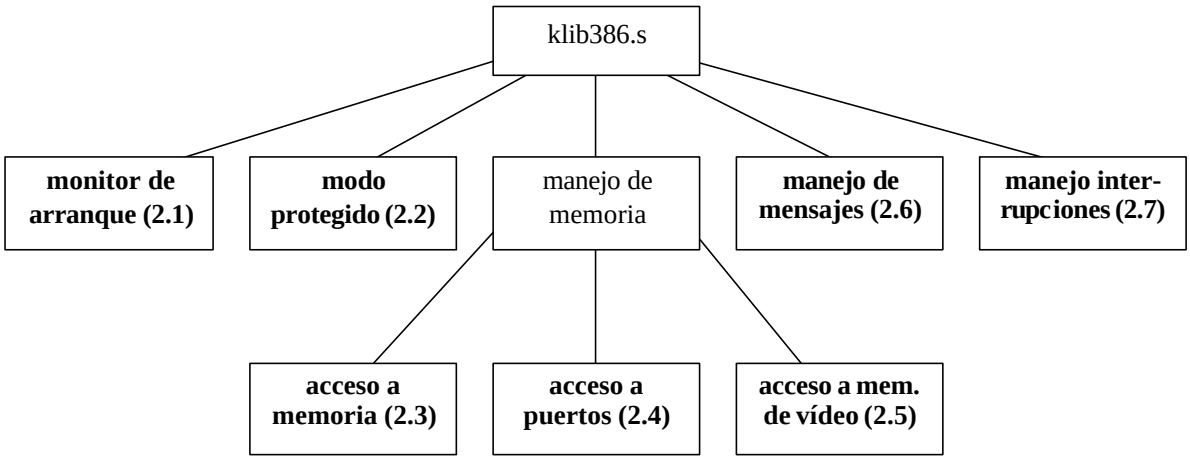
5.- Librería del kernel

Ahora que sabemos que el nombre KLIB (Kernel LIBrary) tiene su origen en las palabras cuyo significado en inglés es "librería del núcleo", nos queda otra cosa importante por aprender: el código ensamblador de las funciones del KLIB no se encuentran realmente en el fichero klib.s. ¿Y esto qué significa? Para entenderlo véase primero el contenido del fichero klib.s:

```
! Fichero klib.s

#include <minix/config.h>
#if _WORD_SIZE == 2
#include "klib88.s"
#else
#include "klib386.s"
#endif
```

Minix es un sistema operativo preparado para trabajar tanto en ordenadores con tamaño de palabra de 16 bits (8086, 8088, 80286) como para los de tamaño de palabra de 32 bits (80386 y superiores). Para mantener la compatibilidad con ambas clases de PC en el fichero klib.s se comprueba el tamaño de la palabra del ordenador. Si el tamaño es 16 bits se incluyen las funciones de soporte para sistemas de 16 bits que se encuentran en el fichero klib88.s. Si el tamaño no es 16 bits (se supone que es 32), se incluye el fichero klib386.s donde están las rutinas de soporte para sistemas que funcionan con palabra de 32 bits.

En este trabajo nos centraremos en las funciones de soporte para sistemas de 32 bits. En total son 23 funciones que hemos clasificado en los siguientes módulos para mayor comprensión:



En la siguiente tabla se muestran todas las funciones implementandas en el fichero klib386.s clasificadas de la manera indicada en la figura anterior:

Módulo	Función	Descripción
2.1	*_monitor*	Finaliza Minix y retorna al monitor
2.2	*_level0*	Invoca una función en nivel 0
2.3	*_phys_copy*	Copia datos de una posición de memoria a cualquier otra
	_check_mem	Revisa un bloque de memoria y devuelve el tamaño válido
	_mem_rdw	Copia una palabra de *segmento:desplazamiento*
2.4.	*_in_byte*	Lee un byte de un puerto de E/S
	_in_word	Lee una palabra de un puerto de E/S
	_out_byte	Escribe un byte a un puerto de E/S
	_out_word	Escribe una palabra a un puerto de E/S
	_port_read	Transfiere datos de un puerto (el controlador de disco) a memoria
	_port_read_byte	Igual que *_port_read* pero byte a byte
	_port_write	Transfiere datos de memoria a un puerto (el controlador de disco)
	_port_write_byte	Igual que *_port_write* pero byte a byte
2.5.	*_mem_vid_copy*	Copia datos desde la memoria a la ram de vídeo
	_vid_vid_copy	Copia datos desde la ram de vídeo a la ram de vídeo
2.6	*_cp_mess*	Copia mensajes de una fuente a un destino
2.7	*_lock*	Inhabilita las interrupciones
	_unlock	Habilita las interrupciones
	_reset	Vuelve a arrancar el sistema
	_enable_irq	Habilita una irq del controlador 8259
	_disable_irq	Inhabilita una irq del controlador 8259
	_bios_13	Realiza una llamada al servicio 13h de la BIOS para E/S a disco

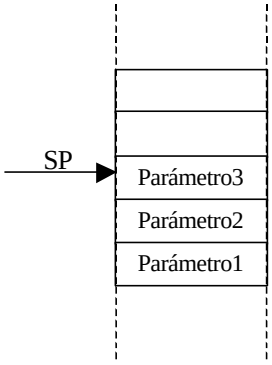
Antes de pasar directamente a detallar cada una de las funciones anteriores, explicaremos conceptos comunes a todas ellas. En primer lugar se mostrará se mostrará el paso de parámetros a funciones y luego se mostrará la estructura general de las mismas.

1. Paso de parámetros a funciones

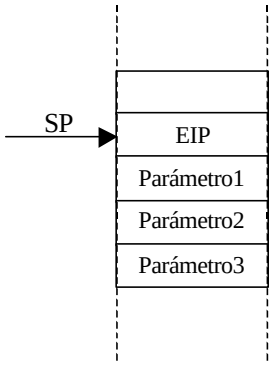
El primer hecho a considerar es que en lenguaje ensamblador es el propio programador o, en su defecto, el ensamblador, el que debe controlar el paso de parámetros a funciones. Para ello debe guardar dichos parámetros en la pila o en registros antes de realizar la llamada a la función y una vez "dentro" de la misma debe recuperar esos valores.

En las funciones que se muestran a continuación el paso de parámetros se realiza a través de la pila de la forma que se muestra a continuación.

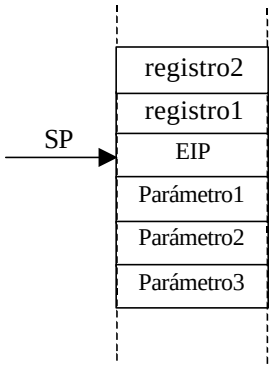
- Antes de la llamada a la función se ubican en la pila los parámetros a pasar a ésta.



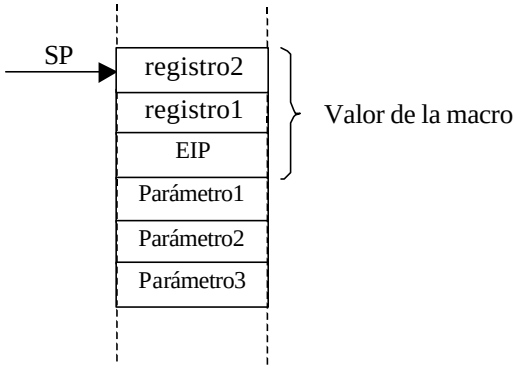
- Se realiza un CALL, que automáticamente salva el registro EIP en la pila.



- Dentro de la función invocada se guardan los registros que se vayan a utilizar.



- Dentro de la función, para acceder a los parámetros se accede a la zona de memoria apuntada por el registro ESP más el valor de una macro, que define el número de entradas de la pila por 4 (es decir número de bytes), desde el top de la misma hasta la entrada correspondiente al primer parámetro.



2. Estructura general de las funciones

A continuación se muestran las partes principales de las que se componen las funciones que se describen en esta memoria:

([1]) Los únicos registros que se salvan y restauran de la pila si se van a utilizar, son ebx, esi, edi, esp, los registros de segmento (CS, DS, ES) y el bit de dirección del registro de flags. El registro EIP siempre es guardado en la pila por la instrucción CALL que llama la función en cuestión.

([2]) Si hay que retornar un valor este se devuelve en el registro EAX.

5.1.- Monitor de arranque

En este módulo sólo se encuentra la función _monitor que pasamos a describir en el siguiente subapartado.

5.1.1.- _monitor

Se encarga de finalizar Minix y retornar al monitor de arranque. Desde el punto de vista del monitor de arranque, todo Minix no es más que una subrutina y cuando se inicia Minix se deja una dirección de retorno al monitor en la pila de éste. La función _monitor es la que se encarga de restaurar los diversos selectores de segmento y el apuntador a la pila que se guardó cuando Minix se inició, y luego regresar como una subrutina cualquiera.

```
! PUBLIC void monitor();

_monitor:
       mov     eax, (_reboot_code)   ! dirección de los nuevos parámetros
       mov     esp, (_mon_sp)        ! restaurar el puntero a pila del monitor
    o16 mov    dx, SS_SELECTOR       ! segmento de datos del monitor
       mov     ds, dx
       mov     es, dx
       mov     fs, dx
       mov     gs, dx
       mov     ss, dx
       pop     edi
       pop     esi
       pop     ebp
    o16 retf                         ! retornar al monitor
```

5.2.- Modo protegido

Este módulo también está formado por una única función, en este caso _level0, que describimos a continuación.

5.2.1.- _level0

Esta función se utiliza para llamar a una función con permiso de nivel 0. Esto permite al kernel ejecutar tareas que sólo son posibles al nivel de CPU más privilegiado. Estas tareas comprenden el poder reiniciar el equipo, acceder a las rutinas de la ROM BIOS, etc.

Existe una posición de memoria en la cual se carga la dirección de la función a tratar con permiso de nivel 0. El apuntador a dicha posición está en *_level0_func*. Para ejecutar una función con permiso de nivel 0 hay que guardar su dirección en la posición apuntada por *_level0_func* y llamar a la interrupción LEVEL0_VECTOR. Esta función cambia el EIP para que la próxima instrucción a ejecutar sea el contenido de la dirección por *_level0_func.*

El único parámetro de esta rutina es la dirección de una función que se ejecutará con permiso de nivel 0.

