

INCLUYE
CD-ROM



PROGRAMACIÓN

ESPECIALIZACIÓN
EN PROGRAMACIÓN
DE COMPUTADORES

FRANCISCO CHARTE OJEDA



ANAYA
MULTIMEDIA

Ensamblador

Edición 2009

Francisco Cnarte Ojeda



Todos los nombres propios de programas, sistemas operativos, equipos hardware, etc. que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

Reservados todos los derechos. El contenido de esta obra está protegido por la Ley, que establece penas de prisión y/o multas, además de las correspondientes indemnizaciones por daños y perjuicios, para quienes reprodujeren, plagiaren, distribuyeren o comunicaren públicamente, en todo o en parte, una obra literaria, artística o científica, o su transformación, interpretación o ejecución artística fijada en cualquier tipo de soporte o comunicada a través de cualquier medio, sin la preceptiva autorización.

© EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S.A.), 2009
Juan Ignacio Luca de Tena, 15. 28027 Madrid
Depósito legal: M. 49.604-2008
ISBN: 978-84-415-2511-5
Printed in Spain
Impreso en: Closas-Orcoyen, S. L.

Agradecimientos

La terminación de un nuevo libro, que sigue su camino hacia el lector, es siempre una experiencia gratificante. En este caso esa sensación es, si cabe, aún mayor, dado que el libro que ahora tiene en sus manos ha recorrido un largo camino y tiene una historia de muchos años detrás.

Comencé a programar en ensamblador a mediados de la década de los 80 utilizando un microordenador MSX, con un Z80 en su interior, y un ensamblador/desensamblador en formato de cartucho. Dos años después inicié la redacción de un libro sobre programación en ensamblador Z80 que, no mucho después, quedó detenido en favor de la redacción de otros títulos.

Mi primer agradecimiento va dirigido a mis antiguos alumnos de los cursos de ensamblador 8086 en *Centro de Estudios Informáticos* (también a ti, querido Andrés) de Jaén, en los años 89 y 90. Su entusiasmo por aprender, y la ilusión que ponían en el desarrollo de cada proyecto, me llevó a reciclar aquel antiguo libro y convertirlo en un libro de programación en ensamblador 8086 para DOS. Unas doscientas páginas, recogiendo la formación de aquellos cursos, volvieron a quedar en el olvido cuando un año después cambié de trabajo.

Dada la imposibilidad de finalizar y publicar aquel libro, sacié mis ganas de escribir sobre ensamblador publicando diversos artículos en la desaparecida revista RPP, a cuyo director, Daniel Alonso, agradezco su apoyo siempre incondicional a mis ideas. Me alegra que te vaya bien en tu nueva singladura y espero que nuestros caminos vuelvan a cruzarse algún día.

Terminando el siglo pasado mis editores de *Anaya Multimedia* me propusieron la elaboración de un libro sobre programación en ensamblador. Obviamente acepté de inmediato, era algo que llevaba esperando años. Me puse manos a la obra y comencé mi nuevo libro pero, una vez más, quedó estancado al dar prioridad a otros proyectos (sí Visual Studio .NET, tú tienes la culpa 8-D). Finalmente, en los últimos meses de 2002 pude dedicarme a este libro y no podría haberlo hecho sin la confianza que siempre depositan en mí personas como José María Mourelle o Víctor Manuel Ruiz. Gracias a ambos.

Cinco años después, en 2008, la aceptación que tuvo esa primera edición de *Programación en ensamblador para DOS, Linux y Windows* llevó a mis editores a pedirme una actualización del título. Éste es el libro que tiene en sus manos, con ocho nuevos capítulos y la revisión y corrección de los procedentes de la edición previa. Mi agradecimiento a todos los lectores que han confiado en este trabajo y que, con su beneplácito, han propiciado la aparición de esta segunda edición.

Aunque para redactar la primera edición de este libro rescaté parte del trabajo que tenía esperando desde hacía una década, mi objetivo era actualizarlo para dar cabida a temas de actualidad que, por entonces, no había planteado tratar. Busqué las herramientas adecuadas para ello, parte de las cuales están incluidas en el CD-ROM que incluye el libro. Mi agradecimiento a todas las personas que en su momento, en la primera edición de este libro, me autorizaron a incluir su trabajo y, especialmente, a Javier Izquierdo, entonces en Microsoft Ibérica, por autorizar la inclusión de MASM y a Rob Anderton, autor de NASM-IDE quien, a cambio del permiso para incluirlo con el libro, sólo me pidió un

ejemplar de éste. En cuanto lo recibió añadió un enlace en su sitio Web, indicando que NASM-IDE había sido incluido en un libro que le pareció excelente. No deje de visitar la Web de Rob, en http://uk.geocities.com/rob_anderton/, para contar siempre con la última versión de NASM-IDE.

Como suele ocurrir en todos los proyectos, cuando se llega a la fecha límite, en este caso la fecha de entrega del material, todo son prisas y hay que acabar cuanto antes. Esto no sería posible sin el buen trabajo de Tomás Eisman (la persona que convierte mis archivos de texto e imágenes en un libro), la ayuda de mi esposa, María Jesús Luque, ocupándose de todas las tareas que habitualmente compartimos y, finalmente, la alegría que me aportan mis hijos, David y Alejandro. Gracias a todos, sin vosotros no estaría escribiendo esto.

índice de contenidos

Agradecimientos.....	6
Introducción.....	22
Microprocesadores.....	24
Sistemas hardware y sistemas operativos.....	25
Objetivos de este libro.....	25
1. Microprocesadores y sistemas basados en microprocesador.....	28
Del circuito integrado al microprocesador.....	30
Evolución de los microprocesadores.....	31
Fueron los primeros.....	31
Microprocesadores de 8 bits.....	32
Microprocesadores de 16 bits.....	33
CISC versus RISC.....	34
Microprocesadores modernos.....	35
Microprocesadores versus microcontroladores.....	36
Arquitectura común de una CPU.....	37
Arquitectura común de un sistema basado en microprocesador.....	41
Resumen.....	43
2. Representación de datos en ordenadores.....	44
Bases de numeración.....	46
Sistemas de numeración informáticos.....	46

Índice de contenidos

Cálculo del valor de una cifra.....	48
Conversión entre bases de numeración.....	49
Conversión a la base decimal desde cualquier base.....	49
Conversión a cualquier base desde la base decimal.....	50
Operar con números binarios.....	50
Bits, nibbles y bytes.....	51
Números con signo.....	52
Operar con números hexadecimales.....	54
De binario a hexadecimal y viceversa.....	54
Números negativos en hexadecimal.....	56
Números en base ocho.....	56
Identificación de la base de un número.....	56
Representación de números enteros.....	57
Big endian vs Little endian.....	58
Representación BCD.....	60
Representación de números en coma flotante.....	60
Normalización de la mantisa.....	61
Codificación del exponente.....	63
Codificación del signo.....	63
Representación de caracteres y cadenas.....	64
Resumen.....	65
 3. Arquitectura de la familia de microprocesadores x86.....	66
Estructura de bloques.....	67
Banco de registros.....	69
El registro de estado.....	71
Generación de direcciones.....	73
Patillaje del 8086.....	74
Buses de direcciones y datos.....	74
Modos de funcionamiento.....	76
Gestión de interrupciones.....	77
Los sucesores del 8086.....	77
Resumen.....	79
 4. Sistemas basados en microprocesadores x86.....	80
Estructura de bloques.....	81
Generador de reloj - 8284.....	83
Controlador de bus - 8288.....	84
Reloj programable - 8253.....	85
Interfaz programable de periféricos - 8255.....	89
Controlador programable de interrupciones - 8259.....	92
Controlador de acceso directo a memoria - 8237.....	94
Resumen.....	96

5. Modos de direccionamiento.....	98
Direccionamiento por registro.....	99
Direccionamiento inmediato.....	100
Direccionamiento directo.....	102
Direccionamiento indirecto.....	103
Direccionamiento indexado.....	104
Registros de segmento por defecto.....	105
Modos de direccionamiento del 80386.....	106
Resumen.....	106
6. Conjunto de instrucciones.....	108
Instrucciones aritméticas.....	110
Instrucciones lógicas y de rotación/traslación.....	111
Instrucciones de conversión.....	111
Instrucciones de cadena.....	112
Instrucciones de transferencia de datos.....	112
Instrucciones de control de flujo.....	113
Instrucciones de entrada/salida.....	115
Instrucciones de control.....	116
Otras instrucciones.....	116
Resumen.....	117
7. Herramientas necesarias.....	118
Editores.....	119
DOS.....	121
Windows.....	123
Linux.....	124
Ensambladores.....	125
MASM.....	126
NASM.....	128
Otros ensambladores.....	130
RAD y ensamblador.....	130
Enlazadores.....	130
Depuradores.....	132
Otras herramientas.....	133
Resumen.....	133
8. Nuestro primer programa.....	136
Esqueleto de un programa mínimo.....	137
Programas COM en DOS.....	138
El código.....	138
Ensamblado y enlace.....	140

Índice de contenidos

Programas EXE en DOS.....	142
Versión MASM.....	142
Versión NASM.....	143
Ensamblado y enlace.....	145
Programas Linux.....	146
El código.....	147
Ensamblado y enlace.....	147
Programas Windows.....	148
El código.....	149
Ensamblado y enlace.....	150
Resumen.....	151
 9. Ejecución de un programa.....	152
Formatos de archivo ejecutable.....	153
Ejecutables en DOS.....	154
Ejecutables en Linux.....	155
Ejecutables en Windows.....	155
Detalles sobre formatos de archivo.....	156
Preparación del programa por parte del sistema.....	157
Recuperación de la cabecera del ejecutable.....	158
Asignación de bloques de memoria.....	158
Creación de un proceso.....	159
Configuración de registros.....	160
Segmentos de código, datos y pila.....	160
El puntero de instrucción.....	161
Base y puntero de la pila.....	161
Acceso a los datos.....	162
Resumen.....	162
 10. Registros y memoria.....	164
Unidades de información.....	165
Palabras y dobles palabras.....	166
Múltiplos del byte.....	167
Capacidad de direccionamiento.....	167
Registros de segmento.....	168
Párrafos y segmentos.....	169
Modelos de memoria.....	170
Registros de uso general.....	172
Asignación de valores.....	173
Valores inmediatos.....	173
Asignación entre registros.....	174
Lectura de datos de la memoria.....	175
Escritura de datos en la memoria.....	176

Definición de datos en el programa	177
Campos simples.....	177
Conjuntos de campos.....	178
Referencias al segmento de datos.....	179
Un ejemplo.....	179
Resumen.....	182
11. Depuración.....	184
Puesta en marcha del depurador.....	186
Nombres de archivos DOS.....	186
Apertura desde DEBUG.....	187
Análisis del programa.....	188
Direcciones, instrucciones y código máquina.....	189
Traducción de etiquetas.....	191
Examen del contenido de datos.....	191
Estado inicial de los registros.....	193
Ejecución paso a paso.....	194
Depuración de rutinas y BIOS.....	194
Ejecución hasta un cierto punto.....	196
Alteración del curso del programa.....	197
Modificar el contenido de un registro.....	197
Cambiar los datos en memoria.....	197
Ensamblar nuevas instrucciones.....	200
Otras posibilidades de DEBUC.....	202
Resumen.....	203
12. Operaciones aritméticas.....	204
Suma de dos números.....	205
Desbordamiento y acarreo.....	207
Suma con acarreo.....	209
Sumas de 32 bits con registros de 16.....	210
Restar un número de otro.....	212
Multiplicar dos números.....	212
Dividir un número entre otro.....	214
Incrementos y reducciones.....	215
Aritmética BCD.....	217
Números BCD empaquetados y sin empaquetar.....	217
Suma de números BCD.....	219
Otras operaciones con números BCD.....	221
Negativos, palabras y dobles palabras.....	223
Uso de la unidad de punto floLante.....	225
Registros de la FPU.....	226
Tipos de datos.....	227
Introducción de datos en la FPU.....	227

Ejecución de operaciones.....	228
Recuperación de datos de la FPU.....	229
Un sencillo ejemplo.....	229
Resumen.....	231
13. Condicionales.....	232
El registro de indicadores.....	233
Obtención y restauración del registro de indicadores.....	235
Comparación de valores.....	238
Igualdad y desigualdad.....	238
Menor y mayor que.....	241
Instrucciones de manipulación de bits.....	242
Activación de bits individuales.....	243
Desactivación de bits individuales.....	244
Otras operaciones lógicas.....	245
Comprobación de bits individuales.....	246
Rotación y desplazamiento de bits.....	248
Resumen.....	250
14. Bucles.....	252
Bucles con saltos condicionales.....	253
Instrucciones para implementar bucles.....	257
Casos concretos.....	258
Bucles con condición compuesta.....	258
Bucles anidados.....	260
Transferencia de datos.....	263
Resumen.....	267
15. Estructuración del código.....	268
Procedimientos.....	269
Llamada a un procedimiento.....	270
Retorno de un procedimiento.....	271
Salvaguarda de los registros.....	274
Transferencia de parámetros.....	278
Una rutina de espera.....	278
Instrucciones de E/S.....	279
Comunicación con el reloj del sistema.....	279
Código de la rutina.....	280
Un ejemplo de uso.....	281
Macros.....	286
Macros simples.....	286
Expansión de lamacro.....	287
Macros complejas.....	288

Archivos de macros y procedimientos.....	289
Resumen.....	290
16. Manipulación de secuencias de bytes.....	292
Orígenes, destinos e incrementos.....	293
Recuperación y almacenamiento de datos.....	294
Conversión de binario a decimal.....	296
Almacenamiento de valores.....	299
Repetición automática de la operación.....	300
Transferencia de una secuencia de datos.....	301
Búsqueda de un dato.....	302
Comparación de cadenas.....	304
Resumen.....	310
17. La BIOS.....	312
¿Qué es la BIOS?.....	313
El mecanismo de interrupciones.....	315
El área de parámetros de la BIOS.....	316
Acceso a variables de la BIOS.....	318
Servicios de la BIOS.....	320
Acceso al adaptador de vídeo.....	321
Lectura del teclado.....	322
Configuración del sistema.....	323
Memoria disponible.....	327
Acceso a unidades de disco.....	328
Puertos serie y paralelo.....	330
Fecha y hora.....	332
Interrupciones hardware.....	335
Excepciones.....	336
Manipulación de los vectores de interrupción.....	337
Resumen.....	338
18. Servicios de vídeo.....	340
Detección del tipo de adaptador.....	341
Modos de visualización.....	343
Obtener y modificar el modo de visualización.....	346
Servicios para trabajar con texto.....	349
Posición y aspecto del cursor.....	350
Caracteres y atributos.....	352
Cambio de la página activa.....	356
Desplazamiento del texto.....	359
Servicios para trabajar con gráficos.....	361
Escritura y lectura de puntos.....	361
El color en adaptadores CGA.....	364

Fecha y hora.....	473
Gestión de vectores.....	473
Finalización y ejecución de programas.....	474
Gestión de memoria.....	476
Un programa que ejecuta otros.....	477
Resumen.....	483
24. Tratamiento de archivos.....	484
Apertura y creación de archivos.....	486
Creación de un nuevo archivo.....	486
Creación de archivos temporales.....	487
Apertura y cierre de archivos.....	487
Lectura y escritura de datos.....	488
Guardar y restaurar pantallas.....	489
Borrado, renombrado y otras operaciones con archivos.....	492
Unidades y directorios.....	493
La unidad por defecto.....	493
El directorio actual.....	495
Creación y borrado de directorios.....	495
Archivos existentes en un directorio.....	496
UDisk.....	497
Resumen.....	531
25. Acceso a sectores de disco.....	532
Servicios del DOS.....	533
Unidades de más de 32 Mb.....	535
Unidades de más de 2 Gb.....	535
Servicios de la BIOS.....	536
Copia de discos.....	538
Resumen.....	546
26. Memoria expandida y extendida en DOS.....	548
Bits, direccionamiento y modos de operación.....	549
Memoria expandida.....	550
Memoria extendida.....	552
Memoria alta.....	552
Memoria superior.....	555
Memoria extendida.....	556
La especificación XMS.....	556
El gestor XMS.....	559
Asignación de EMB.....	562
Transferencia de datos.....	565
Resumen.....	569

27. Programas residentes en DOS.....	570
Aplicación y problemática.....	571
Tipos de código residente.....	572
Limitaciones del código residente.....	573
Métodos para dejar código residente en memoria.....	574
Longitud del código residente.....	574
Activación del código.....	575
Asignación de un vector de interrupción.....	579
Ocupación en memoria.....	585
Fiabilidad del método.....	589
La interrupción múltiple.....	591
Engancharse a la interrupción múltiple.....	591
Un primer ejemplo.....	592
Cómo evitar la reinstalación.....	595
Facilitar la desinstalación.....	598
Restauración de los vectores.....	598
Liberar la memoria.....	599
Tercera versión de INT2F.....	600
A vueltas con la pila y el PSP.....	605
Una pila para la parte residente.....	606
Cambio del PSP activo.....	606
Estado del DOS y la BIOS.....	609
La reentrada y el DOS.....	610
Los indicadores InDOS y ErrorMode.....	610
La interrupción 28h.....	619
Estructura del programa residente.....	620
Los servicios de entrada y salida de caracteres.....	621
Las interrupciones BIOS.....	621
Tiempo de interrupción de un residente.....	622
Estado de otros elementos del sistema.....	623
Intercambio de la DTA.....	623
Gestión de errores críticos.....	625
Respuesta del controlador de error crítico.....	626
Otros aspectos a tener en cuenta.....	627
División por cero.....	627
Tratamiento de excepciones.....	628
Tratamiento de Control-C y Control-ínter.....	631
Inhibición del tratamiento de Control-C.....	632
Inhibición del tratamiento de Control-ínter.....	634
Otros aspectos a tener en cuenta.....	634
Acceso a la pantalla.....	634
Salvaguarda del contenido de la pantalla.....	635
Estado del teclado.....	635
Estado del ratón.....	636

Activación por teclado.....	636
Interceptar la interrupción de teclado.....	636
Control del teclado a bajo nivel.....	637
Códigos de teclado.....	638
Combinaciones de teclas.....	642
Bytes de estado del teclado.....	643
Esquema general de un programa residente.....	644
Instalación.....	645
Desinstalación.....	645
Supervisión.....	645
Gestor de INT 9h.....	646
Gestor de INT 1Ch.....	646
Gestor de INT 28h.....	646
Gestor de INT 10h e INT 13h.....	646
Activación.....	646
Otros factores a tener en cuenta.....	647
Una labia de códigos ASCII residente.....	648
La instalación.....	648
Petición de activación.....	650
Estado de la BIOS.....	653
La activación.....	654
Mostrar la tabla de códigos ASCII.....	659
Otros gestores de interrupción.....	664
Procedimientos adicionales.....	664
Funcionamiento del programa.....	665
Aplicaciones residentes y Windows.....	666
Residentes globales y locales.....	666
Problemas de un residente global.....	667
Iniciación de Windows.....	668
Funcionamiento bajo Windows.....	669
A vueltas con las VM.....	669
Copias individuales de datos.....	670
Secciones críticas.....	672
Ejecución de programas Windows.....	672
Resumen.....	675
28. Servicios de Windows.....	676
Herramientas necesarias.....	678
Inclusión de definiciones y bibliotecas.....	679
Ensamblado y enlace.....	679
Invocación a funciones Windows.....	680
Estructura básica de una aplicación Windows.....	681
La clase de ventana.....	682
Creación de ventanas.....	685

Proceso de mensajes.....	688
El programa completo.....	689
Uso de controles.....	692
Añadir un control a una ventana.....	693
Botones.....	694
Envío de mensajes a ventanas.....	695
Un ejemplo.....	696
Textos	701
Otros controles.....	703
Dibujar en la ventana.....	703
Resumen.....	706
29. Servicios de Linux.....	708
Herramientas necesarias.....	709
Servicios del núcleo de Linux.....	711
Devolución del control al sistema.....	711
Entrada y salida por consola.....	712
Macros de ayuda.....	715
Trabajo con archivos.....	716
Apertura y creación de archivos.....	716
El puntero de lectura/escritura.....	719
Constantes y macros.....	719
Acceso a la memoria de pantalla.....	721
Dispositivos ves y vesa.....	721
Guardar el contenido de la pantalla en un archivo.....	723
Manipulación del contenido de la pantalla.....	725
Acceso a discos.....	727
La biblioteca de funciones de Linux.....	730
Entrega de parámetros.....	730
Servicios disponibles.....	731
Resumen.....	732
30. 32 bits en DOS.....	734
El modo protegido.....	736
Registros de control del procesador.....	737
Modificación de los registros de control.....	739
Segmentos y selectores.....	740
Descriptores de segmentos.....	741
Tipos de segmentos.....	743
Tablas de descriptores.....	744
De vuelta a los selectores de segmento.....	745
Direccionamiento en modo protegido.....	746
Entrada y salida del modo protegido.....	747

Preparación de la GDT.....	748
Cálculo de direcciones físicas.....	751
Núcleo del programa.....	753
Interrupciones en modo protegido.....	756
DPMI.....	757
Anfitriones DPMI.....	757
Clients DPMI.....	758
Detectar la presencia de un anfitrión DPMI.....	758
Activación del modo protegido.....	762
Servicios DPMI.....	763
Un ejemplo.....	763
Extensores DOS.....	766
Resumen.....	767
31. Interfaz entre ensamblador y C/C++.....	768
Ensamblador embebido.....	770
Visual C++.....	770
GCC.....	772
Procedimientos externos en ensamblador.....	774
Prólogo y epílogo.....	775
Acceso a los parámetros de entrada y devolución de resultados.....	775
Compilación, ensamblado y enlace.....	776
Resumen.....	779
32. Recursos de interés.....	780
A. Contenido del CD-ROM.....	788
índice alfabético.....	791

Introducción

INTRODUCCION

Al inicio del siglo XXI, escribiendo esta introducción, cabe preguntarse qué sentido tiene dedicar nuestro tiempo a aprender a programar en un lenguaje, como es el ensamblador, de tan bajo nivel, teniendo a nuestra disposición sofisticadas herramientas de desarrollo rápido que, en minutos, son capaces de generar las aplicaciones más complejas que podamos imaginar.

Sería lo mismo, sin embargo, que preguntarse por qué dedicamos tiempo al bricolaje casero cuando hay estanterías y muebles modulares y prefabricados, listos para montar. No es comparable hacer un mueble uno mismo a comprar las piezas y montarlas, o cocinar uno mismo un guiso a comprarlo en lata y calentarlo. Lo segundo siempre será más rápido, pero en lo primero está el gusto y la satisfacción de lo hecho por uno mismo, aparte de que suele salir más barato.

Programando en ensamblador nos convertiremos prácticamente en artesanos del desarrollo de programas, ocupándonos de todas las tareas en las que deseemos intervenir personalmente.

El tiempo empleado para crear cualquier programa será superior pero, a cambio, obtendremos programas muchísimo más pequeños e infinitamente más rápidos que los que podamos crear con cualquier lenguaje de alto nivel.

El tamaño de los programas, en un mundo conectado a través de redes cada vez más saturadas, puede marcar diferencias muy importantes para los usuarios, aparte de que la memoria de los ordenadores siempre es un bien escaso. La velocidad nunca es despreciable, especialmente cuando se programa para entornos que ejecutan simultáneamente múltiples aplicaciones, como es el caso de Linux y Windows, o cuando se crean programas para consolas, como la XBox de Microsoft.

Más que la velocidad de ejecución, el tamaño compacto del código o la satisfacción de controlar hasta el más mínimo detalle, programar en un lenguaje como es el ensamblador resulta una tarea prácticamente imprescindible para comprender cómo funciona a bajo nivel cualquier sistema basado en un microprocesador. El estudio teórico de la arquitectura de dichos sistemas es la base, el punto de partida, pero su programación resulta fundamental para familiarizarse con este tipo de tecnología y, de paso, apreciar en su justa medida lo que ofrecen los lenguajes de alto nivel.

Microprocesadores

Los ordenadores personales, los grandes servidores, los pequeños Palm y Pocket PC, los teléfonos móviles y hasta la mayoría de los electrodomésticos tienen en su interior uno o más microprocesadores.

Cada microprocesador reconoce un cierto conjunto de instrucciones, cuenta con un determinado conjunto de registros y tiene una capacidad de direccionamiento concreta. Utilizando el lenguaje C, por mencionar uno de los más conocidos, puede desarrollarse un programa con el mismo código fuente y funcionaría en un antiguo MSX, un Power Mac o un PC.

Usando ensamblador, por el contrario, esto no es posible porque los microprocesadores Z80, PowerPC y Pentium/Core son totalmente distintos.

Aunque existen conceptos, que conocerá en este libro, generales a la programación en ensamblador usando cualquier microprocesador, cuando se inicia la práctica las diferencias suelen ser muy considerables.

Por ello es necesario optar por estudiar el conjunto de instrucciones y características de un microprocesador concreto. Conociendo éste, aprender a programar para otro será bastante más sencillo.

El dilema está, en principio, en qué procesador elegir para comenzar. Programar para un microprocesador Z80, muy conocido por ser el utilizado hace años en casi todos los microordenadores y actualmente en sistemas embebidos, es bastante sencillo, pero el salto a un micro como los utilizados en los actuales PC y Mac, por ejemplo, sería muy grande.

Lo mismo puede aplicarse a otros microprocesadores surgidos a finales de los setenta y principios de los ochenta, como el conocido Intel 8085. Éste sigue utilizándose en universidades con el objetivo de introducir a los alumnos en los conceptos más básicos de la programación de sistemas basados en microprocesador, pero fuera de este contexto en la actualidad tienen muy poca aplicación.

Actualmente la arquitectura de procesador más usada mundialmente en ordenadores personales, el dispositivo más accesible a la programación, es la conocida como *x86*. Ésta surgió con el ahora anciano microprocesador Intel 8086 y, a lo largo de casi treinta años, ha ido evolucionando hasta llegar a los últimos Pentium 4, AMD Athlon, Core y Core 2 y otros micros compatibles.

Existen multitud de herramientas para programar con ensamblador en la plataforma *x86*, por lo que parece una elección lógica.

Sistemas hardware y sistemas operativos

Al programar en ensamblador no sólo se utiliza el conjunto de instrucciones y registros de un cierto microprocesador sino que, además, se usarán dichas instrucciones para acceder a elementos hardware, como el adaptador de vídeo, el teclado o los buses de comunicaciones de una cierta arquitectura de ordenador. De igual manera, para efectuar ciertas tareas se utilizarán servicios puestos a disposición de las aplicaciones por el sistema operativo.

La disposición de la memoria en el ordenador, localización de los puertos para acceder a dispositivos externos al micro, interrupciones y otros parámetros dependen directamente de la arquitectura hardware del sistema. Es totalmente distinto el diseño hardware, por ejemplo, de un PC respecto a un Power Mac. Esto hace que un simple programa que pretenda mostrar un mensaje en pantalla y solicitar una entrada por teclado, todo ello escrito en ensamblador, sea totalmente distinto para uno y otro sistema hardware.

Cuando las tareas a efectuar son algo más complejas, como puede ser recuperar información de un archivo en disco o abrir una ventana en un entorno gráfico, suele recurrirse a los servicios que ofrece el sistema operativo, dejando que éste controle el hardware en lugar de hacerlo nosotros directamente. Es otro punto en el cual se crean fuertes dependencias entre programa y plataforma, en este caso plataforma software. Son distintos, lógicamente, los servicios ofrecidos por Linux, DOS, Mac OS y Windows, no estando disponibles las mismas funciones en cada uno de ellos.

Los sistemas operativos mencionados son cuatro de los más utilizados, si bien DOS, Windows y Linux pueden encontrarse para una misma arquitectura de procesador y hardware, mientras que Mac OS es un caso aparte por la arquitectura de los procesadores PowerPC, si bien recientemente Apple ha comenzado a usar los mismos microprocesadores de Intel que se utilizan en los PC, y el propio diseño del hardware del Mac como sistema. Por eso en este libro se ha optado por utilizar DOS, Windows y Linux como objetivo para los programas, ya que es posible usar un mismo procesador y, en general, una misma arquitectura hardware, aunque los servicios del sistema sean distintos. Incluso es posible usar la misma herramienta, el mismo ensamblador, en los tres casos, lo cual simplifica el proceso de aprendizaje considerablemente.

Objetivos de este libro

La finalidad de este libro es servir como guía de aprendizaje para todos aquellos programadores que desean introducirse en el desarrollo a bajo nivel, utilizando el lenguaje ensamblador de los procesadores *x86* en la plataforma PC y con los sistemas operativos DOS, Windows y Linux. Para ello se facilitará toda la información, teórica y práctica, llevándole desde un nivel de inicio hasta un nivel medio.

Como puede observarse en la tabla de contenidos, el libro está estructurado en dos bloques claramente diferentes. El primero, que abarca la primera media docena de capítulos, se ocupa de cubrir los fundamentos básicos sobre microprocesadores, sistemas

basados en microprocesadores, representación de la información y, ya de forma más específica, algunos detalles sobre la arquitectura de la familia x86, sus modos de direccionamiento y conjunto de instrucciones. Esta parte del contenido es nueva en la segunda edición de este libro, redactada a finales de 2008.

El resto de los capítulos, del séptimo en adelante, corresponden en su mayor parte a la primera edición de este libro, redactada durante 2002, y cubre distintos aspectos de la programación en ensamblador, tanto en DOS como en Windows y Linux. Los capítulos se han revisado y corregido, introduciéndose también algunos aspectos nuevos como los que tratan los servicios de uso de palancas de juegos o la interfaz entre ensamblador y C/C++.

Como acaba de decirse, este libro va dirigido a programadores que desean comenzar a usar ensamblador, pero no a usuarios de informática sin conocimientos previos de programación, ya que en este libro no se explicarán términos y conceptos básicos sobre el desarrollo de programas. Si nunca ha programado necesitará, antes de abordar este libro, leer otro más básico que le permita adquirir esos fundamentos básicos. El libro *Introducción a la programación*, del mismo autor y editorial, puede ser una buena opción.

Tampoco es éste un libro de referencia, como los típicos títulos sobre ensamblador en los que sencillamente se enumera cada una de las instrucciones del procesador, cada una de las interrupciones de acceso a servicios de DOS, etc., sin ninguna explicación adicional ni ejemplos demostrativos. En su lugar, se ha optado por un enfoque más didáctico, persiguiendo objetivos en cada capítulo tales como la realización de operaciones aritméticas, ejecución repetitiva de un conjunto de instrucciones, etc. El objetivo es hacer el aprendizaje del lenguaje lo más simple posible, sin entrar en detalles innecesarios y todas las posibilidades que, pudiéndose encontrar en materiales de referencia, no resultan totalmente imprescindibles en un principio.

A parte de los conocimientos indicados, no necesitará nada más para comenzar a trabajar programando en ensamblador. En el CD-ROM y las páginas de este libro encontrará todos los recursos necesarios: ensambladores, enlazadores y editores, que serán usados a lo largo del libro en los tres sistemas citados.

1

Microprocesadores y sistemas basados en microprocesador

Los primeros ordenadores que pueden considerarse similares a los actuales, en el sentido de que ejecutaban un programa almacenado en memoria y su utilidad era general en contraposición a los ordenadores con objetivos específicos y cuya reprogramación implicaba cambios en el hardware (generalmente en el cableado), surgieron en la década de 1940, bastante antes de que se fabricase el primer microprocesador.

La tecnología basada en relés mecánicos dio paso a las válvulas de vacío, mucho más rápidas y fiables, y posteriormente a los transistores, más pequeños y con menores requerimientos de energía. Era la época de los *mainframes* y miniordenadores, de firmas como IBM, DEC, SDS y Honeywell y de sistemas con arquitecturas heterogéneas hasta en los aspectos que hoy se consideran casi obvios. Actualmente a nadie se le ocurriría diseñar un sistema con un ancho de palabra que no fuese una potencia de 2: 8, 16, 32, 64 bits, pero por entonces hasta ese detalle dependía del fabricante de cada ordenador, utilizándose anchos de palabra de 36 bits, de 12 bits, de 10 dígitos decimales y otros tamaños que hoy se considerarían atípicos.

La invención del microprocesador, y su posterior uso en la construcción de ordenadores sustituyendo paulatinamente a los transistores y otros componentes electrónicos, marca un claro punto de inflexión en la historia de la informática, un antes y un después en la forma en que se construyen los ordenadores y una simplificación de su estructura que contribuiría a reducir su coste, tamaño y consumo.

Todo ello favorecería la llegada de los sistemas informáticos, hasta ese momento restringidos a organizaciones militares, científicas y a grandes empresas, a un público mucho más amplio y heterogéneo hasta alcanzar, desde mediados de los ochenta, el campo del usuario particular o doméstico.

El microprocesador puede ser considerado, desde la perspectiva que ofrecen cuatro décadas desde su aparición, como uno de los inventos más importantes del periodo post-industrial, dada su presencia en todo tipo de dispositivos, no solamente ordenadores; la influencia que tiene en segmentos tan diversos como la automoción o la telefonía y su impacto económico a escala mundial.

En este capítulo se lleva a cabo una breve introducción a la evolución y arquitectura general de los microprocesadores, sin entrar en detalles sobre ninguno en concreto, así como al diseño habitual, muy a grandes rasgos, de sistemas basados en un microprocesador, sean éstos ordenadores o no.

Del circuito integrado al microprocesador

Hasta agosto de 1958, los circuitos con que se diseñaban los ordenadores estaban constituidos de componentes discretos (resistencias, diodos, condensadores, transistores, etc.) independientes, fabricado cada uno de ellos con materiales distintos y conectados entre sí mediante cables. En esa fecha, hace ahora medio siglo, Jack Kilby diseñó un circuito en el que todos los componentes, aunque seguía siendo independientes, estaban hechos del mismo material: silicio. En septiembre del mismo año construyó el mismo circuito a partir de una única pieza de material, en este caso germanio, en lugar de obtener los componentes por separado y conectarlos después. Las conexiones también eran del mismo material. Había nacido el *circuito integrado*, base de la miniaturización de circuitos que posibilitaría diseños cada vez más complejos y baratos.

Desarrollada la tecnología necesaria para fabricar circuitos integrados (IC en adelante) de silicio, a principios de la década de los sesenta, fue posible la construcción de componentes encapsulados relativamente complejos: biestables, puertas lógicas y elementos similares. Combinando un cierto número de estos elementos resultaba mucho más sencillo diseñar una unidad aritmético-lógica, una unidad de control y, con el tiempo, sustituir las memorias de núcleo de ferrita por circuitos integrados de memoria de mucha mayor capacidad, menor tamaño y consumo. Aun así, el diseño de las CPU de los ordenadores de aquella época seguía siendo considerablemente complejo, tratándose de un circuito formado por un importante número de IC y cableado. El tamaño, sin embargo, se había reducido de manera significativa, introduciendo en una caja del tamaño de una mesa de escritorio lo que antes requería una habitación completa. El consumo y coste disminuyó de manera proporcional, al tiempo que se incrementaba la fiabilidad y velocidad.

A medida que se mejoró la tecnología de fabricación de IC, reduciendo el tamaño de los componentes discretos y, en consecuencia, posibilitando la inclusión de un mayor número de ellos en un mismo encapsulado, se llegó al punto en que resultó factible la construcción de una CPU completa en un único IC. Este potencial, sin embargo, no fue visto con interés por las empresas que en aquel momento fabricaban ordenadores. De hecho el que está considerado el primer microprocesador de la historia, el Intel 4004 presentado en 1971, se diseñó por encargo de la empresa Busicom para producir una calculadora con avanzadas funciones matemáticas. Por entonces Intel era una empresa dedicada a la fabricación de IC de memoria para calculadoras, un dispositivo que

comenzaba a ser accesible para el gran público gracias a su progresivo abaratamiento al fabricarse con circuitos integrados en lugar de los circuitos clásicos que aún seguían utilizándose en los ordenadores.

El diseño que Intel hizo para Busicom puede ser considerado el primer sistema basado en microprocesador, a pesar de que fuese una calculadora, formado por cuatro circuitos integrados: la CPU 4004 de 4 bits (suficientes para representar los diez dígitos decimales en formato BCD), un IC de memoria RAM, otro de memoria ROM y por último uno encargado de controlar teclado y pantalla, es decir, un circuito de E/S.

Nota

Obtendrá la hoja de características original del 4004, en formato PDF, de la dirección http://download.intel.com/museum/archives/pdf/4004_datasheet.pdf.

En <http://www.4004.com>, sitio creado por el 35 aniversario de la presentación del 4004, puede descargarse también información y software, principalmente simuladores, tanto del 4004 como de la calculadora Busicom en la que se utilizó.

Evolución de los microprocesadores

Según distintas referencias, los diseños más antiguos de lo que hoy se conoce como microprocesador (originalmente no se denominaban como tales), datan de finales de 1968 y principios de 1969, si bien el primero en materializarse fue, al parecer, el ya citado Intel 4004, presentado en noviembre de 1971. Esos diseños que, a día de hoy, podrían considerarse primitivos, se limitaban a conseguir que un único circuito integrado pudiese realizar las funciones para las que anteriormente era necesario utilizar múltiples IC.

En 1964, mucho antes de que se llegase a materializar el primer microprocesador, Gordon Moore, uno de los cofundadores de Intel y por entonces empleado de Fairchild, predijo que el número de transistores que era posible introducir en un circuito integrado se duplicaría, gracias a procesos de fabricación cada más sofisticados, aproximadamente cada 12 meses. Este cálculo, basado en datos de casi una década de fabricación de IC, permitió a Moore predecir el momento en que sería posible introducir en un circuito integrado elementos suficientes como para encapsular toda una CPU. Su predicción fue acertada y, de hecho, la que hoy se conoce como *Ley de Moore* sigue cumpliéndose después de cuatro décadas.

Fueron los primeros

Según distintos historiadores el microprocesador surgió paralelamente, sin ninguna conexión entre sí, al menos en tres empresas distintas. Una de ellas fue Intel con el ya mencionado 4004, otra TI con el diseño de un IC para una calculadora y el tercero estuvo asociado a un proyecto militar, no conociéndose muchos detalles al respecto.

El IC desarrollado por TI estaba diseñado específicamente para una familia de calculadoras, de forma que las funciones posibles se encontraban codificadas en el propio microprocesador, si es que podía denominarse así. En contraposición, el Intel 4004 ejecutaba código recuperado de una memoria ROM, de forma que podía realizar funciones distintas sencillamente cambiando esa memoria por otra con un programa diferente. Esta es la razón de que se le considere el primer microprocesador.

Según la hoja de especificaciones de Intel, el 4004 contaba con un bus de datos de 4 bits, operaba a una frecuencia de 0,1 Mhz y estaba compuesto de 2300 transistores. Podía direccionar como máximo 4 Kbytes de memoria ROM y 640 bytes de RAM, más que suficiente para una calculadora avanzada. También era capaz de comunicarse con hasta 16 puertos de E/S, utilizándose en la calculadora de Busicam para controlar la pantalla y el teclado.

Dada su limitada capacidad de direccionamiento, así como el reducido conjunto de instrucciones con que contaba, el 4004 no era adecuado para construir un ordenador. Ciertas instrucciones, como las de salto condicional, estaban diseñadas para operar siempre sobre memoria ROM, por lo que no resultaba factible la carga de programas en la reducida memoria RAM y su ejecución desde la misma. En este sentido el 4004 estaba diseñado siguiendo la arquitectura Harvard explicada más adelante, por la que la memoria dedicada a datos está separada de la que aloja los programas.

Otra característica de esta arquitectura era la pila interna con que contaba el 4004 que permitía hasta 3 niveles de anidamiento, en contraposición al uso de parte de la memoria RAM como pila.

Microprocesadores de 8 bits

El primer procesador de 8 bits de la historia también vino de las manos de la firma Intel, de hecho su diseño comenzó antes de que se finalizara el desarrollo del 4004. Al igual que éste, el 8008 también fue un encargo de una tercera empresa, concretamente Datapoint, en este caso dirigido a la fabricación de un terminal. Su construcción concluyó solamente cinco meses después que la del 4004, pero con retraso respecto a la fecha acordada lo que provocó que Datapoint finalmente fabricase su terminal usando circuitos integrados más básicos, en lugar de un microprocesador. Intel decidió entonces comercializar por su cuenta el 8008, producto que se convirtió en un éxito inesperado y provocó que, en pocos meses, muchas otras empresas comenzasen a desarrollar sus propios microprocesadores.