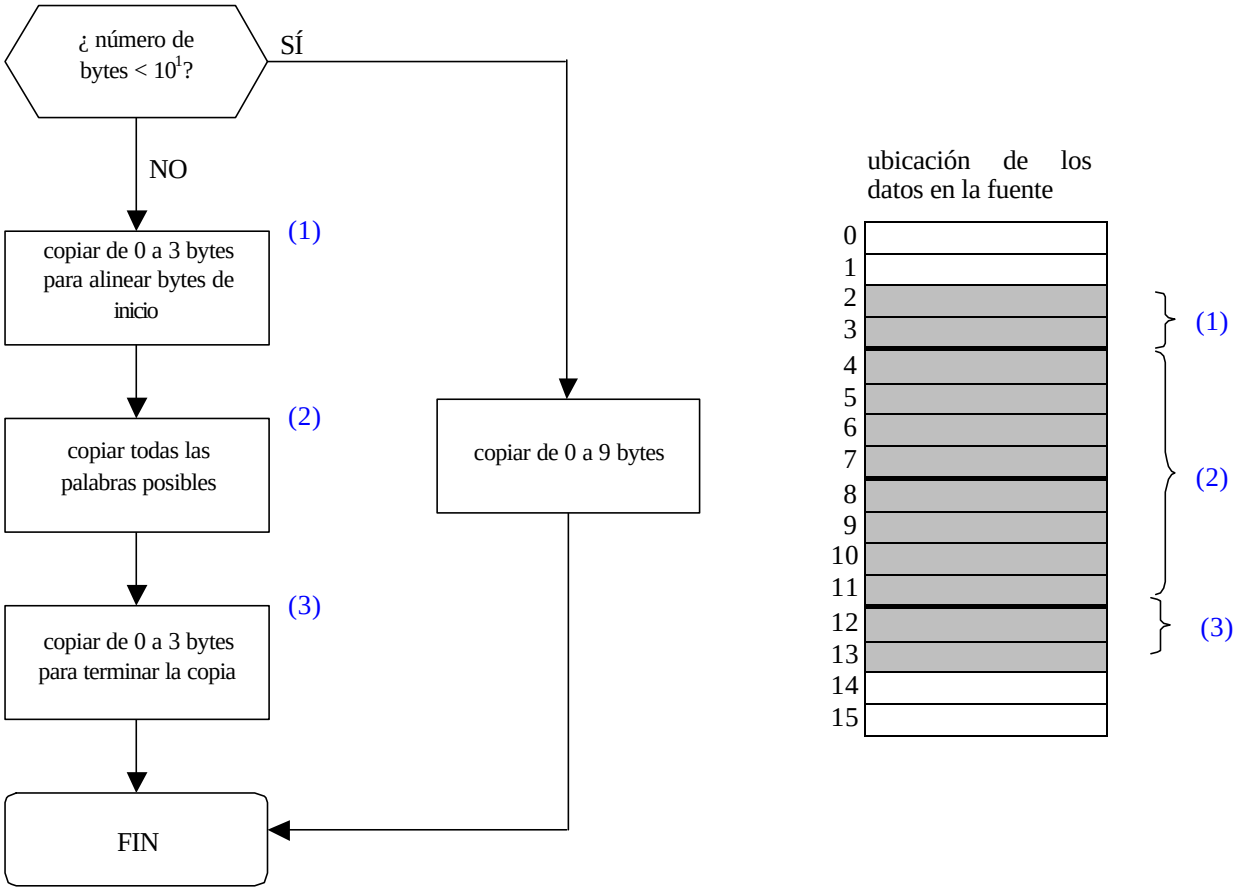
```
! PUBLIC void level0(void (*func)(void))

_level0:
       mov     eax, 4(esp)           ! eax ← parámetro (dirección de una función)
       mov     (_level0_func), eax   ! guardar eax en la direc. para funciones a nivel0
       int     LEVEL0_VECTOR         ! interrupción para tratar la función a nivel 0
       ret
```

5.3.- Acceso a memoria

Se incluyen aquí las funciones que operan sólo sobre la memoria principal, es decir las que tienen como fuente y/o destino la memoria principal.

5.3.1.- _phys_copy

Copia un bloque de datos de cualquier parte de la memoria física a cualquier otro lugar de la misma. Ambas direcciones son absolutas, es decir, la dirección 0, por ejemplo, se refiere realmente al primer byte de todo el espacio de direcciones.



([1]) Este es un valor empírico, que indica que por debajo del mismo no es rentable realizar el proceso intentando copiar palabra a palabra, sino que es más eficiente realizar la copia byte a byte.

Los tres parámetros indican la dirección fuente de los datos a copiar, la dirección destino de dichos datos a copiar y el número de bytes a copiar respectivamente. Todos los parámetros son del tipo unsigned long int.

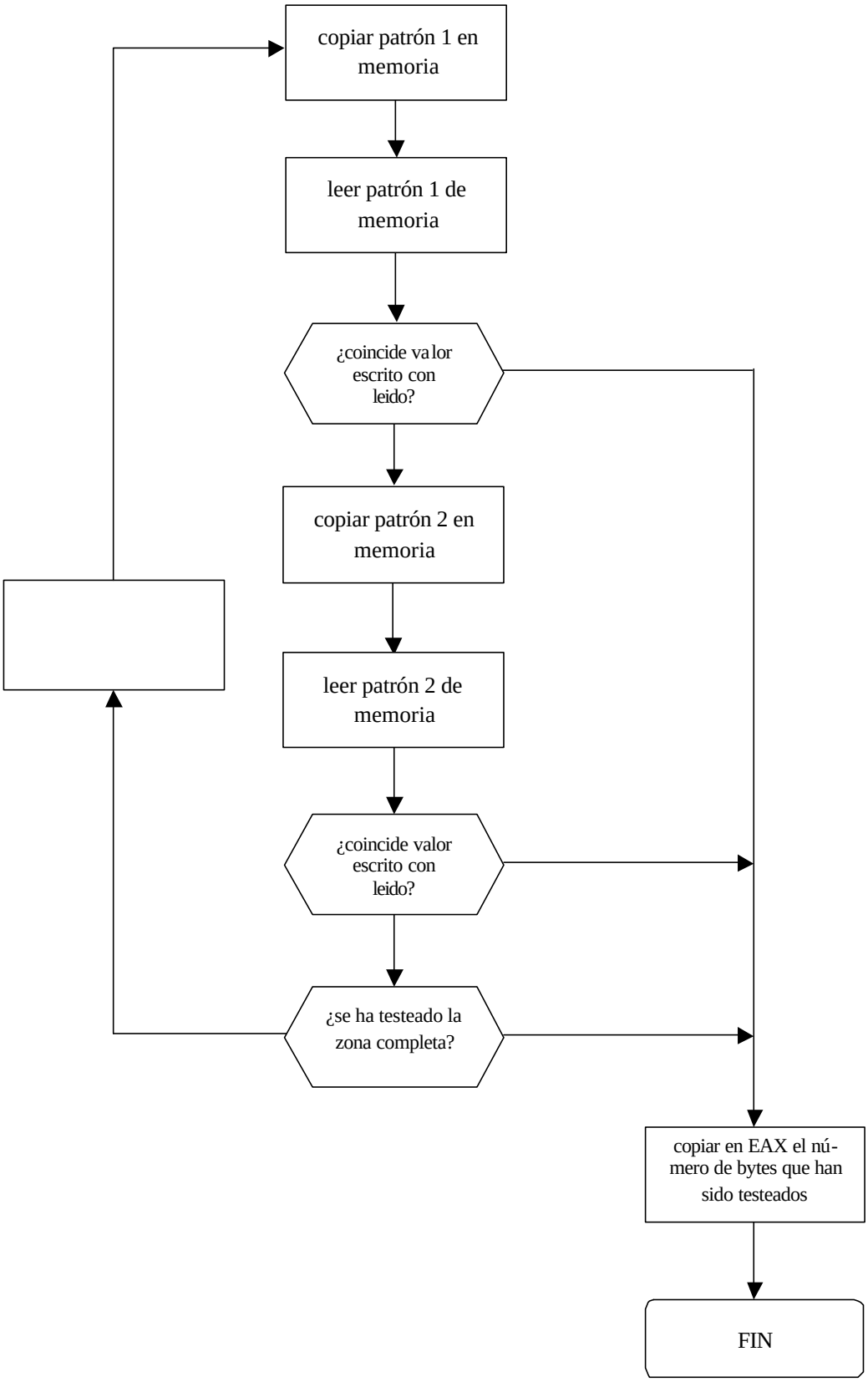
```
! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
!                       phys_bytes bytecount);

PC_ARGS=      4 + 4 + 4 + 4  ! 4 + 4 + 4
!             es edi esi eip  src dst len

        .align 16
_phys_copy:
        cld                          ! dirección de las oper. con ristras hacia adelante
        push    esi
        push    edi
        push    es                   ! salvar los registros que se van a utilizar

        mov     eax, FLAT_DS_SELECTOR
        mov     es, ax               ! establecer los selectores de segmento

        mov     esi, PC_ARGS(esp)    ! esi ← dirección fuente  (ds:esi)
        mov     edi, PC_ARGS+4(esp)  ! edi ← dirección destino (es:edi)
        mov     eax, PC_ARGS+4+4(esp) ! eax ← cantidad de bytes a copiar

        cmp     eax, 10              ! si cantidad a copiar < 10 bytes ir al bucle final
        jb      pc_small
        mov     ecx, esi             ! alinear fuente
        neg     ecx
        and     ecx, 3               ! contador para el alineamiento
        sub     eax, ecx
        rep
   eseg movsb                        ! ecx ← resto de bytes q quedan por copiar (inicio)
        mov     ecx, eax
        shr     ecx, 2               ! si dividimos por 4 → nº de palabras a copiar
        rep
   eseg movs
        and     eax, 3               ! falta copiar bytes restantes de la última palabra
pc_small:
        xchg    ecx, eax             ! ecx ← resto bytes q quedan por copiar(fin)
        rep
   eseg movsb

        pop     es
        pop     edi
        pop     esi                  ! recuperar los registros modificados
        ret                          ! retornar
```

5.3.2.- _check_mem

Esta función se invoca en el momento de iniciarse Minix para chequear un bloque de memoria. Esta función realiza una prueba sencilla con cada decimosexto byte, pues chequear todos los bytes sería muy costoso, usando dos patrones que prueban cada bit con valores tanto "0" como "1" (véase en el código TEST1PATTERN y TEST2PATTERN).

Los parámetros indican la dirección física del bloque de memoria a chequear y el tamaño de dicho bloque respectivamente. Si el tamaño del bloque es cero entonces hay que chequear todo.

```
! PUBLIC phys_bytes check_mem(phys_bytes base, phys_bytes size);

CM_DENSITY     =      16             ! densidad de comparación, se chequea cada 16º byte
CM_LOG_DENSITY =      4              ! logaritmo en base 2 de CM_DENSITY
TEST1PATTERN   =      0x55           ! patrón de chequeo 1 =01010101
TEST2PATTERN   =      0xAA           ! patrón de chequeo 2 =10101010
CHKM_ARGS      =      4 + 4 + 4      ! 4 + 4
!                     ds ebx eip     base size

_check_mem:
       push    ebx
       push    ds                    ! salvar los registros que se van a utilizar

    o16 mov    ax, FLAT_DS_SELECTOR  ! establecer los selectores de segmento
       mov     ds, ax

       mov     eax, CHKM_ARGS(esp)   ! eax ← dirección base del segmento a chequear
       mov     ebx, eax              ! ebx ← eax
       mov     ecx, CHKM_ARGS+4(esp) ! ecx ← tamaño de bloque en bytes
       shr     ecx, CM_LOG_DENSITY   ! ecx ← número de bytes a chequear realmente,
                                     ! es decir, número de iteraciones del bucle

cm_loop:
       movb    dl, TEST1PATTERN      ! dl ← patrón 1
       xchgb   dl, (eax)             ! escribir el patrón 1 en memoria
       xchgb   dl, (eax)             ! volver a escribirlo en dl
       cmpb    dl, TEST1PATTERN      ! deben coincidir si la memoria actual esta OK
       jnz     cm_exit               ! si son diferentes, hay un fallo en la memoria

       movb    dl, TEST2PATTERN      ! IGUAL PARA EL SEGUNDO PATRÓN
       xchgb   dl, (eax)
       xchgb   dl, (eax)
       add     eax, CM_DENSITY
       cmpb    dl, TEST2PATTERN

       loopz   cm_loop               ! repetir este proceso mientras ecx!=0 y ZF==1.

cm_exit:
       sub     eax, ebx              ! devuelve en eax tamaño del bloque correcto (bytes)
       pop     ds
       pop     ebx                   ! recuperar los registros modificados
       ret                           ! retornar
```


copiar patrón 1 en memoria
leer patrón 1 de memoria
¿coincide valor escrito con leido?
copiar patrón 2 en memoria
leer patrón 2 de memoria
¿coincide valor escrito con leido?
¿se ha testeado la zona completa?
copiar en EAX el número de bytes que han sido testeados
FIN

5.3.3.- _mem_rdw

Devuelve una palabra de 16 bits de cualquier lugar de la memoria. El resultado se rellena con ceros dentro del registro EAX de 32 bits. Es una función muy específica para los procesadores Intel.

El primer parámetro indica el segmento base y el segundo un puntero al desplazamiento a añadir a dicho segmento.

```
! PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);

        .align 16
_mem_rdw:
        mov     cx, ds                  ! salvar registro ds

        mov     ds, 4(esp)              ! ds  ← segmento
        mov     eax, 4+4(esp)           ! eax ← desplazamiento

        movzx   eax, (eax)              ! eax= M[segmento:desplazamiento]= M[ds:eax]

        mov     ds, cx                  ! restablecer ds
        ret                             ! retornar
```

5.4.- Acceso a puertos

Las funciones *_in_byte*, *_in_word*, *_out_byte* y *_out_word* que a continuación se describen proporcionan acceso a puertos de E/S. El uso de estas funciones específicas y no de las funciones de lectura/escritura a memoria tiene su origen en el tipo de arquitectura que las máquinas Intel poseen: los puertos de E/S ocupan un espacio distinto del de la memoria, como ya hemos visto en el apartado tres de este trabajo.

Además existen las funciones *_port_read*, *_port_read_byte*, *_port_write* y *_port_write_byte* que se encargan de transferir bloques de datos entre puertos de E/S y la memoria. Estas se usan principalmente para transferencias desde y hacia el disco que deben efectuarse con mucha mayor rapidez que la que es posible con las otras llamadas de E/S.

En estas últimas funciones, las versiones de bytes leen 8 bits en lugar de 16 bits en cada operación, a fin de manejar dispositivos periféricos antiguos de 8 bits.

5.4.1.- _in_byte

Lee un byte sin signo del puerto de E/S indicado y devuelve dicho valor.

```
! PUBLIC unsigned in_byte(port_t port);

        .align 16
_in_byte:
        mov     edx, 4(esp)           ! edx ← puerto origen
        sub     eax, eax              ! eax ← 0 (pues leemos sólo un byte)
        inb     dx                    ! leemos 1 byte del puerto que se vuelca en eax
        ret                           ! retornar
```

5.4.2.- _in_word

Lee una palabra sin signo del puerto de E/S indicado y devuelve dicho valor.

```
! PUBLIC unsigned in_word(port_t port);

        .align 16
_in_word:
        mov     edx, 4(esp)           ! edx ← puerto origen
        sub     eax, eax              ! eax ← 0
    o16 in      dx                    ! leemos 1 palabra del puerto que se vuelca en eax
        ret                           ! retornar
```

5.4.3.- _out_byte

Escribe un valor, que convierte a byte, al puerto indicado.

Los parámetros que se pasan a la función son el puerto destino y el valor de 8 bits a escribir en dicho puerto.

```
! PUBLIC void out_byte(port_t port, u8_t value);

        .align 16
_out_byte:
        mov     edx, 4(esp)           ! edx ← puerto destino
        movb    al, 4+4(esp)          ! ax(low) ← valor a escribir
        outb    dx                    ! escribir al puerto indicado 1 byte
        ret                           ! retornar
```

5.4.4.- _out_word

Escribe un valor, de tamaño palabra, al puerto indicado.

Los parámetros que se pasan a la función son el puerto destino y el valor de 16 bits a escribir en dicho puerto.

```
! PUBLIC void out_word(Port_t port, U16_t value);

       .align 16
_out_word:
       mov    edx, 4(esp)           ! edx ← puerto destino
       mov    eax, 4+4(esp)         ! ax ← valor a escribir
    o16 out   dx                    ! escribir al puerto indicado 1 palabra
       ret                          ! retornar
```

5.4.5.- _port_read

Transfiere información de un puerto (el controlador de disco) a memoria.

El primer parámetro indica el puerto del que se van a leer los datos, el segundo parámetro la dirección de destino en la que se van a escribir los datos y el último parámetro indica la cantidad de bytes a copiar.

```
! PUBLIC void port_read(port_t port, phys_bytes destination, unsigned bytcount);

PR_ARGS =      4 + 4 + 4             ! 4 + 4 + 4
!              es edi eip            port dst len

       .align 16
_port_read:
       cld
       push   edi
       push   es                    ! salvar registros

       mov    ecx, FLAT_DS_SELECTOR
       mov    es, cx

       mov    edx, PR_ARGS(esp)     ! edx ← puerto que se va a leer
       mov    edi, PR_ARGS+4(esp)   ! edi ← dirección de destino
       mov    ecx, PR_ARGS+4+4(esp) ! ecx ← cantidad de bytes
       shr    ecx, 1                ! ecx ← ecx/2  (cantidad de palabras de 16 bits)
       rep
    o16 ins                         ! leer todo

       pop    es
       pop    edi                   ! restablecer registros

       ret                          ! retornar
```

5.4.6.- _port_read_byte

Transfiere información de un puerto (el controlador de disco) a memoria, byte a byte.

El primer parámetro indica el puerto del que se van a leer los datos, el segundo parámetro la dirección de destino en la que se van a escribir los datos y el último parámetro indica la cantidad de bytes a copiar.

```
! PUBLIC void port_read_byte(port_t port, phys_bytes destination,
!                                         unsigned bytcount);

PR_ARGS_B =    4 + 4 + 4               ! 4 + 4 + 4
!              es edi eip              port dst len

_port_read_byte:
       cld
       push    edi
       push    es                      ! salvar registros

       mov     ecx, FLAT_DS_SELECTOR
       mov     es, cx

       mov     edx, PR_ARGS_B(esp)     ! edx ← puerto que se va a leer
       mov     edi, PR_ARGS_B+4(esp)   ! edi ← dirección de destino
       mov     ecx, PR_ARGS_B+4+4(esp) ! ecx ← cantidad de bytes
       rep
       insb                           ! leer todo

       pop     es
       pop     edi                     ! restablecer registros

       ret                             ! retornar
```

5.4.7.- _port_write

Transfiere información de memoria a un puerto (el controlador de disco).

El primer parámetro indica el puerto en el que se van a escribir los datos, el segundo parámetro la dirección de destino de la que se van a leer los datos y el último parámetro indica la cantidad de bytes a copiar.

```
! PUBLIC void port_write(port_t port, phys_bytes source, unsigned bytcount);

PW_ARGS=       4 + 4 + 4             ! 4 + 4 + 4
!              es edi eip            port src len

       .align 16
_port_write:
       cld
       push    esi
       push    ds                    ! salvar registros

       mov     ecx, FLAT_DS_SELECTOR
       mov     ds, cx

       mov     edx, PW_ARGS(esp)     ! edx ← puerto en el que se va a escribir
       mov     esi, PW_ARGS+4(esp)   ! esi ← dirección fuente
       mov     ecx, PW_ARGS+4+4(esp) ! ecx ← cantidad de bytes
       shr     ecx, 1                ! ecx ← ecx/2 (cantidad de palabras)
       rep
    o16 outs                         ! escribir todo

       pop     ds
       pop     esi                   ! restablecer registros

       ret                           ! retornar
```

5.4.8.- _port_write_byte

Transfiere información byte a byte de memoria a un puerto (el controlador de disco).