Tanto el 4004 como el 8008 fueron microprocesadores no diseñados en un principio para actuar como motores de un ordenador, todo lo contrario que el 8080, presentado por Intel en 1974, un microprocesador que, aunando una mayor capacidad de direccionamiento de memoria, mayor velocidad de ejecución y precisando menos IC de apoyo, ofrecía la misma potencia de proceso que los miniordenadores usados pocos años antes en grandes empresas, universidades y otras corporaciones.

El 8080 se convirtió en el corazón del primer ordenador personal de la historia, el Altair 8800, si bien en los meses previos era posible adquirir *kits* para construir ordenadores basados en el 8008 como el Mark-8.

Uno año después, en 1975, MOS Technology presentaba su microprocesador 6502, y en 1976, un grupo de ingenieros que abandonaron Intel y fundaron Zilog anuncianaban el Z80. El Commodore PET fue posiblemente el primer ordenador en el que se utilizó el 6502, mientras que el Z80 formó parte del TRS-80, presentado en 1977. A partir de ese momento ambos microprocesadores, Z80 y 6502, y varios derivados de ellos ganarían en popularidad a medida que fueron surgiendo decenas de microordenadores en el mercado, procedentes de firmas como Atari, Commodore, Sinclair, Philips, Sony, Thomson y un largo etcétera.

El último microprocesador de 8 bits desarrollado por Intel fue el 8085, presentado en 1976 y que representaba una evolución significativa sobre el 8080, al precisar una única fuente de alimentación y fabricarse con una tecnología de 3 micrómetros frente a los 6 micrómetros del 8080. Por lo demás se trataba de un microprocesador compatible con todo el código escrito para el 8080, al ofrecer el mismo conjunto de instrucciones, modos de direccionamiento, banco de registros, etc.

La mayoría de los microprocesadores de 8 bits utilizaban un encapsulado estándar de 40 patillas, contaban con un bus de datos de 8 bits y un bus de direcciones de 16 bits, generalmente multiplexado con el de datos. Esto les permitía direccionar un máximo de 64 Kbytes de memoria. También incorporaban las señales necesarias para atender interrupciones, característica de la que carecían diseños previos. Las frecuencias habituales de funcionamiento oscilaban entre los 0,8 Mhz y 4 Mhz.

Micropocesadores de 16 bits

La evolución desde los 8 a los 16 bits no se produjo de manera secuencial, al contrario, de forma paralela a los últimos microprocesadores de 8 bits fueron surgiendo diseños de 16 bits que, en su mayor parte, tenían el cometido de emular el funcionamiento de sistemas ya existentes. A esta categoría pertenecen los microprocesadores desarrollados por DEC y TI en la primera mitad de los setenta. Por su propio diseño no resultaban adecuados para la construcción de ordenadores distintos a aquellos que debían sustituir, lo que limitó su difusión a nichos muy concretos.

En 1978 Intel anunció el 8086, su primer microprocesador de 16 bits, y un año más tarde el 8088, idéntico al anterior salvo por usar externamente un bus de 8 bits, en lugar de 16, lo cual le permitía aprovechar toda la circuitería procedente de microprocesadores de 8 bits y, en consecuencia, fabricar sistemas más baratos. El 8088 fue elegido por IBM para desarrollar su primer ordenador personal, el IBM PC 5150. Esta decisión, y el hecho de que la arquitectura abierta de ese sistema contribuyese a que proliferaran los ordenadores compatibles, fue decisiva para que el 8086/8088 y sus sucesores se convirtieran en la familia de microprocesadores con mayor éxito desde entonces.

Zilog, Motorola y otros fabricantes también desarrollaron sus propios microprocesadores de 16 bits, pero su éxito no alcanzó el nivel de los de Intel. La familia 68K de Motorola, un microprocesador con estructura interna de 32 bits y externa de 16, fue utilizado en sistemas tan conocidos como los Atari ST y Commodore Amiga.

Respecto a los microprocesadores de 8 bits, los de 16 no solamente contaban con una superior capacidad de direccionamiento de memoria y mayor ancho de palabra, sino

que también incorporaron conjuntos de instrucciones más complejos. El 8086/8088, por ejemplo, disponía de instrucciones aritméticas para el producto y la división, así como para el tratamiento de cadenas de bytes. También se incrementaron las posibilidades de direccionamiento, con modos más sofisticados de acceso a la memoria.

CISC versus RISC

A medida que los microprocesadores incrementaban tanto su frecuencia de funcionamiento como el número de transistores que integraban, la complejidad de su diseño fue también creciendo, con juegos de instrucciones cada vez más extensos.

A principios de los ochenta surgió el debate sobre qué tipo de arquitectura resultaba más adecuada para el diseño de los microprocesadores, la empleada hasta ese momento, a la que se denominó CISC (*Complex Instruction Set Computer*), o bien la alternativa RISC (*Reduced Instruction Set Computer*).

Un microprocesador RISC cuenta con un extenso conjunto de registros internos y un reducido conjunto de instrucciones, relativamente simples y que se caracterizan por tener una longitud homogénea, de forma que es posible descodificarlas y preparar su ejecución utilizando siempre el mismo número de cíelos de reloj. La idea es que pueda ejecutarse un mayor número de instrucciones en menos tiempo, pero indirectamente se hace más complejo el diseño de los compiladores para lenguajes de alto nivel, que tienen que dividir las operaciones en un mayor número de instrucciones máquina.

Durante las décadas de los ochenta y noventa empresas como IBM, Sun, HP, DEC y Motorola fabricaron sus propios microprocesadores RISC, dirigidos en su mayor parte a estaciones de trabajo y no a ordenadores personales salvo excepciones concretas, como es el caso de los microprocesadores PowerPC usados en los Apple Mac hasta hace pocos años, cuando también la firma de la manzana se pasó al lado de Intel y comenzó a utilizar sus microprocesadores.

Los ordenadores personales fueron incrementando su potencia a medida que Intel desarrollaba nuevos procesadores, siempre manteniendo una compatibilidad hacia atrás que garantizaba el funcionamiento de todo el software existente, llegando con el tiempo a alcanzar y superar la potencia de las estaciones de trabajo. Los microprocesadores RISC introdujeron en su momento innovaciones que, como la arquitectura superescalar que les permitía ejecutar más de una instrucción por ciclo, fueron implementadas también con posterioridad en los microprocesadores CISC de Intel.

Con el tiempo, la evolución de los microprocesadores se ha ido acelerando en las últimas décadas, las diferencias entre arquitecturas RISC y CISC han ido reduciéndose paulatinamente.

Una gran parte de los microprocesadores RISC ya no se encuentran en estaciones de trabajo, sino en dispositivos tan heterogéneos como las PDA, los teléfonos móviles y en otros tipos de sistemas embebidos. Los microprocesadores más avanzados de Intel, como los Core y Core 2, aunque externamente presentan un conjunto de instrucciones CISC, internamente llevan a cabo una traducción a instrucciones mucho más simples de ejecución más rápida, presentando así una arquitectura combinada CISC/RISC que aprovecha lo mejor de ambos mundos.

Microprocesadores modernos

A punto de alcanzarse el final de la primera década de este tercer milenio, el microprocesador es un componente electrónico presente en multitud de dispositivos diferentes, si bien los más avanzados y potentes siguen siendo los empleados en ordenadores.

De los 2300 transistores que integraba el Intel 4004 se ha pasado a los 731 millones de transistores que utilizan los microprocesadores *Nehalem*, nombre en clave de la arquitectura en la que basarán los productos de Intel desde finales de 2008. Esta proporción, que permitiría introducir más de 300000 microprocesadores 4004 en un *Nehalem*, es bastante significativa de cómo se ha mejorado la tecnología de fabricación e incrementado la potencia de proceso.

Pero no solamente se ha aumentado el número de transistores, sino que también se ha mejorado considerablemente el diseño interno de los microprocesadores, aumentando el ancho de palabra y la cantidad de memoria que es posible direccionar, incluyendo unidades de cálculo en coma flotante de gran sofisticación, ofreciendo conjuntos de instrucciones específicas para operar sobre datos multimedia, incluyendo varios núcleos de procesamiento en el mismo microprocesador, etc.

En la tabla 1.1 se han resumido algunas características de los microprocesadores más representativos de Intel, a fin de que pueda compararlos y apreciar de un vistazo la rápida evolución a lo largo de estas cuatro décadas.

Tabla 1.1. Características de algunos microprocesadores de Intel.

Denominación	Año	Frecuencia	Transistores	Ancho palabra	Direccionamiento
4004	1971	0,1 Mhz	2300	4 bits	640 Bytes.
8008	1972	0,2 Mhz	3500	8 bits	16 KBytes.
8080	1974	2 Mhz	6000	8 bits	64 KBytes.
8085	1976	2 Mhz	6500	8 bits	64 KBytes.
8086	1978	4,77 Mhz	2900	16 bits	1 MByte.
80286	1982	6 Mhz	134000	16 bits	16 MBytes.
80386	1985	16 Mhz	275000	32 bits	4 GBytes.
80486	1989	25 Mhz	1,2 mill.	32 bits	4 GBytes.
Pentium	1993	60 Mhz	3,1 mill.	32 bits	4 GBytes.
Pentium Pro	1995	150 Mhz	5,5 mill.	32 bits	64 GBytes.
Pentium II	1997	233 Mhz	7,5 mill.	32 bits	64 GBytes.
Pentium III	1999	450 Mhz	28 mill.	32 bits	64 GBytes.
Pentium 4	2000	1,4 Ghz	42 mill.	32 bits	64 GBytes.
Pentium D	2005	2,8 Ghz	230 mill.	32 bits	64 GBytes.

Denominación	Año	Frecuencia	Transistores	Ancho palabra	Direccionamiento
Core 2	2007	3 Ghz	291 mill	64 bits	64 GBytes.
Core 2 Quad	2007	3 Ghz	582 mill.	64 bits	64 GBytes.
Core 2 Extreme	2008	3,2 Ghz	820 mill.	64 bits	64 GBytes.

Nota

Puede encontrar más detalles sobre todos los procesadores fabricados por Intel en <http://www.intel.com/pressroom/kits/quickrefyr.htm>.

Microprocesadores versus microcontroladores

Los microprocesadores no son los únicos circuitos integrados que incorporan registros, unidad de control y aritmético-lógica, buses, etc. Existen otros tipos de diseños, como las GPU (*Graphics Processing Unit*), con potencias equiparables e incluso superiores a las CPU, integrando hasta mil millones de transistores. Otro tipo interesante de circuito integrado es el conocido como *microcontrolador* que, básicamente, no es más que un microprocesador con algunas características específicas.

La principal diferencia existente entre un microcontrolador, también conocido como ordenador integrado (*Computer-on-a-chip*), y un microprocesador es que el primero incorpora en el mismo IC elementos fundamentales, como la memoria RAM, memoria ROM (normalmente EPROM) e interfaces de E/S para comunicarse con el exterior. Esto permite conectar el microcontrolador directamente al sistema que va a supervisar/controlar, a través de sus terminales, sin precisar ningún otro circuito de apoyo. Ésta es la razón de que se le denomine, como se apuntaba antes, ordenador integrado.

Además de los elementos que integra, el diseño de un microcontrolador también difiere de la de un microprocesador actual en la arquitectura interna. El primero tiene una arquitectura *Harvard*, la misma empleada en el Intel 4004, mientras que el segundo usa la arquitectura *Von Neumann*.

La arquitectura Harvard establece una clara separación entre la memoria dedicada a alojar el código del programa a ejecutar y la memoria en la que se escribirán y leerán los datos procesados por dicho programa, hasta tal punto que dichas memorias pueden ser de tecnologías totalmente distintas.

En la familia de microcontroladores más popular, conocida como PIC, la memoria dedicada a programa es una EPROM cuyo tamaño es bastante superior a la memoria RAM asignada al almacenamiento de datos.

En la arquitectura Von Neumann programa y datos se almacenan en la misma unidad de memoria, elemento independiente y que debe mostrar una interfaz homogénea al microprocesador, con independencia de que sea ROM, RAM o de cualquier otra clase.

Debe existir, por lo tanto, un medio de comunicación entre la memoria y el microprocesador, concretamente un conjunto de buses tal y como se detalla en el siguiente apartado de este capítulo.

Nota

No todos los microcontroladores usan la arquitectura Harvard ni todos los microprocesadores la arquitectura Von Neumann, aunque lo más usual es que sea así.

La memoria interna de los microcontroladores suele ser de reducido tamaño, por lo que se aplican a sistemas embebidos relativamente simples, como equipos de control industrial y dispositivos similares.

Los programas se codifican por regla general en ensamblador y, a continuación, se graban en una memoria EPROM, de forma que no es necesario cargarlos cada vez que se inicia el sistema.

Arquitectura común de una CPU

Habiendo situado temporal e históricamente la invención del microprocesador, e introducido algunos conceptos básicos sobre arquitecturas CISC/RISC y Harvard/Von Neumann, el siguiente objetivo es conocer la arquitectura interna general de cualquier CPU (*Central Processing Unit*), distinguiendo sus principales componentes. Éstos ya existían en los ordenadores previos a la aparición del microprocesador, pero implementados como circuitos independientes y conectados entre sí.

En la figura 1.1 puede verse el diagrama de bloques simplificado de una CPU cualquiera. El borde alrededor de los bloques representa el límite del microprocesador, del IC físico. Los buses de datos y direcciones le permiten comunicarse con el exterior, enviando y recibiendo información. Físicamente se corresponden con parte de los conectores o patillas del microprocesador.

También existe un bus de control, así como conectores adicionales para la gestión de interrupciones, recepción de señal de reloj, alimentación, etc.

De los bloques representados en esta figura los cinco que ocupan el margen derecho son elementos de memoria, mientras que los dos del margen izquierdo son los elementos lógicos, encargados de recuperar, descodificar y ejecutar las instrucciones. Entre todos ellos existen conexiones internas que hacen posible la comunicación, almacenamiento y recuperación de datos.

La finalidad de cada bloque, brevemente, es la siguiente:

- **Unidad de control:** Es la encargada de recuperar la instrucción apuntada por el contador de programa, descodificarla y establecer el estado interno del resto de los elementos para preparar su ejecución.

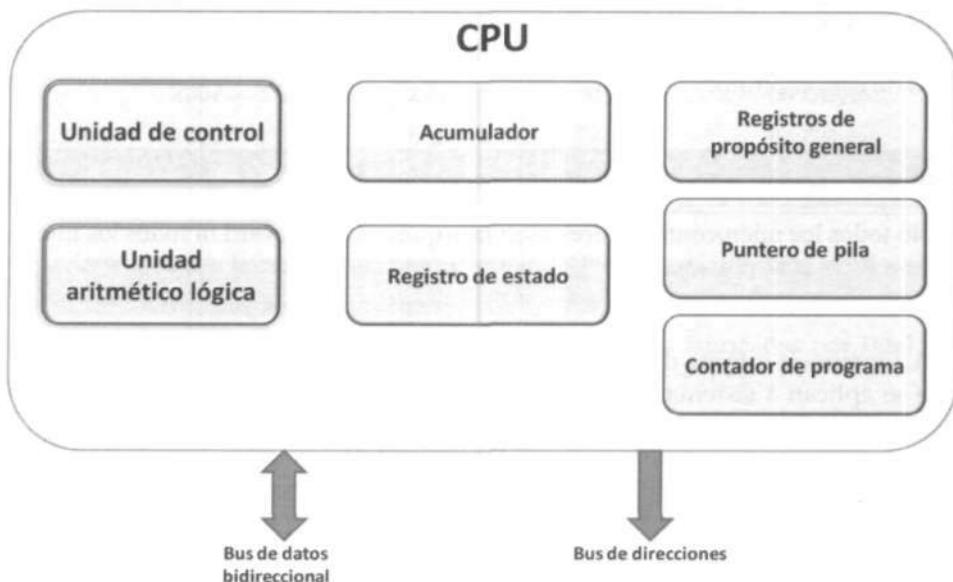


Figura 1.1. Diagrama de bloques de un microprocesador básico.

- **Unidad aritmético lógica:** Es el elemento que lleva a cabo las operaciones aritméticas, como suma y resta, y lógicas, al nivel de bits según el álgebra de Boole. En la mayoría de los casos la ALU (*Arithmetic Logic Unit*) asume como operando implícito el dato almacenado en el acumulador, pudiendo existir, según la instrucción a ejecutar, operandos adicionales. También influye en su funcionamiento el contenido del registro de estado (véase la figura 1.2).



Figura 1.2. Esquema de bloques de una ALU típica.

- **Acumulador:** Se trata de un registro, una zona de memoria interna a la CPU, en la que se almacenan los resultados de las operaciones. El tamaño del acumulador,

medido en bits, es lo que se conoce habitualmente como *ancho de palabra* del microprocesador, soliendo coincidir con el ancho del bus de datos.

- **Registro de estado:** Cada vez que se ejecuta una instrucción el estado interno de la CPU cambia, utilizándose este registro para mantener información sobre dicho estado. De esta forma es posible que la ejecución de instrucciones posteriores se vea afectada por las anteriores, teniendo en cuenta, por ejemplo, el acarreo de una suma.
- **Registros de propósito general:** Todos los datos sobre los que opera la CPU, así como los resultados generados, han de estar almacenados en el microprocesador, utilizándose para ello zonas de memoria internas denominadas registros. De éstos siempre existe un grupo de *propósito general*, registros que los programas pueden usar según necesiten y que no tienen asignada una función específica.
- **Puntero de pila:** Los primeros microprocesadores (4004 y 8008 entre ellos) y los microcontroladores cuentan con una pila interna de pequeño tamaño, con 3, 7 ó 15 niveles de anidamiento máximo para llamadas a subrutinas. En las CPU actuales la pila se almacena en la misma memoria que aloja el código del programa y los datos, utilizándose este registro, el puntero de pila, para saber dónde está localizada.
- **Contador de programa:** Es un registro que, como el puntero de pila, tiene un propósito específico, concretamente almacenar la dirección donde se encuentra la siguiente instrucción del programa a ejecutar. La unidad de control va incrementando el valor de este registro a medida que recupera instrucciones y los operandos asociados. También lo modifica cuando se ejecuta un salto, condicional o incondicional, estableciendo la nueva dirección de la que se recuperarán los códigos de instrucción.

Sobre la base de esta arquitectura abstracta, cada fabricante lleva a cabo la arquitectura física de sus microprocesadores según parámetros reales: ancho de palabra, ancho del bus de direcciones, cantidad de registros internos, operaciones que pueda llevar a cabo la ALU, etc.

En la figura 1.3 puede verse esquemáticamente la arquitectura de uno de los microprocesadores de 8 bits más difundidos: el 8085. Junto con el Z80, que era prácticamente idéntico, se estima que llegaron a fabricarse unos mil millones de microprocesadores con esta arquitectura.

En este esquema puede apreciarse que existen seis registros de propósito general y los registros específicos antes mencionados: acumulador, puntero de pila y contador de programa.

La unidad aritmético lógica es un elemento con acceso directo al acumulador y que se comunica con el resto del microprocesador a través de un bus de datos interno de 8 bits, en su funcionamiento influye también el registro de estado.

La unidad de control no aparece explícitamente, estando formada por el bloque de control de interrupciones, el de temporización y control así como el descodificador de instrucciones.

La comunicación con el exterior se lleva a cabo a través de un bus de datos de 8 bits, un bus de direcciones de 16 bits (multiplexado con el de datos) y un bus de control que utiliza el resto de las conexiones del circuito integrado, entrantes o salientes de los bloques ya citados de control de interrupciones, temporización y control.

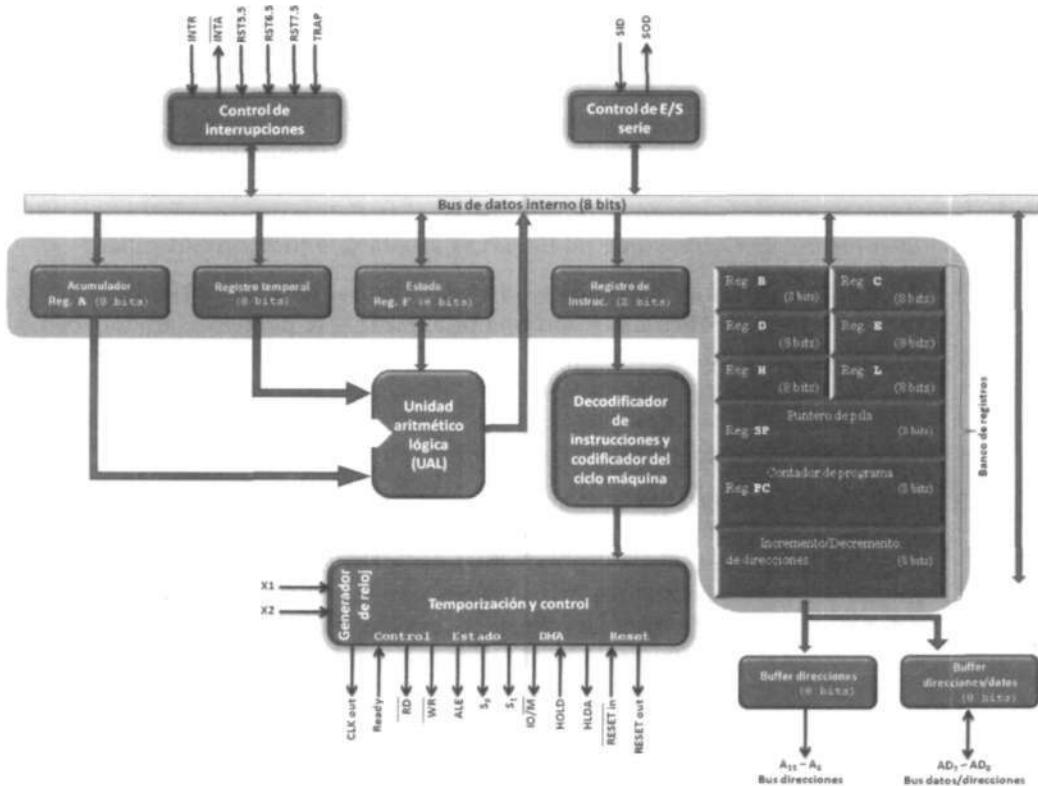


Figura 1.3. Esquema de bloques del Intel 8085.

En total, todas las señales del 8085 se canalizan a través de 40 conectores o patillas, como se aprecia en la figura 1.4. Éstas deberán conectarse a los buses que permitan la comunicación con el resto de componentes del sistema.

El Intel 8085 es el antecesor directo del 8086 y, en consecuencia, de toda la familia x86, de la que forman parte los microprocesadores Core y Athlon que pueden encontrarse actualmente en todos los ordenadores personales. La arquitectura del 8085, sin embargo, es más simple al tratarse de un microprocesador de 8 bits y no de 16 bits, de ahí que se utilice en muchas universidades como base para estudiar la estructura y tecnología de los ordenadores.

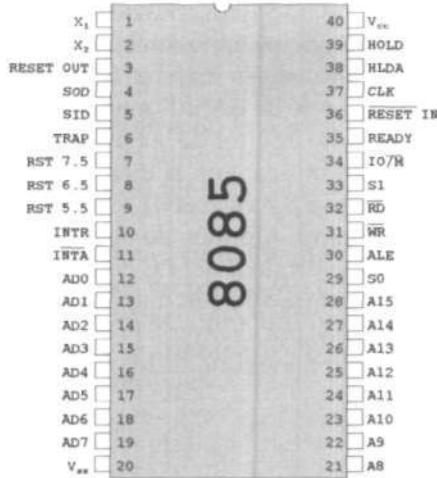


Figura 1.4. Patillaje del 8085.

Arquitectura común de un sistema basado en microprocesador

Por sí solo un microprocesador no resulta suficiente para construir un sistema completo, un ordenador, sino que precisa elementos auxiliares tan indispensables como la memoria. A diferencia de un microcontrolador, un microprocesador tampoco incorpora la circuitería apropiada para conectarse directamente a dispositivos externos, como una pantalla o un teclado, de ahí que necesite circuitos externos de apoyo.

Para comenzar, el microprocesador tiene que contar con una fuente de alimentación, así como con una señal externa de reloj que le permita conocer la frecuencia a la que debe operar. Es habitual que la frecuencia del reloj externo no sea utilizada directamente, sino que es multiplicada/divida por algún factor. El resto de las patillas del microprocesador formarán los tres buses antes citados: datos, direcciones y control. Estos buses se extienden por el sistema facilitando la comunicación con el resto de los componentes, como puede apreciarse en la figura 1.5. Ésta corresponde al esquema de un sistema simple, compuesto del microprocesador y seis IC adicionales: tres de memoria y tres de E/S.

Puesto que los buses no pueden ser usados simultáneamente por más de un IC, el microprocesador se encarga de activar, ya sea mediante líneas de control o del bus de direcciones, el IC con el que quiere operar en cada momento. Para ello todos los IC cuentan con una señal, denominada habitualmente CS (*Chip-Select*), que mantiene desactivado el circuito mientras está a nivel alto, activándolo cuando el procesador coloca un 0 (nivel bajo) en dicha patilla.

El número de IC de memoria dependerá de las características del ordenador que se esté diseñando y de la capacidad de cada IC. En la figura 1.5, por ejemplo, cada pastilla de RAM podría aportar 16 KBytes de memoria, de forma que en total el sistema dependerá

de 32 KBytes de memoria para almacenamiento de programas y datos. La memoria ROM se usa para contener el código que permite iniciar el sistema, contenido en el programa que se ejecuta en cuanto se pone éste en marcha. Hace dos o tres décadas en esa memoria ROM solía residir un intérprete de BASIC, mientras que en la actualidad contiene la conocida BIOS. Los IC de la parte inferior de la figura 1.5 son de E/S. Los números 8251, 8279 y 8255 son los códigos identificativos, si bien son conocidos también por denominaciones como USART (*Universal Synchronous Asynchronous Receiver Transmitter*) o PPI (*Programmable Peripheral Interface*). Algunos de ellos, como los mencionados USART y PPI, formaban parte del diseño original del PC como sistema basado en el microprocesador 8086, facilitando la comunicación con dispositivos serie (módem por ejemplo) y dispositivos paralelo (impresoras por ejemplo). El microprocesador no tiene conocimiento directo de los periféricos que tiene conectados el sistema, comunicándose con ellos a través de estos IC de apoyo.

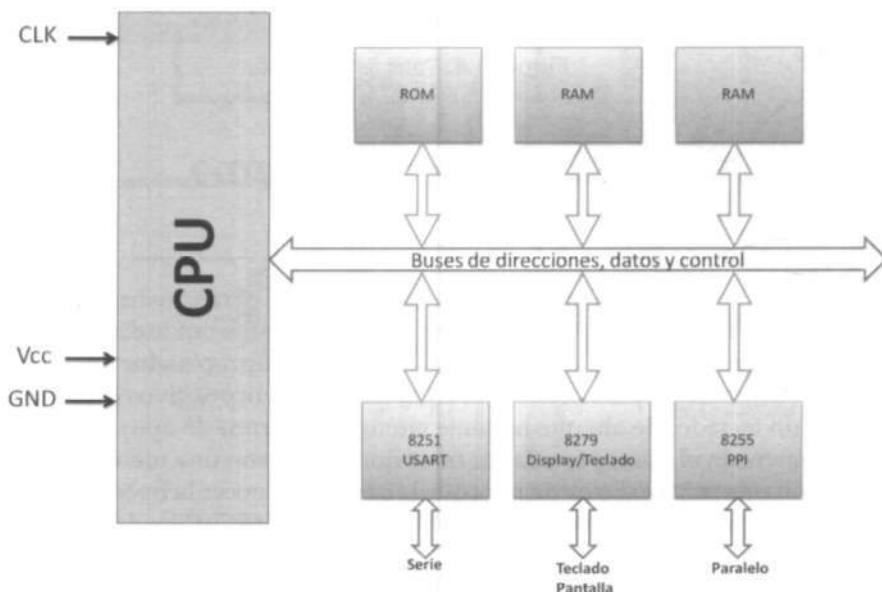


Figura 1.5. Esquema de bloques de un sistema simple basado en microprocesador.

Nota

Lógicamente la arquitectura de los actuales sistemas informáticos es considerablemente más compleja que la esbozada en la figura 1.5, incluyendo multitud de circuitos integrados con funciones específicas como la comunicación a través de redes Ethernet, la visualización de gráficos, generación de audio y cantidades muy superiores de memoria. No obstante, la diferencia se reduce básicamente a contar con buses de datos y direcciones de mayor anchura y contar con IC de apoyo más sofisticados, lo fundamental no cambia.

Resumen

Este capítulo le ha ofrecido una visión general de la evolución de los microprocesadores, desde la aparición del primer circuito integrado hace ahora medio siglo, hasta los desarrollos más actuales del primer fabricante del sector: Intel. Ya conoce la diferencia entre microprocesadores de 8, 16 ó 32 bits, así como las dos arquitecturas más usuales: CISC y RISC.

También se han descrito los bloques fundamentales que componen un microprocesador, aunque sin entrar en detalles de un diseño concreto. Finalmente se ha esbozado la estructura de un sistema basado en microprocesador, destacando la necesidad de contar con circuitos de apoyo para la comunicación con el exterior y un subsistema de memoria como elementos imprescindibles.

2

Representación de datos en ordenadores

En los circuitos integrados actuales toda la información se almacena como una secuencia de bits, de ceros y unos, empaquetados de ocho en ocho dado que éste es el tamaño asignado a cada una de las celdillas de los chips de memoria: un byte.

Cualquier dato que se desee almacenar en un sistema informático, por tanto, habrá de representarse seleccionando un cierto formato, un patrón de bits que determine el significado de cada valor.

Los lenguajes de alto nivel cuentan con tipos de datos muy complejos: fechas, vectores, matrices, estructuras, etc. Todos ellos, sin embargo, no son más que secuencias de bits, siendo el compilador el encargado de generar el código que facilite su correcta interpretación por parte de los programas.

Al programar en ensamblador el conjunto de tipos de datos sobre los que se operará será mucho más reducido, limitándose en su mayor parte a números enteros de distintos tamaños y a números en punto flotante de distintas precisiones. También es habitual trabajar con bits individuales y secuencias de bytes, pudiendo interpretarse éstas como cadenas de caracteres.

Otro aspecto fundamental, a la hora de trabajar con datos en un programa en ensamblador, es el conocimiento de las bases de numeración más apropiadas para cada caso concreto. Las personas estamos acostumbradas a utilizar la base 10, pero ésta no suele ser la más adecuada muchas veces, resultando más fácil la representación en base hexadecimal, octal o incluso binaria.

El objetivo de este capítulo es sentar las bases sobre representación de la información que se precisarán en el resto del libro, comenzando por efectuar una introducción al trabajo con distintas bases de numeración, detallando cómo se representan números

enteros de diferentes tamaños, con o sin signo, continuando con la representación BCD, de números en coma flotante y, finalmente, una somera explicación sobre representación de caracteres y cadenas de caracteres.

Bases de numeración

El ser humano ha tenido siempre la necesidad de contar aquello que tenía a su alrededor, ingenierando para ello el sistema o base y, posteriormente, los símbolos necesarios para reflejar en algún medio esos números.

Un sistema o base de numeración puede definirse como un conjunto de símbolos que sirven para contar y realizar operaciones aritméticas, operaciones básicas desde nuestros orígenes. A lo largo de la historia han ido surgiendo diversos sistemas de numeración, siendo el más conocido el que utiliza la base diez, o sistema decimal, y los símbolos árabigos que todos conocemos. La mayoría de estudiosos coincide en afirmar que esto es debido al uso de los dedos como elementos básicos a la hora de contar. Ciertas culturas, no obstante, usaron bases de numeración consistentes en 5, 20 y hasta 60 símbolos diferentes.

A pesar de todo, independientemente del número de símbolos o elementos distintos empleados, el mecanismo para contar prácticamente era siempre el mismo: llegado al último de los símbolos posibles, éstos comenzaban a repetirse cambiando su posición. Esto da lugar, en el sistema de numeración decimal, a las unidades, decenas, centenas, millares, etc. Por ello a estos sistemas de numeración se les llama habitualmente *posicionales*, dado que un mismo símbolo representará un valor u otro dependiendo de la posición que ocupe dentro de la cifra.

Nota

En realidad los sistemas de numeración posicionales eran una minoría, en las culturas antiguas, frente a los no posicionales o sistemas de numeración aditivos. El ejemplo más conocido actualmente de un sistema de numeración de este tipo es el romano, compuesto por símbolos como I, V, X, C, D, M y Q que siempre representan un mismo valor, sin importar su posición, que se suma o resta al resto de los símbolos de la cifra.

Sistemas de numeración informáticos

A la hora de programar en ensamblador necesitaremos conocer, aparte del sistema de numeración decimal que todos conocemos, otro que resulta imprescindible: el sistema binario. Hay dos sistemas adicionales, el octal y el hexadecimal, que facilitan, en ciertas situaciones, la codificación de valores sin tener que escribir una larga secuencia de ceros y unos.

El sistema binario se basa en la utilización de los dígitos 0 y 1, necesitando, por tanto, muchos más dígitos para representar un cierto valor. Este sistema es habitual al trabajar a bajo nivel, por ejemplo programando en ensamblador, porque internamente los procesadores operan sobre transistores que sólo pueden tener dos estados: abierto o cerrado, apagado o encendido, cierto o falso, 0 ó 1. Como tendrá ocasión de ver en posteriores capítulos, un solo dígito en un número binario nos indicará si una condición se cumple o no, por poner un ejemplo.

Es habitual llamar *bit* a un solo dígito binario. Un *bit*, por tanto, es una unidad de información que sólo puede representar dos valores: 0 ó 1. En contraposición, un dígito decimal puede tener diez estados distintos, desde el 0 hasta el 9.

Dada la incomodidad de representar valores grandes como secuencias de ceros y unos, como números en el sistema binario o base dos, el sistema más usado por los programadores en ensamblador es el hexadecimal.

Este se basa en la utilización de dieciséis símbolos diferentes: los números, del 0 al 9, y las letras de la A a la F. Cada dígito hexadecimal representa a una cifra binaria de cuatro dígitos (véase la tabla 2.1), por lo que es relativamente fácil convertir un número binario en hexadecimal y viceversa, especialmente si se compara con la conversión desde el sistema decimal.

Tabla 2.1. Correspondencia binario-hexadecimal.

Binario	Hexadecimal	Binario	Hexadecimal
0000	0	0001	1
0010	2	0011	3
0100	4	0101	5
0110	6	0111	7
1000	8	1001	9
1010	A	1011	B
1100	C	1101	D
1110	E	1111	F

A medio camino, entre las bases 2 y 16, se encuentra la base 8 o sistema octal. Éste emplea ocho símbolos, los números del 0 al 7, de forma que cada dígito equivale a una cifra binaria de tres dígitos. La equivalencia entre binario y octal coincidiría con la indicada en las cuatro primeras filas de la tabla 2.1.

Cálculo del valor de una cifra

Cuando leemos una cifra decimal, como puede ser un precio en un supermercado, calculamos automáticamente el valor sin darnos cuenta del proceso que llevamos a cabo. Un mismo dígito representa un valor u otro dependiendo de la posición en la que aparezca en la cifra, cuyo valor global se obtiene mediante una serie de multiplicaciones y sumas.

Suponga que ve escrita la cifra 2034 que, de inmediato, se traduce en su mente como "dos mil treinta y cuatro".

Para llegar ahí, sin embargo, habrá efectuado una serie de cálculos. Ve el primer dígito, el 2, y, por su posición, lo multiplica por 1000, sumándole a continuación el resultado de sumar el dígito siguiente por 100, el siguiente por 10 y el siguiente por 1. Es decir, obtiene el valor 2034 a partir de $2000+0+30+4$.

¿Por qué multiplica un dígito por 1000, otro por 100 y otro por 10? Simplemente porque su posición indica que son millares, centenas y decenas, respectivamente. Al ser la base de numeración la decimal, la obtención de ese multiplicador es muy sencilla ya que basta con ir añadiendo ceros a medida que la posición del dígito está más a la derecha. No obstante, matemáticamente lo que estamos haciendo es una operación de potenciación tomando como base la base de numeración, en este caso 10, y como exponente la posición del dígito, que sería 0 para el que está más a la derecha, 1 para el siguiente y así sucesivamente.

El proceso de descomposición de la cifra en sus componentes es el mostrado en la figura 2.1. En la parte inferior aparece el valor de cada dígito multiplicado por la base de numeración elevada al exponente que indique la posición. Más arriba aparece la operación de potenciación ya resuelta, en el nivel superior se han resuelto las multiplicaciones y sólo quedan sumas que, al final, producen el número 2034.

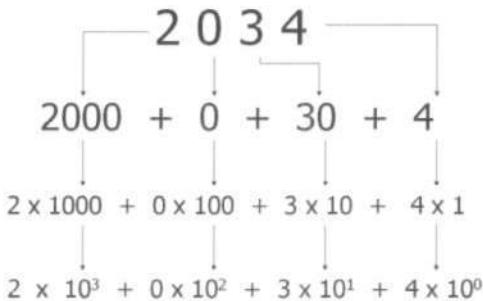


Figura 2.1. Descomposición de un número en las diversas operaciones de suma, multiplicación y potenciación que lo componen.

Aunque le parezca algo totalmente obvio, comprender cómo calculamos el valor de una cierta cifra es totalmente indispensable a la hora trabajar con números expresados en otras bases, especialmente si no estamos acostumbrados a trabajar con ellas como suele ocurrir con la base 2 y 16.

Conversión entre bases de numeración

Si pasamos años trabajando con números en una cierta base de numeración, como la binaria o hexadecimal, al final conoceremos su valor sin necesidad de conversión alguna pero, en principio, lo habitual es que precisemos la conversión a la base decimal, la que estamos acostumbrados a manejar a diario. Las operaciones posibles son dos: conversión de un número de cualquier base a la base decimal, para lo cual se emplean las operaciones de potenciación, multiplicación y suma que acaban de describirse en el punto previo, o bien la conversión inversa, desde la base diez a cualquier otra, caso en el que deberemos efectuar una serie de divisiones sucesivas.

Conversión a la base decimal desde cualquier base

Si ha entendido el proceso representado en la figura 2.1, convertir un número desde cualquier base de numeración a la base decimal no le resultará especialmente difícil. Lo único que tiene que hacer es sustituir, en la línea inferior, la base 10 por la que corresponde al número a convertir.

Supongamos que nos entregan el número 3121 expresado en base 4 y que queremos conocer cuál es su valor decimal. Separaríamos los dígitos y los multiplicaríamos por 4, que es la base, elevado a 0, 1, 2 y 3, respectivamente, como se ha hecho en la figura 2.2. Esos resultados los sumaríamos obteniendo el valor decimal: 217.

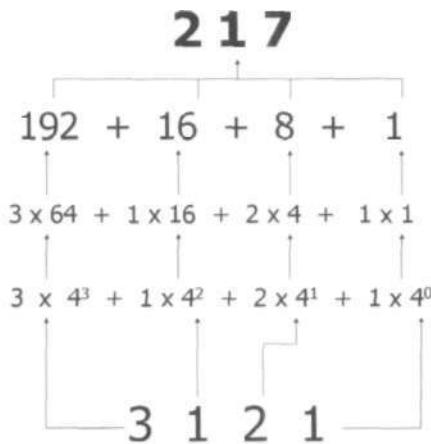


Figura 2.2. Conversión de un número de base 4 a base 10.

El proceso sería idéntico para cualquier otro número y cualquier otra base, tan sólo hay que cambiar la base de la operación de potenciación y conocer el valor decimal de cada dígito. En este ejemplo, los dígitos 0, 1, 2 y 3, únicos válidos en la base 4, coinciden en valor con los mismos dígitos de la base 10. Otras bases de numeración, como es el caso de la hexadecimal, emplean dígitos que no tienen una correspondencia directa

con un dígito decimal, pero sabemos que los símbolos A a F corresponden a los valores 10 a 15 en decimal.