El primer parámetro indica el puerto en el que se van a escribir los datos, el segundo parámetro la dirección de destino de la que se van a leer los datos y el último parámetro indica la cantidad de bytes a copiar.

```
! PUBLIC void port_write_byte(port_t port, phys_bytes source,
!                                           unsigned bytcount);

PW_ARGS_B =    4 + 4 + 4               ! 4 + 4 + 4
!              es edi eip              port src len

_port_write_byte:
       cld
       push    esi
       push    ds                      ! salvar registros

       mov     ecx, FLAT_DS_SELECTOR
       mov     ds, cx

       mov     edx, PW_ARGS_B(esp)     ! edx ← puerto en el que se va a escribir
       mov     esi, PW_ARGS_B+4(esp)   ! esi ← dirección fuente
       mov     ecx, PW_ARGS_B+4+4(esp) ! ecx ← cantidad de bytes
       rep
       outsb                           ! escribir todo

       pop     ds
       pop     esi                     ! restablecer registros

       ret                             ! retornar
```

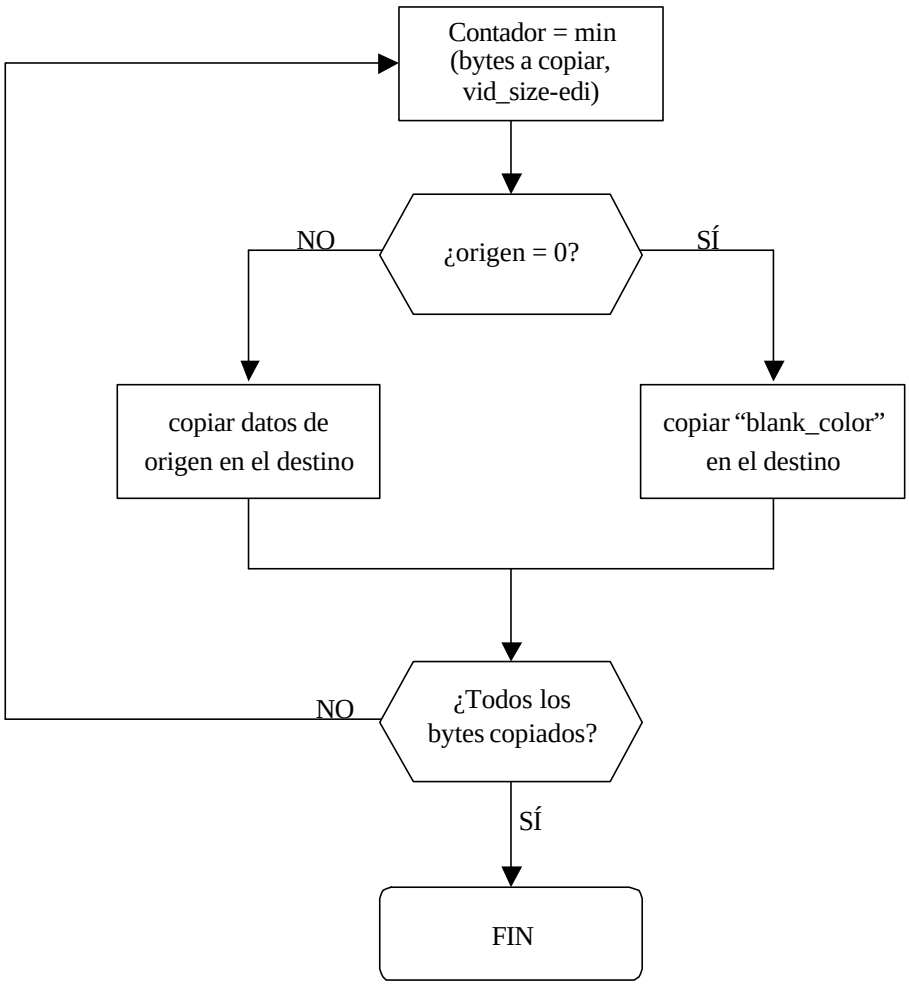
5.5.- Acceso a ram de vídeo

A continuación se detallan dos funciones que se usan para el volcado de caracteres en pantalla.

5.5.1.- _mem_vid_copy

Copia una cadena de palabras que contienen bytes alternos de caracteres y atributos desde la región de memoria del kernel a la memoria de vídeo.

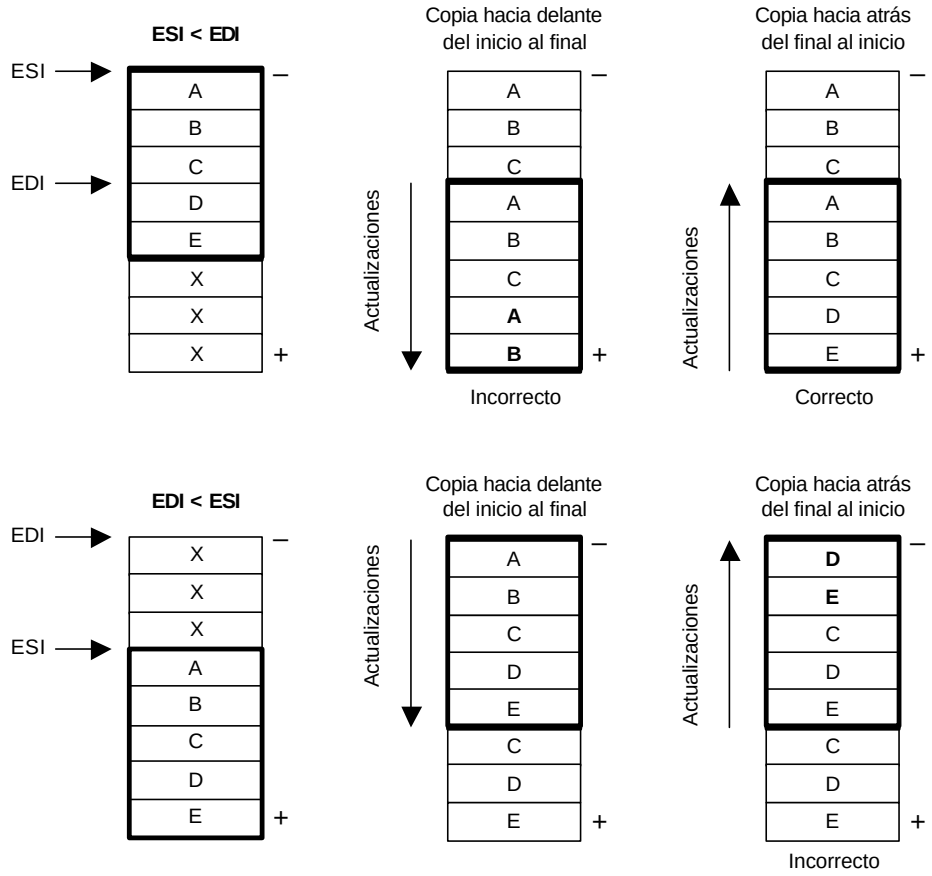
Los tres parámetros son desplazamientos y cantidades de la memoria vídeo basadas en caracteres (tamaño palabra). Se copian tanto caracteres como indique el tercer parámetro desde la memoria del núcleo a la memoria de vídeo. Si el primer parámetro es nulo se borra la memoria de vídeo llenándola con el valor definido por la macro blank_color.



```
! PUBLIC void mem_vid_copy(u16 *src, unsigned dst, unsigned count);

MVC_ARGS      =      4 + 4 + 4 + 4  ! 4 + 4 + 4
!                    es edi esi eip  src dst ct

_mem_vid_copy:
       push    esi
       push    edi
       push    es                    ! salvamos los registros

       mov     esi, MVC_ARGS(esp)    ! esi ← origen
       mov     edi, MVC_ARGS+4(esp)  ! edi ← destino
       mov     edx, MVC_ARGS+4+4(esp)! edx ← cantidad
       mov     es, (_vid_seg)        ! el destino es el segmento de vídeo
       cld                           ! aseguramos que la dirección es hacia delante

mvc_loop:
       and     edi, (_vid_mask)      ! truncar la dirección destino
       mov     ecx, edx              ! ecx ← contador
       mov     eax, (_vid_size)      ! eax ← tamaño bloque de vídeo
       sub     eax, edi
       cmp     ecx, eax
       jbe     0f
       mov     ecx, eax              ! ecx ← min(ecx, _vid_size - edi)
0:     sub     edx, ecx              ! edx ← resto que queda por copiar
       shl     edi, 1                ! arreglar dirección a byte
       test    esi, esi
       jz      mvc_blank             ! si origen == 0 borrar pantalla

mvc_copy:
       rep                           ! copiar palabras en memoria vídeo
    o16 movs
       jmp     mvc_test

mvc_blank:
       mov     eax, (_blank_color)   ! eax ← carácter de borrado
       rep
    o16 stos                         ! copiar blancos en memoria de vídeo
       !jmp    mvc_test

mvc_test:
       shr     edi, 1                ! arreglar direcciones a palabra
       test    edx, edx
       jnz     mvc_loop              ! si quedan palabras por copiar repetir bucle

mvc_done:
       pop     es
       pop     edi
       pop     esi                   ! restablecer registros

       ret                           ! retornar
```

5.5.2.- _vid_vid_copy

Copia un bloque dentro de la memoria de vídeo misma. Esto se utiliza para desplazar, insertar o borrar caracteres y líneas en pantalla. Esta rutina es algo más complicada ya que tiene que tener en cuenta el hecho de que los bloques origen y destino pueden solaparse, para lo cual hemos creado el siguiente diagrama donde se muestran cómo debe ser la copia según sea la dirección origen (ESI) menor que la dirección destino (EDI) o viceversa.



Los tres parámetros son desplazamientos y cantidades de la memoria vídeo basadas en caracteres (tamaño palabra). Se copian tanto caracteres como indique el tercer parámetro desde un lugar de la memoria de vídeo a otro dentro de la misma.

```
! PUBLIC void vid_vid_copy(unsigned src, unsigned dst, unsigned count);

VVC_ARGS        =       4 + 4 + 4 + 4  ! 4 + 4 + 4
!                       es edi esi eip  src dst ct

_vid_vid_copy:
        push    esi
        push    edi
        push    es                      ! salvar registros

        mov     esi, VVC_ARGS(esp)      ! esi ← origen
        mov     edi, VVC_ARGS+4(esp)    ! edi ← destino
        mov     edx, VVC_ARGS+4+4(esp)! edx ← contador
        mov     es, (_vid_seg)          ! el destino es el segmento de vídeo
        cmp     esi, edi                ! copiamos hacia delante o hacia atrás?
        jb      vvc_down
```

```
vvc_up:
       cld                          ! la dirección es hacia delante

vvc_uploop:
       and     esi, (_vid_mask)     ! truncamos las direcciones destino y origen
       and     edi, (_vid_mask)
       mov     ecx, edx             ! ecx ← contador
       mov     eax, (_vid_size)     ! eax ← tamaño bloque de vídeo
       sub     eax, esi
       cmp     ecx, eax
       jbe     0f
       mov     ecx, eax             ! ecx ← min(ecx, vid_size - esi)

0:     mov     eax, (_vid_size)
       sub     eax, edi
       cmp     ecx, eax
       jbe     0f
       mov     ecx, eax             ! ecx = min(ecx, vid_size - edi)

0:     sub     edx, ecx             ! edx ← resto que queda por copiar
       shl     esi, 1
       shl     edi, 1               ! arreglar direcciones a byte
       rep
eseg o16 movs                       ! copiar palabras en memoria de vídeo

       shr     esi, 1
       shr     edi, 1               ! arreglar direcciones a palabra
       test    edx, edx
       jnz     vvc_uploop           ! si quedan palabras por copiar repetir bucle
       jmp     vvc_done

vvc_down:
       std                          ! la dirección es hacia atrás
       lea     esi, -1(esi)(edx*1)  ! empezar a copiar por el final
       lea     edi, -1(edi)(edx*1)

vvc_downloop:
       and     esi, (_vid_mask)     ! truncamos las direcciones destino y origen
       and     edi, (_vid_mask)
       mov     ecx, edx             ! ecx ← contador
       lea     eax, 1(esi)
       cmp     ecx, eax
       jbe     0f
       mov     ecx, eax             ! ecx ← min(ecx, esi + 1)
0:     lea     eax, 1(edi)
       cmp     ecx, eax
       jbe     0f
       mov     ecx, eax             ! ecx ← min(ecx, edi + 1)
0:     sub     edx, ecx             ! edx ← resto que queda por copiar
       shl     esi, 1
       shl     edi, 1               ! arreglar direcciones a byte
       rep
eseg o16 movs                       ! copiar palabras en memoria de vídeo

       shr     esi, 1
       shr     edi, 1               ! arreglar direcciones a palabra
       test    edx, edx
       jnz     vvc_downloop         ! si quedan palabras por copiar repetir bucle
       cld                          ! la dirección hacia delante para compilador de C
       !jmp    vvc_done

vvc_done:
       pop     es
       pop     edi
       pop     esi                  ! restablecer registros

       ret                          ! retornar
```

5.6.- Manejo de mensajes

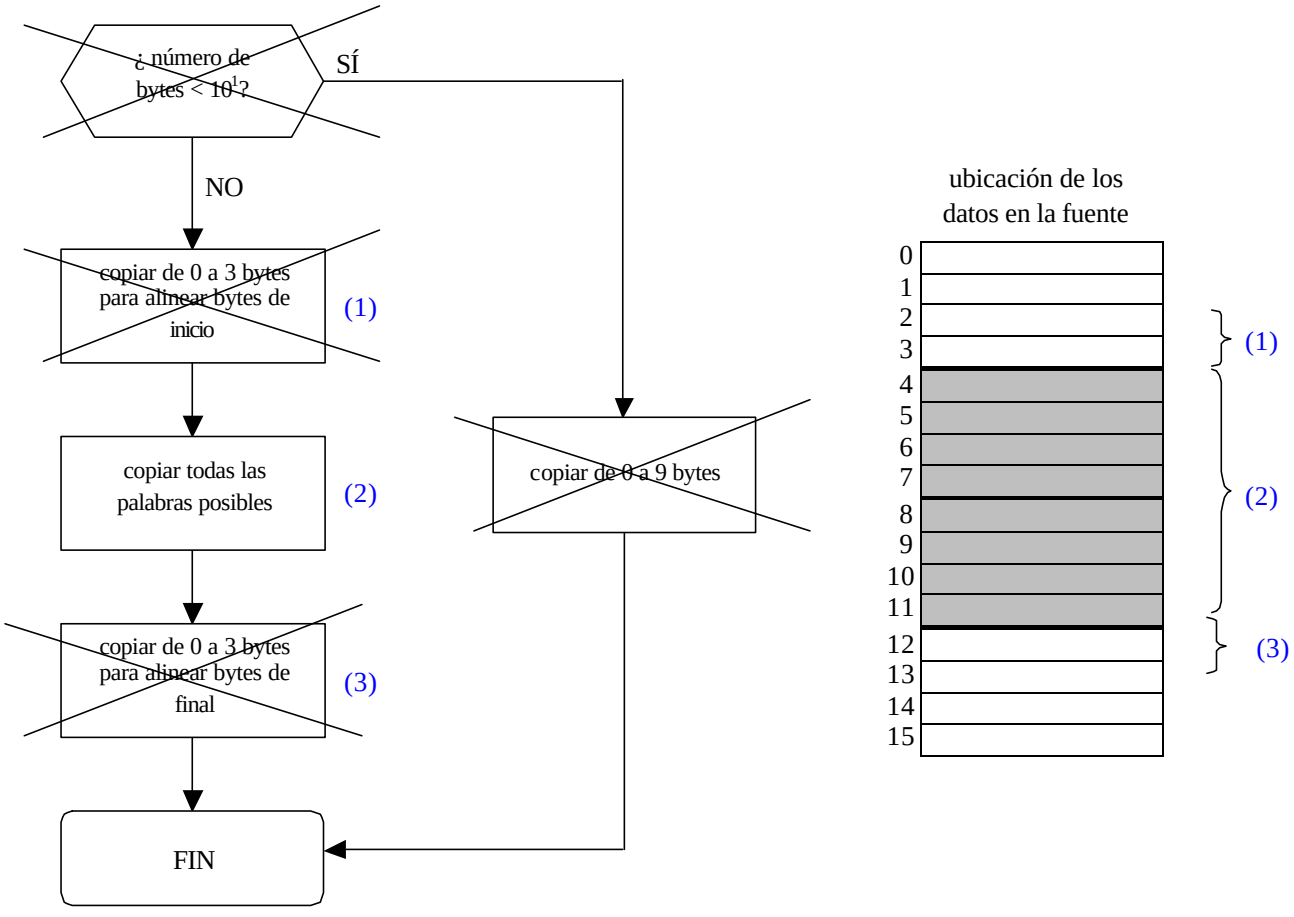
En este módulo sólo se proporciona una función para la copia de mensajes en memoria.

5.6.1.- _cp_mess

Copia mensajes de una zona de memoria a cualquier otra. También copia la dirección origen que se pasa como primer parámetro en la primera palabra de la zona destino en la que se copia el mensaje.

Aunque se podría haber utilizado la función *_phys_copy* para realizar esta tarea, se ha suministrado *_cp_mess*, pues es un procedimiento especializado más rápido.

En el siguiente diagrama se muestra la diferencia entre esta función y *_phys_copy*. La rutina se simplifica bastante pues el tamaño del mensaje es fijo y está alineado a palabra:



Los parámetros son los siguientes: el primero indica el número del proceso emisor del mensaje; el segundo indica una dirección de memoria en clicks (para máquinas Intel un click=256 bytes); el tercero indica el desplazamiento en bytes que hay que añadirle al parámetro anterior; el cuarto indica la dirección destino del mensaje a copiar, también en clicks; y el quinto es el desplazamiento en bytes que hay que añadirle al parámetro anterior. ¿Y cuál es el tamaño del mensaje a copiar? Este es un parámetro fijo definido en la constante Msize que para máquinas con palabra de 32 bits es de 9 palabras.

```
! PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
!                   phys_clicks dst_clicks, vir_bytes dst_offset);
!
! Note that the message size, "Msize" is in DWORDS (not bytes) and must be set
! correctly.  Changing the definition of message in the type file and not
! changing it here will lead to total disaster.

CM_ARGS=      4 + 4 + 4 + 4 + 4     ! 4 + 4 + 4 + 4 + 4
!             es  ds edi esi eip    proc scl sof dcl dof

        .align 16
_cp_mess:
        cld                         ! la dirección es hacia delante
        push    esi
        push    edi
        push    ds
        push    es                  ! salvar registros

        mov     eax, FLAT_DS_SELECTOR
        mov     ds, ax
        mov     es, ax              ! inicializamos registros de segmento

        mov     esi, CM_ARGS+4(esp)           ! esi ← dirección fuente en clicks
        shl     esi, CLICK_SHIFT              ! esi ← dirección fuente en bytes
        add     esi, CM_ARGS+4+4(esp)         ! esi ← dirección fuente completa en bytes
        mov     edi, CM_ARGS+4+4+4(esp)       ! edi ← dirección destino en clicks
        shl     edi, CLICK_SHIFT              ! edi ← dirección destino en bytes
        add     edi, CM_ARGS+4+4+4+4(esp)     ! edi ← dirección destino completa en bytes

        mov     eax, CM_ARGS(esp)     ! eax ← número de proceso del emisor
        stos                          ! número que copiamos en el destino
        add     esi, 4                ! saltamos la primera palabra
        mov     ecx, Msize - 1        ! ecx ← Msize - primera palabra ya copiada
        rep
        movs                          ! copiar el mensaje

        pop     es
        pop     ds
        pop     edi
        pop     esi                   ! restablecer los registros

        ret                           ! esto es todo amigos!
```

5.7.- Manejo de interrupciones

En este módulo se describen las funciones que tienen que ver con las interrupciones.