Conversión a cualquier base desde la base decimal

La conversión de un número decimal a cualquier otra base, como puede ser la 4, se efectúa mediante una serie de divisiones sucesivas. Se toma el número decimal y se divide por la base, obteniendo un cociente y un resto. Éste será el dígito que quede más a la derecha en el número resultante ya convertido. El cociente obtenido, en caso de ser superior a la base usada como divisor, vuelve a dividirse, obteniéndose un nuevo cociente y un nuevo resto. El último cociente, cuando ya no sea posible seguir dividiendo, se convertirá en el primer dígito del número convertido. En realidad se tarda más en explicar e imaginarse el proceso que en efectuarlo. Tome el número 217 en decimal y divídalo sucesivamente por 4, como se ha hecho en la figura 2.3, hasta obtener un cociente menor a 4. En ese momento ya tiene el número 217 pero en base 4, tomando el último cociente y los restos en orden inverso a como se han ido obteniendo.

De manera análoga se convertiría a cualquier otra base, tan sólo hay que cambiar el divisor por el que nos interese. Intente convertir el mismo número a base 3 y base 2. Obtendrá diversas representaciones del mismo valor pero en distintos sistemas de numeración.

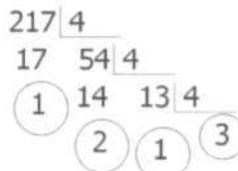


Figura 2.3. Conversión de un número decimal a su equivalente en otra base.

Nota

Aunque es posible pasar números de cualquier base a cualquier otra, sin pasar por la base decimal, ese paso intermedio nos facilita el trabajo ya que estamos acostumbrados a sumar, multiplicar y dividir en decimal, no en otras bases.

Operar con números binarios

La base 2 o binaria es particularmente interesante a la hora de programar en ensamblador, ya que un dígito binario puede tener tan sólo dos estados y, consecuentemente, ser usado para representar estados de puertas lógicas como el registro de indicadores que tiene el procesador y que conocerá en un capítulo posterior.

Los únicos dígitos válidos en un número binario son el 0 y el 1. La conversión a y desde la base 10 la efectuaríamos según las indicaciones dadas antes. Por ejemplo, el número 217 decimal daría como resultado, mediante la división sucesiva entre 2, el número binario 11011001. La división, si el número no es muy grande como en este caso, resulta especialmente fácil, ya que tan sólo tenemos que ver si el número es impar, caso en el que nos quedaría un resto de 1, y partirlo por la mitad, realizando la misma operación con el cociente obtenido hasta que fuese 0 ó 1.

En la figura 2.4 se ha presentado la conversión del número 11011001 binario a base 10. Si conocemos el valor de cada dígito según su posición, algo fácil ya que en este caso se trata simplemente de ir multiplicando por 2, el proceso también es sencillo cuando nos acostumbramos a él, ya que después tan sólo hay que mantener ese valor, si el dígito es un 1, o desecharlo, en caso de ser 0.

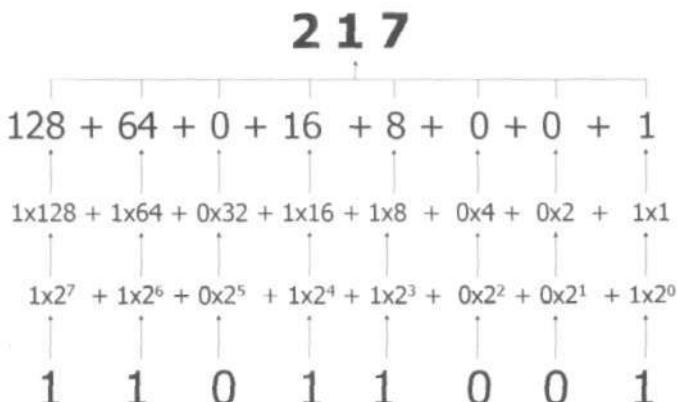


Figura 2.4. Conversión de un número de base 2 a base 10.

Bits, nibbles y bytes

Estos tres términos hacen referencia a conjuntos formados por 1, 4 y 8 dígitos binarios, respectivamente. Un bit, por tanto, está compuesto por un solo dígito binario, un 0 o un 1, pasando por ser la menor unidad de información que es posible manipular en un ordenador. Podría decirse, por tanto, que un dígito binario es un bit, si bien como unidad de medida el bit se define como la entropía de un sistema con dos sucesos equiprobables. Se llama *nibble* al conjunto de cuatro bits, mediante el cual es posible formar 16 (2^4) combinaciones distintas, es decir, 16 números que, en decimal, serían los comprendidos entre el 0 y el 15 según se aprecia en la figura 2.5. Recordar estas 16 combinaciones, cuando se lleva un tiempo trabajando con ensamblador, es bastante corriente y, como verá en un momento, facilita en gran medida la conversión de números al sistema hexadecimal.

Por su parte, se conoce como *byte* al conjunto de 8 bits. Con un byte, por tanto, es posible representar 256 (2^8) valores distintos que, en decimal, serían los comprendidos entre 0 y 255. Las celdillas de memoria de los ordenadores tienen capacidad para almacenar

exactamente un byte, siendo ésta también la capacidad de algunos registros internos de los procesadores.

0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Figura 2.5. Correspondencia entre los valores que puede tomar un *nibble* y números decimales.

Tomando como base la secuencia mostrada en la figura 2.5, le resultará fácil obtener la representación binaria de todos los valores que puede contener un byte y su equivalente en decimal.

Números con signo

Para representar números negativos en la base decimal, la que usamos a diario, basta con anteponer el símbolo - al número en cuestión. Esta notación, sin embargo, no es habitual al operar con números binarios. En ensamblador no puede ponerse un signo - delante de un número binario y esperar que se interprete como negativo, puesto que el número completo sólo puede contener ceros y unos, ningún otro símbolo.

En el punto anterior se indicaba que un byte puede representar 256 valores distintos, concretamente los números del 0 al 255. En realidad, esto no tiene necesariamente que ser así, pudiendo utilizar ese rango de valores para representar cualquier otro conjunto de números que nos convenga. Podemos, por ejemplo, usar la mitad del rango para representar números positivos y la otra mitad para los negativos. Esto es lo que se consigue con el complemento a dos.

Usando el complemento a dos se asume que el séptimo bit del byte, el que aparece más a la izquierda, actúa como bit de signo. Si está a cero, el número es positivo, mientras que si está a 1, se interpretará como negativo. Los restantes siete bits se emplean para

representar el número, que estará comprendido entre 0(0000000) y 127 (1111111). El número 0000000 se interpreta como 0, mientras que 1000000 sería -128. Por tanto con el complemento a dos podemos representar los números desde el -128 al 127, un total de 256 valores al incluir el 0.

Para convertir cualquier número positivo en negativo, siempre hablando de la base 2, hay que dar dos pasos: lo primero es invertir todos los bits del número positivo, poniendo a 0 los que están a 1 y viceversa. A continuación se sumará 1 al número invertido. Suponga, por ejemplo, que quiere representar el número -16 en binario. Los pasos serían los siguientes:

- Convertir 16 a su representación binaria, que es 00010000.
- Invertir todos los bits del número, obteniendo 11101111.
- Sumar 1 al resultado, obteniendo 11110000.

Por tanto, -16 en binario sería 11110000. Podemos efectuar una comprobación bastante simple para estar seguros de que el resultado es el correcto. Sume a 11110000 el número 00010000, que es la representación binaria de 16. Al sumar -16 y 16 deberíamos obtener el resultado 0, como ocurre en la figura 2.6.

A binary addition diagram. It shows two numbers: 11110000 and 00010000. The first number has a circled '1' under the fourth bit from the left. The second number has a circled '1' under the fifth bit from the left. A horizontal line with a plus sign (+) is between them. Below the line is a sum: 10000000. An arrow points down to the rightmost zero of the sum, indicating the result is zero.

Figura 2.6. Sumamos las representaciones binarias de -16 y 16.

Para comprender la operación mostrada en la figura 2.6 es imprescindible saber cómo sumar números binarios y, para ello, basta con analizar cómo sumamos normalmente números decimales. Cuando la suma de dos dígitos supera el máximo valor representable con un dígito se produce un acarreo, lo que normalmente en el colegio se enseña como nos llevamos una. Ese acarreo se suma al dígito siguiente y, ocasionalmente, puede producir otro acarreo. Las posibles sumas de dígitos binarios aislados serían éstas:

- $0 + 0 = 0$
- $1 + 0 = 1$
- $0 + 1 = 1$
- $1 + 1 = 10$

Observe el último resultado. $1 + 1$ es igual a 2, pero en la base binaria ese dígito no existe y, por ello, se convierte en 0 y se produce el acarreo, obteniendo 10. Es lo mismo

que ocurre al sumar 9 y 1 en la base decimal. Teniendo esto en cuenta, analice la operación representada en la figura 2.6. Observe que el resultado final es 100000000, es decir, un número de nueve bits. Al quedarnos sólo con los ocho primeros, ya que un byte se compone de ocho bits, tenemos el valor 0. Se demuestra, por tanto, que 11110000 es el número -16.

Nota

Dedique algún tiempo a efectuar sumas de números binarios hasta que logre efectuar el acarreo de forma más o menos automática. Intente restar un número binario de otro. Puede convertir a y desde decimal para comprobar que los resultados son correctos.

Operar con números hexadecimales

Como ha podido ver en los puntos previos, operar con números binarios en ocasiones puede ser algo complicado porque se necesitan muchos dígitos para representar números relativamente pequeños.

Sin embargo, a veces precisaremos usarlos cuando queremos estar seguros de controlar el estado de cada bit, y la conversión a decimal requiere una serie de operaciones que ya conoce.

La base hexadecimal, o dieciséis, emplea seis símbolos más que la base decimal, las letras A a la F, para representar los números 10 a 15. Su ventaja es que cada dígito hexadecimal puede representar los mismos valores que un *nibble* binario, siendo relativamente fácil recordar la correspondencia entre número binario y dígito hexadecimal. De esta forma resulta más fácil introducir números relativamente grandes conociendo el estado de cada bit pero sin usar directamente la base 2.

De binario a hexadecimal y viceversa

Los pasos para convertir números entre bases explicados anteriormente, en el punto *Conversión entre bases de numeración*, son perfectamente aplicables con números hexadecimales, sólo hay que cambiar la base 2 por la 16 o dividir entre 16 en lugar de hacerlo entre 2, según el sentido de la conversión.

En teoría, para convertir un número de binario a hexadecimal tendríamos que efectuar en realidad dos conversiones: de binario a decimal y de decimal a hexadecimal. Lo mismo sería aplicable en sentido inverso.

Conociendo la secuencia de bits que representa cada dígito hexadecimal, sin embargo, la conversión es mucho más rápida al poder efectuarse directamente. Como se decía antes, con un dígito hexadecimal es posible representar cualquier *nibble* binario. La figura 2.7 muestra la correspondencia entre cada *nibble* posible y los valores decimales y hexadecimales.

Binario	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Figura 2.7. Correspondencia binario-decimal-hexadecimal.

Suponga que quiere convertir el número binario 11011001 a su correspondiente hexadecimal. Primero lo dividimos en dos *nibbles*, obteniendo 1101, por una parte, y 1001, por otra. Buscamos los dígitos hexadecimales que corresponden a esos *nibbles*: D y 9, obteniendo así el número hexadecimal D9. Si convierte D9 a decimal (véase la figura 2.8) observará que el resultado es 219, el mismo obtenido al convertir 11011001 a decimal.

Para convertir cualquier número hexadecimal a binario el proceso sería análogo. Por ejemplo, el número 3B2D (15149 decimal) puede convertirse directamente a binario, sin necesidad de pasar a decimal y después dividir sucesivamente entre 2 un número tan grande, separando cada dígito y sustituyéndolo por el correspondiente *nibble* binario, obteniendo 0011101100101101. Eliminando los ceros a la izquierda, no significativos, nos quedaría 1101100101101.

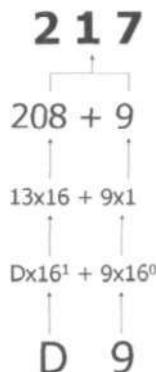


Figura 2.8. Conversión de hexadecimal a decimal.

Números negativos en hexadecimal

Al igual que ocurre con los números binarios, al usar la base dieciséis no es habitual el empleo del símbolo - para indicar que un número es negativo, utilizándose en su lugar un mecanismo idéntico al descrito anteriormente para la base dos. Si con un byte pueden representarse los valores hexadecimales 00 a FF, un total de 256, éstos pueden interpretarse como todos positivos (0 a 255) o bien como positivos desde el 00 al 7F y negativos del 80 al FF. Puede efectuar algunas comprobaciones convirtiendo cualquier número entre 80 y FF a binario. Observe que el séptimo bit siempre se obtiene a 1, indicando que es un número negativo. La forma más fácil de convertir cualquier número hexadecimal en negativo consiste en convertirlo en binario y aplicarle el complemento a dos explicado en un punto previo.

Números en base ocho

^ ^ ^

Aunque de uso mucho menos corriente que los sistemas binario y hexadecimal y, por supuesto, decimal, tampoco es extraño encontrar en un programa en ensamblador números octales o de base ocho. Como puede suponer, en este sistema de numeración tan sólo existen los dígitos 0 a 7, de tal forma que el número 10 octal sería el 8 decimal.

Podríamos preguntarnos qué sentido tiene la utilización de otra base de numeración más, cuando ya contamos, como de uso habitual, con la base decimal, la binaria y la hexadecimal. Dejando de lado los aspectos históricos, y centrándonos tan sólo en los prácticos, encontramos una sola razón: cada dígito octal se corresponde con una combinación de tres bits, de tal forma que es posible representar los valores 000 a 111. Sabiendo de memoria tan sólo esos valores, podemos representar cualquier número binario con menos dígitos, mediante la base ocho, y sin necesidad de recurrir a la base hexadecimal, para la cual es necesario conocer de memoria el doble de combinaciones binarias.

Tomemos una vez más el número binario 11011001 de ejemplos anteriores. Para representarlo en octal lo dividiríamos, de derecha a izquierda, en grupos de tres bits, quedándonos con 11, 011 y 001, obteniendo el número octal 331. Éste sería equivalente al D9 hexadecimal que obteníamos en un punto previo, o bien al número 217 decimal. La diferencia es que a conversión de binario a octal, y viceversa, resulta más fácil, ya que en dicha base sólo hay ocho dígitos que se corresponden con la combinaciones 000 a 111.

Identificación de la base de un número

Dado que una parte de los dígitos empleados en las distintas bases de numeración coinciden, al introducir un número es necesario identificar de alguna forma la base de numeración en que está expresado. El número 10, por ejemplo, podría ser el diez decimal, pero también el número dos en binario, el dieciséis en hexadecimal o el ocho en octal.

Matemáticamente la base de un número suele indicarse como un subíndice a modo de sufijo, por ejemplo 10_2 para indicar que el número está en base dos o 10_{16} para comunicar que es un número hexadecimal. Al codificar en ensamblador, sin embargo, lo que se usa como sufijo o prefijo no es un subíndice sino un cierto carácter. En NASM (una de las herramientas que conocerá posteriormente), por ejemplo, se emplean las letras B, Q y H, a modo de sufijo, para indicar que el número es binario, octal o hexadecimal, respectivamente. Puede efectuarse una prueba muy sencilla mediante el código mostrado a continuación. Introdúzcalo en un archivo llamado Bases .asm, ensámblelo con NASM y enlácelo con ALINK, obteniendo así un ejecutable con extensión EXE.

```
segment Datos

segment Pila stack
resb 256
InicioPila:

segment Código
.start:
    mov ax, Pila
    mov ss, ax
    mov sp, InicioPila

    mov ex, 10 ; número decimal
    mov ex, 10q ; número octal
    mov ex, 10b ; número binario
    mov ex, 10h ; número hexadecimal

    mov ah, 4ch
    int 21h
```

Observe que se introducen en CX, uno de los registros existentes en el procesador, varios valores sin hacer nada con ellos, simplemente queremos obtener un código ejecutable para poder analizarlo con un desensamblador, en este caso la herramienta DEBUG de DOS. invoque a esa herramienta facilitando como parámetro el nombre de archivo Bases .exe, es decir, el ejecutable obtenido a partir del código introducido. Tras un momento debe aparecer un guión en el margen izquierdo, es el indicador de que DEBUG está esperando nuestras indicaciones. Introduzca el comando U y pulse **Intro**. Debe aparecer una lista de instrucciones ensamblador similar a la que se ve en la figura 2.9. Tenga en cuenta que todos los números aparecen en hexadecimal.

Observe las instrucciones mov con las que se asigna valor al registro CX. Los valores hexadecimales son A, 8, 2 y 10 (en decimal serían 10, 8, 2 y 16) correspondientes a los números 10, 10q, 10b y 10h introducidos en el código fuente.

Representación de números enteros

Ya sabemos cómo es posible representar números enteros, positivos y negativos, utilizando 8 bits, es decir, un byte, la capacidad que tiene cada una de las unidades en que se dividen las actuales memorias.

Figura 2.9. Obtenemos con DEBUG la lista de instrucciones que componen el ejecutable.

Con dicha capacidad, sin embargo, no es posible representar números mayores que 255, asumiendo que se prescinda del signo, ¿cómo es posible, entonces, trabajar con números más grandes?

La respuesta es obvia: para números mayores habrá que utilizar más bits. Con 16 bits se pueden componer $2^{16}=65536$ valores distintos que, según la representación elegida, pueden ser los números 0 a 65535 o bien -32768 a 32767. Con bits adicionales, por ejemplo 32, pueden representarse números aún mayores, del orden de miles de millones. Las técnicas para convertir números de ese tamaño entre bases, o efectuar el complemento a dos para valores negativos, son exactamente las mismas que se han descrito ya, lo único que cambia es la cantidad de dígitos con la que se trabaja.

El único aspecto que es necesario tener en cuenta, por lo tanto, es cómo se dividirá esa secuencia de 16, 32 o más bits a la hora de ser almacenada en memoria de 8 en 8, porque de ello dependerá que al recuperar los números éstos se interpreten correctamente o no. Lógicamente el orden de almacenamiento debe ser el mismo que se utilice en la posterior recuperación.

Big endian vs Little endian

Un dato de 16 bits, como puede ser una dirección de memoria en un sistema basado en el microprocesador 8086, se dividiría en dos partes de 8 bits, almacenándose esos dos bytes en dos posiciones de memoria consecutivas. Dependiendo del orden en que se almacenen esos dos bytes se dice que el dato está almacenado en formato *little endian* o *big endian*. El formato *little endian* es el que suele utilizarse en procesadores Intel y consiste en almacenar el byte de menor peso o LSB (*Least Significant Byte*) seguido del MSB (*Most*

Significant Byte). Es decir, el byte menos significativo está en la dirección de memoria más baja y el más significativo en la más alta. En el formato *big endian* el orden es el inverso, siendo el utilizado en procesadores de Motorola e IBM, por ejemplo. En la figura 2.10 puede apreciarse la diferencia entre un formato de almacenamiento y otro.

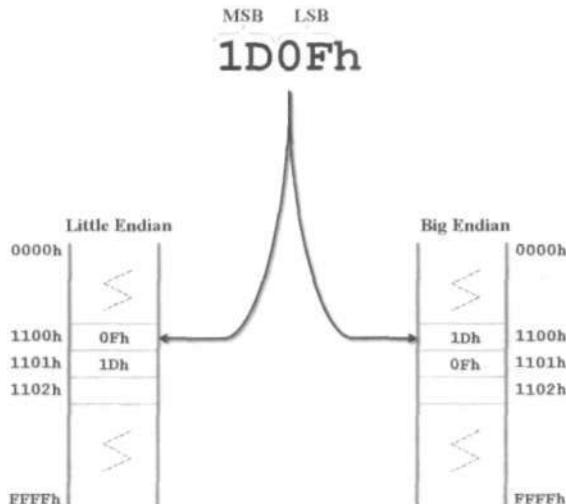


Figura 2.10. Almacenamiento de números de 16 bits.

Al aplicar estos formatos sobre números de 32 bits, dan lugar a una primera división en dos trozos de 16 bits (MSW y LSW), cada uno de los cuales se partiría en dos bytes (MSB y LSB). El esquema de la figura 2.11 muestra esta división y la forma en que se almacenaría el dato en la memoria en *little endian* y *big endian*. Como puede apreciarse, en formato *little endian* los bytes que forman el dato se almacenan en orden inverso, mientras que en formato *big endian* el orden es el mismo y, por tanto, resulta más natural.

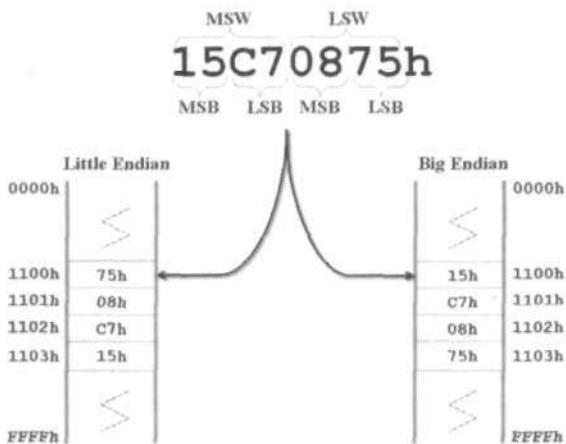


Figura 2.11. Almacenamiento de números de 32 bits.

En caso de tener que operar con números de mayor tamaño, 64 ó 128 bits por ejemplo, el proceso de división en partes, dependiendo del formato elegido, sería el mismo.

Nota

En su momento, al estudiar las instrucciones aritméticas con que cuenta la familia x86, aprenderá a operar con números enteros de distintos tamaños.

epresentacion BCD

, -----,,--

La aritmética BCD (*Binary Codea Decimal*) se fundamenta, básicamente, en la representación de un dígito decimal (0 a 9) usando 4 bits, de forma que en un byte es posible introducir dos dígitos decimales. De esta forma la secuencia de bits 10010011 no se correspondería con el número 147, que sería el resultado al operar en binario, sino con el número 93($1001=9$ y $0011=3$). La conversión entre BCD y decimal resulta muy rápida y cómoda, por eso la mayoría de los microprocesadores están preparados para utilizar este tipo de aritmética. El 8086 no es una excepción, contando con un bit en el registro de estado, el bit AC (*Auxiliary Cary*), que señala el acarreo en BCD. También hay instrucciones específicas para operar con aritmética BCD.

Este tipo de representación surgió con las calculadoras y los primeros microprocesadores de 4 bits, como el Intel 4004, pero sigue teniendo aplicación en contextos determinados. En un capítulo posterior aprenderá a realizar cálculos con aritmética BCD.

Representación de números en coma flotante

La representación en un ordenador de números con parte decimal puede efectuarse básicamente de dos formas distintas: con un formato de coma fija o bien recurriendo a la coma flotante. Los formatos de coma fija se limitan a establecer un número fijo de posiciones decimales en cada número, existiendo un algoritmo de conversión decimal-binario-decimal basado en productos y divisiones, como el explicado para los números enteros aunque ligeramente más complejo, separando la parte entera de la decimal.

Los microprocesadores no tienen ningún soporte especial para operar con números en coma fija, pero sí para trabajar con números en coma flotante, normalmente siguiendo el estándar IEEE-754. Este formato es similar al científico y divide el número en tres partes: signo, mantisa y exponente. Al primero siempre se dedica un bit, el de mayor peso, repartiéndose el resto entre mantisa y exponente. En el mencionado estándar se establecen cuatro precisiones diferentes: precisión simple utilizando 32 bits, precisión simple extendida con 44 o más bits, precisión doble con 64 bits y precisión doble extendida con 80 o más bits. La precisión simple extendida no suele utilizarse, siendo la precisión simple y la doble las más usuales.

En la figura 2.12 puede apreciarse la división que se efectúa de los 32, 64 u 80 bits, dependiendo de la precisión elegida. Observe tanto el número de bits asignado a exponente y mantisa como su colocación: primero el bit de signo, a continuación el exponente y finalmente la mantisa.

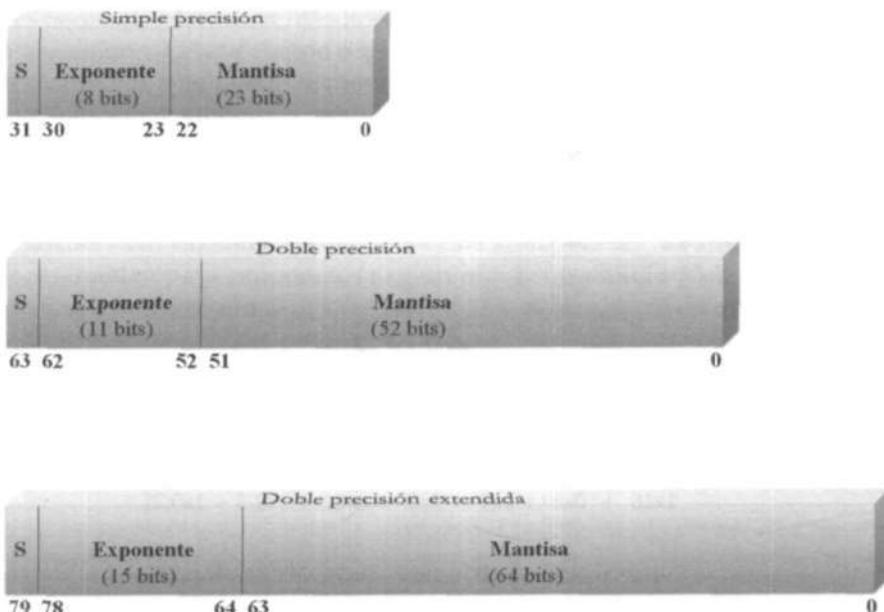


Figura 2.12. Estructura de números en coma flotante y distintas precisiones según el estándar IEEE-754.

Normalización de la mantisa

El estándar IEEE-754 exige que la mantisa de los números en coma flotante esté normalizada, con el objetivo de conservar el mayor número de dígitos significativos. Lo primero que debe hacerse es obtener, a partir del número decimal con parte fraccionaria, su representación en binario. Para ello se procesa por separado la parte entera y la parte fraccionaria. La primera se convierte en binario por divisiones sucesivas, como se explicó en un punto previo. Para convertir la segunda, la parte fraccionaria, se efectúan multiplicaciones sucesivas por 2, tomando como bits la parte entera que vaya obteniéndose y repitiendo el proceso hasta que el resultado sea 0 o se hayan obteniendo los bits fraccionarios deseados.

Tomemos, por ejemplo, el número decimal 23,75 y veamos cómo se efectuaría la conversión a binario. Los pasos serían:

1. $23 / 2 = 11$. Resto 1.
2. $11 / 2 = 5$. Resto 1.

3. $5 / 2 = 2$. Resto 1.

4. $2 / 2 = 1$. Resto 0.

La parte entera en binario sería 10111. Continuamos:

5. $0,75 * 2 = 1,5$ 0. Parte entera 1. Queda 0,50.

6. $0,50 * 2 = 1,00$. Parte entera 1. Queda 0, 00, no hay que continuar.

La parte decimal en binario sería , 11. El número completo en decimal, por tanto, es 10111,11. Para comprobar que, en efecto, este valor en binario corresponde al número decimal 23,75, basta con realizar la operación inversa (véase la figura 2.13). Fíjese en cómo los dígitos binarios que están a la derecha de la coma tienen exponentes negativos, de forma que $1x2^{-1}$ se convierte en $1/2^1$, cuyo resultado es 0,5. De manera análoga: $1x2^{-2}=1/2^2=1/4=0,25$.

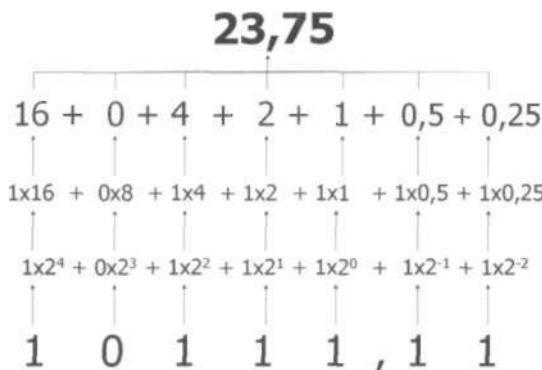


Figura 2.13. Conversión binario-decimal de un número con parte fraccionaria.

El valor 10111,11 podría expresarse en notación científica como $10111,11 \times 2^0$, de igual forma que 23,75 puede escribirse como $23,75 \times 10^0$. Ninguna de estas cifras está normalizada. Para ello habría que ajustar mantisa y exponente de forma que a la izquierda de la coma quedase un único dígito.

En el caso del número decimal bastaría con dividir la parte entera entre 10, sumando 1 al exponente: $2,375 \times 10^1$. El número sigue siendo el mismo, únicamente se expresa de manera distinta.

Para normalizar la mantisa en binario se procede de igual manera: se desplaza la coma hacia la izquierda las posiciones que sea preciso, hasta que quede solamente el dígito 1 a su izquierda, incrementando el exponente de acuerdo con el desplazamiento. De esta forma $10111,11 \times 2^0$ se expresaría como $1,01111 \times 2^2$.

De la mantisa normalizada finalmente se prescinde del dígito 1 inicial, de la parte entera, introduciéndose en los bits reservados a la mantisa (véase la figura 2.12) los bits de la parte fraccionaria, teniendo en cuenta que en ésta los ceros no significativos están a la derecha y no a la izquierda. Es decir, la mantisa para ese número expresado en coma flotante de simple precisión sería 011110000000000000000000, un total de 23 bits.

Codificación de! exponente

Tras ajustar la mantisa, a fin de normalizarla, obtenemos el exponente definitivo. Los números en notación de coma flotante pueden representar tanto números muy grandes como muy pequeños, muy cercanos a 0. En estos últimos el exponente será negativo, por lo que es necesario plantearse, a la hora de codificar el exponente en binario, cómo se representa ese signo. El bit de signo que aparece en la figura 2.12 corresponde al signo del número completo, que puede ser un número positivo o negativo, y no tiene nada que ver con el signo del exponente, que determinará el número de posiciones que hay que desplazar la mantisa a izquierda o a derecha.

Para almacenar exponentes tanto negativos como positivos el estándar IEEE-754 usa la representación *en exceso a N*, consistente en sumar al exponente original un valor N. Éste dependerá del número de bits dedicados al exponente, de forma que la mitad de los valores queden para exponentes negativos y la otra mitad para positivos, calculándose mediante la expresión $N=2^{BE-1}-1$, donde BE es el número de bits existentes para el exponente.

El valor N a sumar al exponente, por tanto, será 127 para simple precisión, 1023 para doble precisión y 16383 para doble precisión extendida. El exponente del número 23,75, del que ya se había obtenido la mantisa, sería $4+127 = 131$. Sería este valor, 131, el que se convertiría en binario: 10000011 se colocaría en la posición dedicada al exponente.

Codificación del signo

La codificación del signo en coma flotante IEEE-754 es la habitual al representar números en sistema binario, la misma utilizada en el formato de complementado a 2. Si el número es positivo el bit de signo quedará a 0 y si es negativo quedará a 1.

Uniendo todas las piezas, el número 23,75 se representaría en simple precisión, según el estándar IEEE-754, como 0 10000011 01111100 00000000 00000000. Agrupándolos en bytes, como es habitual, tendríamos 01000001 10111110 00000000 00000000. La conversión de este formato a decimal se efectuaría según la fórmula siguiente:

$$-1^{BS} \times 2^{exp} \times mantisa$$

En este caso tendríamos:

- BS=0->-1° = 1
- exp=10000011=131, $2^{131}-127=2^4$
- mantisa=1.011111
- $1 \times 1.011111 \times 2^4 = 10111.11 = 23,75$

También se podría convertir la mantisa a decimal y a continuación multiplicar por el exponente:

- $1.011111 \times 2^4 = 1.484375 \times 16 = 23,75$

Nota

Elija un número decimal con parte fraccionaria, alternando entre mayor y menor que cero, positivo y negativo, y repita todo el proceso descrito en estos últimos puntos para obtener su representación en simple precisión IEEE-754. De esta forma se familiarizará rápidamente con la forma en que los microprocesadores actuales tratan los números en coma flotante.

Representación de caracteres y cadenas

Además de números, enteros o con parte fraccionaria y de distintas precisiones, muchas informaciones puede representarse como números: el día del mes, el peso de una persona, la distancia entre dos ciudades, la solución a una ecuación, etc., el segundo tipo de información más importante representable en un ordenador es el texto.

Internamente, en la memoria de un ordenador, únicamente pueden almacenarse secuencias de ceros y unos. Que se agrupen de ocho en ocho, formando bytes, se debe a las características físicas de la memoria. La interpretación que se les dé, como se ha visto ya en el caso de los números, es algo que depende totalmente de la lógica de las aplicaciones.

El texto se compone de caracteres individuales, tomados de un conjunto finito al que se conoce como *alfabeto*. Existen multitud de alfabetos, con distinto número de elementos y símbolos también diferentes. Los alfabetos italiano y chino, por poner un ejemplo, son dos extremos en cuanto al tamaño del conjunto de símbolos y su grafía. Cuantos más símbolos existan en un alfabeto tantos más bits se necesitarán para poder representarlos.

En los ordenadores también se utilizan distintos alfabetos para representar caracteres. Dos de los más conocidos son el ASCII y Unicode. El primero utiliza 8 bits por símbolo, si bien solamente los primeros 7 tienen una asignación estándar. En este alfabeto, limitado a 128 símbolos, no existe la eñe, las vocales acentuadas ni, por supuesto, muchos otros símbolos específicos de lenguas no occidentales. En Unicode el tamaño es de 16 bits, ofreciendo espacio suficiente para representar todos los símbolos de todas las lenguas conocidas.

Lo que se hace en los ordenadores es asignar un código a cada símbolo del alfabeto, de forma que en la memoria se almacena dicho código, no el carácter en sí. La apariencia de éste dependerá tanto del dispositivo de destino (una pantalla, una impresora, etc.) como de la aplicación que lo trate, recuperándose normalmente de archivos de fuente de letra. Por ello es importante que la codificación (el alfabeto) que se ha empleado para representar un cierto carácter, por ejemplo a la hora de guardarlo en un archivo o enviarlo a través de una red, esté indicada claramente, de lo contrario el resultado puede ser ilegible.

En cuanto a las cadenas de caracteres, no son más que secuencias de códigos almacenados en memoria uno tras otro. Cómo se establezca el final de la cadena es algo que depende normalmente de cada aplicación o, para ser precisos, del lenguaje que se emplee

en su desarrollo. En lenguajes como C todas las cadenas tienen al final un código 0 que indica la posición en que finaliza. Otros lenguajes, sin embargo, colocan delante de la cadena una cabecera en la que se indica el número de caracteres con que cuenta.

En capítulos posteriores aprenderá a trabajar con cadenas de caracteres en ensamblador, indicando dónde terminan usando el código apropiado.

Resumen

Este capítulo le ha servido para analizar cómo interpretamos normalmente un número cualquiera, así como para conocer tres bases de numeración que resultan fundamentales a la hora de programar en ensamblador: la binaria, la hexadecimal y la octal, especialmente las dos primeras. Asimismo, sabe cómo convertir un número desde cualquier base al sistema decimal y viceversa.

Aunque es posible sumar, restar, multiplicar y, en general, efectuar cualquier operación aritmética utilizando cualquiera base de numeración, lo cierto es que para ello se requiere una cierta práctica ya que, de manera inconsciente, siempre tendemos a pensar en base diez. Por eso, generalmente es más fácil convertir los números al sistema decimal y, tras efectuar todas las operaciones que se necesiten, convertir de nuevo a la base deseada. Esto, lógicamente, no es preciso en caso de que almacenemos los valores en un registro del procesador o en memoria y efectuemos operaciones aritméticas mediante las instrucciones que conoceremos en un capítulo posterior, dado que en ese caso será el propio programa quien efectúe las conversiones apropiadas.

También ha conocido dos formatos para representar números negativos, en complemento a 2 y en exceso a N, así como la representación en BCD y los fundamentos del estándar IEEE-754 que se utiliza para representar números en coma flotante. Finalmente se ha introducido el modo en que se codifican caracteres y cadenas de caracteres en un ordenador.

3

Arquitectura de la familia de microprocesadores x86

Todos los microprocesadores empleados actualmente en ordenadores personales, incluidos los conocidos iMac de Apple, así como los de muchas estaciones de trabajo son herederos, de una forma u otra, del 8086, de ahí que todos ellos formen una familia, conocida genéricamente como x86.

En algunos casos esa herencia es lógica, como en el caso de los microprocesadores de Intel ya que incluso los más modernos han ido evolucionando paulatinamente desde ese origen común, respetando siempre una compatibilidad hacia atrás que es parte de su éxito. En otros casos, como ocurre con los microprocesadores de AMD, existe una relación similar puesto que ésta, y otras empresas en las últimas décadas, han fabricado microprocesadores compatibles x86 gracias a licencias obtenidas de Intel.

El objetivo de este capítulo es la de describir a grandes rasgos la arquitectura interna de esta familia de microprocesadores, especialmente del 8086 por ser éste la base de dicha familia y porque la complejidad de los microprocesadores más modernos, como los Core 2 o Core i7, es tan grande que prácticamente se necesitaría este libro completo para poder estudiarla.

Estructura de bloques

Como cualquier otro microprocesador, el 8086 consta de una unidad de control (CU), una unidad aritmético lógica (ALU) y una serie de registros que le permiten alojar internamente la información que precisa para su funcionamiento.

La figura 3.1 muestra un esquema de bloques muy simplificado de un microprocesador x86 y los elementos que le permiten comunicarse con la memoria a fin de recuperar instrucciones y ejecutarlas. Asumiendo que en la memoria hay preparado un programa, los pasos que daría el procesador serían los siguientes:

- La BU (*Bus Unit*) recupera de la memoria las instrucciones y las va almacenando en una cola interna, de tal manera que se encuentren ya en el microprocesador con antelación para acelerar la ejecución.
- Las instrucciones van pasando de esa cola interna de la BU a la IU (*Instruction Unit*), que se encarga de descodificar las instrucciones y recuperar los datos que pudieran llevar asociados. La IU, conjuntamente con la BU, conforman la parte del procesador que se ocupa de ir recuperando y preparando las instrucciones a ejecutar.
- Los dos bloques que aparecen a la izquierda, en el interior del procesador, forman la unidad de ejecución, compuesta de la CU (*Control Unit*) y la ALU (*Arithmetic Logic Unit*). Observe que estas unidades comparten el acceso a los registros.
- La instrucción pasa de la IU a la CU, encargada de ejecutarla sirviéndose para ello de los datos alojados en los registros y apoyándose en la ALU. En consecuencia, se modificarán dichos registros, por ejemplo el de indicadores que conocerá más adelante, y si es necesario se enviará de vuelta a la memoria el dato que corresponda. Para ello la CU se comunica con la BU.

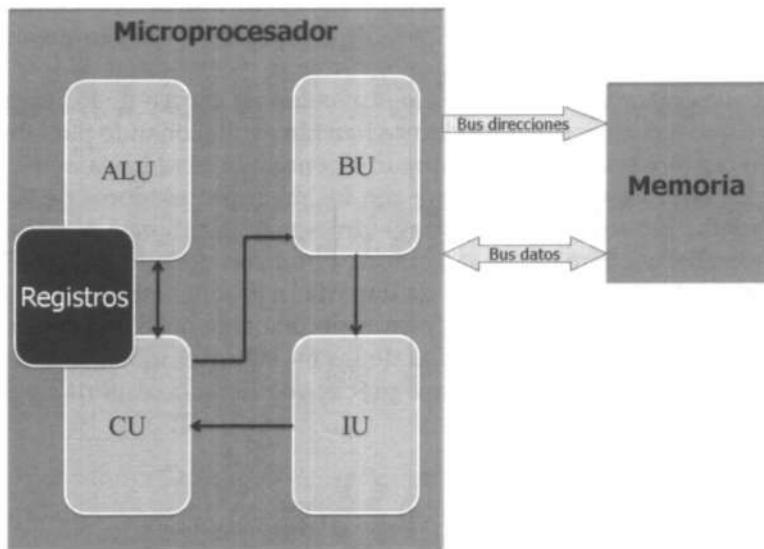


Figura 3.1. Esquema de bloques simplificado del x86.