5.7.1.- _lock

Es una sencilla función sin parámetros que únicamente inhabilita las interrupciones de la CPU.

```
! PUBLIC void lock();

        .align 16
_lock:
        cli                             ! inhabilita las interrupciones
        ret
```

5.7.2.- _unlock

Análogamente a la función anterior ésta vuelve a habilitar las interrupciones de la CPU.

```
! PUBLIC void unlock();

        .align 16
_unlock:
        sti
        ret                             ! habilita las interrupciones
```

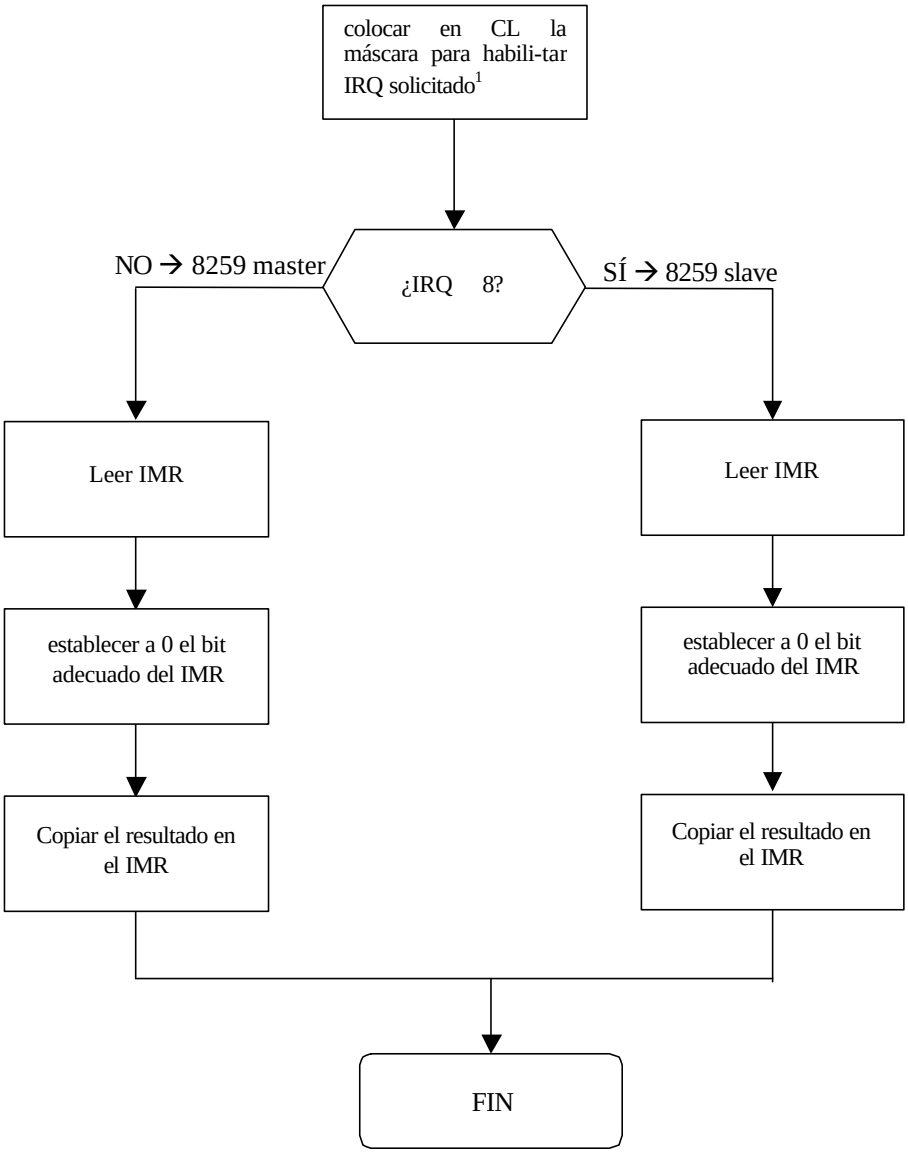
5.7.3.- _reset

Esta función restablece el procesador cargando el registro de la tabla de descriptores de interrupciones del procesador con un apuntador nulo y ejecutando a continuación una interrupción software. Esto tiene el mismo efecto que un “reset” de hardware.

```
! PUBLIC void reset();

_reset:
        lidt    (idt_zero)      ! carga en el registro IDTR un apuntador nulo
        int     3               ! todo vale; al 386 no le gustará

.sect .data
idt_zero:       .data4 0, 0
.sect .text
```

5.7.4.- _enable_irq

Esta función es algo complicada: opera en el nivel de los chips controladores de interrupciones para habilitar interrupciones de hardware individuales.



([1]): debe realizarse de tal forma que cuando se quiera habilitar la IRQ 12, por ejemplo, se habilite la IRQ 4 del esclavo, es decir, el rango es IRQ = [8, 15] equivalente a IRQ = [0, 7] del 8259 que actúa como esclavo.

El único parámetro que necesita esta función es el número de irq.

Para habilitar interrupciones hardware con irq menor que 8 también se podría haber usado la siguiente combinación de las funciones *_out_byte* e *_in_byte*:

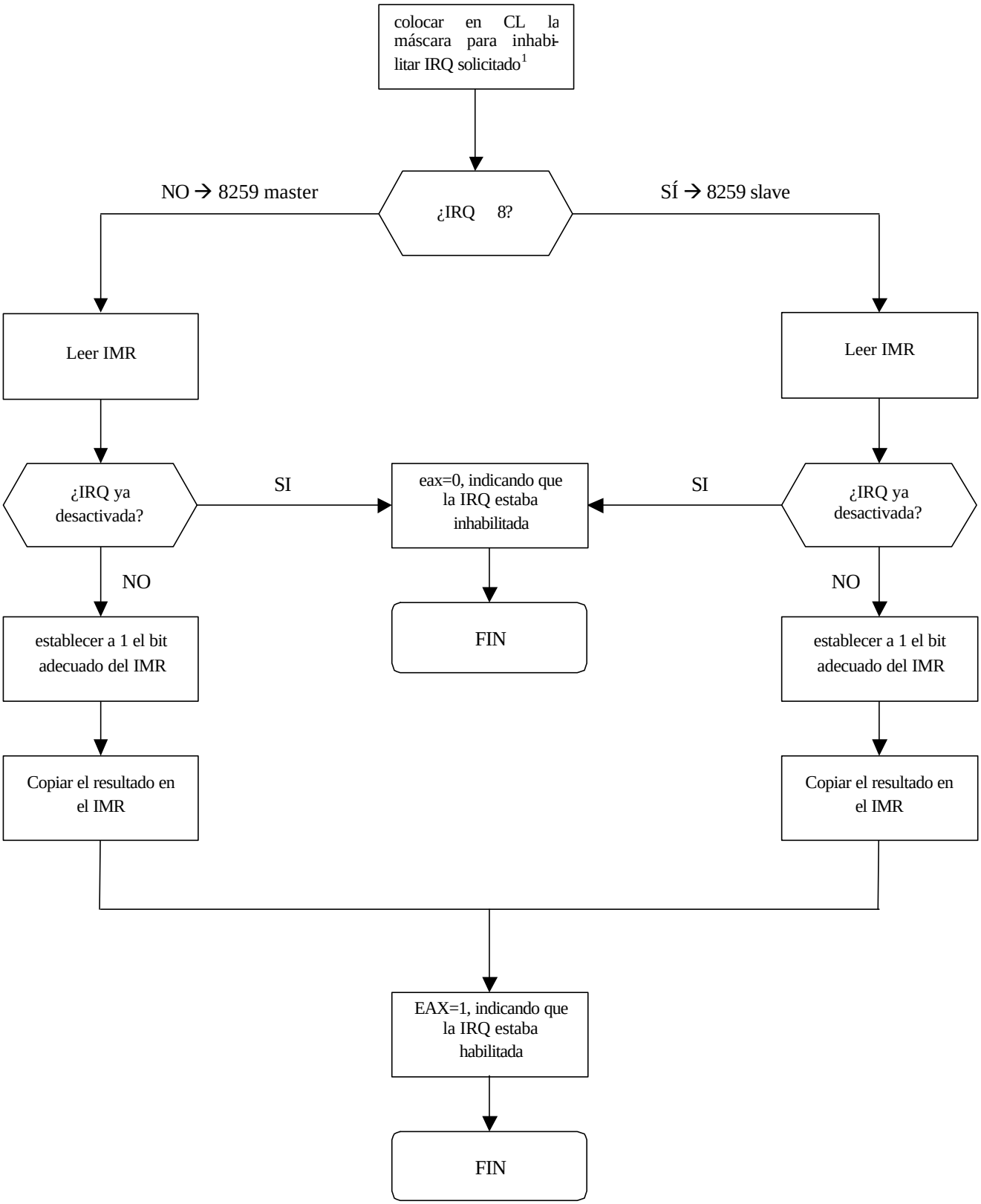
```
out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) & ~(1 << irq));
```

```
! PUBLIC void enable_irq(unsigned irq)

        .align 16
_enable_irq:
        mov     ecx, 4(esp)             ! ecx ← número de irq
        pushf                           ! guardamos el registro de flags en la pila
        cli                             ! deshabilitamos las interrupciones software

        movb    ah, ~1
        rolb    ah, cl                  ! ah ← ~(1<<(irq % 8)) máscara para habilitar irq
        cmpb    cl, 8
        jae     enable_8                ! si irq >= 8 tratar con chip 8259 slave

enable_0:
        inb     INT_CTLMASK             ! al ← IMR
        andb    al, ah                  ! al ← máscara & IMR
        outb    INT_CTLMASK             ! bit=0 en el chip 8259 master

        popf                            ! restablecemos registro de flags
        ret                             ! retornar
        .align 4

enable_8:
        inb     INT2_CTLMASK            ! al ← IMR
        andb    al, ah                  ! al ← máscara & IMR
        outb    INT2_CTLMASK            ! bit=0 en el chip 8259 slave

        popf                            ! restablecemos registro de flags
        ret                             ! retornar
```

5.7.5.- _disable_irq

Análogamente a la función anterior esta sirve para deshabilitar interrupciones hardware individuales.