Las distintas unidades del procesador se comunican a través de buses internos, muy rápidos, mientras que para acceder al exterior se emplean, básicamente, dos buses: el de

direcciones y el de datos. En el primero la BU dispone la dirección donde va a escribirse o leerse, mientras que por el segundo, de tipo bidireccional, viajará el dato en dirección a la memoria o al procesador, según los casos.

Nota

Además de los dos buses mencionados, existe entre la memoria y el procesador uno adicional, podríamos denominarlo bus de control, mediante el cual se indica si la operación a efectuar va a ser de lectura o escritura.

La CU y la ALU forman la unidad de ejecución o EU (*Execution Unit*), compartiendo el banco de registros de propósito general y algunos específicos, como el acumulador o el registro de estado. La BU y la IU, conjuntamente con la cola de instrucciones y la lógica de generación de direcciones, dan lugar a la unidad de interfaz con el bus o BIU (*Bus Interface Unit*). A esta unidad pertenecen los registros de segmento y el contador de programa.

Banco de registros

El banco de registros del 8086 se compone de un grupo de registros de propósito general, otro de registros de segmento, una más de registros índices y algunos específicos como el contador de programa y el registro de estado.

Este banco se ha ido ampliando en microprocesadores posteriores, extendiendo tanto el tamaño de los registros, primero a 32 bits y más adelante a 64 bits, como su número, incluyendo registros adicionales de segmento y también de propósito general, así como un conjunto de registros de 128 bits para instrucciones de tipo SIMD (*Simple Instruction Múltiple Data*).

La figura 3.2 es una representación de los registros de propósito general con que cuenta cualquier microprocesador x86 actual. Las denominaciones AX, BX, CX, DX, SI, DI, BP y SP corresponden al 8086 y son registros de 16 bits. Con el 80386 su tamaño se amplió a 32 bits, pasando a denominarse EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP. En los microprocesadores actuales, de 64 bits, los registros de propósito general tienen ese tamaño y se denominan RAX, RBX, RCX, RDX, RSI, RDI, RBP y RSP, a los que se suman ocho registros adicionales de 64 bits llamados R9 a R15. Las denominaciones xH y xL, existentes desde el 8086, facilitan el acceso a los 8 bits de mayor peso y menor peso, respectivamente, de cada registro de 16 bits.

Observe que en un microprocesador de 64 bits, como los Core y Phenom por ejemplo, se puede acceder a los 64 bits completos que forman un registro, por ejemplo con la denominación RAX; a los 32 bits de menor peso, con la denominación EAX, y a los 16 bits de menor peso, con el nombre AX. No se pueden recuperar directamente, por el contrario, los 32 bits de mayor peso de un registro de 64 bits, o los 16 bits de mayor peso de un registro de 32.

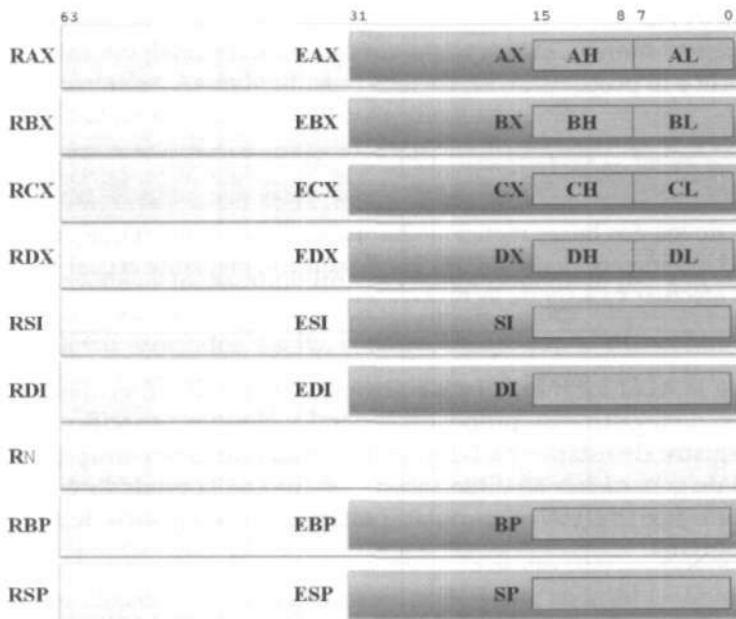


Figura 3.2. Banco de registros de propósito general.

Los nombres de los registros no están elegidos arbitrariamente, en contra de lo que pudiera parecer en un principio dado el orden alfabético de los primeros, sino que tienen su significado:

- **AX:** La denominación A (*Accumulator*), para hacer referencia al registro que actúa como acumulador, proviene de los inicios de la informática, mucho antes de que apareciesen los primeros microprocesadores. El ENIAC, por ejemplo, constaba de múltiples acumuladores y de la lógica necesaria para cargar en ellos un valor, sumarles un valor o restarlos. Cada acumulador estaba construido de decenas de válvulas de vacío. En los microprocesadores x86 el acumulador sigue siendo el destino de ciertas operaciones, como productos y divisiones, pero también puede emplearse como un registro de propósito general cualquiera en la mayoría de las situaciones.
- **BX:** También el nombre B (*Base*) como denominación de un registro, en concreto aquél encargado de mantener una dirección base, a la cual se aplicará un desplazamiento para recuperar datos, apareció con los primeros ordenadores en la década de los años cuarenta. En los x86 sigue teniendo dicha función con ciertas instrucciones.
- **CX:** Prácticamente todos los microprocesadores cuentan con un registro llamado C (*Counter*) al que se asigna una función específica en ciertas situaciones, por ejemplo al codificar bloques de código que han de repetirse un cierto número de veces o ejecutar instrucciones que pueden repetirse automáticamente.

- DX: El nombre de este registro: D (*Data*), se debe al hecho de que habitualmente se usa para contener un dato adicional que participa en una operación conjuntamente con el acumulador, como puede ser una multiplicación o una división.
- SI y DI: En los lenguajes de alto nivel es habitual trabajar con punteros para acceder a secuencias de datos en memoria, una función similar a la que tienen estos dos registros en los x86. El registro SI (*Source Index*) contendrá la dirección de los datos origen y el registro D1 (*Destination Index*) la dirección de destino. La operación puede ser la copia de un bloque de datos, una búsqueda, etc.
- SP y BP: Aunque teóricamente son registros que podrían utilizarse para otras funciones, su finalidad en los x86 es facilitar el trabajo con la pila. SP (*Stack Pointer*) contiene la dirección de inicio de la pila, mientras que BP (*Base Pointer*) actúa como puntero base a esa misma estructura de datos. En capítulos posteriores aprenderá a preparar estos registros y usar la pila.

Los prefijos E y R que se anteponen a estos registros, cuando se utilizan en programas de 32 ó 64 bits, no tienen ningún significado útil salvo la correcta identificación de la parte del registro a la que se quiere acceder.

El banco de registros (obviando los de control del microprocesador y MMX/SIMD) se completa con los mostrados en la figura 3.3. El registro IP/EIP/RIP (*Instruction Pointer*) es el puntero de instrucción, el registro que contiene la dirección donde está alojada la próxima instrucción a ejecutar.

En otros microprocesadores es conocido como PC (*Program Counter*). El registro F (*Flags*), también conocido como registro de estado, se compone de una serie de bits cuyo significado se estudia más adelante en este mismo capítulo. Tanto IP como F son registros de 16 bits en el 8086, de 32 en el 80386 y posteriores y de 64 bits en los microprocesadores x86 actuales.

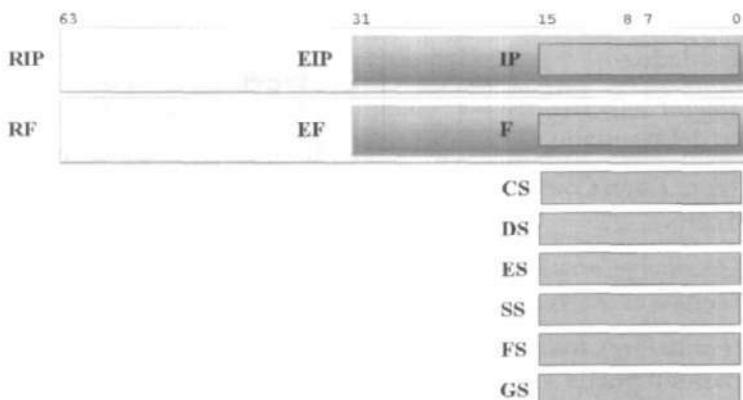


Figura 3.3. Registros de segmento, estado y puntero de pila.

El resto de los registros que aparecen en la figura 3.3, todos ellos de 16 bits, se denominan *registros de segmento*. Los dos últimos, FS y GS, no existen en los 8086/80186/80286.

Los otros cuatro son: CS (*Code Segment*), DS (*Data Segment*), SS (*Stack Segment*) y ES (*Extra Segrnent*), siendo su finalidad seleccionar el segmento de código, datos, pila y adicional, respectivamente.

El registro de estado

Como todos los microprocesadores, los de la familia x86 cuentan con un registro de estado que, como su propio nombre indica, es un reflejo del estado interno del microprocesador en cada momento. Dicho registro, a diferencia del resto, no almacena un dato que pueda tratarse como un todo, sino que cada uno de sus bits tiene un significado concreto. Dependiendo del estado de cada bit, que estará a 0 o a 1, se sabrá, por ejemplo, cuál ha sido el resultado de una operación previa o cómo deberá comportarse una cierta instrucción que va a ejecutarse a continuación.

La figura 3.4 es una representación del registro de estado del 8086. Fíjese en que solamente parte de sus 16 bits tienen asignada una función. Los bits señalados en dicha figura pueden agruparse en tres categorías distintas: indicadores de estado: OF, SF, ZF, AF, PF y CF, indicadores de control: DF e indicadores de sistema: IF y TF.



Figura 3.4. Estructura del registro de estado en un 8086.

Aunque en su momento aprenderá a utilizar muchos de estos indicadores en la práctica, no está de más que conozca de manera general cuál es su finalidad. Ésta es una breve descripción de cada uno de ellos:

- **OF (*Overflow Flag*):** Indica que la última operación ha generado un resultado que no puede almacenarse en el destino correspondiente, es decir, que el destino se ha desbordado.
- **SF (*Sign Flag*):** Tras realizar cualquier operación aritmética, este bit indicará el signo del resultado.
- **ZF (*Zero Flag*):** Comunica que el resultado de la última operación efectuada ha sido cero.
- **AF (*Auxiliar*) *Carry Flag*:** Este indicador es útil al trabajar con aritmética BCD, ya que señala el acarreo del primer *nibble* al segundo dentro de un byte.
- **PF (*Parity Flag*):** Indica la paridad del resultado obtenido tras la última operación. La paridad puede ser par o impar, dependiendo del número de bits a 1 que haya en ese resultado.
- **CF (*Carry Flag*):** Este indicador comunica que se ha producido un acarreo durante una operación aritmética, por ejemplo al sumar dos números de 16 bits y obtener un resultado que necesita un bit adicional para ser almacenado.

- DF (*Dírection Flag*): Controla el sentido en que se recorren zonas de memoria con determinadas instrucciones de repetición, incrementando o reduciendo el contenido de los registros SI y DI.
- IF (*Interrupt Flag*): Determina si el microprocesador atenderá o no las interrupciones enmascarables. Es un tema que conocerá con detalle más adelante.
- TF (*Trap Flag*): Este bit es utilizado por los depuradores para facilitar la ejecución paso a paso de los programas.

Nota

En capítulos posteriores conocerá instrucciones cuyo funcionamiento se ve afectado por los bits del registro de estado, como las de salto condicional, así como instrucciones que modifican el estado de los indicadores de control y sistema.

Generación de direcciones

El microprocesador 8086 cuenta con registros de 16 bits, parte de los cuales actúan como punteros según se ha indicado en el punto previo. Es el caso de IP, SP, SI y DI, por ejemplo. El bus de direcciones de este microprocesador, sin embargo, es de 20 bits, lo cual le permite direccionar exactamente un megabyte de memoria. La pregunta es ¿cómo pueden generarse direcciones de 20 bits a partir de registros de 16 bits? La respuesta se encuentra en la unidad de interfaz con el bus, donde se produce la generación de direcciones combinando un registro de segmento con otro registro de 16 bits.

La lógica que hay tras la unidad que genera las direcciones se compone de un multiplicador y un sumador que operarían según el esquema mostrado en la figura 3.5.

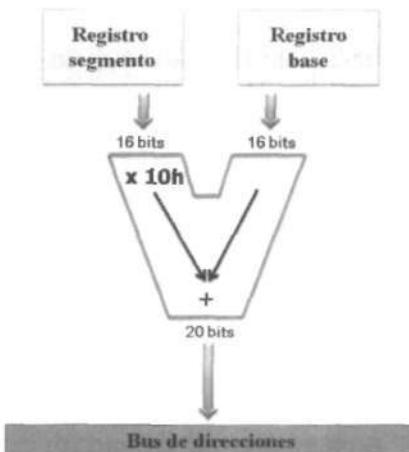


Figura 3.5. Esquema del generador de direcciones efectivas.

Los operandos de entrada son un registro de segmento y un registro base (como los antes citados IP, SP, SI o DI). El primero es multiplicado por 10h (16 decimal), lo cual equivale a desplazarlo cuatro bits a la izquierda. A continuación a ese valor se suma el registro base y se obtiene la dirección efectiva de 20 bits.

Posteriormente podrá aprender a utilizar este esquema, notado habitualmente como segmento: desplazamiento, para acceder desde sus propios programas a la memoria instalada en cualquier PC.

Patillaje del 8086

Acercándonos un poco más a los elementos físicos del microprocesador, aquellos que resulta más fácil identificar desde el exterior puesto que su arquitectura interna es microscópica, nos encontramos con los conectores que le permiten comunicarse con el mundo exterior, vitales para poder diseñar un sistema basado en microprocesadores. Esos conectores en realidad resultan demasiado pequeños, y están demasiado juntos, como para poder usarlos directamente, de ahí que el microprocesador se encapsule en una *pastilla* plástica con una serie de patillas a ambos lados, facilitando así la conexión.

El 8086, al igual que el 8085, el Z80, 8088 y muchos otros microprocesadores de la época, se empaquetaba en un DIP (*Dual In-line Package*) de 40 patillas, lo que hacía posible su inserción en los zócalos estándar de entonces. Los microprocesadores actuales se empaquetan en cápsulas con varios cientos de patillas dispuestas en forma de matriz, con un diseño considerablemente más complejo.

La figura 3.6 muestra las designaciones que se dan normalmente a las 40 patillas del 8086 en los dos modos de funcionamiento de este microprocesador. Las diferencias entre ambas se encuentran en las patillas 24 a 31. El encapsulado del 8088 es práctica idéntico, salvo por el hecho de que las patillas AD9 a AD15 pasa a denominarse A9 a A15.

Buses de direcciones y datos

pire ce 10€5*

El bus de-dfttrafc del 8086 tiene 20 bits, por lo que precisa de 20 terminales o patillas para su funcionamiento. El bus de datos precisa 16 bits (8 en el 8088) que, sumados a los anteriores, haría un total de 36 terminales. Si a éstos se suman los de alimentación y el de entrada de señal reloj se estarían ocupando los 40 conectores del empaquetado, no quedando patillas para ninguna otra función. A la vista de la figura 3.6, sin embargo, esto no es así y hay patillas para la gestión de interrupciones, controlar el acceso a memoria y dispositivos externos, etc.

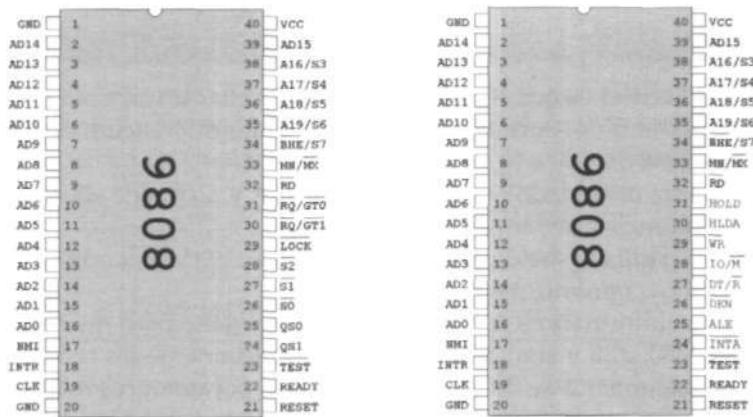


Figura 3.6. Esquema del patillaje del 8086.

Este problema no es nuevo y, de hecho, la solución aplicada proviene de microprocesadores más antiguos como el 8085, en el que bus de direcciones y de datos se encuentran multiplexados en el interior del microprocesador. Esto significa que dichos buses comparten un conjunto de terminales, en los cuales se alternan en el tiempo las señales correspondientes al bus de direcciones y de datos.

Los sistemas basados en un microprocesador de este tipo, por lo tanto, deben hacer uso de biestables externos que almacenen el contenido de esos terminales comunes, separando las direcciones de los datos. El terminal que indica si en un momento dato el contenido es una dirección o un dato es ALE (*Address Latch Enable*). Cuando esta señal se activa, el sistema debe tomar el contenido del bus de direcciones y almacenarlo en los biestables. De esta forma dispondrá de la dirección cuando, en el siguiente ciclo de bus, el microprocesador envíe o reciba por los mismos terminales un dato.

En el 8086 los terminales multiplexados son los comprendidos entre ADO y AD15, ya que el bus de datos externo (el del sistema) es de 16 bits. En el 8088, por el contrario, ese bus externo es de 8 bits, de ahí que los terminales multiplexados sean ADO a AD7.

Nota

El hecho de que el bus externo de los 8088 sea de 8 bits, en lugar de 16, se debe únicamente a temas económicos. A principios de los ochenta la mayoría de dispositivos de E/S contaban con un bus de datos de 8 bits, resultando mucho más barato diseñar sistemas con buses externos de 8 bits que empleando dispositivos más sofisticados y caros.

El hecho de que los buses estén multiplexados, o que el bus de datos tenga 8 ó 16 bits, no nos afectará como programadores ya que no conllevan diferencia alguna a la hora de escribir programas en ensamblador. Sí habrían de tenerse en cuenta, sin embargo, a la hora de diseñar un sistema basado en estos microprocesadores.

Modos de funcionamiento

Una de las decisiones que han de tomarse al diseñar un sistema hardware en torno al microprocesador 8086, puesto que implica notables diferencias, es el modo de operación: mínimo o máximo.

Este modo está controlado por el terminal MNMX y no puede alterarse una vez que el sistema esté en funcionamiento.

Por ello esta patilla se suelda directamente a GND o VCC, dependiendo de que se desee el modo máximo o mínimo, respectivamente.

Los sistemas informáticos construidos tomando como base microprocesadores de 8 bits, como el 8080, Z80 y similares, generaban directamente las señales de control que regían el funcionamiento de dispositivos externos, por entonces relativamente simples y que no requerían que dichas señales tuvieran una gran intensidad (en términos eléctricos) para funcionar correctamente.

El 8086 en modo mínimo se comporta de manera análoga, lo cual permite utilizar este microprocesador en sistemas cuyo diseño es sencillo.

La mayoría de los ordenadores personales, incluso aquellos construidos a principios de los ochenta, utilizan dispositivos externos que requieren una intensidad mayor que la que puede proporcionar directamente el microprocesador. Estos sistemas se diseñan de tal forma que el microprocesador se limita a indicar el estado actual a un circuito externo, un controlador de bus que, a la postre, será el que genere las señales por las que se regirán el resto de dispositivos. El 8086 en modo máximo opera así y, lógicamente, precisa el apoyo de ese circuito externo, un controlador de bus que originalmente era el IC Intel 8288.

Configurado en modo máximo, los terminales SO, SI y S2 del 8086 comunican al 8288 el estado que debe generar en el bus del sistema, dependiendo de la instrucción que se esté ejecutando.

La tabla 3.1 enumera los posibles estados así como la señal que se generaría en el bus del sistema.

Tabla 3.1. Señales de control de bus en modo máximo.

S2	S1	SO	Estado
0	0	0	Reconocimiento de interrupción.
0	0	1	Leer de puerto de E/S.
0	1	0	Escribir en puerto de E/S.
0	1	1	Condición de parada.
1	0	0	Obtener código de operación.
1	0	1	Leer de memoria.
1	1	0	Escribir en memoria.

Nota

La existencia de dos modos de funcionamiento es una característica propia del 8086, inexistente en microprocesadores posteriores como el 80286, 80386, etc. En éstos siempre se recurre a un circuito externo encargado de generar las señales de control del bus.

Gestión de interrupciones

Un aspecto fundamental en el diseño de cualquier sistema informático en las últimas décadas es el relativo a la gestión de las interrupciones, las señales que permiten a los dispositivos externos captar la atención del microprocesador para que éste realice un trabajo que no puede esperar. Las interrupciones son las que hacen posible que el puntero del ratón se desplace por la pantalla a pesar de que el ordenador esté realizando otra tarea, o de que una cierta combinación de teclas puede detener la ejecución de un programa.

El directo antecesor del 8086, el Intel 8085 de 8 bits, contaba con varios terminales que podían conectarse directamente a dispositivos externos, gestionando por sí mismo tres interrupciones hardware con distinta prioridad, una interrupción adicional de tipo genérico, una interrupción 110 enmascarable y también una interrupción por software.

Comparativamente hablando la capacidad del 8086 en este sentido es inferior a la que tenía su directo predecesor, ya que este microprocesador carece de las tres interrupciones hardware con niveles de prioridad que tenía el 8085 (denominadas RST7 . 5, RST6 . 5 y RST5 . 5). Conserva, sin embargo, las terminales INTR, NMI e INTA, que también tenía dicho microprocesador.

El hecho, no obstante, es que el mecanismo de gestión de interrupciones de los antiguos microprocesadores de 8 bits resultaba insuficiente en el diseño de sistemas más complejos, con mayor número de dispositivos y, en consecuencia, con la necesidad de más señales de interrupción y más niveles de prioridad. Por ello en el 8086 se optó por reducir el número de señales dedicadas a esta tarea, delegándola en un circuito externo especializado: el Intel 8259A. Este 1C recibe las solicitudes de interrupción de los dispositivos, las prioriza y se las hace llegar al microprocesador a través del terminal INTR. Cuando es posible atenderla se activa entonces la señal INTA, reconociendo la solicitud y procesándola.

Los sucesores del 8086

A pesar de que el 8086 es un microprocesador diseñado hace más de dos décadas, la compatibilidad hacia atrás que Intel han mantenido en todos sus sucesores consigue que su estudio en la actualidad siga siendo completamente útil, ya que la arquitectura

básica: banco de registros, conjunto de instrucciones, modos de direccionamiento, etc., sigue siendo la misma.

Obviamente los nuevos desarrollos de Intel en este tiempo han ido incorporando multitud de avances y novedades, pero de cara a su programación en ensamblador no son demasiados los aspectos importantes: mayor capacidad de direccionamiento, al ampliarse los buses de datos y direcciones; algunas instrucciones nuevas, como las conocidas MMX/SIMD, y también, en los últimos años, la posibilidad de ejecución en paralelo al incorporar en un microprocesador varios núcleos.

Los hitos más importantes en la evolución de la familia x86 tienen los nombres que se describen a continuación:

- **80286:** El microprocesador usado en los denominados IBM AT seguía teniendo un bus de datos de 16 bits, pero su bus de direcciones se amplió hasta los 24 bits, lo cual le permitía usar un máximo de 16 Mbytes de memoria, una cantidad considerable en 1982 que fue cuando se presentó. Fue el primer microprocesador de Intel en incorporar una unidad de gestión de memoria o MMU (*Memory Management Unit*). Al iniciarse se comporta como un 8086, aunque más rápido, operando en el que a partir de entonces se denominaría *modo real*. Para aprovechar sus características más avanzadas era necesario activar el llamado *modo protegido*.
- **80386:** Presentado en 1985, se trata de un microprocesador de 32 bits, lo cual supuso un avance enorme respecto a los anteriores y el despegue de aplicaciones mucho más potentes, especialmente sistemas operativos que podían aprovechar su capacidad de direccionamiento, de hasta 4 Gbytes, y las sofisticadas funciones para intercambio de procesos. La MMU del 80386 era mucho más avanzada que la de su predecesor, contemplando esquemas de protección de memoria y de memoria virtual basados en paginación. Al igual que el 80286, el 80386 se comporta como un 8086 muy rápido al iniciarse, operando en modo real. Además del modo protegido también incorporó el *modo virtual 8086*, en el que podía funcionar como varios 8086 independiente.
- **80486:** Manteniendo la misma arquitectura básica del 80386, este nuevo microprocesador se caracterizó por integrar en el mismo circuito integrado una unidad de coma flotante. Hasta dicho momento si se requería velocidad en la realización de cálculos en coma flotante, que normalmente se realizaban mediante software, era preciso añadir al sistema un coprocesador matemático: el 8087, 80287 u 80387. El diseño del 80486 estaba formado por 80386, un 80387, memoria caché interna y no incorporaba muchas más novedades.
- **Pentium:** En realidad habría que hablar de una familia completa de microprocesadores Pentium, incluyendo el diseño original de 1993, el posterior Pentium MMX con instrucciones multimedia, el Pentium Pro con un bus de direcciones de 36 bits que le permitía direccionar hasta 64 Gbytes de memoria, los Pentium II y Pentium III que incluyeron nuevas instrucciones SIMD bajo la denominación SSE y el Pentium 4 con HT (*HyperThreading*) capaz de ejecutar dos hilos en paralelo con un solo microprocesador.

- Core: Bajo esta denominación existe también una familia de microprocesadores, como los Core Solo, Core Dúo, Core 2 Dúo, Core 2 Quad, Core i7, etc. Es la nomenclatura que utiliza actualmente Intel para sus nuevos microprocesadores que, en general, se caracterizan por incorporar un nuevo modo de funcionamiento, llamado *Long*, que opera con un bus de datos de 64 bits y un bus de direcciones de 64 bits. Además todos ellos incluyen dos, cuatro u ocho núcleos dentro del mismo circuito integrado, mayores cantidades de memoria caché e innovaciones en el tratamiento de datos multimedia.

Como se apuntaba en el capítulo previo, la complejidad de los microprocesadores actuales se ha incrementado notablemente respecto a los primeros diseños, lo cual se evidencia rápidamente por el número de transistores que integran. Existen nuevas instrucciones y los modos de funcionamiento más potentes resultan más difíciles de configurar, por lo que esta tarea suele delegarse en los sistemas operativos modernos. Dicho esto, cabe recordar que incluso los últimos procesadores de Intel siguen comportándose, en el momento en que se inician, como un 8086, aunque miles de veces más rápido. Esto permite seguir utilizando el mismo modelo de programación en ensamblador, al tiempo que se aprovechan las características avanzadas que interesen.

Resumen

Al finalizar la lectura de este capítulo ya conoce de manera general la arquitectura de los microprocesadores de la familia x86, descendientes todos ellos del 8086 fabricado por Intel a finales de la década de los setenta. De esta forma, ya sabe qué registros podrá utilizar a la hora de escribir programas en ensamblador, la forma en que se generan las direcciones o cuál es el significado de cada uno de los bits del registro de estado.

A pesar de no resultar imprescindible para poder programar en ensamblador, en este capítulo también se ha descrito el conexionado del 8086 y el diseño de sistemas basados en este microprocesador, optando por uno de sus dos modos de funcionamiento o explicando cómo se efectúa la gestión de interrupciones. Para terminar, se han destacado las diferencias más importantes entre el 8086 y algunos de sus sucesores.

4

**Sistemas
basados en
microprocesadores
x86**

Los microprocesadores pueden encontrarse actualmente en multitud de dispositivos, sistemas hardware tan heterogéneos como los teléfonos móviles, televisores, consolas de juegos, hornos microondas, sistemas de alarma, automóviles y, por supuesto, ordenadores personales. Cada uno de estos sistemas tiene, como es obvio, un diseño completamente distinto que, en parte, se verá influenciado por el microprocesador que se haya escogido como base. En torno a la familia de microprocesadores x86 se han desarrollado muchos tipos de equipos pero, sin duda alguna, el más importante fue el IBM PC del que derivan los actuales ordenadores personales. Por ello el diseño de este sistema resulta especialmente interesante y su estudio, aunque no sea de manera profunda, un aspecto fundamental. Conocer la arquitectura del IBM PC, aunque sea superficialmente, nos ayudará a comprender qué ocurre cuando se usan ciertos servicios desde un programa en ensamblador, cuando se accede a una determinada zona de memoria, en el momento en que un dispositivo externo genera una interrupción, etc.

La finalidad de este capítulo es introducir la arquitectura general del IBM PC original, el más importante sistema desarrollado tomando como base el microprocesador 8086 y cuyo diseño, evolucionado a lo largo de los años, sigue siendo el núcleo de los actuales ordenadores personales.

Estructura de bloques

Partamos de una visión general de la arquitectura que tenían los primeros sistemas PC, representada en el esquema de la figura 4.1. En éste los buses de datos y direcciones aparecen unidos, ocultando la lógica de multiplexación que almacenaría la dirección,

por una parte, y los datos, por otra, en unos registros externos que son los que se conectarían a la memoria y resto de elementos. Desde una perspectiva lógica, no obstante, éste es un detalle que no interesa especialmente.

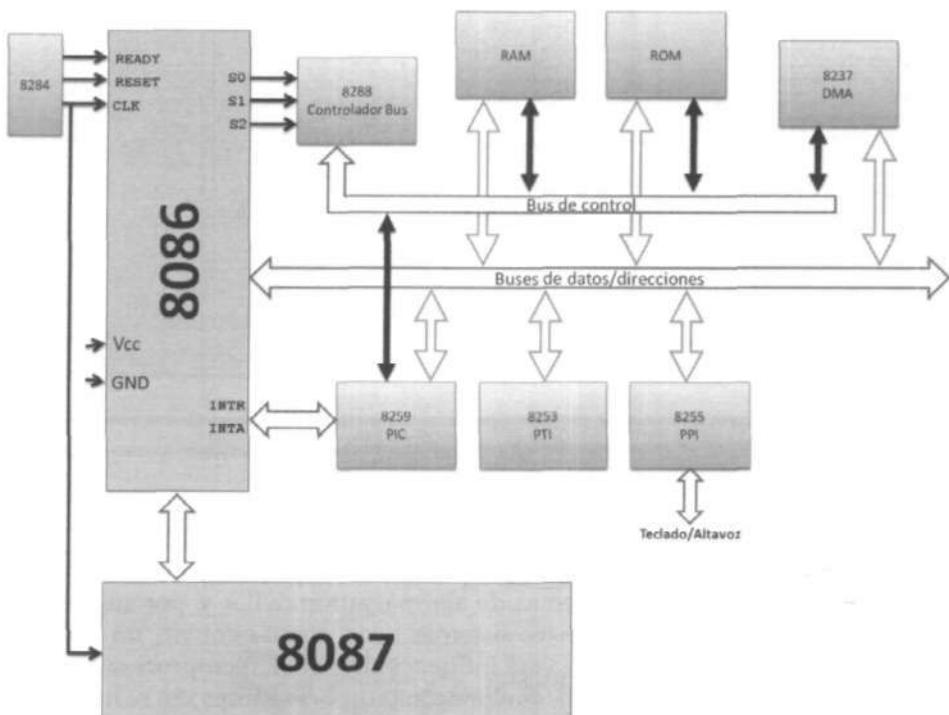


Figura 4.1. Diagrama de bloques de un sistema PC.

No a todos los dispositivos que aparecen en la figura 4.1 conectados al bus de datos/direcciones les llegan las señales del bus completo. Los circuitos 8255, 8259 y 8253, por ejemplo, son dispositivos con un bus de datos de 8 bits, utilizados en el diseño original del PC porque resultaban más baratos.

Parte de las señales del bus de direcciones actúan sobre una lógica de selección, implementada por un descodificador 3 a 8 que se encarga de activar el IC (a través de la señal CS) que corresponda según la dirección enviada por el microprocesador. Una vez activado, el circuito recibe/envía por los 8 bits bajos del bus de datos la información que proceda. La figura 4.2 muestra con algo más de detalle la lógica de selección que permite al 8086 comunicarse con los IC 8255, 8259, 8237 y 8253.

El 8087 es un elemento opcional y, de hecho, la mayoría de los PC carecían de este coprocesador matemático. Los sistemas, sin embargo, estaban diseñados para poder agregarlo. La comunicación con el microprocesador se efectúa a través de un bus local, entre ambos circuitos, empleando señales específicas como la del conector TEST.

Los buses de datos, direcciones y control se extienden a través de los buses de ampliación, ISA originalmente, EISA y PCI posteriormente, que permiten agregar al sistema

otros tipos de dispositivos, generalmente especializados. El más habitual es el adaptador de vídeo, mediante el que es posible conectar el sistema a una pantalla externa para mostrar texto y, por regla general, también gráficos.

Sin perder la visión general del sistema que nos ofrece la figura 4.1, en los apartados siguientes se detallan los distintos elementos que lo componen.

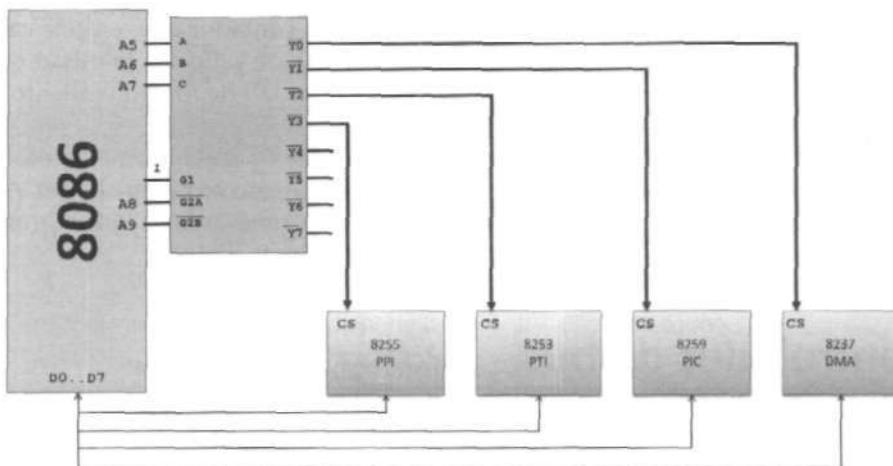


Figura 4.2. Lógica que permite al microprocesador seleccionar un dispositivo del sistema.

Generador de reloj - 8284

Los dispositivos que forman el sistema mostrado en la figura 4.1, incluyendo el propio microprocesador, han de operar de manera sincronizada. Esto implica que exista una señal de reloj compartida que marque los ciclos de funcionamiento y evite, por ejemplo, conflictos de acceso a los buses. En sistemas más antiguos, como los basados en el 8085, el propio microprocesador aceptaba la entrada de un oscilador, normalmente un cristal de cuarzo, y generaba la señal de reloj que se transmitía a los demás elementos. Se ocupaban de esta forma tres de los terminales del circuito. En el diseño del PC se optó por delegar la generación de la señal de reloj en un circuito externo: el 8284. En la figura 4.3 puede ver los conectores de este IC.

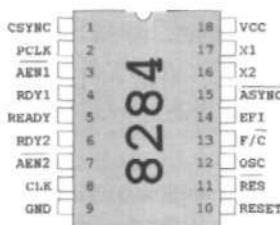


Figura 4.3. Diagrama de conexiones del 8284.

A los terminales XI y X2 se conectará el oscilador externo, con una frecuencia que debe ser el triple de aquella a la que se quiere que opere el microprocesador. En el PC original ese cristal oscila a una frecuencia de 14,31 Mhz. Un contador interno del 8284 se incrementa con cada oscilación y, cuando llega al valor 3, genera un pulso por el terminal **CLK**. Este se conecta a la entrada de reloj del 8086 que, como es fácil deducir, operará a $14,31 / 3 = 4,77$ Mhz.

El terminal PCLK actúa de manera similar, con un contador que, en este caso, genera un pulso cada 6 oscilaciones, es decir, la frecuencia de salida es la mitad que por la patilla **CLK**. Sus niveles TTL son los que rigen la velocidad de funcionamiento del resto de elementos del sistema.

Además de facilitar la señal de reloj, el 8284 también participa en el reinicio del sistema, facilitando una señal RESET sincronizada tal y como requiere el 8086. Asimismo facilita la señal READY sincronizada que se utiliza para generar los estados de espera que permiten al microprocesador operar con dispositivos más lentos.

Controlador de bus - 8288

En los PC el 8086 se configuraba para operar en modo máximo (ver capítulo previo), de forma que las señales de control del bus han de ser generadas externamente, a partir del estado del procesador que éste facilita mediante los terminales SO, SI y S2. El circuito encargado de descodificar ese estado, generando las señales de salida adecuadas, es normalmente un 8288 o derivado. La figura 4.4 muestra los conectores correspondientes a este integrado.

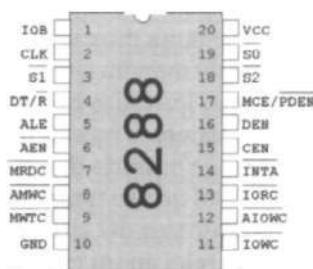


Figura 4.4. Diagrama de conexiones del 8288.

Desde una perspectiva lógica el 8288 se compone de cuatro bloques diferentes:

- **Descodificador de estado:** A partir de las tres señales de entrada S0, S1 y S2 determina el estado en que se encuentra el microprocesador, configurando el estado interno del propio 8288.
- **Lógica de control:** Los terminales **CLK**, AEN, CEN e IOB actúan como señales de entrada respecto al 8288, facilitando la operación síncrona con el microprocesador y determinando qué señales de control de salida deben activarse.

- Generador de comando: A partir del estado obtenido por el descodificador envía al bus el comando que debe ejecutarse, utilizando para ello los terminales indicados en la tabla 4.1.
- Generador de señales de control: Controla los terminales de salida DT/R, DEN, MCE/PDEN y ALE, con los que se controla el funcionamiento del bus de acuerdo con las señales de entrada y el comando generado.

Tabla 4.1. Señales de control de bus en modo máximo.

S2	S1	S0	Estado	Comando de salida
0	0	0	Reconocimiento de interrupción	INTA
0	0	1	Leer de puerto de E/S	IORC
0	1	0	Escribir en puerto de E/S	TOWC, AIOWC
0	1	1	Condición de parada	Ninguno.
1	0	0	Obtener código de operación	MRDC
1	0	1	Leer de memoria	MRDC
1	1	0	Escribir en memoria	MWTC, AMWC

Al igual que el generador de reloj 8284, el controlador de bus 8288 carece de elementos programables y, por ello, resulta interesante más a los diseñadores de sistemas que a los programadores. Son dos circuitos integrados cuyas señales de entrada y salida se configuran durante la fabricación del sistema y rigen su funcionamiento durante toda su vida, no siendo accesibles desde programas en ensamblador.

Reloj programable - 8253

En los ordenadores hay multitud de elementos que necesitan operar a una determinada velocidad, con independencia de cuál sea la frecuencia de funcionamiento del microprocesador o el resto del sistema. Las unidades de disco, el reloj de tiempo real o el sistema de refresco de memoria, por ejemplo, trabajan a un cierto ritmo que no puede fijarse directamente a partir de las señales de salida del 8284. Ésta es la razón de que en los PC se incluya un PTI/PIT (*Programmable Interval Timer*), un circuito que actúa como un reloj programable. Como se ve en la figura 4.5, este circuito se encapsula en un DIP de 24 terminales donde es fácil distinguir, aparte de un bus de datos, uno de direcciones y algunas señales de control, tres grupos formados por una señal CLK, una GATE y una OUT. Esos tres grupos corresponden a los tres contadores internos con que cuenta el PTI.