El único parámetro que necesita esta función es el número de irq y retorna verdadero y si la interrupción no estaba todavía deshabilitada.


colocar en CL la máscara para inhabilitar IRQ solicitado[1]
¿IRQ 8?
NO → 8259 master
SÍ → 8259 slave
Leer IMR
Leer IMR
¿IRQ ya desactivada?
SI
eax=0, indicando que la IRQ estaba inhabilitada
SI
¿IRQ ya desactivada?
NO
NO
FIN
establecer a 1 el bit adecuado del IMR
establecer a 1 el bit adecuado del IMR
Copiar el resultado en el IMR
Copiar el resultado en el IMR
EAX=1, indicando que la IRQ estaba habilitada
FIN

Para habilitar interrupciones hardware con irq menor que 8 también se podría haber usado la siguiente combinación de las funciones _out_byte e _in_byte:

```
out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) | (1 << irq));
```

```
! PUBLIC int disable_irq(unsigned irq)

        .a! PUBLIC int disable_irq(unsigned irq)

        .align 16
_disable_irq:
        mov     ecx, 4(esp)             ! ecx ← número de irq
        pushf                           ! guardamos el registro de flags en la pila
        cli                             ! deshabilitamos las interrupciones software
        movb    ah, 1
        rolb    ah, cl                  ! ah ← (1<<(irq %8)) máscara para deshabilitar irq
        cmpb    cl, 8
        jae     disable_8               ! si irq >= 8 tratar con chip 8259 slave

disable_0:
        inb     INT_CTLMASK
        testb   al, ah
        jnz     dis_already             ! already disabled?
        orb     al, ah
        outb    INT_CTLMASK             ! set bit at master 8259
        popf
        mov     eax, 1                  ! disabled by this function
        ret
disable_8:
        inb     INT2_CTLMASK
        testb   al, ah
        jnz     dis_already             ! already disabled?
        orb     al, ah
        outb    INT2_CTLMASK            ! set bit at slave 8259
        popf
        mov     eax, 1                  ! disabled by this function
        ret
dis_already:
        popf
        xor     eax, eax                ! already disabled
        ret
```

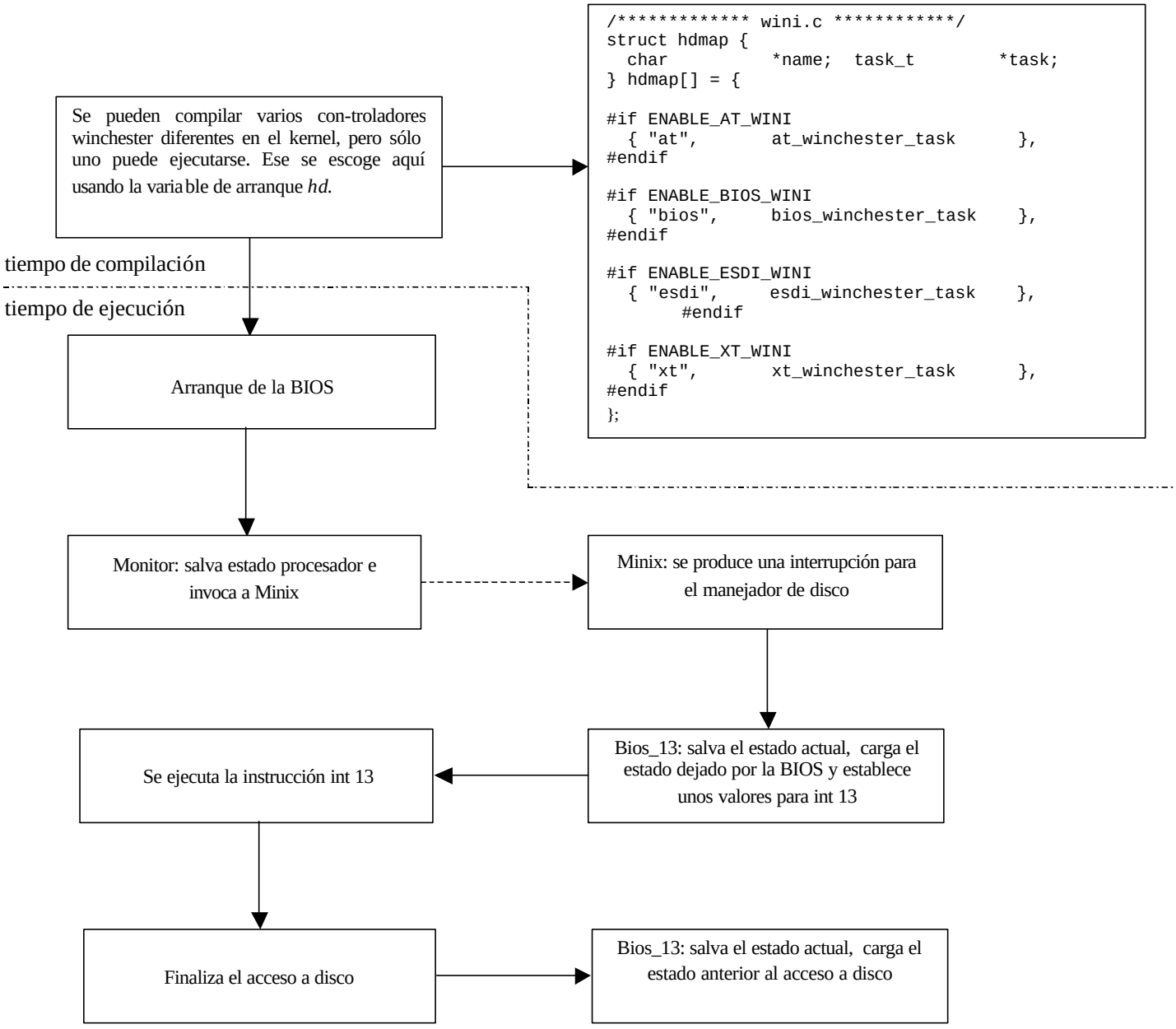
5.7.6.- _bios_13

Esta función no es compleja en sí, sino el marco en el que está ubicada. Pasamos a explicarlo: Minix no utiliza normalmente los servicios de E/S que le ofrece la BIOS, sino que define sus propios drivers para acceder a los dispositivos. Sin embargo, en los ordenadores que no son completamente compatibles con el estándar IBM PC, debe utilizar ésta para tal fin.

El marco de esta función engloba sólo al disco duro. Cuando éste no es AT, Minix debe utilizar el servicio 13 de la bios para poder leer y escribir en el mismo. El código de que realiza dicha tarea se encuentra en el fichero bios_wini.c.

Como paso previo, para poder realizar esta tarea, es necesario restablecer todos los registros y tablas del procesador al estado en el que se encontraban antes de que Minix se cargase (estado que fue salvado por el monitor). Una vez finalizada la operación de disco es necesario realizar el proceso inverso que deja el estado del procesador tal como se encontraba antes del acceso a disco. Ambas tareas las lleva a cabo la función *bios_13*.

Así el proceso aparece resumido en el siguiente diagrama:



```
! PUBLIC void bios13();

        .define _bios13
_bios13:
        cmpb    (_mon_return), 0      ! comprueba si está el monitor
        jnz     0f
        movb    (_Ax+1), 0x01         ! "comando no válido"
        ret
0:      push    ebp                   ! salvar los registros de C
        push    esi
        push    edi
        push    ebx
        pushf                         ! salvar el registro de flags
        cli                           ! deshabilitar las interrupciones

        inb     INT2_CTLMASK          ! al ← IMR del chip 8259 slave
        movb    ah, al
        inb     INT_CTLMASK
        push    eax                   ! salvar ambas máscaras de interrupción
        mov     eax, (_irq_use)       ! mapear los IRQs en uso
        and     eax, ~[1<<CLOCK_IRQ]  ! manejador de reloj especial
        outb    INT_CTLMASK           ! habilitar todos los IRQs sin usar y viceversa
        movb    al, ah
        outb    INT2_CTLMASK

        mov     eax, cr0
        push    eax                   ! salvar la palabra de estado de la máquina

        mov     eax, SS_SELECTOR      ! segmento de datos del monitor
        mov     ss, ax
        xchg    esp, (_mon_sp)        ! intercambia punteros de pila

        mov     ds, ax                ! restantes selectores de datos
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        push    cs
        push    return                ! dirección y selector de retorno del kernel
    o16 jmpf    16+2*4+5*2+2*4(esp)   ! realizar la llamada

return:
    o16 pop     (_Ax)
    o16 pop     (_Bx)
    o16 pop     (_Cx)
    o16 pop     (_Dx)
    o16 pop     (_Es)
        lgdt    (_gdt+GDT_SELECTOR)   ! volver a cargar la tabla de descriptores globales
        jmpf    CS_SELECTOR:csinit    ! restaurar todo

csinit: mov     eax, DS_SELECTOR
        mov     ds, ax
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        mov     ss, ax
        xchg    esp, (_mon_sp)        ! volver a intercambiar punteros de pila
        lidt    (_gdt+IDT_SELECTOR)   ! volver a cargar la tabla de descr. de interrup.
        andb    (_gdt+TSS_SELECTOR+DESC_ACCESS), ~0x02  ! pone a cero el bit de ocupado
                                                        ! de la TSS

        mov     ax, TSS_SELECTOR
        ltr     ax                    ! cargar el registro de la TSS

        pop     eax
        mov     cr0, eax              ! restaurar la palabra de estado de la máquina
```

```
        pop     eax
        outb    INT_CTLMASK             ! restaurar las máscaras de interrupción
        movb    al, ah
        outb    INT2_CTLMASK

        popf                            ! restaurar el registro de flags
        pop     ebx                     ! restaurar los registros de C
        pop     edi
        pop     esi
        pop     ebp
        ret

.sect .bss
.define _Ax, _Bx, _Cx, _Dx, _Es         ! variables de registro del 8086
.comm  _Ax, 4
.comm  _Bx, 4
.comm  _Cx, 4
.comm  _Dx, 4
.comm  _Es, 4
.sect .text
```

6.- Utilidades del kernel

En este apartado mostramos cuatro funciones especializadas escritas en lenguaje C y que son usadas por las demás rutinas del núcleo del sistema operativo.

6.1.- mem_init

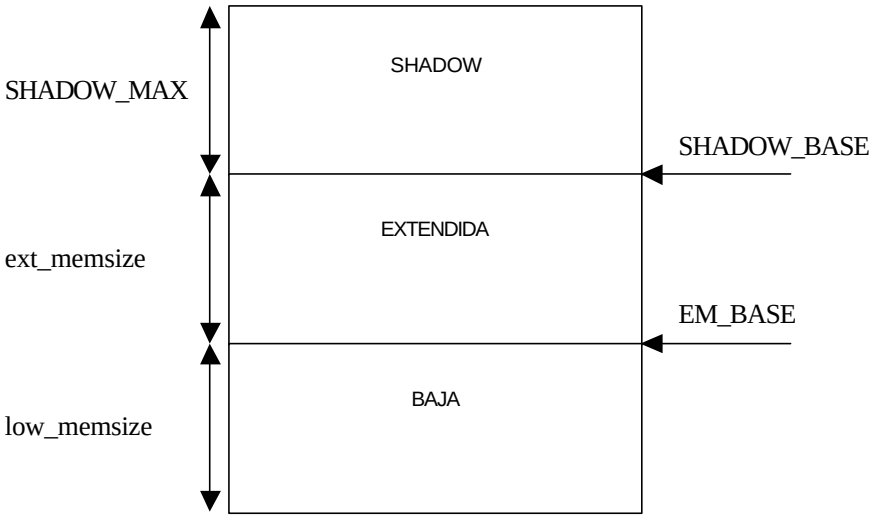
Las direcciones a veces se almacenan en clicks para reducir el espacio necesario para su almacenamiento. Una dirección en clicks, es una dirección que está alineada a 256 bytes, por lo que su últimos 8 bits son siempre cero. Es por ello que estos últimos bits nunca se almacenan.