Nota

El conocido como PTI recibe la denominación 8253 o bien 8254, dependiendo de las frecuencias máximas de reloj que acepte como entrada. También puede encontrarse bajo denominaciones como 8254-2, 8254-5 y similares.

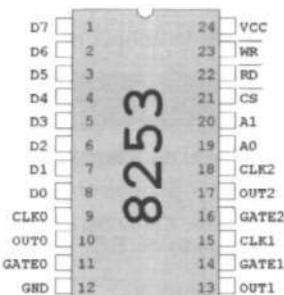


Figura 4.5. Diagrama de conexiones del PTI.

Las señales CS, AO y Al son de entrada y proceden del bus de direcciones del sistema. Con la primera se activa el circuito, de lo contrario no atendería a los comandos enviados por el microprocesador a través de las demás líneas. Una vez que el circuito está activo, las señales RD y WR determinan si va a efectuarse una lectura o una escritura. En ese momento las señales AO y Al pueden tomar los valores indicados en la tabla 4.2.

Tabla 4.2. Selección de contador en el PTI.

A0	A1	Se escribe/lee
0	0	El contador 0.
0	1	El contador 1.
1	0	El contador 2.
1	1	El registro de control.

Cada uno de los contadores con que cuenta el PTI es de 16 bits y acepta una señal de reloj independiente que marcaría su frecuencia de funcionamiento. En los PC la señal de las tres entradas de reloj es la misma: 1,1925 Mhz. Dependiendo del valor que se escriba en un contador, se obtendrá una frecuencia de salida equivalente a dividir la de entrada entre ese valor. El valor del contador 0, por ejemplo, tiene asignado por defecto el valor FFFFh, de forma que genera un pulso cada $1192500 \text{ Mhz} / 65535 = 18,196 \text{ Hz}$. Ésta es la frecuencia con que se actualiza el reloj del sistema, aproximadamente unas 18,2 veces por segundo. La configuración del PTI se establece a través del registro de control, en el que únicamente se permite la escritura. La estructura de este registro es la indicada en la figura 4.6, estando compuesto de cuatro campos. Su finalidad es la siguiente:

- **SEL:** Los dos bits de mayor peso se usan para indicar al PTI el contador en el que se va a escribir o del que se va a leer. Sus posibles valores son: 0 0 - Contador 0, 0 1 - Contador 1 y 1 0 - Contador 2.
- **R/W:** Determinan la forma en que se va a leer/escribir el contador indicado. Es posible efectuar una lectura/escritura directa de los ocho bits de menor peso o de mayor peso, indicar que va a leerse/escribirse el contador completo o bien realizar una lectura a través de un *latch* intermedio, de forma que el contador no vea afectado su funcionamiento.
- **Modo:** Fija el modo en que operará el contador y el tipo de señal que se generará por la correspondiente patilla OUT. Hay seis modos de funcionamiento diferentes, gracias a ellos el PTI puede actuar como un contador de eventos, como un generador de pulsos a intervalos regulares, produciendo como salida una onda cuadrada de la frecuencia establecida, como un disparador único rearmable, etc.
- **BCD:** Con este bit se establece si el contador utilizará el sistema binario o bien contará en BCD. En el primer caso el valor máximo que puede establecerse es 65535 (FFFFh), mientras que en el segundo será 9999.



Figura 4.6. Registro de control del PTI.

Cada uno de los tres contadores puede operar en modo distinto y, obviamente, tener un valor de partida también diferente. En los sistemas PC los contadores del PTI se utilizan de la siguiente forma:

- **Contador 0:** Al ponerse en marcha, la ROM del PC configura este contador para que genere un pulso cada 55 milisegundos (1 seg. / 18,196 Hz). La salida OUT0 está conectada a la señal IRQ0 del PIC (descrito más adelante). Esta interrupción periódica puede utilizarse desde programas propios y, además, es la encargada de mantener actualizado el reloj del sistema. En el capítulo 27, dedicado a la implementación de programas residentes, tiene un ejemplo de cómo aprovechar la salida de este contador a través de la interrupción que genera.
- **Contador 1:** Este contador también lo programa el sistema al ponerse en marcha, operando conjuntamente con el 8237 (DMA) para refrescar periódicamente la memoria RAM dinámica. Su configuración dependerá de las necesidades que tenga la memoria instalada en el sistema.
- **Contador 2:** La salida OUT2, correspondiente a este contador, está conectada al altavoz con el que suelen contar todos los PC. Si se programa el contador para que genere una onda cuadrada de una determinada frecuencia, el altavoz producirá un sonido de dicha frecuencia. Tomando una tabla de las frecuencias que corresponden a cada nota, y controlando el tiempo que se mantiene activa la señal, es relativamente fácil generar música mediante esta técnica.

Obviamente la programación de los dos primeros contadores no debe ser alterada por ningún programa, ya que esto podría afectar al funcionamiento normal del sistema. El tercero, por el contrario, puede ser utilizado sin ningún problema por los programas. Para determinar cómo pueden estos acceder al PTI, a fin de poder programar el contador mencionado, hay que tomar como referencia la figura 4.2, correspondiente a la lógica de selección de dispositivos. En la figura 4.7 se ha detallado exclusivamente el direccionamiento del PTI, prescindiendo de los elementos que no interesan en este momento.

La señal CS del PTI es activada por la salida Y2 del descodificador 3 a 8. Esa salida actúa cuando por las entradas A, B y C del descodificador se recibe un 010. Dado que esas entradas se encuentran conectadas a las líneas A5, A6 y A7 del bus de direcciones, esto significa que la dirección puesta por el microprocesador deberá seguir el patrón indicado en la tabla 4.3 en su byte menos significativo.

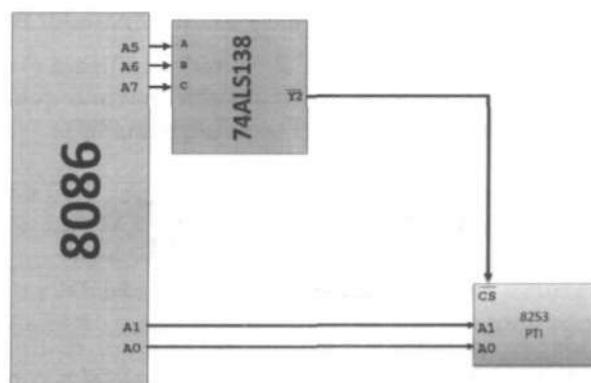


Figura 4.7. Direccionamiento del PTI en un PC.

Tabla 4.3. Patrón de direccionamiento del PTI.

A7	A6	A5	A4	A3	A2	A1	A0	Hex	Corresponde a
0	1	0	X	X	X	0	0	40h	Contador 0.
0	1	0	X	X	X	0	1	41h	Contador 1.
0	1	0	X	X	X	1	0	42h	Contador 2.
0	1	0	X	X	X	1	1	43h	Registro de control.

La secuencia a seguir, a la vista de estos datos, sería:

- Enviar al puerto de E/S 43h, correspondiente al registro de control, la configuración que se quiere establecer para el contador 2 del PTI.
- A continuación se enviaría el divisor de la frecuencia maestra, de 16 bits, al puerto 42h (Contador 2), colocando primero el byte menos significativo y después el más significativo.

El altavoz quedaría sonando a la frecuencia establecida hasta que el propio programa lo desconectase.

Interfaz programable de periféricos - 8255

Este integrado, conocido genéricamente como PPI (*Programmable Peripheral Interface*), permite al microprocesador comunicarse con dispositivos externos a través de 24 líneas en paralelo que pueden configurarse de distintas formas. El encapsulado es el habitual de 40 patillas y la distribución de las señales es la mostrada en la figura 4.8.

Desde un punto de vista lógico, 24 de las 40 líneas se unen en tres grupos de 8 a los que se denomina puertos A, B y C. Son las patillas PA_n, PB_n y PC_n. En el caso del puerto C, los ocho bits pueden tratarse como dos partes independientes de 4 bits cada una. El 8255 contempla tres modos de funcionamiento diferentes que, de manera simplificada, serían los descritos a continuación:

- Modo 0: Los tres puertos se utilizan como entrada o salida de datos, sin ningún control sobre la comunicación con los dispositivos. Es el modo más simple de funcionamiento del PPI.
- Modo 1: Los puertos A y B se pueden usar como entrada o salida, repartiéndose el puerto C en 4 bits para control de la comunicación por el puerto A y otros 4 bits para el control por el puerto B.
- Modo 2: Este modo permite que el puerto A funcione de manera bidireccional, es decir, como entrada y salida simultáneamente, apoyándose para ello en ciertos bits del puerto C que se usan para controlar el diálogo con los dispositivos externos. El puerto B se puede configurar para operar en modo 0 ó 1, con independencia del puerto A.

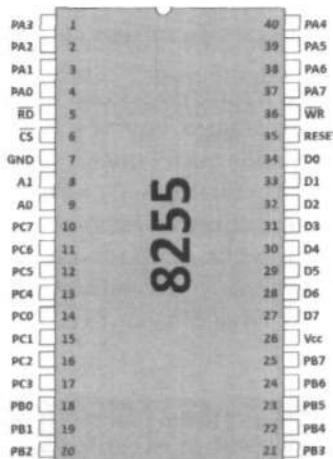


Figura 4.8. Diagrama de conexiones del PPI.

Estas 24 patillas se conectan desde el 8255 hacia los dispositivos externos. En un sistema completo podrían utilizarse para conectar al ordenador una impresora, un teclado u otros periféricos de comunicación en paralelo.

En el diseño del PC original los tres puertos del PPI están configurados para operar en modo 0, es decir, de manera unidireccional, habiéndose establecido el puerto B como de salida (solamente escritura) y los otros dos como entrada (solamente lectura). A través del puerto A se lee tanto el teclado como ciertos parámetros de la configuración del sistema, fijados mediante microinterruptores, dependiendo de un bit del puerto B. El puerto C se usa para obtener parámetros adicionales de esa configuración, mientras que con el puerto B es posible modificar parte de ellos. En la figura 4.9 se muestra esquemáticamente la forma en que se agruparían las señales.

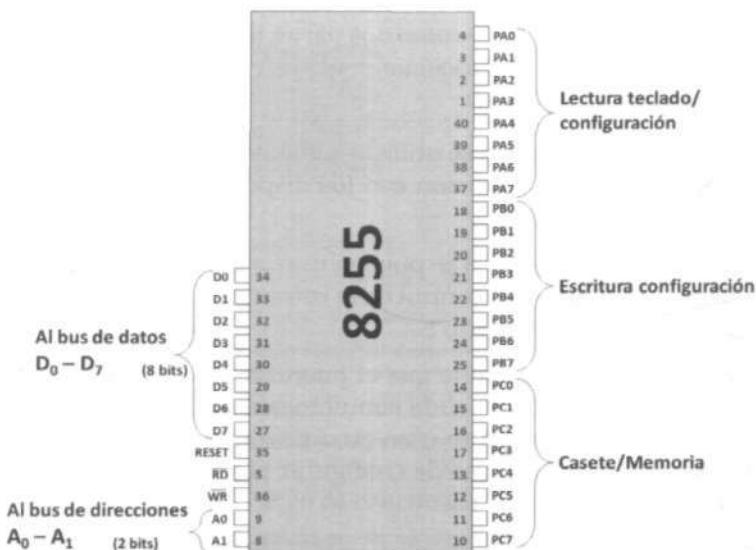


Figura 4.9. Distribución de señales del PPI en un sistema PC.

De estas patillas las del lado izquierdo son las que permiten al microprocesador comunicarse con el PPI para programarlo, enviar y recibir datos. Por una parte están los pines AO y Al, dos bits mediante los que el microprocesador selecciona el puerto donde se escribirá o del que se leerá: 0 0 - Puerto A, 0 1 - Puerto B, 1 0 - Puerto C y 1 1 - Registro de control. Hecha la selección, se utilizarán los pines DO a D7 para enviar o recuperar los 8 bits de datos o configuración, según el estado de las patillas RD y WR que son las que establecen si se va a leer o a escribir. Para establecer la configuración del PPI se recurre a un registro de control interno, cuya estructura es la indicada en la figura 4.10.



Figura 4.10. Registro de control del PPI.

La finalidad de cada bit o conjunto de bits es la siguiente:

- **CFG:** Determina el tipo de configuración que va a efectuarse. Puesto a 1 permite establecer el modo y dirección de comunicación de cada puerto, con el patrón de bits de la figura 4.10. Si se pone a 0 facilita la configuración individual de cada uno de los bits del puerto C.
- **Modo A:** Estos dos bits establecen el modo de funcionamiento para el puerto A: 00 - Modo 0, 01 - Modo 1 y IX - Modo 2.
- **PA:** Configura el puerto A para: 0 - Salida, 1 - Entrada.
- **PCH:** Éste configura los cuatro bits de mayor peso del puerto C (PC7-PC4) para: 0 - Salida, 1 - Entrada.
- **MB:** Este bit establece el modo de funcionamiento para el puerto B más la parte baja del puerto C: 0 - Modo 0,1 - Modo 1.
- **PB:** Configura el puerto B para: 0 - Salida, 1 - Entrada.
- **PCL:** Este configura los cuatro bits de menor peso del puerto C (PC3-PC0) para: 0 - Salida, 1 - Entrada.

Con toda esta información, la secuencia habitual de trabajo con el PPI desde el punto de vista del programador sería la siguiente:

1. Configuración del PPI mediante el registro de control, estableciendo el modo de funcionamiento y sentido de la comunicación. Para ello se pondría un 11 en las patillas A0-A1 y se enviaría por D0-D7 el byte con la configuración, según los bits descritos antes.
2. Selección mediante A0 -A1 del registro del que se va a leer o donde se escribirá.
3. Envío o lectura por D0-D7 del byte de datos.

En realidad para el programador la selección de registro mediante las patillas A0 -A1 se hará automáticamente según la dirección colocada en el bus de direcciones y la configuración del sistema. La señal CS del PPI se encuentra conectada (véase la figura 4.2) a la salida Y3 del decodificador, por lo que la entrada A deberá estar a 0 y las otras dos a 1, es decir, en A5 habrá un 0 y en A6 y A7 un 1, dando lugar al patrón de dirección amien-to indicado en la tabla 4.4.

Tabla 4.4. Patrón de direccionamiento del PPI.

A7	A6	A5	A4	A3	A2	A1	A0	Hex	Corresponde a
1	1	0	X	X	X	0	0	60h	Puerto A.
1	1	0	X	X	X	0	1	61h	Puerto B.
1	1	0	X	X	X	1	0	62h	Puerto C.
1	1	0	X	X	X	1	1	63h	Registro de control.

Nota

Puesto que en los PC la configuración del PPI se encuentra ya establecida, según se indicó antes, en la práctica el registro de control no se utilizará. Los programas podrán escribir en el puerto B y leer del puerto A y c. En capítulos posteriores podrá ver algunos ejemplos de estas operaciones.

Controlador programable de interrupciones - 8259

Como se explicaba en el capítulo previo, el 8086 no incluye la circuitería necesaria para atender a múltiples interrupciones con prioridad, contando únicamente con una señal de solicitud de interrupción enmascarable: INTR, y una señal de salida que notifica el reconocimiento: INTA. Por esta razón el diseño original del PC incluye un PIC (*Programmable Interrupt Controller*), un circuito integrado especializado en esta tarea.

A diferencia de otros circuitos que forman el sistema, el PIC no precisa una señal de reloj para funcionar.

Como se aprecia en la figura 4.11, no existe una señal CLK puesto que el estado interno del circuito no cambia a lo largo del tiempo, sino únicamente cuando recibe una interrupción o el microprocesador atiende una interrupción.

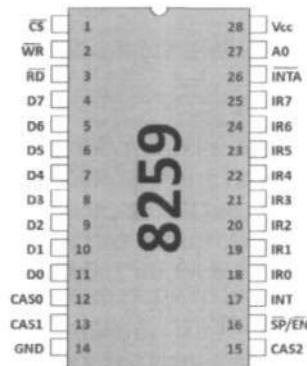


Figura 4.11. Diagrama de conexiones del PIC.

Las señales IRO a IR7 son de entrada y son las que permiten al PIC aceptar solicitudes de interrupción de hasta 8 dispositivos, unas señales que en el sistema pasan a denominarse IRQ0-IRQ7. En el PC original los dispositivos conectados a cada una de esas líneas son los indicados en la tabla 4.5.

En caso de que el diseño del sistema requiera atender más de ocho señales de interrupción, el PIC puede ser configurado como maestro para controlar, a través de las señales

CASO, CASI y CAS2, hasta ocho circuitos PIC adicionales configurados como esclavos. De esta forma podrían atenderse un máximo de 64 señales de interrupción.

Internamente el PIC cuenta con tres registros que le permiten saber qué líneas han solicitado interrupción, cuáles están siendo atendidas y cuáles están enmascaradas. Dichos registros son de 8 bits y se denominan, según se aprecia en la figura 4.12, IRR (*Interrupt Request Register*), ISR (*In-Service Register*) e IMR (*Interrupt Mask Register*).

Tabla 4.5. Dispositivos conectados a las líneas de entrada del PIC.

Señal	Dispositivo
IRQ0	Salida del contador 0 del PTI.
IRQ1	Teclado.
IRQ2	Reservado.
IRQ3	COM2.
IRQ4	COM1.
IRQ5	Controlador de disco duro.
IRQ6	Controlador de disquetes.
IRQ7	Salida paralelo para impresora.

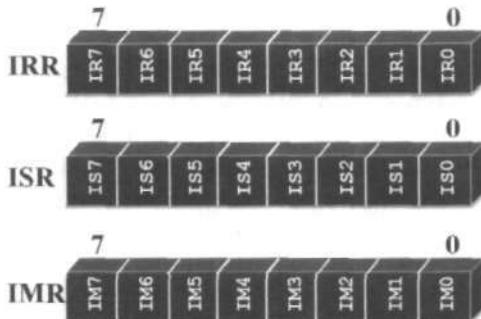


Figura 4.12. Los tres registros internos con que cuenta el PIC.

La secuencia de funcionamiento del PIC, una vez conectado en el sistema, es ésta:

1. Se recibe una solicitud de interrupción a través de una de las líneas IRn.
2. El PIC activa el bit correspondiente del registro IRR, por lo que la solicitud queda registrada aún cuando el nivel de la línea por la que ha entrado cambie. El PIC puede de estar forma registrar varias solicitudes de interrupción procedentes de distintos dispositivos.
3. Simultáneamente al registro de solicitud en el registro IRR, el PIC activa la salida INT para avisar al 8086 que tiene pendiente de atender una o más interrupciones.

4. Cuando el 8086 activa la señal INTA, conectada a la entrada INTA del PIC, éste toma del registro IRR la señal correspondiente a la interrupción con mayor prioridad y activa el bit correspondiente en el ISR. Queda en dicho registro marcada la interrupción que va a atenderse.
5. Al recibir una segunda señal INTA procedente del 8086, el PIC envía por el bus de datos el número de vector de interrupción que corresponde a la interrupción marcada en el registro ISR.
6. El 8086 toma del bus de datos el vector de interrupción y ejecuta la rutina de servicio correspondiente. Ésta termina enviando al PIC el comando EOI, con la que notifica que se ha concluido la interrupción. En ese momento se desactiva el bit del ISR correspondiente a la interrupción ya atendida.

Este proceso puede repetirse de manera inmediata si en el IRR hay más de un bit activo. Siempre se asigna mayor prioridad a las señales de interrupción con índice menor, es decir, la señal IRO tiene la mayor prioridad y la señal IR7 la menor.

Nota

El microprocesador atenderá las solicitudes de interrupción que le lleguen por la señal INT siempre y cuando en el registro de estado, descrito en el capítulo previo, no se haya desactivado el bit que las enmascara globalmente.

El PIC cuenta como señal de direccionamiento únicamente con la patilla A0, procedente del bus de direcciones. A la vista de la lógica de selección ya analizada en otros puntos previos (véase la figura 4.2), los dos puertos a utilizar serían el 20h y el 21h. A través de éstos se procede a la inicialización del PIC, un trabajo que efectúa el propio sistema al ponerse en marcha, y es posible utilizar tres comandos distintos. Las operaciones básicas son la lectura de los registros IRR, ISR e IMR, la escritura del registro IMR para enmascarar/desenmascarar líneas de interrupción y la finalización de una interrupción ya atendida.

Controlador de acceso directo a memoria - 8237

De los circuitos integrados de apoyo con los que se diseña un sistema basado en el microprocesador 8086, un PC, posiblemente el controlador DMA (*Direct Memory Access*) sea el más complejo. Al igual que el PPI o el propio 8086, y como se ve en la figura 4.13, cuenta con 40 señales. Parte de ellas, como puede deducirse de la denominación, operan conjuntamente formando cuatro grupos que corresponden a los cuatro canales de acceso directo a memoria con que cuenta el 8237.



Figura 4.13. Diagrama de conexiones del controlador de DMA.

A menos que se escriba software de muy bajo nivel, la programación del 8237 no resulta especialmente interesante. Su finalidad es facilitar la transferencia de información desde y hacia la memoria sin necesidad de pasar por el microprocesador, dejando por tanto que éste realice mientras tanto otras tareas. Sus canales están ocupados en el diseño del PC con los siguientes elementos:

- **Canal 0:** Lleva a cabo el refresco de memoria cada vez que el PTI señala que es necesario.
- **Canal 1:** Lo ocupa el controlador de comunicaciones serie.
- **Canal 2:** Usado por las unidades de disquete.
- **Canal 3:** Transferencias desde y hacia el disco duro.

Internamente el 8237 cuenta con 18 registros de 16 bits y 9 más de 8 bits, mediante los cuales se programan las transferencias a través de los cuatro canales. Cada uno de ellos tiene asociada una dirección de 16 bits y un contador que permite transferir un máximo de 64 Kbytes de una vez.

También existe la posibilidad de configurar varios 8237 en cascada, como ocurría con el 8259, estableciendo uno como maestro y otros como esclavos.

Nota

Puede encontrar información detallada sobre este circuito integrado, todos los descritos en los puntos previos y otros muchos en <http://www.datasheetcatalog.curia.com>. Deberá conocer la denominación del IC para realizar una búsqueda directa o bien examinar las hojas de especificaciones disponibles por fabricante.

El único elemento importante del diseño esbozado en la figura 4.1 que queda por cubrir es el acceso a la memoria, controlador por las señales 10/M, RD y WR del bus de control que establece cuándo se va a efectuar una lectura o una escritura, momento en el que los buses de direcciones y datos participan de la operación. En capítulos posteriores se detallará la distribución de esa memoria y aprenderá a direccionarla correctamente.

Resumen

Los microprocesadores son la base de cualquier sistema informático actual, pero para funcionar precisan una serie de circuitos de apoyo y vías de comunicación que se establecen durante la fase de diseño del hardware. Este capítulo le ha servido, fundamentalmente, para conocer cuáles fueron las decisiones tomadas al diseñar el PC original.

En el capítulo se han descrito los aspectos más importantes de los circuitos integrados que, conjuntamente con el microprocesador, conforman los pilares del diseño del PC: el generador de reloj, el gestor de interrupciones, el controlador de bus, el controlador de DMA y la interfaz programable de periféricos o PPL. En capítulos posteriores accederá a algunos de estos circuitos, por ejemplo para leer una pulsación del teclado o dar por atendida una interrupción.

5

Modos de direcccionamiento

Cuando se escribe un programa en lenguaje ensamblador, un importante número de las sentencias utilizadas tienen la finalidad de transferir datos, ya sea dentro del propio microprocesador, desde la memoria al microprocesador o viceversa. No en vano una de las instrucciones del 8086 más utilizadas es MOV, implicada en prácticamente todas las transferencias de información controladas por el microprocesador. Las operaciones que más variantes ofrecen son las de transferencia entre el microprocesador y la memoria, ya que la dirección de origen o destino puede obtenerse de múltiples formas distintas, como tendrá ocasión de ver en este capítulo. En conjunto, a las diferentes formas de trasladar los datos que ofrece un microprocesador se las denomina modos de direccionamiento. La finalidad de este breve capítulo es enumerar esos modos proporcionando sencillos ejemplos que faciliten su comprensión. En capítulos posteriores, a medida que vaya escribiendo programas, tendrá ocasión de usarlos en la práctica.

Direccionamiento por registro

Se habla de direccionamiento por registro, sin más calificativos, cuando en la operación la información procede de un registro y el destino es otro registro. Dicho de otra forma: la transferencia tiene lugar en el interior del microprocesador, sin emplear en ningún momento los buses de datos o direcciones.

Un ejemplo de direccionamiento por registro sería el siguiente:

```
MOV AH, AL
```

En este caso se traslada el contenido del registro AL al registro AH, lo que también equivale a copiar los 8 bits de menor peso del registro AX en los 8 bits de mayor peso. Obviamente el contenido que tuviese anteriormente AH se pierde de forma definitiva.

Una de las operaciones más habituales en las que se ve implicado este tipo de direccionamiento es la inicialización de registros de segmento, para establecer por ejemplo el área de datos de un programa. Asumiendo que en el registro AX se tuviese la dirección del segmento de datos, una de las primeras instrucciones del programa sería la siguiente:

```
MOV DS, AX
```

Al usar este tipo de direccionamiento deben tenerse en cuenta las reglas siguientes:

- El registro de destino y el de origen ser del mismo tamaño, por lo que no sería válida una instrucción como **MOV AH, DX**, ya que AH es un registro de 8 bits y DX de 16. Esta regla es aplicable en general a cualquier transferencia de información.
- Determinados registros no pueden aparecer como destino en un direccionamiento por registro. Los casos más significativos son CS e IP, siendo inválidas instrucciones como **MOV IP, BX**, por poner un ejemplo.
- No es posible usar como origen y destino un registro de segmento, siendo incorrectas instrucciones del tipo **MOV DS, CS**. La alternativa sería un código como:

```
MOV AX, CS  
MOV DS, AX
```

Otra alternativa sería enviar el origen a la pila y recogerla de ésta en el registro de destino, pero ya no se estaría utilizando direccionamiento por registro que es el método más rápido.

Nota

MOV no es la única instrucción en la que se utiliza direccionamiento por registro, hay otras como **XCHG**, **CMP** o **TEST** que también emplean este tipo de direccionamiento. En el capítulo siguiente conocerá todas estas instrucciones.

Direccionamiento inmediato

Este tipo de direccionamiento se caracteriza porque uno de los operandos, concretamente el de origen, forma parte del código del programa, situándose en la memoria a continuación del código de operación correspondiente a la instrucción que va a utilizarlo. Por ello se dice que es de acceso o direccionamiento inmediato. Uno de los casos más típicos es la inicialización de un registro con algún valor, por ejemplo:

```
MOV AH, 20h
```

El ensamblador, al traducir esta instrucción, generaría el siguiente patrón de bits:

1011 0 100 0010 0000

Los 4 bits de mayor peso: 1011 indican al microprocesador que la instrucción a ejecutar es **MOV** y que el direccionamiento empleado es inmediato, siendo el destino un registro. El siguiente bit, contando de izquierda a derecha, establece que se trata de un registro de 8 bits. Los tres bits siguientes: 0 0 0, indican que el registro de destino es AH. El byte siguiente es el valor 20h que se asignará a ese registro.

En cuanto se recupera esta sentencia de la memoria, el primer byte, la unidad que se encarga de descodificar la instrucción detecta que lleva asociado un operando que está en la siguiente posición de memoria y la recupera. De esta forma cuando la unidad de ejecución va a realizar su trabajo, ejecutando la instrucción, se sabe cuál es el operando de destino (AH en este caso) y el de origen se encuentra también en el microprocesador, en un registro temporal.

El operando de destino, el que recibirá el dato inmediato, puede ser prácticamente cualquier registro. La excepción son los registros de segmento, cuya inicialización debe efectuarse mediante direccionamiento por registro, por ejemplo:

```
MOV AX, 0B800h  
MOV ES, AX
```

Las posibles combinaciones de los cuatro bits de menor peso del código de operación, que indican tamaño y registro, limitan en cierta forma los posibles destinos. La tabla 5.1 resume esas posibles combinaciones.

Tabla 5.1. Posibles destinos para el código de operación 1011 W RRR.

W	RRR	Registro
0	000	AL
0	001	CL
0	010	DL
0	011	BL
0	100	AH
0	101	CH
0	110	DH
0	111	BH
1	000	AX
1	001	CX
1	010	DX
1	011	BX

W	RRR	Registro
1	100	SP
1	101	BP
1	110	SI
1	111	DI

Nota

Cuando las instrucciones forman parte de un programa de 32 o de 64 bits, en lugar de 16 bits, los códigos de operación son algo más complejos a fin de permitir un mayor número de combinaciones ya que hay más registros posibles de destino.

La regla a respetar al usar este tipo de direccionamiento, ya mencionada en el punto anterior, es que el valor inmediato debe tener un tamaño idéntico al del registro de destino. En cuanto al dato en sí, puede expresarse en cualquiera de las bases que conoció en un capítulo previo, el dedicado a la representación de la información. También existe la posibilidad de entregar uno o dos caracteres entre comillas simples, según se necesiten 8 ó 16 bits, dejando que el ensamblador introduzca en el destino el código de los caracteres.

Direccionamiento directo

En este tercer modo de direccionamiento uno de los operandos, puede ser el destino o bien el origen, es una dirección de memoria. Esta, al igual que ocurre con el dato inmediato del modo de direccionamiento previo, es recuperada durante la descodificación de la instrucción, por lo que el procesador la tiene a su disposición cuando va a ejecutar la operación. En ese momento la coloca en el bus de direcciones e indica a la memoria si tiene que leerla, y poner su contenido en el bus de datos, o bien si debe recuperar la información enviada por el bus de datos y escribirla en esa dirección.

Para diferenciar un dato de 16 bits, como puede ser 12 80h, de la dirección de memoria 12 80h, ya que el ensamblador no puede saber ante una instrucción como **MOV AX, 12 80h** si el segundo operando es un dato inmediato o bien una dirección de memoria, existen dos posibilidades:

- Colocar la dirección entre corchetes, de manera que la instrucción quedaría como **MOVAX, [1280h]**.
- Declarar la dirección como una etiqueta, una capacidad que ofrecen todos los ensambladores, de forma que la instrucción sería del tipo **MOV AX, Valor**, siendo **Valor** la etiqueta previamente definida.

Al acceder a memoria la dirección efectiva (real) se obtiene a partir de un registro de segmento y la dirección indicada en la instrucción, que sería el desplazamiento. Cada instrucción asume un registro de segmento por defecto, registro que puede ser cambiando por otro indicado explícitamente en la propia instrucción. En capítulos posteriores conocerá con exactitud la forma en que se generan estas direcciones y la función que tienen los registros de segmento, tanto en modo real (16 bits) como en modo protegido (32 bits).

El número de bytes transferidos desde o hacia la memoria, cuando se utiliza este direccionamiento directo o cualquier otro que opere sobre memoria, será deducido por el propio ensamblador siempre que sea posible, observando el tamaño del otro operando. En una instrucción como `MOV AX, [1280h]`, por ejemplo, se recuperaría el contenido de la posición de memoria indicada y la siguiente, 16 bits en total, ya que el registro AX es de 16 bits. En caso de que no sea posible deducir el tamaño, éste puede especificar explícitamente con las palabras `BYTE`, `WORD`, `DWORD`, etc., que reconocen la mayoría de los ensambladores.

Pireccionamiento indirecto

Se trata de un direccionamiento similar al anterior, dado que de lo que se trata es de transferir un dato desde o hacia la memoria, pero en este caso la dirección no se facilita en la propia sentencia, de ahí que cuando el microprocesador vaya a ejecutar la instrucción no pueda realizar directamente esa transferencia. En su lugar la dirección está almacenada en un registro que, desde el punto de vista del programa, actuaría como un puntero. Por ejemplo:

`MOV AL, [RX]`

Como en el direccionamiento directo, el tamaño de uno de los operandos marca el número de bytes a transferir. La dirección de memoria puede ser el destino o el origen, también como en el direccionamiento directo.

Para ejecutar una instrucción como la anterior el microprocesador tiene que obtener el código de operación y descodificarlo. En ese momento sabe que uno de los operandos, en el caso anterior el registro BX, contiene una dirección que debe colocar en el bus de direcciones, llevando a cabo la lectura o escritura.

Una variante de este tipo de direccionamiento es la que incluye un desplazamiento sobre la dirección base contenida en el registro. La forma de expresar el desplazamiento sería la siguiente:

`MOV AL, [BX+4]`

Asumiendo que el registro BX contuviese la dirección donde está alojada una cierta estructura de datos que el programa necesita utilizar, el desplazamiento adicional sería útil para acceder a campos de dicha estructura.

Nota

Los registros que pueden usarse como base (puntero) en este tipo de direccionamiento son BX, BP, SI y DI.

Direccionamiento indexado

Es el modo de direccionamiento más sofisticado con que cuenta el 8086, permitiendo el acceso a memoria calculando la dirección a partir de tres componentes. Es también un direccionamiento indirecto, si bien la dirección no se obtiene sin más de un registro sino que es calculada a partir de dos registros y, opcionalmente, un desplazamiento. Podría decirse, por tanto, que el direccionamiento indexado es un caso particular de direccionamiento indirecto.

Al usar el modo de direccionamiento indexado uno de los registros, normalmente BX o BP, contendrá una dirección base a la que se sumará el contenido de un registro índice, DI o SI.

El contenido del registro índice suele ir cambiando, incrementándose o reduciéndose, en el interior de un bucle. Opcionalmente puede sumarse también un desplazamiento, dando lugar a instrucciones como la siguiente:

```
MOV AL, [BX+SI+4]
```

Suponiendo que un programa tuviese alojada en memoria una lista de estructuras de datos, el registro BX contendría la dirección base donde comienza la lista, es decir, la dirección del primer campo de la primera estructura de esa lista. El registro SI serviría para direccionar cada uno de los elementos de la lista, de forma que se le sumaría o restaría, según se quiera avanzar o retroceder, un número que sería el tamaño de cada elemento. Finalmente el desplazamiento sería el que establecería el campo al que se quiere acceder dentro de ese elemento.

Nota

1

Para la mayoría de los ensambladores es indistinto el orden en que se faciliten los operandos en un direccionamiento indexado, por lo que $[BX+SI+4]$ podría también escribirse como $[SI+BX+4]$, $[SI+4+BX]$, $[4+BX+SI]$, etc. También es usual escribir cada parte en una pareja de corchetes independiente, por ejemplo: $[BX][SI]+4$.

Registros de segmento por defecto

Las direcciones obtenidas al utilizar el direccionamiento indirecto e indexado, sumando el puntero base, el índice y el desplazamiento, según los casos, son siempre direcciones relativas a un cierto segmento. De donde proceda la dirección de segmento (o el selector en modo protegido) es algo que dependerá en parte de los registros utilizados como operandos.

La tabla 5.2 recoge las posibles combinaciones de tipo de direccionamiento, registros que aparecen como operandos y el registro de segmento que se usaría por defecto, salvo que en la propia instrucción se indique lo contrario.

Tabla 5.2. Registros de segmento asignados por defecto.

Direccionamiento	Operando	Ejemplo	Registro de segmento asignado
Directo	Dirección	MOV AX, [1200h]	DS
Indirecto simple	BX, ST, DI	MOV AX, [BX]	DS
Indirecto con desplazamiento	BX, SI, DI	MOV AX, [BX+nn]	DS
Indirecto con desplazamiento	BP	MOV AX, [BP+nn]	SS
Indexado	BX	MOV AX, [BX+DI+nn]	DS
Indexado	BP	MOV BP, [BP+SI+nn]	SS

Conocer el registro de segmento asignado por defecto es importante, ya que de ello dependerá que una instrucción opere con la información correcta o, por el contrario, use datos que no son útiles. Siempre es posible preceder una dirección con un prefijo de segmento, como se muestra en la instrucción siguiente:

`MOV AL, ES:[BX+SI]`

En este caso, dados los registros que se utilizan en el direccionamiento, si no se utiliza el prefijo ES : la suma de BX+SI se aplicaría como desplazamiento sobre el registro de segmento DS a fin de obtener la dirección efectiva. Al aparecer el prefijo, sin embargo, se usará el segmento indicado por ES que, lógicamente, habría sido inicializado previamente con un valor adecuado.

En el direccionamiento por registro y el direccionamiento inmediato no existe un registro de segmento por defecto, ya que no es necesario acceder a memoria para obtener los operandos.

Modos de direccionamiento del 80386

Los modos de direccionamiento descritos en los puntos previos están disponibles en toda la familia de microprocesadores x86, partiendo desde el 8086. Existen modos más complejos introducidos en el 80386 y miembros posteriores de esta familia. En su mayor parte son modos de direccionamiento indirecto a los que se agregan operandos adicionales.

Un ejemplo de estos modos de direccionamiento es el direccionamiento indexado escalado que apareció con el 80386, en el cual es posible aplicar un factor multiplicador al registro que actúa como índice. Por ejemplo:

```
MOV AL, [EBX+ESI*3+4]
```

En este caso el registro de 32 bits EBX contiene la dirección base, a la que se suma el contenido de 32 bits de ES I multiplicado por 3. Finalmente se suma el desplazamiento: 4, obteniendo la dirección en la que se encuentra el dato de 8 bits que se llevará a AL.

Una vez que conocemos los modos de direccionamiento que podemos usar a la hora de transferir información, solamente nos resta tener una visión general sobre el juego de instrucciones con que cuenta el 8086, descrito en el capítulo siguiente, para comenzar a escribir nuestros primeros programas en ensamblador.

Resumen

Los microprocesadores cuentan con una limitada capacidad para almacenamiento de datos, de ahí que sea fundamental contar con una memoria suficiente y, de manera complementaria, con métodos que facilitan el acceso a la misma: los modos de direccionamiento.

El 8086 dispone de los modos habituales en muchos microprocesadores: inmediato, directo e indirecto, por ejemplo, conjuntamente con otros más sofisticados, como el modo indexado. Todos los microprocesadores derivados del 8086 disponen de esos modos, a los cuales han ido agregando otros más complejos. A partir del 80386, según se ha explicado, existe también un modo de direccionamiento indexado con escalado.

6

**Conjunto
de instrucciones**

El 8086 es un microprocesador CISC (*Complex Instruction Set Computer*), de ahí que cuente con un conjunto de instrucciones relativamente complejo, especialmente si hablamos de los últimos integrantes de la familia, a partir de la presentación de los Pentium. No tiene demasiado sentido detallar la finalidad, posibilidades y funcionamiento de cada una de ellas, instrucción por instrucción, salvo para contar con un material de referencia fácil de consultar en cualquier momento.

Al redactar este libro el objetivo que se ha perseguido ha sido el de conseguir un contenido con un enfoque principalmente didáctico, de ahí que las explicaciones y detalles sobre las instrucciones se encuentren en aquellos capítulos donde va a abordarse su utilización en la práctica, con ejercicios demostrativos de su funcionamiento.

Por ello el propósito de este capítulo no es más que el de recoger en una serie de tablas el conjunto de instrucciones con que cuenta el 8086, a fin de que el lector pueda contar con ese material de referencia que le permita consultar, de manera rápida, qué instrucción es la que debe utilizar para una tarea concreta o saber qué bits del registro de estado se verán afectados por su ejecución.

Las instrucciones se han agrupado en varias categorías: aritméticas, lógicas, de control, etc. La primera columna de cada tabla contiene el nombre de una instrucción, la segunda establece brevemente su finalidad y la tercera indica los bits del registro de estado que pueden verse afectados al ejecutarla. Debe tener en cuenta que sin leer el resto del libro esta información no le será muy útil, incluso es posible que algunas indicaciones no llegue a comprenderlas.

Una vez que esté habituado a programar en ensamblador, sin embargo, estas tablas serán un interesante material de apoyo.

instrucciones aritméticas

Mediante las instrucciones aritméticas con que cuenta el 8086 es posible efectuar sumas, restas, productos y divisiones usando como operandos números enteros, con o sin signo y en formato binario o BCD. El trabajo con BCD requiere algunos ajustes antes o después de las operaciones.

Para operar en coma flotante hay que recurrir al coprocesador matemático, actualmente integrado en todos los microprocesadores de la familia x86 y cuyas instrucciones conocerá en un capítulo posterior.

Tabla 6.1. Instrucciones aritméticas.

Instrucción	Finalidad	Indicadores afectados
AAA	Ajuste ASCII del registro AL tras una operación de suma	AF, CF
AAD	Ajuste ASCII del registro AX antes de una operación de división	SF, ZF, PF
AAM	Ajuste ASCII del registro AX tras una operación de multiplicación	SF, ZF, PF
AAS	Ajuste ASCII del registro AL tras una operación de resta	AF, CF
ADC	Suma con acarreo	OF, SF, ZF, AF, PF, CF
ADD	Suma sin acarreo	OF, SF, ZF, AF, PF, CF
DAA	Ajuste decimal del registro AL tras una operación de suma	SF, ZF, AF, PF, CF
DAS	Ajuste decimal del registro AL tras una operación de resta	SF, ZF, AF, PF, CF
DEC	Decremento	OF, SF, ZF, AF, PF
DIV	División	-
IDIV	División con signo	-
IMUL	Producto con signo	OF, CF
INC	Incremento	OF, SF, ZF, AF, PF
MUL	Producto	OF, CF
NEG	Complemento a 2 de un operando	CF
SBB	Resta con acarreo	OF, SF, ZF, AF, PF, CF
SUB	Resta sin acarreo	OF, SF, ZF, AF, PF, CF

Instrucciones lógicas y de rotación/traslación

Mediante las instrucciones de este grupo es posible realizar operaciones al nivel de bits, activándolos, desactivándolos, comprobándolos, desplazándolos, etc.

También hay instrucciones, como CMP y TEST, que se limitan a modificar los bits del registro de estado simulando una operación que, finalmente, no modifica el operando de destino.

Tabla 6.2. Instrucciones lógicas.

Instrucción	Finalidad	Indicadores afectados
AND	Efectuar la operación lógica Y entre dos operandos	SF, ZF, PF, OF=0, CF=0
CMP	Comparar dos operandos	OF, SF, ZF, AF, PF, CF
NOT	Invertir el estado de todos los bits del operando	-
OR	Efectuar la operación lógica O entre dos operandos	SF, ZF, PF, OF=0, CF=0
RCL	Rotación a la izquierda a través del acarreo	OF, CF
RCR	Rotación a la derecha a través del acarreo	OF, CF
ROL	Rotación a la izquierda	OF, CF
ROR	Rotación a la derecha	OF, CF
SAL	Desplazamiento a la izquierda	SF, ZF, AF, PF, CF
SAR	Desplazamiento aritmético a la derecha	SF, ZF, AF, PF, CF
SHL	Igual a SAL	SF, ZF, AF, PF, CF
SHR	Desplazamiento a la derecha	SF, ZF, AF, PF, CF
TEST	Como AND pero sin modificar el operando de destino	SF, ZF, AF, PF, OF=0, CF=0
XOR	Efectuar la operación lógica O exclusivo entre dos operandos	SF, ZF, PF, OF=0, CF=0

Instrucciones de conversión

Las tres instrucciones de este grupo realizan operaciones de conversión de datos, ampliando su tamaño o devolviendo un valor de una tabla de búsqueda.

Tabla 6.3. Instrucciones de conversión.

Instrucción	Finalidad	Indicadores afectados
CBW	Convertir byte (8 bits) en palabra (16 bits)	-
CWD	Convertir palabra (16 bits) en doble palabra (32 bits)	-
XLAT	Sustituye el contenido de AL a partir de una tabla de búsqueda	-

Instrucciones de cadena

Mediante las instrucciones de cadena se facilita el procesamiento de secuencias (cadenas) de bytes, palabras o dobles palabras, elemento a elemento o bien con repetición automáticamente mientras se cumpla una cierta condición.

En la mayor parte de las operaciones el operando origen se obtiene mediante direccionamiento indirecto, a través de DS : [ST], mientras que el de destino es el apuntado por ES:[DI].

Tabla 6.4. Instrucciones de cadena.

Instrucción	Finalidad	Indicadores afectados
CMPS	Comparar operandos de origen y destino	OF, SF, ZF, AF, PF, CF
LODS	Transfiere al acumulador un elemento de la cadena	-
MOVS	Transferir un elemento del origen al destino	-
REP	Automatizar la repetición de una operación de cadena	ZF
SCAS	Buscar un elemento en una cadena	OF, SF, ZF, AF, PF, CF
STOS	Transfiere a un elemento de la cadena el contenido del acumulador	-

Instrucciones de transferencia de datos

Se encuentran en esta categoría todas aquellas instrucciones que facilitan el acceso a datos alojados en memoria, para leerlos o escribirlos, y que no han sido introducidas en otros grupos, como ocurre con algunas de las que operan con cadenas de bytes.

Tabla 6.5. Instrucciones de transferencia de datos.

Instrucción	Finalidad	Indicadores afectados
LDS	Carga puntero en DS y registro de destino	-
LEA	Carga dirección efectiva en registro de destino	-
LES	Carga puntero en ES y registro de destino	-
MOV	Transfiere el origen al destino	-
XCHG	Intercambia el contenido de los dos operandos	-

Instrucciones de control de flujo

Como la mayoría de los microprocesadores, el 8086 dispone de instrucciones que facilitan la transferencia de control de un punto del programa a otro, modificando el contenido del puntero de instrucción (registro IP). Por una parte están las instrucciones de salto, condicionales o incondicionales, y por otra aquellas que antes de modificar IP preservan su contenido en la pila, facilitando el retorno. Son las instrucciones de llamada a subrutinas y, obviamente, las complementarias que se encargan de volver.

Tabla 6.6. Instrucciones de control de flujo.

Instrucción	Finalidad	Indicadores afectados
CALL	Transferir el control a una subrutina	-
INT	Llamar a una rutina de atención a interrupción	-
IRET	Devolver el control desde una rutina de atención a interrupción	-
Jxx	Transferir el control a una dirección según registro de estado	-
JMP	Transferir incondicionalmente el control a una dirección	-
LOOP	Transferir el control a una dirección según el valor de CX	-
RET	Devolver el control desde una subrutina	-

Las transferencias de control de forma condicional, dependiendo de ciertos bits del registro de estado, forman un grupo en sí mismas dado el gran número de sinónimos existentes. Los sinónimos son instrucciones con distinto nombre pero que tienen el mismo código de operación, en realidad hay una única instrucción con varios nombres. La idea es que el nombre de la instrucción sugiera la comprobación que quiere efectuarse,

pudiendo existir comprobaciones diferentes que se apoyen en los mismos bits del registro de estado.

En la tabla 6.7 se han resumido todas las instrucciones de salto condicional, indicando allí donde es necesario si en la comparación previa entre los dos operandos se ha de tener en cuenta el signo o no. Observe la segunda columna y apreciará que muchas de las instrucciones se basan en la comprobación de los mismos bits del registro de estado.

Tabla 6.7. Instrucciones de transferencia de control condicional.

Instrucción	Bits de estado a comprobar	Descripción
JA	CF=0, ZF=0	Saltar si el primer operando es mayor que el segundo (sin signo).
JAE	CF=0	Saltar si el primer operando es mayor o igual al segundo (sin signo).
JB	CF=1	Saltar si el primer operando es menor que el segundo (sin signo).
JBE	CF=1 o ZF=1	Saltar si el primer operando es menor o igual al segundo (sin signo).
JC	CF=1	Saltar si hay acarreo.
JE	ZF=1	Saltar si los dos operandos son iguales (sin signo).
JG	SF=OF o ZF=0	Saltar si el primer operando es mayor que el segundo (con signo).
JGE	SF=OF	Saltar si el primer operando es mayor o igual al segundo (con signo).
JL	SF<>OF	Saltar si el primer operando es menor que el segundo (con signo).
JLE	SF<>OF o ZF=1	Saltar si el primer operando es menor o igual al segundo (con signo).
JNA	CF=1 o ZF=1	Saltar si el primer operando no es mayor que el segundo (sin signo).
JNAE	CF=1	Saltar si el primer operando no es mayor o igual al segundo (sin signo).
JNB	CF=0	Saltar si el primer operando no es menor que el segundo (sin signo).
JNBE	CF=0, ZF=0	Saltar si el primer operando no es mayor o igual que el segundo (sin signo).
JNC	CF=0	Saltar si no hay acarreo.
JNE	ZF=0	Saltar si los dos operandos no son iguales (sin signo).

Instrucción	Bits de estado a comprobar	Descripción
JNG	SF<>OF o ZF=1	Saltar si el primer operando no es mayor que el segundo (con signo).
JNGE	SF<>OF	Saltar si el primer operando no es mayor ni igual al segundo (con signo).
JNL	SF=OF	Saltar si el primer operando no es menor que el segundo (con signo).
JNLE	SF=OF o ZF=0	Saltar si el primer operando no es menor ni igual al segundo (con signo).
JNO	OF=0	Saltar si no se ha producido desbordamiento.
JNP/JPO	PF=0	Saltar si la paridad del resultado es impar.
JNS	SF=0	Saltar si el resultado no es negativo.
JNZ	ZF=0	Saltar si el último resultado no es cero.
JO	OF=1	Saltar si se ha producido desbordamiento.
JP/JPE	PF=1	Saltar si la paridad del resultado es par.
JS	SF=1	Saltar si el resultado es negativo.
JZ	ZF=1	Saltar si el último resultado es cero.

Instrucciones de entrada/salida

El objetivo de este grupo de instrucciones es hacer posible la comunicación con dispositivos externos, tales como pueden ser los circuitos integrados PTI, PPI o bien PIC descritos en el capítulo previo, enviando y recibiendo datos a través de los puertos de E/S adecuados.

Tabla 6.8. Instrucciones de entrada/salida.

Instrucción	Finalidad	Indicadores afectados
IN	Lee un byte o una palabra del puerto indicado	-
INS	Lee un elemento del puerto indicado y lo transfiere a la cadena (puede repetirse automáticamente)	-
OUT	Envia un byte o una palabra al puerto indicado	-
OUTS	Envia un elemento de la cadena al puerto indicado (puede repetirse automáticamente)	-

Instrucciones de control

Son muchas las instrucciones, como puede comprobar examinando las tablas de este capítulo, que modifican uno o más bits del registro de estado del microprocesador. Asimismo, también son numerosas las instrucciones cuyo funcionamiento se ve alterado por el estado de dichos bits. No solamente las de salto condicional, sino también otras como las de trabajo con cadenas o algunas aritméticas. Por último están las instrucciones de control, recogidas en la tabla 6.9, que manipulan directamente varios de los indicadores del registro de estado.

Tabla 6.9. Instrucciones de control.

Instrucción	Finalidad	Indicadores afectados
CLC	Poner a cero el indicador de acarreo	CF=0
CLD	Poner a cero el indicador de dirección	DF=0
CLI	Poner a cero el indicador de interrupción	IF=0
CMC	Complementar el indicador de acarreo	CF=NOT CF
STC	Poner a uno el indicador de acarreo	CF=1
STD	Poner a uno el indicador de dirección	DF=1
STI	Poner a uno el indicador de interrupción	IF=1

Otras instrucciones

En esta última categoría se reúnen el resto de instrucciones del 8Ü86, aquellas cuya finalidad no se ajusta a ninguno de los grupos de puntos previos. Es una decena de instrucciones con finalidades heterogéneas.

Tabla 6.10. Otras instrucciones.

Instrucción	Finalidad	Indicadores afectados
HLT	Detiene el procesador	-
LAHF	Transfiere parte del registro de estado al registro AH	-
LOCK	Facilitar la sincronización entre varios microprocesadores	-
NOP	No operación	-
POP	Recupera un dato de la pila y actualiza SP	-

Instrucción	Finalidad	Indicadores afectados
POPF	Recupera de la pila el registro de estado y actualiza SP	Todos
PUSH	Introduce un dato en la pila y actualiza SP	-
PUSHF	Introduce el registro de estado en la pila y actualiza SP	-
SAHF	Transfiere AH a ciertos bits del registro de estado	SF, ZF, AF, PF, CF
WAIT	Esperar al coprocesador matemático	-

Resumen

Las tablas de instrucciones de este capítulo le ofrecen una referencia rápida del conjunto de instrucciones con que cuenta el 8086, un material que le permitirá, una vez que conozca todas esas instrucciones, conocer de manera rápida cuál es su finalidad y cómo se verán afectados los bits del registro de estado.

Será en capítulos posteriores, a medida que escriba programas en lenguaje ensamblador, cuando aprenda a usar la mayor parte de estas instrucciones. En ese momento este material de referencia le resultará de mayor utilidad.

7

Herramientas necesarias

Para poder programar en ensamblador precisará algunas herramientas básicas, como un editor para introducir el código, un ensamblador y un enlazador. Aparte también puede necesitar módulos con declaraciones de macros, estructuras de datos y funciones, utilidades para generar archivos de recursos, etc. Todo ello, lógicamente, para el sistema operativo concreto sobre el que se pretende programar.

Antes siquiera de que conozca la estructura general de un programa en ensamblador, algo que trataremos en el próximo capítulo, es necesario que obtenga, instale y se familiarice con las herramientas que va a utilizar continuamente para seguir los ejemplos del libro.

Algunas de esas herramientas las encontrará en el CD-ROM que acompaña al libro, mientras que otras puede obtenerlas a través de Internet.

El objetivo de este capítulo es enumerar las herramientas y utilidades principales que se utilizarán en capítulos posteriores, con el objetivo de no tener que explicar su uso en los capítulos siguientes y, así, poder en ellos centrarnos en aspectos concretos de programación y no de usuario.

Editores

^ ^

La primera pieza necesaria para poder crear un programa es, lógicamente, un editor que nos permita introducir el código, guardarlo en un archivo, recuperarlo y, en general, efectuar todas las tareas habituales de edición. Podemos usar virtualmente cualquier

editor de texto sin formato, desde el EDIT de DOS, el Bloc de notas de Windows o vi de Unix hasta los editores integrados en entornos de desarrollo como Visual Studio, Eclipse y similares.

Utilizando un editor simple, como los citados EDIT o vi, cada vez que efectuemos una modificación en el código deberemos guardar el archivo, recurriendo a continuación a la línea de comandos para efectuar el proceso de ensamblado y enlace. Existen editores específicos, podríamos denominarlos entornos integrados de desarrollo o IDE, que pueden ahorrarnos trabajo al contar con opciones para efectuar el ensamblado, enlace e, incluso, la ejecución sin necesidad de abandonar la edición de código.

En la figura 7.1 puede ver el editor ASM Edit funcionando en DOS. Se trata de un editor con múltiples posibilidades y que puede ser usado con diferentes ensambladores. La figura 7.2 corresponde al editor ASMCrunch, una aplicación Windows específica para trabajar con el ensamblador NASM.

Se puede observar a la derecha las distintas paletas de botones que sirven como ayuda a la codificación.

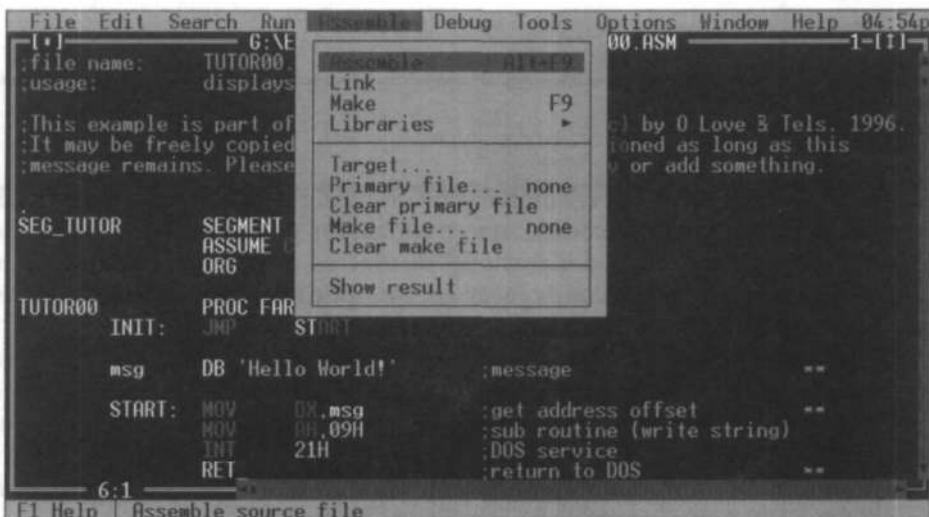


Figura 7.1. Un editor sobre DOS con el menú de opciones de ensamblado abierto.

Como se aprecia en la figura 7.1, algunos de estos editores cuentan incluso con la característica de diferenciación sintáctica del código mediante colores, lo cual contribuye a facilitar la identificación rápida de los distintos elementos.

A parte de estos editores con posibilidades de ensamblado, enlace y ejecución, también es posible encontrar verdaderos entornos de desarrollo que cuentan con herramientas como asistentes, generadores de código, ayuda en línea de funciones e interrupciones, etc. Para comenzar, no obstante, lo mejor es usar un editor sencillo y acostumbrarse a dar todos los pasos manualmente. Después, cuando ya se haya adquirido una cierta familiaridad con la mecánica de trabajo, es posible automatizar ciertas tareas sirviéndonos de uno de estos entornos.



Figura 7.2. Un editor para Windows desde el que podemos ensamblar y ejecutar.

DOS

El número de editores y entornos de desarrollo para trabajar con ensamblador en DOS es bastante grande. En el CD-ROM le facilitamos uno de los más fáciles de usar, conocido como Assembler Editor. Lo encontrará en Herramientas\ DOS \AssemblerEditor. Tan sólo tiene que copiar los tres archivos, en total 100 kbytes, a un directorio de su sistema y ya podrá usarlo. Véase la figura 7.3.

Este editor está preparado para trabajar con Turbo Assembler (TASM) y Microsoft Macro Assembler (MASM), tan sólo hay que modificar el archivo de configuración ae.cfg para establecer el camino y nombre de ensamblador y enlazador, facilitando también las opciones que interesen. Desde el propio editor es posible configurar algunos de estos parámetros.

Si en lugar de TASM o MASM va a utilizar el Netwide Assembler (NASM), una opción muy buena es el entorno NASM-TDE. Como se aprecia en la figura 7.4, es posible editar múltiples archivos simultáneamente, editando y ejecutando desde el propio editor. Puede encontrar esta herramienta en http://uk.geocities.com/rob_anderton. Es de distribución libre, por lo que no le costará nada adquirirlo, al igual que el propio NASM. Encontrará la última versión disponible en el momento de escribir esto en la carpeta Herramientas\ DOS \NASMEdit del CD-ROM que acompaña al libro.

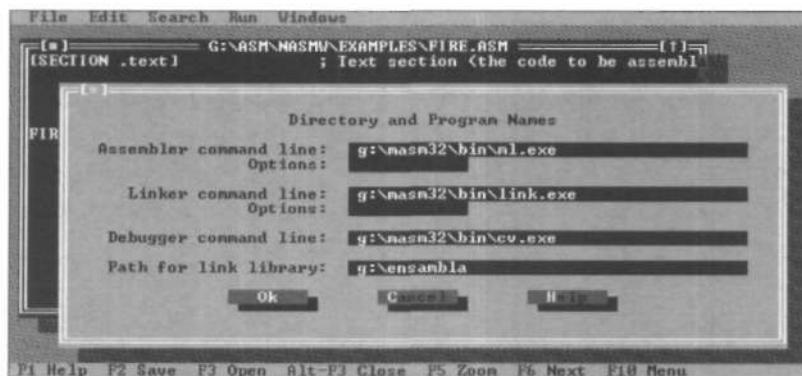


Figura 7.3. Aspecto de Assembler Editor.



Figura 7.4. Aspecto del entorno NASM-IDE.

En dicha Web encontrará también el NasmEdit, un entorno similar al NASM-IDE pero desarrollado para la plataforma Java 2. Esto significa que es posible utilizarlo no sólo en DOS, como el NASM-IDE, sino en cualquier sistema operativo en el que exista un entorno de ejecución Java 2, lo cual es actualmente cierto para Windows, Linux, Solaris y Mac.

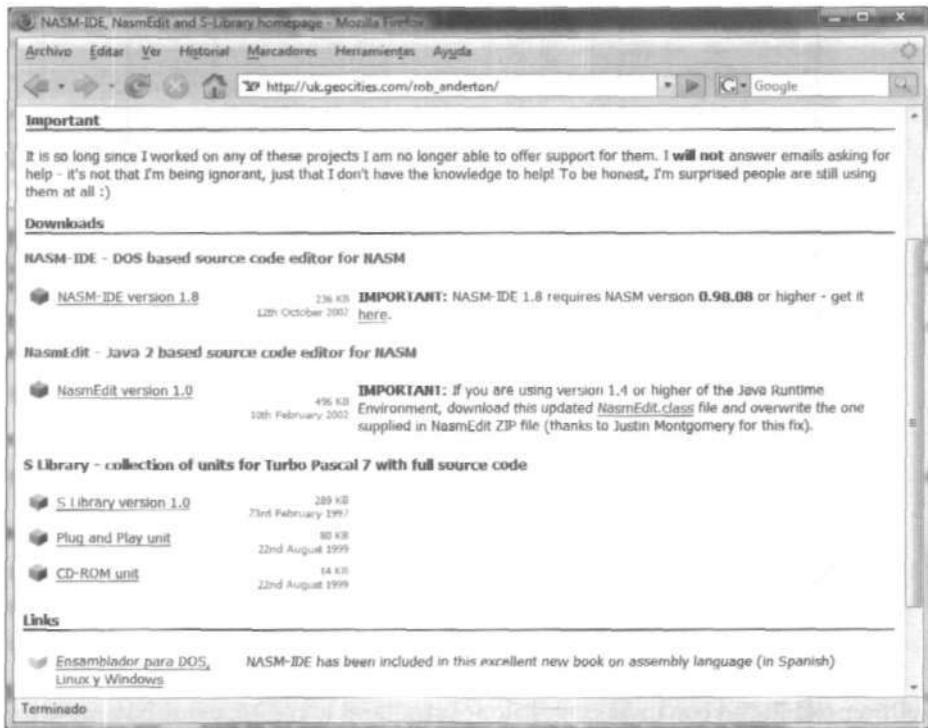


Figura 7.5. Sitio Web de NasmEdit y NASM-IDE.

Windows

Ya que desde Windows es posible ejecutar aplicaciones DOS, cualquiera de los editores citados en el punto anterior podría ser usado para programar en ensamblador desde Windows. Lógicamente, serían precisos algunos cambios de configuración, mediante parámetros al ensamblador y enlazador, para que la aplicación generada fuese Windows y no DOS. Existen algunos editores y entornos de trabajo en ensamblador específicos para Windows, si bien los que a priori parecen mejores se encuentran aún en fase de desarrollo o bien son proyectos abandonados por sus autores. En la figura 7.6 puede ver el Visual Assembler IDE (VASM), un entorno que, en su versión final, pretendía permitir la creación visual de los archivos de recursos y desde el cual es posible editar nuestros archivos y ensamblarlos. Está diseñado para trabajar específicamente con MASM.

Nota

Puede encontrar la última versión de VASM en la carpeta Herramientas\Windows\VASM. Ejecute el programa Setup.exe para instalarlo en su sistema. Tenga en cuenta, no obstante, que se trata de una herramienta cuyo desarrollo se ha abandonado.

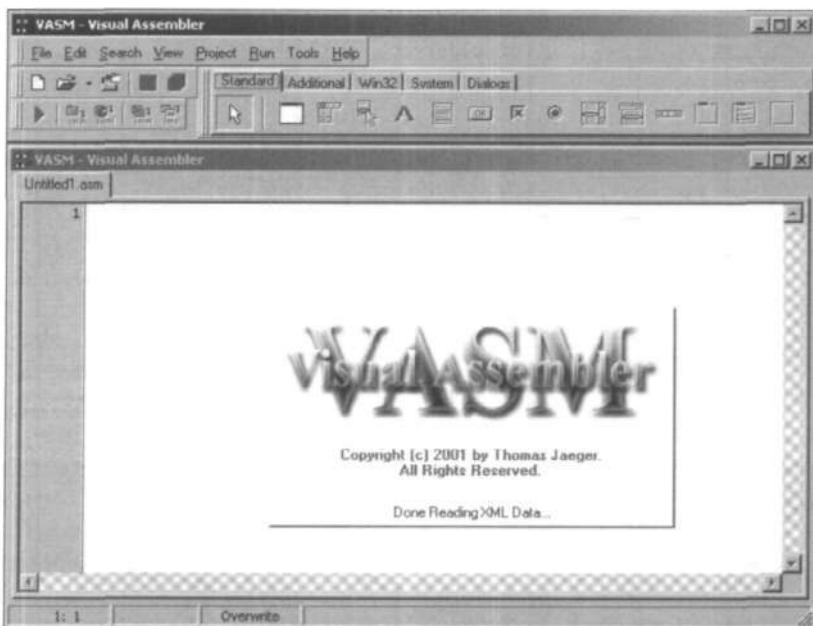


Figura 7.6. Aspecto de VASM.

Otra posibilidad con unas pretensiones similares a VASM, y que ha seguido el mismo camino del abandono, es el editor VisualASM. Su desarrollo se detuvo en algún momento y, aunque es posible descargarlo aún desde algunos sitios Web, el sitio oficial ya desapareció.

Paralelamente, a medida que proyectos como VASM y VisualASM se perdían en los últimos años, han ido apareciendo otros con objetivos similares. Uno de los más interesantes es WinAsm Studio (<http://www.winasm.net>), un entorno en múltiples idiomas preparado para trabajar con MASM y que facilita el desarrollo de programas tanto de 16 como de 32 bits.

Linux

Seguramente es el sistema que cuenta con un mayor número de herramientas de todo tipo y, por supuesto, también de editores avanzados para programadores. Muchos de éstos ya están incluidos en la mayoría de las distribuciones de GNU/Linux, por lo que no es necesario instalar nada aparte de lo que trae el sistema.

Uno de los mejores editores para programadores es, sin duda alguna, emacs, muy conocido por los programadores que utilizan Linux. Realmente es un editor disponible para muchos otros sistemas operativos. Incluso existe una versión para X, llamada xemacs, cuya interfaz de usuario puede observar en la figura 7.7. Toda la potencia de emacs y, al tiempo, la simplicidad de los menús de opciones y botones que ejecutan acciones directamente, sin necesidad de recordar combinaciones de teclas.

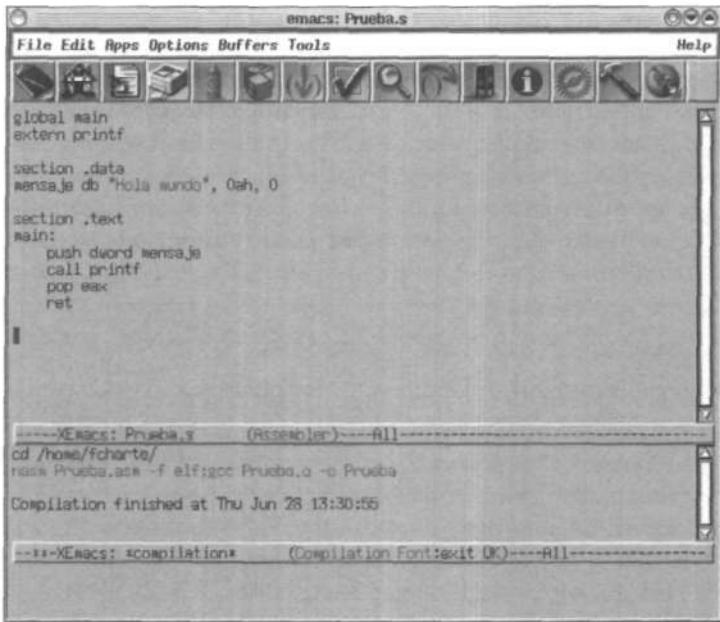


Figura 7.7. El editor xemacs editando un código ensamblador.

Seguramente ya tendrá xemacs instalado en su sistema o bien estará disponible en los CD/DVD de la distribución Linux que haya instalado. En cualquier caso, siempre puede obtener la última versión de este editor en la Web <http://www.xemacs.org>.

Si es de los que prefiere un entorno basado en texto parecido a los usados en DOS, tienen en común muchos elementos, puede recurrir a SetEdit. Encontrará este editor en la carpeta Herramientas/Linux/SetEdit del CD-ROM. Utilice el comando tar con las opciones -xvzf para desempaquetar y descomprimir el archivo, efectuando la instalación. En la figura 7.8 puede ver el aspecto de este editor ejecutándose en una ventana de terminal.

Para comprobar si hay una versión de SetEdit posterior a la entregada en el CD-ROM visite <http://setedit.sourceforge.net>.

Ensambladores

Con los editores podremos crear archivos de texto conteniendo el código fuente, en ensamblador, de nuestros programas.

Esto nos servirá de poco si no tenemos un ensamblador, es decir, la herramienta que traduzca ese código fuente al lenguaje de la máquina. Un ensamblador es como un compilador de cualquier lenguaje: toma un archivo de texto con código fuente como entrada y, en respuesta, genera un nuevo archivo con esas mismas sentencias pero en lenguaje de un determinado procesador.

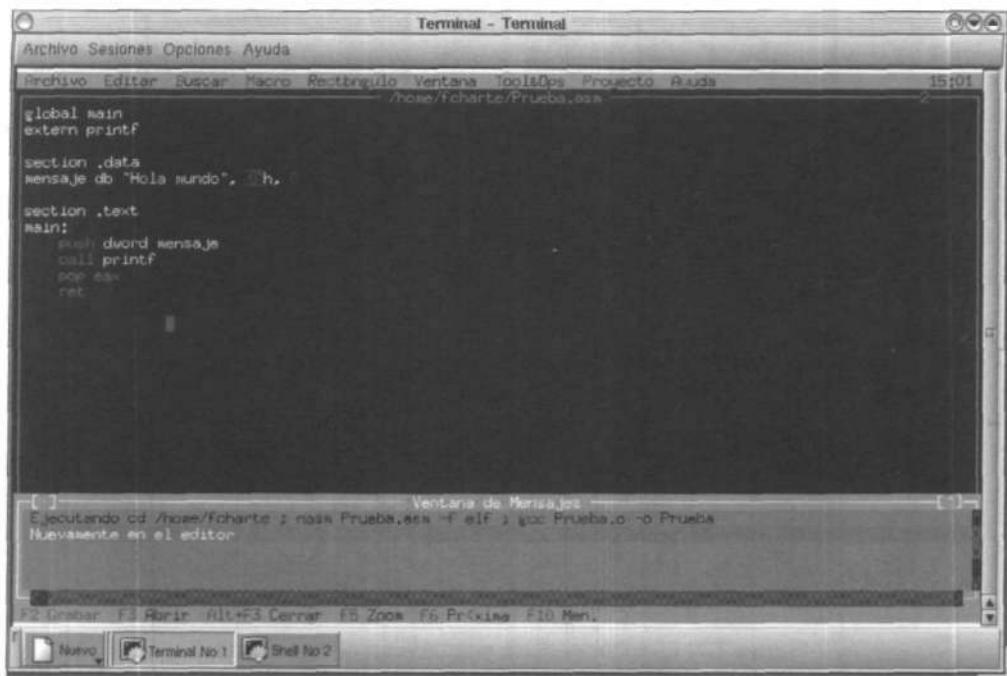


Figura 7.8. Aspecto del editor SetEdit.

Aunque hay multitud de ensambladores para x86, no llegan a ser tantos como los editores y, lógicamente, son aún menos los ampliamente usados. En DOS los reyes fueron TASM y MASM, siendo éste último el más utilizado en la actualidad tanto en DOS como en Windows.

Ambos, no obstante, pierden terreno ante NASM, una solución no sólo para DOS y Windows sino también para otros sistemas, como Linux. Este sistema, además, incorpora como parte indispensable el ensamblador as.

Además del tipo de procesador para el que está pensado, cada ensamblador tiene, por lo general, una sintaxis específica y distinta a los otros. Esto no significa, sin embargo, que sea un ensamblador totalmente diferente, sino que ciertas operaciones tienen variantes.

MASM

Es, seguramente, el ensamblador más usado en DOS y Windows, no en vano está desarrollado por la misma empresa que creó dichos sistemas: Microsoft. Es bastante potente y puede utilizarse para crear programas DOS y Windows, así como bibliotecas y controladores de dispositivos. Como los demás ensambladores, MASM se invoca desde la línea de comandos directamente o, en caso de contar con él, desde un editor previa configuración.

En la carpeta Herramientas \ DOS \ MASM del CD-ROM encontrará la versión 6.15 de MASM. En la subcarpeta bin encontrará el ensamblador propiamente dicho, ML. EXE, así como el enlazador y otras herramientas. Son herramientas que pueden utilizarse directamente desde una ventana Símbolo de sistema o bien desde algunos de los entornos mencionados en puntos previos.

```
Símbolo del sistema
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.

ML [ /options ] filelist [ /link linkoptions ]

/AT Enable tiny model (.COM file)      /omf generate OMF format object file
/B1<linker> Use alternate linker       /Sa Maximize source listing
/c Assemble without linking            /Sc Generate timings in listing
/Cp Preserve case of user identifiers   /Sf Generate first pass listing
/Cu Map all identifiers to upper case  /Sl<width> Set line width
/Cx Preserve case in publics, externs   /Sn Suppress symbol-table listing
/coff generate COFF format object file /Sp<length> Set page length
/D<name>[=text] Define text macro     /Ss<string> Set subtitle
/EP Output preprocessed listing to stdout /St<string> Set title
/F <hex> Set stack size (bytes)        /Sx List false conditionals
/Fe<file> Name executable             /Ta<file> Assemble non-.ASM file
/Fl<file> Generate listing            /w Same as /W0 /WX
/Fm<file> Generate map               /WX Treat warnings as errors
/Fo<file> Name object file           /N<number> Set warning level
/FPi Generate 80x87 emulator encoding  /X Ignore INCLUDE environment path
/Fr<file> Generate limited browser info /Zd Add line number debug info
/FR<file> Generate full browser info  /Zf Make all symbols public
/G<c|d|z> Use Pascal, C, or Stdcall calls /Zi Add symbolic debug info
/H<number> Set max external name length /Zm Enable MASM 5.10 compatibility
/I<name> Add include path            /Zp[n] Set structure alignment
/link <linker options and libraries>  /Zs Perform syntax check only
/nologo Suppress copyright message

G:\MASM\MASM\bin>
```

Figura 7.9. Parámetros del ensamblador MASM.

En la subcarpeta doc de MASM encontrará un documento Word, de más de cien páginas, con información sobre los parámetros que acepta, los mensajes de error, etc.

En algunos de los ejemplos desarrollados en el libro para DOS y Windows usará MASM como ensamblador.

Existe una versión posterior de MASM, la 8.0, que puede descargarse gratuitamente de <http://www.microsoft.com/downloads/details.aspx?FamilyID=7alc9da0-0510-44a2-b042-7ef370530c64&DisplayLang=en> para usos no comerciales (véase la figura 7.10).

Esta versión precisa la instalación previa de Visual C++ 2005 Express, herramienta que puede utilizarse como entorno de trabajo para editar, ensamblar y ejecutar.



Figura 7.10. Página de descarga de MASM 8.0.

NASM

Como se indicaba anteriormente, NASM es una de las mejores opciones a la hora de elegir un ensamblador y, lógicamente, para afirmar esto hace falta aportar hechos.

Al igual que MASM y TASM, NASM es un macro ensamblador. Esto significa que no sólo es capaz de ensamblar instrucciones simples sino que, además, podemos definir y ejecutar macros, lo cual ayuda a simplificar tareas. NASM, además, está en continuo desarrollo y contempla el uso de los últimos conjuntos de instrucciones, como las MMX/SSE, así como las extensiones de 64 bits de los últimos microprocesadores, algo que otros, como TASM, no pueden aportar al encontrarse su desarrollo estancado desde hace tiempo.

Si MASM y TASM son soluciones limitadas a un cierto sistema, DOS o Windows, NASM, por el contrario, es un ensamblador disponible para múltiples plataformas, entre

ellas DOS, Windows y Linux. Esto conlleva una gran ventaja, ya que podemos usar exactamente la misma herramienta en todos los sistemas, sin tener que cambiar de sintaxis, parámetros, etc.

Por último, NASM es una solución de libre distribución, a diferencia de MASM y TASM que son productos comerciales. Esto significa que podemos usar NASM, copiarlo, distribuirlo e, incluso, acceder a su código fuente.

MASM y TASM, por el contrario, no tienen estas ventajas. MASM 6.15, por ejemplo, está incluido en el CD-ROM de este libro previa autorización expresa por parte de Microsoft.

En las carpetas Herramientas\ DOS \NASM, Herramientas\ Windows\NASM y Herramientas\Linux\NASM encontrará versiones específicas de NASM para esos sistemas. En el caso de Windows es un paquete autoinstalable, mientras que en DOS y Linux tan sólo hay que extraer los archivos para comenzar a utilizarlos.

En Herramientas\ DOS \NASM también encontrará documentación común a los tres sistemas.

En todos casos se trata de la versión 2.03. Puede comprobar la existencia de versiones posteriores en <http://nasm.sourceforge.net>.

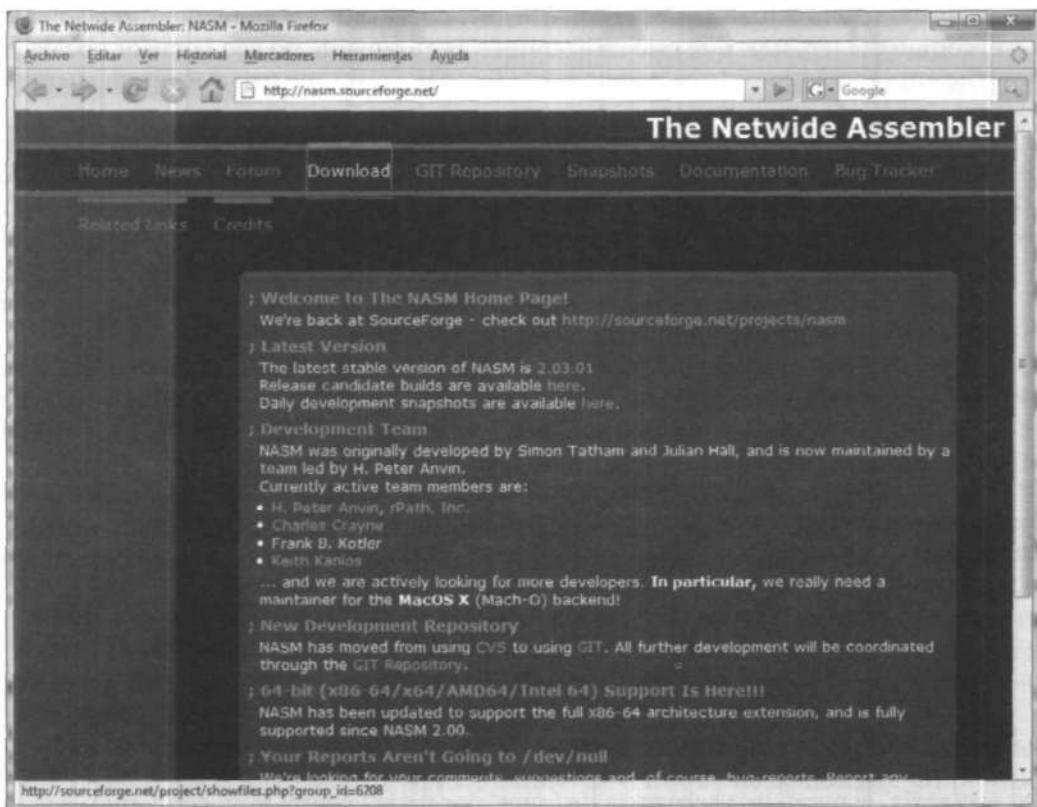


Figura 7.11. Sitio Web de NASM.