Dirección decimal	Dirección en binario	Dirección en clicks
256	0000 0001 0000 0000	0000 0001
512	0000 0010 0000 0000	0000 0010
1024	0000 0100 0000 0000	0000 0100

La definición del tamaño de un click se encuentra en el fichero *const.h*:

```
06857  #if (CHIP == INTEL)
06858  #define CLICK_SIZE       256    /* unit in which memory is allocated */
06859  #define CLICK_SHIFT        8    /* log2 of CLICK_SIZE */
06860  #endif
```

En un ordenador compatible con los IBM PC puede haber dos o tres regiones de memoria no continuas.



La BIOS informa al monitor de arranque el tamaño del intervalo más bajo, que los usuarios del PC conocen como memoria "ordinaria", y el del intervalo de memoria que comienza encima del área de la ROM (memoria "extendida"). A su vez, el monitor pasa los valores como parámetros de arranque, que son interpretados por *cstart* y se escriben en *low_memsize* y *ext_memsize* durante el arranque. La tercera región es la memoria "sombra", en la que puede copiarse la ROM de la BIOS para mejorar el rendimiento, ya que la memoria ROM normalmente es más lenta que la RAM.

Mem_init sólo es invocada por la función principal *main*, cuando se inicia Minix. Su función es la inicializar las tablas donde se almacenan el tamaño y la base de cada una de las tres zonas de memoria anteriores.

```
#define EM_BASE     0x100000L   /* base of extended memory on AT's */
#define SHADOW_BASE 0xFA0000L   /* base of RAM shadowing ROM on some AT's */
#define SHADOW_MAX  0x060000L   /* maximum usable shadow memory (16M limit) */

PUBLIC void mem_init()
{
/* Initialize the memory size tables.  This is complicated by fragmentation
 * and different access strategies for protected mode.  There must be a
 * chunk at 0 big enough to hold Minix proper.  For 286 and 386 processors,
 * there can be extended memory (memory above 1MB).  This usually starts at
 * 1MB, but there may be another chunk just below 16MB, reserved under DOS
 * for shadowing ROM, but available to Minix if the hardware can be re-mapped.
 * In protected mode, extended memory is accessible assuming CLICK_SIZE is
 * large enough, and is treated as ordinary memory.
 */

 u32_t ext_clicks;
 phys_clicks max_clicks;

 /* Se almacena el tamaño de la memoria baja que se ha leído de la Bios. */
 mem[0].size = k_to_click(low_memsize);        /* base = 0 */

 if (pc_at && protected_mode) {
        /* Get the size of extended memory from the BIOS.  This is special
         * except in protected mode, but protected mode is now normal.
         * Note that no more than 16M can be addressed in 286 mode, so make
         * sure that the highest memory address fits in a short when counted
         * in clicks.
         */

       /*se almacena el tamaño de la memoria extendida leída de la BIOS*/
        ext_clicks = k_to_click((u32_t) ext_memsize);
        max_clicks = USHRT_MAX - (EM_BASE >> CLICK_SHIFT);
        mem[1].size = MIN(ext_clicks, max_clicks);
        mem[1].base = EM_BASE >> CLICK_SHIFT;

       /*Se determina si hay memoria shadow y se comprueba que es correcta*/
        if (ext_memsize <= (unsigned) ((SHADOW_BASE - EM_BASE) / 1024)
                        && check_mem(SHADOW_BASE, SHADOW_MAX) == SHADOW_MAX) {
                /* Shadow ROM memory. */
                mem[2].size = SHADOW_MAX >> CLICK_SHIFT;
                mem[2].base = SHADOW_BASE >> CLICK_SHIFT;
        }
  }

  /* Memoria total del sistema */
  tot_mem_size = mem[0].size + mem[1].size + mem[2].size;
}
```

6.2.- env_parse

Esta rutina también se usa durante el inicio del sistema.

El monitor de arranque puede pasar cadenas arbitrarias como “DPETH0=300:10” a Minix en los parámetros de arranque. *Env_parse* trata de encontrar una cadena cuyo primer campo coincida con su primer argumento, y luego extraer el campo solicitado. Los comentarios del código explican el uso de la función, que se proporciona principalmente para ayudar al usuario que desea agregar nuevos controladores que tal vez requieran parámetros. El ejemplo “DPETH0” se usa para pasar información de configuración a un adaptador de Ethernet cuando se incluye apoyo de red durante la compilación de Minix.

```
PUBLIC int env_parse(env, fmt, field, param, min, max)
char *env;              /* variable de entorno a inspeccionar */
char *fmt;              /* template to parse it with */
int field;              /* field number of value to return */
long *param;            /* address of parameter to get */
long min, max;          /* minimum and maximum values for the parameter */
{

  char *val, *end;
  long newpar;
  int i = 0, radix, r;

  if ((val = k_getenv(env)) == NIL_PTR) return(EP_UNSET);
  if (strcmp(val, "off") == 0) return(EP_OFF);
  if (strcmp(val, "on") == 0) return(EP_ON);

  r = EP_ON;
  for (;;) {
        while (*val == ' ') val++;

        if (*val == 0) return(r);       /* the proper exit point */

        if (*fmt == 0) break;           /* too many values */

        if (*val == ',' || *val == ':') {
                /* Time to go to the next field. */
                if (*fmt == ',' || *fmt == ':') i++;
                if (*fmt++ == *val) val++;
        } else {
                /* Environment contains a value, get it. */
                switch (*fmt) {
                case 'd':       radix =   10;   break;
                case 'o':       radix =  010;   break;
                case 'x':       radix = 0x10;   break;
                case 'c':       radix =    0;   break;
                default:        goto badenv;
                }
                newpar = strtol(val, &end, radix);

                if (end == val) break;  /* not a number */
                val = end;

                if (i == field) {
                        /* The field requested. */
                        if (newpar < min || newpar > max) break;
                        *param = newpar;
                        r = EP_SET;
                }
        }
  }
badenv:
  printf("Bad environment setting: '%s = %s'\n", env, k_getenv(env));
  panic("", NO_NUM);
  /*NOTREACHED*/
}
```

6.3.- bad_assertion y bad_compare

Éstas funciones sólo se compilan si la macro DEBUG se define como TRUE, y apoyan las macros de *assert.h*. Aunque no se hace referencia a ellas en ninguna parte del código del sistema operativo, pueden ser de utilidad durante la depuración para el alumno que desee crear una versión modificada de Minix.

```
#if DEBUG


PUBLIC void bad_assertion(file, line, what)
char *file;
int line;
char *what;
{
  printf("panic at %s(%d): assertion \"%s\" failed\n", file, line, what);
  panic(NULL, NO_NUM);
}


PUBLIC void bad_compare(file, line, lhs, what, rhs)
char *file;
int line;
int lhs;
char *what;
int rhs;
{
  printf("panic at %s(%d): compare (%d) %s (%d) failed\n",
         file, line, lhs, what, rhs);
  panic(NULL, NO_NUM);
}


#endif /* DEBUG */
```

7.- Cuestiones

1) **¿Por qué crees que la librería está escrita en ensamblador?**

Hay dos razones:

a) hay una mejora en el rendimiento (en el tiempo de ejecución),

b) Minix no se escribió para compiladores de alto nivel como Turbo C, Borland C, ... que realizan operaciones de bajo nivel.

2) **¿Cuáles son las cuatro categorías principales de interrupciones?**

Interrupciones en respuesta a algún acontecimiento, tal como la pulsación de una tecla (interrupción hardware).

Interrupciones generadas por la CPU debido a alguna anomalía en el programa, como una división por cero (excepciones).

Interrupciones generadas por programas, por ejemplo las llamadas al sistema (interrupciones software).

Interrupciones que requieren intervención inmediata de la CPU, como una caída de voltaje (interrupciones no enmascarables).

3) **¿A qué se debe la existencia de las funciones *port_read_byte* y *port_write_byte*?**

Estas funciones se han añadido para que sean usadas en aquellos dispositivos periféricos antiguos que únicamente pueden realizar transferencias desde memoria al puerto y viceversa, en unidades de byte.

4) **¿Bajo qué circunstancias se utiliza la rutina *_bios_13* y cuál es su función?**

Se utiliza cuando el disco duro del PC no sigue el estándar establecido en el IBM PC, y se requiere acceder a las rutinas de la bios para acceder al disco.

Como paso previo, para poder realizar esta tarea, es necesario restablecer todos los registros y tablas del procesador al estado en el que se encontraban antes de que Minix se cargase (estado que fue salvado por el monitor). Una vez finalizada la operación de disco es necesario realizar el proceso inverso que deja el estado del procesador tal como se encontraba antes del acceso a disco. Ambas tareas las lleva a cabo la función *bios_13*.

5) **¿Cuál es la diferencia entre la función *_phys_copy* y la función *_cp_mess*?**

Ambas funciones sirven para copiar datos dentro de la memoria. La diferencia radica en que *_phys_copy* puede copiar un bloque de datos de cualquier tamaño y sin estar alineado, y la función *_cp_mess* está optimizada para la copia de mensajes entre procesos, en la cual el tamaño del mensaje es fijo y los datos se encuentran alineados en palabra.

8.- Bibliografía

"80386/80286 assembly language programming"
William H. Murray III, Chris H. Pappas
Editorial Osborne McGraw-Hill, 1986

"Sistemas Operativos: Diseño e implementación" Segunda Edición
Andrew S. Tanenbaum, Albert S. Woodhull
Editorial Prentice Hall, 1997

"PC Interno 5"

"Utilidades y Librería del Kernel: klib.s"
Alexis Caballero Caballero, Pablo Pérez González
Ampliación de Sistemas Operativos, curso 1999-2000