Otros ensambladores

Aunque nos hemos centrado en MASM y NASM, éstos no son los únicos ensambladores disponibles. Aparte de TASM, para DOS podemos encontrar también A86, un macro ensamblador de libre distribución pero que debe ser registrado para utilizarse. Puede encontrar este ensamblador, y otras herramientas, en <http://www.ej i . cora/a86/index.htm>.

En Linux puede utilizar as como ensamblador, si bien su sintaxis es algo diferente a la utilizada con TASM, MASM y NASM, aparte de estar pensado más como un ensamblador para gcc que como una herramienta independiente.

Otra posibilidad es gas, un ensamblador que, como NASM, es de libre distribución. Sus posibilidades, no obstante, quedan bastante lejos de las de otros productos.

RAD y ensamblador

Para programar en ensamblador no es indispensable recurrir a un ensamblador externo, como los mencionados en los puntos previos, ya que existen muchas herramientas de desarrollo actuales que permiten la introducción de ensamblador directamente en el código del lenguaje que esté utilizándose. Seguramente no son la mejor opción para crear programas completos en ensamblador, pero sí un recurso a tener en cuenta durante la fase de aprendizaje.

Un ejemplo de este tipo de herramientas es Borland Delphi y su versión para Linux, Borland Kylix. Ambos permiten tanto la introducción de sentencias en ensamblador sueltas como la escritura de procedimientos completos en ensamblador, contemplando el uso de todo el conjunto de instrucciones más actual: Pentium III, MMX, SIMD y 3DNow!.

En la figura 7.12 puede ver el editor de Delphi conteniendo una función muy simple, que sólo suma dos números y devuelve el resultado, escrita en ensamblador. En primer plano una ventana auxiliar nos muestra el estado de la CPU.

La ventaja de usar una herramienta de este tipo, sobre todo al principio, es que uno puede centrarse en lo que está aprendiendo: uso de los registros, operaciones aritméticas, etc., sin tener que preocuparse de cómo mostrar los resultados a efectos de comprobación.

También puede utilizar Visual C++ Express, las últimas versiones son las de 2005 y 2008, como un entorno de trabajo desde el que editar, ensamblar, ejecutar y depurar. El único requisito es agregar reglas de generación de código personalizadas para los módulos con extensión .asm.

Enlazadores

Los ensambladores son traductores que toman el código fuente y, partiendo de cada sentencia, generan una secuencia de bytes, por regla general muy corta, que es el código objeto.

El conjunto de todas las secuencias de bytes, una por instrucción, se almacenan en un archivo que no es directamente ejecutable.

Dicho archivo contiene una versión de nuestro programa en el lenguaje de la máquina, pero no cuenta con la información adicional necesaria para que el sistema operativo sepa gestionarlo.

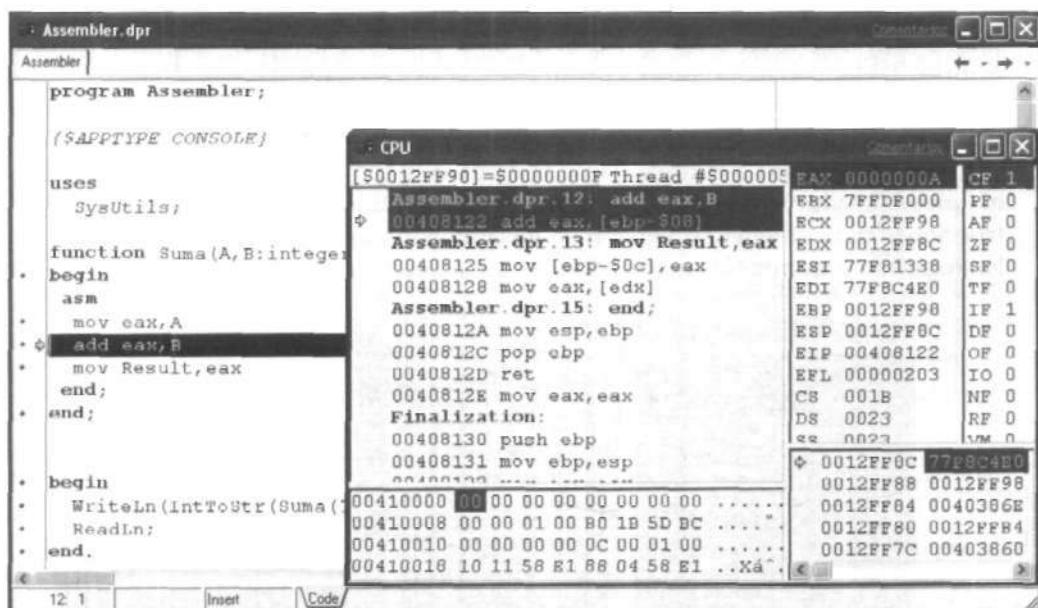


Figura 7.12. Ejecución en Delphi de una función escrita en ensamblador.

Un archivo ejecutable debe contar con uno o varios encabezados con información para el sistema operativo, no instrucciones para el procesador. Esos encabezados indican el tipo de ejecutable, la memoria que necesita, los datos que deben inicializarse en memoria, el punto de entrada, etc.

La herramienta encargada de tomar el código objeto generado por el ensamblador, añadir los encabezados apropiados y producir un nuevo archivo ya ejecutable es el conocido como *linker* o enlazador. Por regla general, cada ensamblador cuenta con su propio enlazador aunque, en realidad, son herramientas distintas y no tiene necesariamente que ser así.

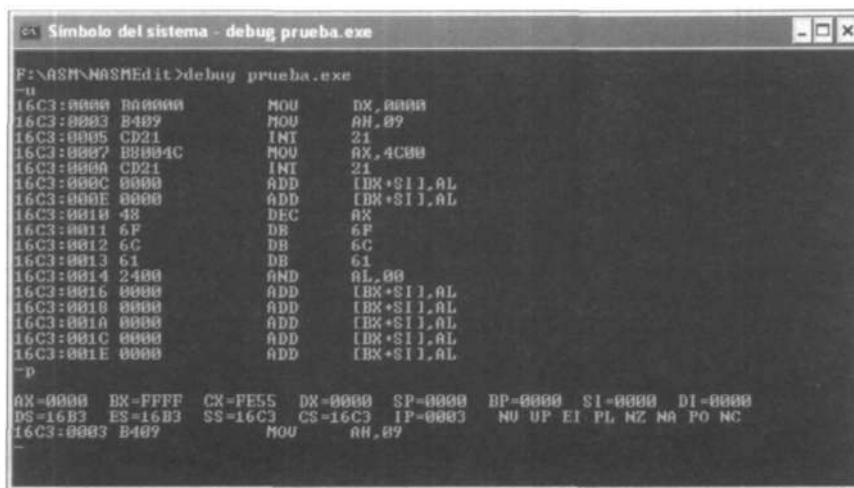
Si utiliza MASM el enlazador será LINK. Para TASM el correspondiente enlazador es TLINK. NASM no se acompaña de un enlazador propio, aunque puede utilizarse cualquiera de distribución libre como es ALINK o bien, en el caso de Linux, usar el propio del sistema.

Una ventaja de utilizar el ensamblador integrado en herramientas de desarrollo actuales, como los mencionados Delphi y Visual C++, es que no se necesita un proceso separado de enlace para obtener un ejecutable, puesto que ese paso está incluido en la compilación.

Depuradores

Como programador, sabrá que una de las fases importantes del desarrollo de cualquier programa es la aplicación de pruebas unitarias que facilite la detección de fallos y que, finalmente, desemboca en el proceso de depuración. Dicho proceso adquiere aún más importancia al programar en ensamblador, dado que las operaciones efectuadas son de muy bajo nivel y cualquier fallo puede provocar un funcionamiento erróneo o, incluso, el bloqueo de la consola o sistema, según los casos.

El depurador más conocido seguramente es la utilidad DEBUG de DOS, que lleva incluida en el sistema operativo prácticamente desde su aparición. Tal como se aprecia en la figura 7.13, es una herramienta de línea de comandos en modo texto. Mediante comandos, introducidos como un solo carácter, es posible desensamblar una serie de instrucciones para obtener su versión en ensamblador, ver el contenido de la memoria o los registros y ejecutar las instrucciones paso a paso.



```

C:\ Símbolo del sistema - debug prueba.exe
F:\ASM\NASMedit>debug prueba.exe
-u
16C3:0000 B00000 MOU DX,0000
16C3:0003 B409 MOU AH,09
16C3:0005 CD21 INT 21
16C3:0007 BB004C MOU AX,4C00
16C3:0009 CD21 INT 21
16C3:000C 0000 ADD [BX+SI],AL
16C3:000E 0000 ADD [BX+SI],AL
16C3:0010 48 DEC AX
16C3:0011 6F DB 6F
16C3:0012 6C DB 6C
16C3:0013 61 DB 61
16C3:0014 2400 AND AL,00
16C3:0016 0000 ADD [BX+SI],AL
16C3:0018 0000 ADD [BX+SI],AL
16C3:001A 0000 ADD [BX+SI],AL
16C3:001C 0000 ADD [BX+SI],AL
16C3:001E 0000 ADD [BX+SI],AL
-p
AX=0000 BX=FFFF CX=PE55 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=16B3 ES=16B3 SS=16C3 CS=16C3 IP=0003 NU UP EI PL NZ NA PO NC
16C3:0003 B409 MOU AH,09

```

Figura 7.13. Aspecto de la utilidad DEBUG depurando un sencillo programa.

Una herramienta similar a DEBUG, pero que contempla el conjunto de instrucciones de los últimos procesadores, es GRDBDL09. La encontrará en la carpeta Herramientas\ DOS\GRDBDL09 del CD-ROM.

Si las aplicaciones a depurar son para Windows, hay múltiples opciones, desde Turbo Debugger y CodeView, que suelen acompañar a TASM y MASM respectivamente, hasta los depuradores integrados en entornos como Delphi y Visual Studio.

En Linux las opciones son similares. El depurador más conocido y usado es gdb. En la carpeta Herramientas\Linux\gdb encontrará un paquete RPM con la última versión de este depurador para instalarlo en su sistema.

La interfaz del programa, como puede verse en la figura 7.14, es igual de simple que la de DEBUG.

The screenshot shows a terminal window titled "Terminal - Terminal". The window contains the following text:

```
Archivo Sesiones Opciones Ayuda
[root@localhost fcharte]# ls
ASM/          Prueba.*  Sout.*  Prueba.bkp  Prueba.s  config*
Desktop/      Infrastr.* Posicion  Prueba.asm  Prueba.o  varnish*  tap/
[root@localhost fcharte]# cd ASM
[root@localhost ASM]# ls
* Prueba.asm  Prueba.o
[root@localhost ASM]# gdb Prueba
GNU gdb 4.17,0.14 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) run
Starting program: /home/fcharte/ASM/Prueba
Hola mundo

Program exited with code 0144.
(gdb)
(gdb)
(gdb)
(gdb)
(gdb)
```

Figura 7.14. El depurador gdb ejecutándose en una ventana de terminal en Linux.

Otras herramientas

Aunque las herramientas mencionadas: editor, ensamblador, enlazador y depurador, son las fundamentales para poder crear programas en ensamblador, en algunos casos necesitaremos otras más específicas. Con ellas podremos compilar archivos de texto describiendo recursos y enlazarlos con nuestro ejecutable, o bien generar módulos de descripción de funciones partiendo de archivos de cabecera, pudiendo así acceder a los servicios del sistema.

En los capítulos concretos donde necesitemos este tipo de herramientas se indicará su función y el lugar donde pueden obtenerse, en caso de que no hayan sido incluidas en el CD-ROM.

Resumen •

Este capítulo nos ha servido para conocer las herramientas más importantes que necesitaremos para programar en ensamblador en tres sistemas operativos distintos: DOS, Windows y Linux.

Aparte de facilitarse en el CD-ROM muchas de las mencionadas, con el fin de que pueda comenzar a trabajar de manera inmediata, también se han indicado los lugares desde donde obtener otras adicionales.

Como es habitual, Internet es una fuente de recursos muy importante para mantenerse actualizado.

Muchas de las herramientas que había disponibles en la primera edición de este libro ya no existen, habiendo aparecido otras, las mencionadas en esta segunda edición, que posiblemente también desaparezcan pasado un cierto tiempo, pero aparecerán otras. A diferencia de otros lenguajes el ensamblador no evoluciona continuamente y, por ello, no precisa nuevas herramientas de trabajo cada 18 ó 24 meses, siendo completamente útiles aquellas que tienen cinco, diez e incluso más años. Una prueba de ello es TASM, un ensamblador que Borland desarrolló en su día y con el desarrollo abandonado desde hace mucho, pero que permite generar programas en ensamblador 8086 de 16 y 32 bits sin ningún problema.

Si en este capítulo tan sólo se han enumerado las herramientas, en el próximo aprenderá a utilizar algunas de ellas creando el que será su primer programa en ensamblador. Será algo muy sencillo, pero suficiente para conocer cuáles son los pasos a dar hasta obtener un ejecutable.

8

Nuestro primer programa

Tras leer los capítulos iniciales, de contenido teórico, y conocer algunas de las herramientas que puede utilizar, seguro que lo que más espera es el momento de escribir su primer programa en ensamblador para poder ver un resultado tangible. Pues bien, no va a tener que esperar mucho ya que en este capítulo, antes de comenzar a introducirnos en los secretos del procesador y el lenguaje, va a tener oportunidad de escribir dicho programa. El objetivo es que sepa cómo usar las herramientas descritas en el capítulo previo, estableciendo una base que, en capítulos posteriores, le sirva para comprobar el código o los ejemplos que vayan proponiéndose. Nuestras pretensiones no van más allá de conseguir ese primer programa que todos escribimos cuando conocemos un nuevo lenguaje y que se limita a emitir un mensaje, ya sea Helio World! o cualquier otro.

Aunque existen herramientas que, como se ha visto en el capítulo previo, integran la edición, el ensamblado y la ejecución, facilitando la tarea de desarrollo, en este capítulo asumiremos que vamos a efectuar todo el trabajo de forma manual. Para editar el código, por tanto, puede usar el editor más básico de que disponga, como el Bloc de notas de Windows o vi de Linux. El proceso de ensamblado, enlazado y ejecución lo realizaremos desde la línea de comandos.

Esqueleto de un programa mínimo

Independientemente de lo que pretendamos hacer en nuestro programa, éste siempre deberá contar con unos elementos que podríamos considerar mínimos. Es algo que ocurre en todos los lenguajes, sin importar su nivel. En el caso concreto del ensamblador,

además, esos elementos dependerán, por una parte, del sistema operativo para el que estemos desarrollando y, por otra, del ensamblador (la herramienta) que vayamos a usar para ensamblar. Al escribir un programa para DOS, por ejemplo, hay que tener en cuenta que estamos creando un programa para un entorno de 16 bits que utiliza un modelo de memoria segmentada. Esto se reflejará en el código del programa en ensamblador, ya que es necesario definir dichos segmentos. La sintaxis, además, es distinta según usemos MASM o NASM, por poner un ejemplo. Cuando el programa va a tener como destino Windows o Linux no existen segmentos, pero sí secciones. Además, puede ser necesaria la inclusión de archivos de cabecera para poder acceder a servicios del sistema.

En los puntos siguientes, donde va a crear un mismo programa para distintos sistemas y con varias versiones específicas para el mismo sistema, verá rápidamente cuáles son esos elementos mínimos. En cualquier caso, no se preocupe si no entiende el sentido de todos esos elementos, es algo que irá conociendo en siguientes capítulos y a medida que vaya adquiriendo experiencia.

Programas COM en DOS

Al crear programas para el sistema operativo DOS existen dos opciones de formato: el formato COM y el EXE. El primero de ellos es el más antiguo y se caracteriza por ser compacto. No dispone de cabeceras con información y, además, todo el programa, código y datos, no debe sobrepasar los 64 kilobytes de ocupación.

i Nota

En capítulos posteriores sabrá qué es un segmento y entonces entenderá el límite de los 64 kilobytes en programas COM.

Vamos a partir creando nuestro programa, que se limitará a mostrar una cadena de texto por la consola, usando este formato compacto para DOS. Los elementos que aprenderemos con él nos permitirán, en los puntos siguientes, centrarnos tan sólo en algunos detalles diferenciadores.

El código

El código necesario para mostrar una cadena de caracteres por la consola generando un programa COM en DOS es el siguiente:

```
Código segment 'code'
org 100h
Entrada:
    jmp Main
```

```
Saludo db '¡Hola COM!$'
```

Main:

```
    mov     dx, offset Saludo
    mov     ah, 9
    int     21h

    mov     ah, 4Ch
    int     21h
```

```
Código ends
end Entrada
```

Todo el código se encuentra contenido en un segmento al que hemos denominado Código. La primera línea es la cabecera de ese segmento. En ella indicamos, por una parte, el nombre con el que será conocido en el programa por parte del ensamblador. Como se ha dicho, ese nombre es Código. Por otra parte, tras la palabra segment aparece, entrecomillado, el nombre que tendrá el segmento de cara al enlazador.

Mientras que los nombres de los segmentos para el ensamblador son variables, a elección del programador, los asignados para el enlazador siempre suelen ser los mismos como verá con posterioridad. El siguiente elemento que encontramos es la directiva org, característica de los programas COM. Básicamente, con ella se indica la posición de memoria en la que deben comenzar a colocarse las instrucciones del programa. El valor siempre será 10 Oh para un programa COM.

i Nota

Recuerde que la letra h detrás de un número indica que éste se ha especificado en base hexadecimal o dieciséis, tal y como se explicó en un capítulo previo.

Los elementos Entrada y Main son etiquetas, marcas con nombre que podemos usar para referirnos a ellas cuando interese. Con la sentencia jmp, por ejemplo, provocamos un salto a la etiqueta Main comenzando a ejecutar las sentencias que hay a partir de ella. El elemento Saludo, por el contrario, no es una etiqueta sino un identificador similar a lo que sería una variable en un lenguaje de alto nivel. Observe que detrás de Entrada y Main se han dispuesto dos puntos, mientras que detrás de Saludo no. El valor de esta variable es una cadena de caracteres finalizada con el carácter \$.

En la parte central del código encontramos el programa propiamente dicho, estructurado en dos grupos de sentencias. El primero, con tres líneas, es el encargado de mostrar el mensaje en la consola, mientras que el segundo tiene la finalidad de devolver el control de nuevo al sistema operativo.

Al trabajar en DOS, una gran parte de los servicios del sistema están accesibles a través de lo que se denominan *interrupciones*. Dichas interrupciones son invocadas mediante la instrucción int, debiendo especificarse a continuación el número de servicio que, en este caso, es el 21h. Un mismo servicio dispone de múltiples funciones. AH es

un registro del procesador y, en este caso, nos sirve para indicar cuál de las funciones del servicio 2 lh queremos usar. La función 9 envía a la consola la cadena de caracteres cuya dirección se haya facilitado en el registro DX, uno a uno hasta encontrar el carácter \$ que, como habrá deducido, señala el final.

El segundo grupo de órdenes utiliza el mismo servicio para ejecutar la función 4ch. Ésta lo que hace es devolver el control al sistema operativo y, en consecuencia, poner fin a la ejecución del programa.

Finalmente tenemos la marca de fin del segmento, compuesta del nombre de éste y la directiva ends, y la marca de fin de código, que es la palabra end. Ésta deberá ir seguida del identificador de la etiqueta por la que deba comenzar la ejecución del programa.

Ensamblado y enlace

Para ensamblar y enlazar el código anterior, que encontrará en el archivo HolaCOM.asm en la carpeta Ejemplos\08\Hola_DOS\COM del CD-ROM, deberá usar las herramientas facilitadas en Herramientas\ DOS \MASM\bin. Puede copiarlas en su sistema y, posteriormente, actualizar la variable PATH para acceder a ellas sin necesidad de anotar el camino completo.

El ensamblador MASM puede utilizarse para crear aplicaciones DOS de 16 y 32 bits, en formatos COM y EXE, así como programas para Windows. El funcionamiento del ensamblador se controla mediante opciones que, como es habitual, habrá que facilitar en la línea de comandos. Si desea conocer todas las opciones disponibles use el comando ML / ?, como se ha hecho en la figura 8.1. Observe que al principio, en la primera línea, existe una opción, /AT, que es la apropiada para generar programas COM.



```

F:\ASM\ DOS\ml /?
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

ML [ /options ] filelist [ /link linkoptions ]

/AT Enable tiny model (.COM file)          /nologo Suppress copyright message
/B[<linker>] Use alternate linker         /Sa Maximize source listing
/c Assemble without linking                /Sc Generate timings in listing
/Cp Preserve case of user identifiers     /Sf Generate first pass listing
/cu Map all identifiers to upper case    /S[<width>] Set line width
/Cx Preserve case in publics, externs     /Sn Suppress symbol-table listing
/coff generate COFF format object file   /Sp[<length>] Set page length
/D<name> [<text>] Define text macro      /Ss<string> Set subtitle
/E[<text>] Define text macro             /St<string> Set title
/F<hex> Set stack size (bytes)           /Sx List false conditionals
/F<file> Name executable                  /Ta<file> Assemble non-.ASM file
/F[<file>] Generate listing               /w Same as /W0 /Wx
/Fm[<file>] Generate map                 /WX Treat warnings as errors
/Fo[<file>] Name object file              /W[<number>] Set warning level
/FPi Generate 80x87 emulator encoding     /X Ignore INCLUDE environment path
/Fr[<file>] Generate limited browser info /Zd Add line number debug info
/FR[<file>] Generate full browser info   /Zf Make all symbols public
/G<<c|d>> Use Pascal, C, or Stdcall calls /ZI Add symbolic debug info
/H<numbers> Set max external name length /Zm Enable MASM 5.10 compatibility
/I<name> Add include path                /Zp[<n>] Set structure alignment
/link <linker options and libraries>     /Zs Perform syntax check only

F:\ASM\ DOS>_

```

Figura 8.1. Opciones que acepta el ensamblador MASM.

Todo lo que necesitamos, por tanto, es invocar al ensamblador facilitándole la citada opción /AT y, por supuesto, el nombre del archivo que contiene el código fuente ensamblador. Asumimos que el enlazador se encuentra también en la ruta de búsqueda de la variable PATH y, como se aprecia en la figura 8.2, es el propio ensamblador el que, cuando ha terminado su trabajo, llama al enlazador para generar el archivo COM.

Si el enlazador se encontrase en otro lugar, o bien deseásemos utilizar un enlazador distinto a link, podríamos usar la opción /c del ensamblador para indicarle que sólo realice su trabajo, pero que no invoque al enlazador. De esta forma, después efectuaríamos nosotros ese trabajo manualmente. En cualquier caso, el ensamblador habrá generado un archivo OBJ a partir del código fuente. Ese archivo tiene el código objeto y también información para el enlazador indicándole, por ejemplo, los segmentos existentes y sus características. El enlazador toma ese archivo intermedio y genera el programa COM, que podemos ejecutar simplemente introduciendo su nombre en la línea de comandos.

F:\ASM\ DOS>type HolaCOM.asm
Codigo segment 'code'

 org 100h
Entrada:
 jmp Main

Saludo db '¡Hola COM!\$'

Main:
 mov dx, offset Saludo
 mov ah, 9
 int 21h

 mov ah, 4ch
 int 21h

codigo ends
end Entrada

F:\ASM\ DOS>ml /AT HolaCOM.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: HolaCOM.asm

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: HolaCOM.obj /t:
Run File [HolaCOM.com]: "HolaCOM.com"
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:

F:\ASM\ DOS>HolaCOM
¡Hola COM!
F:\ASM\ DOS>HolaCOM
¡Hola COM!
F:\ASM\ DOS>

Figura 8.2. Ensamblamos, enlazamos y ejecutamos el programa.

Nota

Observe, tras ensamblar y enlazar el ejemplo, el pequeñísimo tamaño que tiene el programa COM. Tan sólo 24 bytes para un programa que muestra un mensaje por pantalla. Seguramente estará acostumbrado a obtener ejecutables de decenas o incluso de centenas de kilobytes para hacer lo mismo con lenguajes de alto nivel.

Programas EXE en DOS

El formato de ejecutable EXE apareció en DOS cuando las necesidades de los programas excedieron esa capacidad limitada de los 64 kbytes en el formato COM. El formato EXE, más avanzado, permite crear programas mucho más grandes, con múltiples segmentos independientes para datos y código. Salvo esto, no hay muchas más ventajas del formato EXE respecto al COM.

Para generar nuestro programa COM hemos usado MASM como ensamblador, aunque también podríamos haber utilizado NASM. En este punto vamos a ver cómo generar el mismo programa con ambos ensambladores, pudiendo así apreciar las diferencias sintácticas aunque, en definitiva, el resultado obtenido sea el mismo.

tt

Versión MASM

Comencemos editando una primera versión del programa pensada para ensamblarse con MASM. El código sería el siguiente:

```
Pila    segment stack 'stack'
        db 256 dup (?)
Pila    endo

Datos   segment 'data'
        Saludo db ';Hola DOS!$'
Datos   ends

Código  segment 'code'
        assume CS:Código, DS:Datos, SS:Pila

Main    :
        -nr,
        mov     ax, seg Datos
        mov     ds, ax
        mov     dx, offset Saludo
        mov     ah, 9
        int     21h

        mov     ah, 4Ch
        int     21h

Código  ends

end Main
```

Como puede apreciarse, en este caso el programa no cuenta sólo con un segmento sino con tres: uno para utilizar como pila, otro con los datos y un tercero con el código a ejecutar. Todos ellos cuentan con un nombre interno a nuestro programa, como puede ser Datos o Código, y también con un nombre público para el enlazador que, como se indicó anteriormente, va entrecomillado.

El primer segmento es el dedicado a pila del programa, en la que se almacenarán temporalmente parámetros y direcciones de retorno a la hora de llamar a procedimientos y servicios. Observe que tras segmento aparece la palabra stack sin entrecomillar. Con ello indicamos a MASM que se encargue él de disponer apropiadamente los registros para usar la pila. No es algo por lo que deba preocuparse en este momento. La única línea que contiene el segmento de pila tiene por finalidad reservar 256 bytes de memoria. Ése será el tamaño dedicado a la pila.

En segundo lugar tenemos el segmento de datos que, como puede verse, tan sólo contiene la cadena de caracteres a mostrar por la consola.

Por último tenemos el segmento de código que, respecto al ejemplo en el que generábamos un archivo COM, tiene dos particularidades: la directiva assume y la preparación del registro DS para acceder al segmento de datos. Con la primera asociamos ciertos registros con los segmentos previamente definidos. Es algo sobre lo que velvemos posteriormente.

Al usar la función 9 del servicio 21h, para mostrar una cadena de caracteres en la consola, se espera que el registro DS contenga el segmento donde está esa cadena, y el registro DX la dirección dentro del segmento. En nuestro primer ejemplo no nos preocupábamos por el valor de DS, ya que un programa COM sólo tiene un segmento y, por tanto, todos los registros apuntan a él. En un EXE, por el contrario, esto no tiene por qué ser así y, por ello, es preciso recuperar ese segmento y almacenarlo en el mencionado registro DS.

Seguramente no comprenderá por ahora lo relacionado con segmentos y direcciones dentro de segmentos. Es uno de los temas más complejos de la programación en DOS en ensamblador y, en un capítulo posterior, tendrá ocasión de aprender más sobre ello. Por ahora simplemente se mencionan las necesidades del programa, aunque asumiendo algunos puntos como establecidos.

Para ensamblar y enlazar este programa tan sólo tiene que llamar a MASM facilitando como único parámetro el nombre del archivo con el código fuente. En la figura 8.3 puede ver cómo se ensambla, enlaza y ejecuta el programa. Éste genera exactamente el mismo resultado que la versión COM, si bien el archivo EXE ocupa unas treinta veces más. Con todo, son tan sólo 800 bytes.

Versión NASM

El código para realizar una determinada tarea será el mismo indistintamente del ensamblador que se utilice pero, no obstante, la elección del ensamblador influye, por ejemplo, en la sintaxis usada a la hora de definir los segmentos. También ciertas tareas, que pueden darse por asumidas en unos casos, no deben serlo en otros, al cambiar de ensamblador.



Figura 8.3. Generamos y ejecutamos nuestro primer EXE.

Pasemos a analizar el código de esta nueva versión del programa, que es la siguiente:

```

segment Datos
Saludo db 'Hola DOS!$'

segment Pila stack
resb 256
InicioPila;

segment Código
..start:
    mov ax, Pila          jp*
    mov so, ax
    mov sp, InicioPila

    mov ax, Datos
    mov ds, ax
    mov dx, Saludo
    mov ah, 9
    int 21h

    mov ah, 4ch
    int 21h

```

La palabra clave para indicar el inicio de cada segmento es `segment`, como en el caso de MASM, si bien no es necesario indicar más que el nombre con el que se le conocerá y, además, no hay que delimitar el segmento con una indicación de fin. Para reservar el espacio de pila se usa una sintaxis diferente, si bien el resultado es exactamente el mismo.

En el segmento de código no encontramos la directiva `assume` y es que, a diferencia de MASM, en N ASM no se asume la asociación de ciertos registros con determinados segmentos. Al no asumirse, es necesario indicarlo explícitamente cuando proceda.

Con MASM, según se ha dicho en el punto anterior, es el propio ensamblador quien se ocupa de preparar los registros relacionados con la pila, dándoles los valores apropiados para acceder al segmento previamente definido. NASM no lo hace y, por tanto, debemos codificarlo manualmente. Eso es lo que hacen las tres primeras instrucciones del programa. El resto del código es idéntico al visto en los casos anteriores.

Por último, es de notar la diferencia existente a la hora de indicar cuál es el punto de entrada en el programa. Con MASM dicho punto se indica tras el último end, mientras que en NASM se usa la marca .start: para indicar por dónde debe comenzar la ejecución.

Ensamblado y enlace

Al igual que ocurre con MASM, NASM dispone de multitud de opciones. La mayoría de ellas puede verlas invocándolo con la opción -h. Asimismo, NASM es capaz de generar código en múltiples formatos distintos: DOS de 16 y 32 bits, Windows de 32 y 64 bits, Linux, FreeBSD, etc. Con la opción -hf obtendrá una lista de todos los formatos disponibles, visibles en la parte inferior de la figura 8.4. En este caso concreto, generar un EXE para DOS, el que nos interesa es ob j.

El ensamblado, por tanto, lo realizaremos invocando a NASM facilitando como primer parámetro el nombre del archivo que contiene el código y como segundo la opción -f obj, indicando el formato que deseamos.

```
Windows Símbolo del sistema
Warnings:
  error
  macro-params
  macro-selfref
  orphan-labels
  on)
  number-overflow
  n)
  gnu-elf-extensions
  ion (default off)
  float-overflow
  float-denorm
  float-underflow
  float-toolong
  )
  response files should contain command line parameters, one per line.

valid output formats for -f are ('*' denotes default):
  * bin      flat-form binary files (e.g. DOS .COM, .SYS)
  aout     Linux a.out object files
  aoutb    NetBSD/FreeBSD a.out object files
  coff     COFF (i386) object files (e.g. DJGPP for DOS)
  elf32   ELF32 (i386) object files (e.g. Linux)
  elf     ELF (short name for ELF32)
  elf64   ELF64 (x86_64) object files (e.g. Linux)
  as86    Linux as86 (bin86 version 0.3) object files
  obj     MS-DOS 16-bit/32-bit OMF object files
  win32   Microsoft Win32 (i386) object files
  win64   Microsoft Win64 (x86-64) object files
  rdf    Relocatable Dynamic Object File Format v2.0
  ieee   IEEE-695 (LADsoft variant) object file format
  macho  Nextstep/OpenStep/Rhapsody/Darwin/MacOS X object files

D:\NASM\NASM-2~1.01>
```

Figura 8.4. Opciones de formato del ensamblador NASM.

Para obtener un ejecutable a partir del módulo obj generado por NASM necesitamos, como ya sabe, efectuar el enlazado. Con este fin puede usar el enlazador alink, como se ha hecho en la figura 8.5. En ella puede ver también el resultado de ejecutar el programa.

Encontrará el ensamblador nasm y el enlazador alink en la carpeta Herramientas\ DOS\NASM.

```

Símbolo del sistema
F:\ASM\ DOS>nasm HolaDOSNASM.asm -f obj
F:\ASM\ DOS>alink HolaDOSNASM.obj -oEXE
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams,
All Rights Reserved

Loading file HolaDOSNASM.obj
matched Externs
matched ComDefs

F:\ASM\ DOS>
F:\ASM\ DOS>
F:\ASM\ DOS>Hola DOS!
F:\ASM\ DOS>Hola DOS!
|Hola DOS!
F:\ASM\ DOS>

```

Figura 8.5. Ensamblado y enlazado con NASM y ALINK.

Programas Linux

Como se indicaba en el capítulo previo, existen muchos ensambladores, enlazadores y otras herramientas disponibles para Linux. Una de las mejores combinaciones, sin embargo, consiste en utilizar NASM conjuntamente con el compilador g++ para efectuar el enlazado.

Es un mecanismo fácil para acceder a servicios que están habitualmente al alcance de los programas en C o C++.

A pesar de que usemos como ensamblador NASM, que ya hemos utilizado previamente para crear un programa en DOS, el programa será muy distinto, ya que el destino es un sistema operativo diferente y, por lo tanto, la estructura de los ejecutables y los servicios disponibles no son los mismos.

El código

Comencemos, como en los casos anteriores, viendo el código de esta nueva versión del programa y analizándolo. El contenido de HolaLinux.asm es el siguiente:

```
global main
extern printf

section .data
Saludo db ";Hola Linux!", Oah, 0

section .text
main:
    push dworri Saludo
    call printf
    pop eax
```

Seguramente, lo primero que llamará su atención es la brevedad del código si lo comparamos con versiones previas para DOS. Esto se debe, en parte, a que utilizamos una función estándar de C para efectuar la impresión de la cadena. También hay que tener en cuenta que no es necesario llamar a ningún servicio del sistema para devolverle el control, esto ocurre automáticamente cuando el programa termina.

La primera línea del programa, `global main`, indica que el punto de entrada al programa es la etiqueta `main` y, por ello, la hace pública, como si fuese la función `main` de un programa escrito en C. La línea siguiente especifica que la función `printf` es externa y que vamos a usarla en nuestro programa.

A continuación tenemos una sección con los datos del programa, en este caso tan sólo la cadena de caracteres con el mensaje a mostrar. El indicador de fin, en este caso, no es el carácter \$ sino un byte 0 añadido al final.

Por último tenemos la sección de código, con una etiqueta y sólo cuatro sentencias ejecutables. La primera de ellas toma la dirección de `Saludo` y la introduce en la pila, llamando a continuación a la función `printf`. Ésta dará salida a la cadena y retornará un valor de resultado en el registro `EAX`, valor que extraemos con la instrucción `pop`. Finalmente, devolvemos el control con la instrucción `ret`.

Las secciones de datos y código serían equivalentes, sintácticamente hablando, a los segmentos usados en la versión EXE para DOS aunque, en realidad, en Linux no es necesario usar segmentos ya que se trabaja en un espacio de memoria plano de 32 bits.

Ensamblado y enlace

Asumiendo que ya tenemos el código fuente en un archivo de texto, el paso siguiente consistiría en ensamblarlo. Para ello invocaremos a NASM de forma similar a la vista previamente, para generar un EXE en DOS, si bien el parámetro a entregar junto a la opción `-f` no será `obj` sino `elf`, que es el formato estándar en Linux, o bien `elf64` si la distribución de Linux instalada es de 64 bits.

A continuación, en lugar de usar un enlazador propiamente dicho, teniendo que indicarle qué tipo de ejecutable deseamos, qué librerías debe adjuntar para encontrar la función externa printf, etc., nos serviremos del compilador gcc. Tan sólo hay que facilitar como parámetro el nombre del archivo que contiene el código ensamblado, que será el mismo del código fuente pero con extensión o, y, opcionalmente, indicar el nombre del archivo ejecutable.

En la figura 8.6 puede ver los pasos dados en la consola de Linux para ensamblar, enlazar y ejecutar el programa. El resultado, como en el caso de DOS, es la salida de la cadena de caracteres por la consola del sistema.

```
[root@localhost Cap_02]# cat HolaLinux.asm
global main
extern printf

section .data
Saludo db "¡Hola Linux!", Bah, 0

section .text
main:
    push dword Saludo
    call printf
    pop eax
    ret

[root@localhost Cap_02]# nasm HolaLinux.asm -f elf
[root@localhost Cap_02]# gcc HolaLinux.o -o HolaLinux
[root@localhost Cap_02]# ./HolaLinux
¡Hola Linux!
[root@localhost Cap_02]#
```

Figura 8.6. Ensamblado y enlazado del programa en Linux.

Al igual que en DOS, en Linux podríamos usar las funciones de una interrupción de servicio para conseguir el mismo resultado. Por ahora, sin embargo, el uso de la función printf es suficiente y resulta más sencillo.

Programas Windows

A diferencia de DOS y Linux, que son sistemas operativos esencialmente basados en modo texto, Windows utiliza una interfaz gráfica como elemento central para comunicarse con el usuario. Es cierto que podríamos abrir una consola de línea de comandos y, por supuesto, es posible crear aplicaciones de consola para Windows. En la mayoría de los casos, sin embargo, las aplicaciones contarán con una interfaz gráfica.

Para acceder a los servicios de Windows, lo que habitualmente se conoce como su API (*Application Vrogramming Interface*), es necesario contar con definiciones de funciones,

constantes, estructuras de datos, etc. Todas esas definiciones hacen más fácil utilizar dichos servicios. Además, también son precisas bibliotecas que permitan al ensamblador localizar las funciones a las que invoquemos.

En la carpeta Herramientas\Windows\masm32 del CD-ROM encontrará todos esos elementos. Use el ensamblador y enlazador que hay en la subcarpeta bin, los archivos con definiciones de la subcarpeta include y las bibliotecas de la subcarpeta lib. También encontrará multitud de ejemplos y documentación.

El código

Al igual que ocurre en Linux, en Windows trabajaremos siempre sobre un modelo de memoria plano de 32 bits, sin necesidad de usar segmentos. Esto hace que el código, que puede ver a continuación, sea algo más simple.

```
.386
.model flat,stdcall

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

.data

Titulo db "Programación en ensamblador",0
Texto db ";Hola Windows!",0

.code
Main:

invoke MessageBox, 0, offset Texto, offset Titulo, MB_OK
invoke ExitProcess, 0

end Main
```

Las dos primeras líneas del programa son directivas que le indican al ensamblador el conjunto de instrucciones que vamos a usar, así como el modelo de memoria y la convención de llamadas a funciones. Son todos detalles que conocerá posteriormente en mayor detalle.

A continuación puede ver tres `include` y dos `includelib`. Con ellas se recuperan todas las definiciones anteriormente citadas, así como las bibliotecas que permitirán al ensamblador localizar las funciones `MessageBox` y `ExitProcess` que utilizaremos más adelante.

Como en el caso del programa para Linux, tenemos dos secciones: una con los datos del programa y otra con el código. La primera contiene sólo la cadena de caracteres a mostrar y un título adicional para la ventana en la que aparecerá dicha cadena.

El código se compone básicamente de dos sentencias. En la primera se invoca, mediante una macro, a la función MessageBox facilitándole una serie de parámetros, entre ellos las direcciones de la cadena de texto y el título.

MB_OK es una de las constantes importadas en los include. Con ella indicamos que la ventana en la que aparecerá el mensaje deberá contar con un botón **Aceptar**.

La segunda sentencia es equivalente a la función 4 ch de la interrupción 2 lh en DOS, poniendo fin a la ejecución del programa y devolviendo el control a Windows.

Ensamblado y enlace

Tenemos el código del último de nuestros ejemplos en el archivo HolaWindows.asm y, lógicamente, queremos obtener un ejecutable para ver el resultado. El enlazado necesitará algunas opciones más de las que hemos usado hasta ahora, concretamente las opciones /c /Cp /coff.

Con ellas indicamos al ensamblador que sólo ensamble, sin invocar al enlazador; que preserve las mayúsculas y minúsculas en los identificadores, ya que esto es importante en Windows y, por último, que genere código en formato COFF, que es el utilizado en este sistema operativo.

El enlazado lo efectuaremos, como en casos anteriores, con la herramienta link. La única opción necesaria es /SUBSYSTEM:WINDOWS. El programa obtenido ocupa 2560 bytes, mucho más que nuestra versión COM pero, sin duda, infinitamente menos que un programa que muestre esa misma ventana escrita con C++, Visual Basic, Delphi o un RAD similar.

En la figura 8.7 puede ver, en primer plano, la ventana con el mensaje y el título. De fondo aparece la ventana en la que se ha ensamblado y enlazado el proyecto.



Figura 8.7. Nuestro primer programa Windows escrito en ensamblador.

Resumen

Como ha podido ver en este capítulo, crear un programa en ensamblador para mostrar un mensaje en pantalla no es tan fácil como escribir PRINT "Hola" en BASIC, pero, seguramente, tampoco es tan complejo como esperaba.

Hemos creado el mismo programa para tres sistemas operativos distintos, en el caso de DOS en dos formatos diferentes, y usado dos ensambladores y dos enlazadores: MASM, NASM, LINK y ALINK. Lo más importante, el objetivo que tenía este capítulo, era hacerle ver que crear un programa en ensamblador no es una tarea difícil.

Los conocimientos adquiridos con estos ejemplos, ya sabemos cómo mostrar una cadena de caracteres por la consola, nos serán útiles en capítulos posteriores en los que nos centremos más en la manipulación de datos, el uso de registros y ejecución de operaciones diversas. Son casos en los que tan sólo necesitaremos imprimir el resultado para poder efectuar comprobaciones.

A pesar de que en este capítulo se ha facilitado el código de un mismo ejemplo en múltiples versiones, para DOS, Windows y Linux y usando MASM y TASM, en la mayor parte de los capítulos que encontrará a continuación trabajaremos siempre sobre DOS y con NASM. Para comprender cómo se realizan operaciones aritméticas o cómo se codifica un bucle es suficiente una versión del ejemplo, ya que las instrucciones y registros implicados serán los mismos indistintamente del sistema y ensamblador que se emplee. La elección de DOS se debe a su accesibilidad, cualquiera puede ejecutar DOS incluso en un ordenador antiguo, y simplicidad respecto a los otros sistemas a la hora de trabajar y obtener un ejecutable.

9

Ejecución de un programa

En el capítulo anterior, a modo de primer ejemplo, codificábamos nuestro primer programa en ensamblador en distintas versiones, un programa que ejecutábamos simplemente introduciendo su nombre, en una línea de comandos DOS o Linux, o bien haciendo doble clic sobre él, en el caso de Windows. En cualquier caso, en ese momento, al poner en marcha un programa, se desencadena una serie de acciones cuyo fin último es poder transferir el control a la secuencia de instrucciones que componen el programa para así ejecutarlo.

Nuestro objetivo, en este capítulo, es conocer, aunque sea de manera superficial y breve, el proceso que hace posible que una serie de bytes almacenados en un archivo pueda ser ejecutada por el procesador. Por el camino obtendremos algunos fundamentos sobre formatos de archivos ejecutables, preparación de registros del microprocesador o el funcionamiento de la pila. La lectura de este capítulo nos permitirá comprender un poco mejor aspectos de interés a la hora de programar en ensamblador, aspectos que al trabajar con otros lenguajes no tienen una mayor importancia pero que, en nuestro caso, pueden sernos de mucha utilidad.

Formatos de archivo ejecutable

La información con la que trabaja un ordenador por regla general se almacena de forma persistente en algún medio físico, como por ejemplo un disco duro interno, y los programas no son una excepción, alojándose en archivos como también se hace con los

documentos de textos, gráficos, hojas de cálculo, etc. Lo que distingue a unos archivos de otros es su formato, habitualmente determinado por una cabecera, una secuencia de bytes dispuesta al inicio del archivo, que contiene información diversa. Lógicamente, la cabecera de un archivo correspondiente a un documento de Microsoft Word es distinta que la de un programa ejecutable en Linux.

No es extraño que, para almacenar un mismo tipo de información, existan distintos formatos de archivo. Los gráficos, por ejemplo, pueden estar contenidos en archivos BMP, JPG, TIF o GIF, por mencionar cuatro bien conocidos. En estos casos no sólo tenemos una cabecera distinta, según el tipo de archivo, sino que el propio gráfico se almacena en su interior con una estructura u otra, definida como parte del formato.

Al tratar los programas, contenidos en archivos que almacenan datos y código para ser ejecutado, también podemos encontrarnos con múltiples formatos distintos, incluso sobre un mismo sistema operativo. Existe, no obstante, un conjunto de formatos de uso más habitual.

Ejecutables en DOS

El primer formato de archivo ejecutable aparecido en DOS, hace ya casi tres décadas, fue el formato COM, conocido por la extensión que se daba al archivo. Este formato no cuenta con cabecera alguna, sino una simple secuencia de bytes correspondientes a las instrucciones a ejecutar. De hecho, es posible crear un archivo ejecutable en formato COM desde la propia línea de comandos de la consola utilizando tan sólo el teclado, sin herramienta alguna.

DOS fue diseñado inicialmente para un procesador de 16 bits, la familia 8086/8088, y el formato ejecutable COM no aprovechaba las capacidades de segmentación de dicho procesador, por lo que estaba limitado a un tamaño máximo de 64 kbytes. Es decir, un programa en formato COM no puede exceder, conjuntamente entre datos, código y pila de 65536 bytes. Esta limitación, en principio, no suponía un problema ya que los PC de entonces no tenían mucha más memoria.

Nota

El primer modelo de PC presentado por IBM disponía únicamente de 16 kilobytes de memoria RAM, una cantidad que por entonces resultaba más que suficiente.

Las limitaciones del formato COM dieron paso rápidamente al segundo formato de ejecutable para DOS, el más usado, conocido como MZ por ser éstos los dos primeros caracteres de su cabecera. En la cabecera se indica el tamaño del archivo en páginas y las tablas de reubicación del código.

Los programas en formato MZ se alojan en archivos con extensión EXE. A diferencia del formato COM, en el MZ es posible usar múltiples segmentos para código y datos, trabajando con un límite teórico de un megabyte.

Ejecutables en Linux

GNU/Linux es un sistema operativo que tiene sus raíces en Unix y, como tal, ha utilizado y utiliza formatos de archivo ejecutable propios de Unix. Además de Linux existen otras implementaciones libres de Unix, como FreeBSD, en las que también se usan dichos formatos.

El formato de archivo más antiguo en los sistemas Unix es el conocido como a.out, actualmente en desuso en la mayoría de distribuciones Linux pero totalmente vigente en sistemas como FreeBSD. Este formato cuenta con una cabecera compacta que siempre define tres secciones distintas del programa: código, datos y pila. Este formato tiene limitaciones que le hacen poco adecuado para, por ejemplo, la creación de módulos dinámicos contenido código que puede utilizarse desde programas ejecutables.

Base de otros formatos existentes, el segundo que encontramos en Linux es COFF (*Common Object File Format*), generado por múltiples herramientas de ciertas distribuciones, entre ellas los ensambladores y enlazadores. Este formato contempla la posibilidad de que se incluya en el archivo información específica de cada sistema sin, por ello, salirse de la especificación. También facilita la introducción de información simbólica para la depuración, así como una tabla de secciones que supera el límite de las tres fijas existentes en el formato a.out.

Introducido inicialmente por Sun en su sistema Solaris, ELF (*Executable and Linking Format*) es el formato de archivo ejecutable más utilizado en la actualidad en las distintas distribuciones de Linux. Una de las razones para ellos es su idoneidad a la hora de crear esos módulos dinámicos a los que se hacía referencia previamente. Al igual que COFF, el formato ELF también contempla la existencia de múltiples secciones en el programa.

Ejecutables en Windows

Diseñado en sus inicios como un entorno gráfico que se ejecutaba sobre DOS, Windows ha ido evolucionando hasta convertirse en un sistema operativo completo. En esa transición se ha pasado de un entorno de 16 bits, como el de Windows 3.0 y 3.1, a uno de 32, en las actuales versiones de Windows, y de 64 bits en algunos casos. Por ello existen, básicamente, dos formatos de archivo ejecutable de Windows.

El primero de estos formatos, usado en los Windows de 16 bits, es el conocido como Windows-NE (*New Executable*). Por entonces ya existía el formato MZ, pero éste tenía limitaciones que podían ser superadas en Windows y, por ello, se creó el nuevo formato de ejecutable. La aparición de Windows NT y Windows 95 implicó la necesidad de un nuevo formato de archivo, un formato preparado para aprovechar las capacidades de los procesadores de 32 bits. Derivado de COFF, este nuevo formato de ejecutable Windows es el conocido como PE (*Portable Executable*). A diferencia de formatos tales como MZ, basados en la definición de segmentos de código y datos de un máximo de 64 kbytes, el formato PE, al igual que COFF, asume la existencia de un modelo de direccionamiento de 32 bits totalmente plano, por lo que se tiene acceso directo a varios gigabytes de memoria sin necesidad de segmentación.

Si examina un ejecutable Windows, simplemente volcando su contenido a la consola, observará que lo primero que aparece es la cabecera con el prefijo MZ, es decir, el archivo es un ejecutable en formato MZ. Más adelante (véase la figura 9.1) encontramos el prefijo de la cabecera PE. Lo que tenemos, en realidad, es un archivo con dos ejecutables diferentes, uno en formato MZ y otro en formato PE. Si intentamos ejecutar un programa Windows desde DOS, no desde una consola DOS de Windows sino desde el modo DOS del sistema, se ejecutaría el código que sigue a la cabecera MZ, mostrándose un mensaje que indica que el programa no puede utilizarse desde DOS.

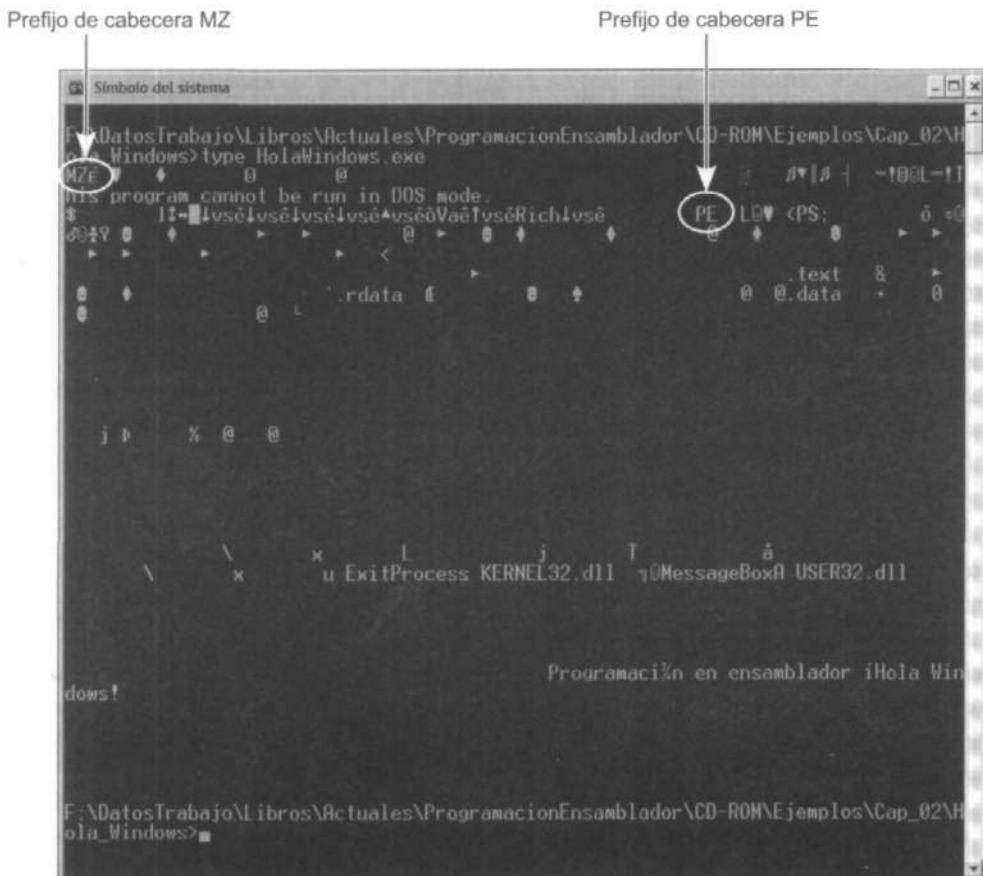


Figura 9.1. Contenido del archivo HolaWindows.exe creado en el capítulo previo.

Detalles sobre formatos de archivo

Es obvio decir que cada formato de archivo ejecutable tiene sus particularidades, si bien todos ellos cumplen un requisito fundamental: informar al sistema operativo sobre los detalles del programa que hay contenido en el archivo, indicando las secciones o

segmentos que existen, facilitando la localización de datos y código, etc. Si quiere saber más sobre formatos de archivo puede recurrir a Internet, donde encontrará cientos de referencias.

Sobre PE y COFF, puede encontrar la especificación de Microsoft en <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>. Para cualquiera de los otros formatos recurra a <http://www.wotsit.org> y busque el formato directamente por su nombre. Encontrará enlaces o documentos, según los casos, describiendo dicho formato.



Figura 9.2. En Wotsit's Format encontrará información diversa sobre formatos de archivo.

Preparación del programa por parte del sistema

Para iniciar la ejecución de un programa el usuario emplea, básicamente, dos recursos distintos: la introducción del nombre del programa en la línea de comandos de la consola o, si el sistema operativo cuenta con una interfaz gráfica, una selección de opción en un menú o un clic sobre una lista tipo Explorador de archivos de Windows.

Indistintamente del recurso empleado, el sistema operativo tiene que abrir el archivo indicado, comprobar que es un ejecutable válido y, en caso afirmativo, preparar todo lo necesario para ponerlo en marcha. Esto implica tareas como la asignación de bloques de memoria, creación de un proceso e hilo de ejecución, configuración de los registros del procesador, el espacio de pila a utilizar, etc.

Recuperación de la cabecera del ejecutable

El primer paso consiste en la apertura del archivo por parte del sistema, como se abriría cualquier otro, con el objetivo de determinar su formato y recuperar la información de la cabecera a.out, MZ, COFF, NE o PE, según los casos. Hay ciertos sistemas, como es el caso de DOS, en los que la extensión del archivo también influye en su tratamiento por parte del sistema operativo.

Un archivo con extensión COM, independientemente de su contenido, siempre es tratado como un ejecutable en formato COM. Es algo que no se da, por ejemplo, en Linux y otras variantes de Unix.

La información obtenida de la cabecera permitirá al sistema operativo determinar si es o no un ejecutable válido. Desde una consola de sistema en Windows, por ejemplo, podrían ejecutarse programas en formatos MZ y PE, es decir, aplicaciones DOS y aplicaciones Windows.

La información de la cabecera indicará al sistema del formato y, en consecuencia, se ejecutará en la misma ventana de consola o bien en su propia ventana independiente como cualquier otra aplicación Windows, respectivamente.

Además del formato, en esa cabecera el sistema también encontrará datos esenciales tales como la longitud del código o los datos, la dirección en la que debe iniciarse el programa, etc. Los detalles concretos dependen de cada formato y podrá encontrarlos al analizar cada uno de ellos más detenidamente, por ejemplo con la información que encontrará Wotsit's Format.

Asignación de bloques de memoria

Para poder ejecutar un programa es necesario que su código y datos, al menos las secciones necesarias inicialmente, se encuentren en memoria RAM. Por ello el paso siguiente que da el sistema operativo, tras verificar que se trata de un ejecutable válido, es asignar los bloques de memoria necesarios dependiendo de la información recuperada de la cabecera.

En este apartado también existen diferencias, dependiendo del sistema operativo y el formato de ejecutable de que se trate.

En DOS se asume que el programa que va a iniciarse puede tomar el control total del sistema y, además, tiene un límite máximo de 64 kilobytes o un megabyte, dependiendo de que sea COM o MZ. En muchos casos, basta la asignación de un bloque de memoria para alojar la imagen del ejecutable, que se encuentra en el archivo, en memoria RAM e iniciar la ejecución.

Nota

En DOS no es posible iniciar la ejecución de un nuevo programa sin antes finalizar el que estuviese ejecutándose, a excepción de casos específicos como los controladores, programas residentes y similares.

Los sistemas que contemplan la ejecución simultánea de varias aplicaciones, como Linux y Windows, necesitan un mayor control de recursos que, como la memoria, debe ser compartida entre todas esas aplicaciones. Dependiendo de los recursos libres que haya, la asignación de la memoria necesaria para poner en marcha un nuevo programa puede implicar operaciones adicionales como el intercambio de páginas de memoria a un soporte externo, típicamente el disco duro.

Creación de un proceso

Asignado el bloque de memoria, y alojado el código y datos en dicho espacio, para poder iniciar la ejecución es preciso crear un proceso. Un proceso se compone de una serie de elementos que, dispuestos en memoria, indican al sistema operativo los atributos del código a ejecutar. Por regla general, también se crea un hilo de ejecución inicial asociado al proceso.

A partir de ese momento, tras preparar adecuadamente los registros del procesador que determinan las direcciones donde se encuentra el código, los datos y la pila, se inicia la ejecución del programa. El área de parámetros del proceso facilita datos como las variables de entorno asociadas a la aplicación, los descriptores de archivos, etc. Los detalles específicos dependen directamente del sistema que se trate. En DOS, por ejemplo, el área de parámetros asociada al proceso se conoce como PSP (*Program Segment Prefix*) y contiene datos como los indicados, además de los parámetros facilitados en la línea de comandos en el momento de invocar al programa.

En sistemas operativos monotarea, como es el caso de DOS, tras la creación del proceso se iniciará la ejecución del programa y no se devolverá el control al sistema hasta salir de dicho programa. Dicho en otras palabras: la aplicación toma el control exclusivo del sistema. Por el contrario, sistemas como Linux y Windows siguen manteniendo el control y ofrecen a las aplicaciones en ejecución porciones de uso del procesador, es decir, se va cambiando la ejecución de un proceso a otro con el fin de conseguir un funcionamiento aparentemente concurrente.

Nota

En los microprocesadores más modernos, que cuentan con cuatro e incluso con ocho núcleos, la ejecución en paralelo de varios hilos es una realidad y la multitarea no se basa exclusivamente en el uso compartido del procesador (técnica que es conocida como *time-sharing*). No obstante la asignación cíclica de tiempos de microprocesador a cada

hilo sigue estando presente ya que, por el momento, el número de procesos que ejecutan simultáneamente sistemas operativos como Windows y Linux es muy superior al número de núcleos integrados en los microprocesadores.

Configuración de registros

En este momento conocemos, aunque sea superficialmente, el trabajo que lleva a cabo el sistema operativo para llegar a iniciar la ejecución de un programa. Vamos ahora a bajar algo de nivel, entrando en los detalles de la configuración de registros del procesador para, a continuación, analizar el mecanismo de recuperación, descodificación y ejecución de instrucciones.

Tal como ya sabe, todos los microprocesadores de la familia x86 y compatibles, como son los Intel Pentium, Core y AMD Athlon, cuentan con un grupo de registros común. Parte de ellos son heredados de los primeros microprocesadores de dicha familia, los 8086/8088, y, por tanto, existen en todos los procesadores aparecidos desde entonces: 80286, 80386, 80486 y los distintos Pentium y compatibles.

Ya conoce todos los registros de uso general del procesador, a continuación nos centraremos exclusivamente en aquellos que, de una forma directa, se configuran al poner en marcha la ejecución de un programa. Lógicamente, el código de éste puede usar cualquier otro de los registros disponibles.

Segmentos de código, datos y pila

El microprocesador cuenta con una serie de registros, entre ellos CS (*Code Segment*), DS (*Data Segment*) y SS (*Stack Segment*), conocidos genéricamente como *registros de segmento* por su función primigenia, si bien en la actualidad, con sistemas de 32 y 64 bits como Windows y Linux, lo que contienen no son direcciones de segmentos sino descriptores. En cualquier caso, este detalle no nos interesa especialmente en este momento.

Las áreas de memoria en las que se ha alojado el código, el área de datos y que se usará para la pila del programa parten de una cierta dirección a la que podría denominarse *dirección base*. Dicha dirección es desconocida por el programa, de tal forma que es el sistema operativo el que decide dónde colocarlo según las áreas que ya estuviesen ocupadas. Es lo que se conoce como *reubicación del código*, un aspecto importante en sistemas operativos capaces de ejecutar múltiples tareas.

Es el sistema operativo, por tanto, el que asigna valor al registro CS, introduciendo en él la dirección base o dirección de memoria donde se encuentra la primera de las instrucciones del programa.

De igual forma, se establecen los valores para los registros DS y SS, si bien éstos suelen ser configurados por el propio programa una vez que es puesto en marcha. El primero determina el área de memoria en la que se encontrarán los datos del programa, mientras que el segundo aloja la dirección base de la zona dedicada a la pila.

El puntero de instrucción

Según acaba de decirse, el registro CS contendrá la dirección base del segmento o sección donde se encuentra el código del programa a ejecutar. Se necesita, además, un segundo registro que vaya indicando al procesador la instrucción que debe ejecutar a cada paso, ya que CS permanecerá durante todo el tiempo con esa dirección base. Aquí es donde entra en juego el registro IP (*Instruction Pointer*) o de puntero de instrucción.

El valor inicial del registro TP dependerá del formato del ejecutable y la información que contenga su cabecera. En cualquier caso, al iniciar la ejecución la pareja CS : IP determinará la posición exacta de memoria en la que se encuentra la primera instrucción a procesar. El procesador recupera la instrucción leyéndola de la memoria, la descodifica y, finalmente, la ejecuta, incrementando el valor del registro IP y repitiendo el proceso continuamente.

Nota

La notación `es : ip` indica una dirección de segmento y un desplazamiento sobre la dirección base. En este caso CS contiene la dirección base y IP el desplazamiento sobre ella, pero existen otras posibles combinaciones de *segmento'.desplazamiento*.

Base y puntero de la pila

Otros registros fundamentales, que deben configurarse al iniciar la ejecución de cualquier programa, son BP (*Base Pointer*) y SP (*Stack Pointer*). Ambos trabajan conjuntamente con el registro de segmento SS para definir la dirección base de la pila, que estaría en la dirección CS : BP, y la posición actual en la que están insertándose o recuperándose datos, que sería CS : SP. La pila es una estructura que crece en orden inverso, es decir, en principio CS contendrá el valor del final de pila, mientras que SP irá reduciéndose a medida que se inserten datos o incrementándose al extraerlos.

La pila es una estructura de datos LIFO (Lasí *In First Out*), de tal forma que el último dato introducido en ella es el primero que se recupera. Puede imaginarla como una pila de platos o cajas dispuestos unos sobre otros. El último plato puesto en la pila es el primero en salir, no es lógico tomar el primero que entró y que se encuentra bajo todos los demás. Ésta es la razón de que el crecimiento sea en orden inverso, para facilitar siempre la extracción en primer lugar del último dato introducido.

Existen dos usos básicos para la pila a la hora de programar en ensamblador: conservar el valor actual del registro IP cuando va a invocarse a una rutina o función que se encuentra en otra dirección, a fin de poder volver posteriormente al punto previo, y guardar temporalmente los valores que contienen uno o más registros. Más adelante, a medida que vayamos conociendo operaciones e instrucciones del lenguaje, sabrá más sobre el funcionamiento de la pila.

Acceso a los datos

Además de código y una zona de pila, la mayoría de programas cuentan, asimismo, con un área de datos. La dirección base de ésta se introduce en el registro de segmento DS y, en ocasiones, también se usa ES (*Extra Segment*). Como en el caso de CS y SS, son necesarios registros adicionales para acceder a los datos, ya que DS y SS sólo contienen la dirección de segmento, siendo necesario un desplazamiento o dirección dentro de ese segmento.

Los procesadores de la familia 8086 disponen de una serie de registros de uso general, entre los que se encuentran los registros BX (*Base Register*) y DX (*Data Register*). Las combinaciones DS : BX, DS : DX, ES : BX y ES : DX facilitan la lectura y almacenamiento de datos de la memoria. También existen algunos registros más específicos, como SI (*Source Index*) y DI (*Destination Index*), que también se usan conjuntamente con DS y ES para desplazar datos a y desde la memoria.

Nota

En este capítulo se está haciendo referencia a los registros de 16 bits que existen en todos los microprocesadores de la familia 8086, como IP, SP, BX o DX. Los actuales microprocesadores disponen de registros de 32 y 64 bits tal y como se describió en un capítulo previo. Su uso con los registros de segmento sería análogo.

Como ocurre con el espacio de la pila, la configuración de los registros de segmento para acceso a los datos suele configurarse en las primeras sentencias del programa, aunque ciertos formatos de ejecutable no precisan dicha operación.

* * *

Resumen

Al finalizar la lectura de este capítulo ya tenemos una idea de general del proceso que hace posible que un programa, alojado en un archivo, llega a ejecutarse, para lo cual es fundamental la intervención del sistema operativo. Éste determina las necesidades del ejecutable, sirviéndose de la información almacenada en su cabecera, y da los pasos de configuración necesarios para alojar código y datos en memoria y transferirle el control.

También ha conocido algunos detalles que podríamos considerar de más bajo nivel, como la configuración de los registros del procesador durante el proceso de carga de un programa, que es necesario conocer para entender su funcionamiento.

10

Registros
y memoria

El código de los programas, los datos con los que éstos trabajan, la pila de direcciones de retorno y muchos otros elementos esenciales a la hora de programar en ensamblador se almacenan temporalmente en la memoria del ordenador. De ahí tienen que pasar, en un momento u otro, al microprocesador para que éste pueda servirse de ellos y, en caso necesario, manipularlos. Con este fin, el procesador cuenta con pequeñas zonas de memoria interna conocidas como *registros*. Al operar sobre microprocesadores de la familia x86, dependiendo del sistema operativo para que programemos, podemos encontrarnos con varios modos de funcionamiento distintos que implican modelos de memoria diferentes. Los más importantes son los modos real y protegido, que llevan asociados un modelo de memoria segmentado y un modelo plano de 32 bits, respectivamente.

Nuestro objetivo, en este capítulo, es recordar algunos de los términos relacionados con la memoria: bits, bytes, palabras, etc., así como con las características más destacables de los modelos de memoria con los que podemos trabajar. También conoceremos las instrucciones necesarias para recoger datos de la memoria en un registro, llevarlos de un registro a otro o devolverlos a la memoria. Esto nos permitirá crear nuestros primeros programas en ensamblador poniendo en práctica lo que vayamos aprendiendo.

Unidades de información^

En un capítulo previo aprendimos a trabajar con números binarios y, en consecuencia, surgió la definición de qué es un bit. Un byte es un conjunto de ocho bits, sirviendo como la unidad básica en la que se mide la información. Cada una de las celdillas lógicas

en que se divide la memoria del ordenador, tanto RAM como ROM, tienen capacidad para un byte. En los sistemas actuales es habitual medir la información en gigabytes, aproximadamente mil millones de bytes, pero existen otras unidades intermedias también muy empleadas, como el kbyte o kilobyte y el megabyte.

Al hablar de procesadores es habitual medir su capacidad de direccionamiento en bits, al igual que la de los registros internos con que cuenta. Así, la mayoría de los procesadores actuales, entre los que se cuentan los compatibles x86 o los PowerPC de los Mac, son procesadores de 32/64 bits, ya que ése es el direccionamiento máximo para el que tienen capacidad.

Palabras y dobles palabras

Además del bit y el byte, son de uso habitual otras unidades de información como la palabra y la doble palabra. Una palabra son dos bytes, es decir, 16 bits, mientras que una doble palabra, como puede suponer, son cuatro bytes o 32 bits.

Al hacer mención a ciertos datos de un programa, o del sistema, es habitual indicar que ocupa una palabra o bien una doble palabra, en lugar de dos o cuatro bytes que sería el equivalente.

En realidad se define como palabra la unidad de medida cuyo tamaño coincide con el bus de datos del microprocesador y sus registros internos. Dado que el 8086 tiene un bus de 16 bits y registros de 16 bits, en este caso una palabra tendría esos 16 bits y una doble palabra 32 bits. Al operar con un microprocesador de 32 bits en modo protegido, sin embargo, una palabra serían 32 bits y no 16.

La existencia de estos tipos de datos se justifica por la presencia, en el microprocesador, de registros de distintos tamaños. Unos tienen capacidad para ocho bits, otros para dieciséis y otros para treinta y dos.

Existen, también, registros específicos, como los usados en las operaciones MMX/SSE de los microprocesadores más modernos, que trabajan sobre datos que ocupan 8 bytes (64 bits) o incluso 16 bytes (128 bits).

Para identificar a estos tipos de datos suele utilizarse la notación siguiente:

- **WORD:** Lo que conocemos como palabra.
- **DWORD:** Viene de *double word*, doble palabra.
- **QWORD:** *Quad Word*, o cuádruple palabra.

Los registros de uso general de los procesadores x86, los registros de segmento, el puntero de instrucción y el de indicadores, algunos de ellos los conocerá con más detalle más adelante en este mismo capítulo, tienen capacidad para una palabra, es decir, 16 bits.

Múltiplos del byte

Si para medir la capacidad de los registros internos del procesador se usan las unidades indicadas en el punto anterior, cuando se trata de indicar la capacidad de un bloque de memoria o de ciertos dispositivos, como los CD/DVD, discos duros y similares, se emplean siempre múltiplos del byte, la mayoría de los cuales seguramente ya conoce.

Iodos estos múltiplos, al operarse habitualmente con la base 2 en lugar de la base 10, usan como multiplicador no el millar sino 1024, que es el resultado elevar 2 a 10. Por ello, un kilobyte, o kbyte, no son 1000 bytes sino 1024 bytes. De manera análoga, un megabyte serían 1024 kbytes, un terabyte 1024 gbytes, etc.

Nota

Existen excepciones a esta norma en ciertos campos. Hay fabricantes, por ejemplo, que al indicar la capacidad de dispositivos como los discos duros asumen que un gabyte son mil millones de bytes, es decir, 1000X1000X1000 en vez de $1024 \times 1024 \times 1024$. Esto explica la diferencia que solemos encontrar entre la capacidad indicada por el fabricante y la que luego nos comunica el sistema operativo.

Al programar en ensamblador y efectuar ciertas tareas, como la reserva de bloques de memoria, lectura de sectores de un disco y similares, tenga en cuenta siempre que el factor de multiplicación es 1024.

Capacidad de direccionamiento

J ————— ^0. —————

Un factor de gran importancia, a la hora de programar para un cierto microprocesador, es su capacidad de direccionamiento o, dicho de otra manera, la cantidad de celdillas de memoria a las que puede acceder. Este factor determina la máxima cantidad de memoria a la que tiene acceso y, por tanto, el límite de datos y código que pueden emplearse.

La capacidad de direccionamiento de un microprocesador viene determinada directamente por el tamaño de los registros encargados de mantener las direcciones de memoria y su bus de direcciones. En el caso de los procesadores x86 éstos son los registros de segmento y otros que se usan como desplazamiento sobre ese segmento. Ciertos procesadores de 8 bits, como es el caso del conocidísimo Z80, cuentan con registros de 16 bits para indicar la sentencia a ejecutar o la posición en la pila, registros PC (*Program Counter*) y SP (*Stack Pointer*), respectivamente. Esto determina que la capacidad de direccionamiento es de 2 elevado a 16, es decir, 64 kbytes como máximo, cantidad de memoria habitual en los ordenadores que incorporaban este microprocesador.

Los primeros procesadores de la familia x86, concretamente el 8086 y 8088, eran microprocesadores de 16 bits, con registros de 16 bits para el direccionamiento. Esto, en principio, limitaría la cantidad de memoria a la que se puede acceder a 64 kbytes, lo

mismo que en los procesadores de 8 bits. Dicha cantidad, sin embargo, era insuficiente prácticamente desde la creación del primer PC, por lo que sería absurdo pensar en el uso de un microprocesador con semejante limitación. La solución a este problema está en la utilización combinada de dos registros, uno de segmento y otro de desplazamiento, para acceder a una mayor cantidad de memoria.

Registros de segmento

Los registros de segmento básicos, existentes en los primeros x86, son cuatro: CS, DS, ES, y SS. Son registros de 16 bits, como el resto de los registros del procesador, pero su uso interno, a la hora de componer direcciones, es un tanto especial. Estos registros no se emplean para acceder directamente a una cierta dirección de memoria, sino que definen una dirección base, o de segmento, sobre la que se aplicará un desplazamiento de 16 bits.

Nota

Los procesadores de 32 bits, 80386 y posteriores, cuentan, además de con los cuatro indicados, con dos registros de segmento adicionales: FS y GS. Éstos, al igual que ES, se emplean para acceder a segmentos de datos adicionales, mientras que ES se asocia al segmento de código, DS al principal de datos y SS al de pila.

No obstante, lo que obtenemos no es una dirección de 32 bits, ya que no se combinan los 16 bits del registro de segmento más los 16 del desplazamiento para crear una dirección completa de 32 bits, sino una dirección de 20 bits.

A la hora de efectuar el desplazamiento, el procesador toma el contenido del registro de segmento y desplaza sus bits hacia la izquierda cuatro posiciones, o lo que es lo mismo, multiplica su valor por 16.

El desplazamiento de 4 bits, sobre una base de 16, da como resultado una dirección de 20 bits en la que los cuatro primeros siempre están a cero. Con 2^{16} podemos direccionar exactamente 1048576 bytes o, lo que es lo mismo, un megabyte. Éste ha sido el límite teórico de memoria que han podido usar las aplicaciones en un PC durante muchos años, hasta la aparición del procesador 80386.

A pesar de ser procesadores de 16 bits, los 8086/8088 disponían de un bus externo de direcciones de 20 bits, mientras que el bus de datos era de 16. La conexión con la memoria, por tanto, contaba con 20 líneas para seleccionar una dirección y 16 líneas para leer o escribir el dato. Ésa es la razón de que sea necesario el uso de dos registros para poder componer la dirección.

Párrafos y segmentos

Al asignar un valor a un registro de segmento, por tanto, es necesario tener en cuenta que después, a la hora de componer la dirección, ese valor se multiplicará por 16. Lo que estamos asignando al registro, por tanto, no es un valor en bytes, sino en paquetes de 16 bytes. El valor que demos al registro DS, por ejemplo, se multiplicará por 16 antes de sumarse al de BX para acceder a una cierta celdilla de memoria. Desde ese punto, y ya que BX puede contener cualquier valor de 16 bits, es posible leer o escribir 65536 celdillas de memoria, es decir, 64 kbytes, como en los micros de 8 bits.

El conjunto de 16 bytes recibe el nombre de párrafo, mientras que el de 65536 bytes, o 64 kbytes, se conoce como segmento. Son dos unidades de medida más, como las palabras o los megabytes. El valor que se asigna a un registro de segmento siempre viene expresado en párrafos, no en bytes, mientras que el espacio de direcciones al que puede accederse partiendo del valor de un registro de segmento es, justamente, un segmento.

Si dividimos el megabyte de memoria al que tenemos acceso en los 8086/8088, así como en el resto de los procesadores de la familia x86 funcionando en modo real, en celdillas de un byte, tenemos justamente 1.048.576 celdillas, numeradas desde la 0 hasta la 1.048.575. Si en lugar de divisiones de un byte dividimos en celdillas de un párrafo, obtendríamos 65.536 celdillas, numeradas desde la 0 hasta la 65.535. La figura 10.1 muestra gráficamente las dos divisiones sobre la misma memoria física.

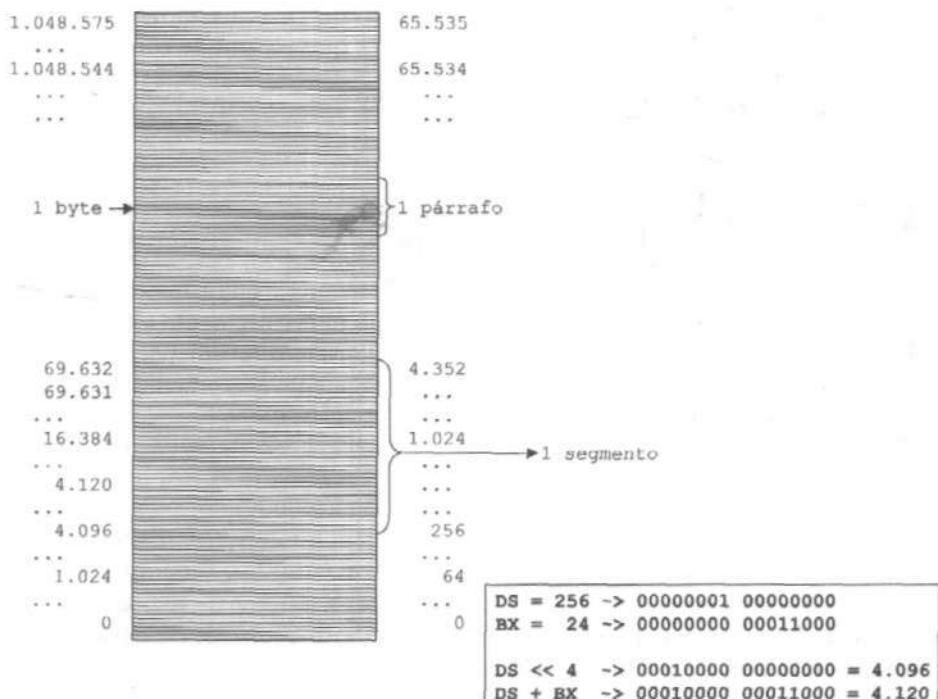


Figura 10.1. Representación de un megabyte de memoria contado en bytes y en párrafos.

Nota

Aunque es posible asignar 65.536 valores diferentes a un registro de segmento, esto no implica que puedan direccionarse otros tantos segmentos consecutivos ya que, como posiblemente haya notado, los segmentos en ocasiones se solapan unos a otros.

Asumiendo que diésemos al registro DS el valor 256 y al registro BX el valor 24, al componer DS : BX, para acceder a la memoria, estaríamos leyendo o escribiendo en la celdilla número 4.120 contando en bytes. Para llegar a dicho resultado hay que dar los pasos siguientes:

1. Tomar el valor del segmento, en este caso el de DS, y desplazarlo 4 bits a la izquierda. Si no estamos trabajando en binario, podemos multiplicar por 16 para obtener el mismo resultado que, en este caso, sería 4.096. Lo que hay en DS es un número de párrafo, el párrafo 256, que al convertirlo a bytes corresponde a la celdilla 4.096.
2. Sumar al valor de la operación anterior el que contenga el registro de desplazamiento, en este caso el 24 que hay en BX. Obtenemos la dirección 4.120.

Manteniendo el valor 256 en DS, y modificando el del registro BX, podríamos acceder a cualquier celdilla de memoria comprendida entre las direcciones 4.096 y 69.631, o, lo que es lo mismo, entre 4.096 y 4.096+65.535, ya que BX puede contener 65.536 valores distintos comprendidos entre 0 y 65.535. En resumen, podemos direccionar un segmento completo, 64 kbytes. En la siguiente celdilla de memoria, la 69.632 que equivale al párrafo 4.352, se iniciaría otro segmento.

Nota

Si necesitásemos acceder a cualquier dirección de memoria que estuviese fuera del rango 4.096-69.631, antes deberíamos seleccionar un nuevo segmento alterando el valor del registro DS. Sumando 1 a DS, por ejemplo, estaríamos accediendo al segmento de memoria comprendido entre 4.112 y 69.647, un segmento que prácticamente solapa al completo al anterior salvo en el primer párrafo.

Modelos de memoria

Ahora que conocemos, aunque sea básicamente, el sistema de segmentación que hace posible acceder a un megabyte de memoria empleando registros de 16 bits, veamos cuáles son los modelos de memoria con que nos podemos encontrar a la hora de crear una aplicación para un procesador de la familia x86, ya que de dicho modelo dependerá el mecanismo de direccionamiento y la cantidad de memoria a la que podamos acceder.

Los primeros procesadores x86, el 8086 y 8088, solamente contaban con un modo de funcionamiento, denominado posteriormente como *modo real*, que se caracteriza porque tan sólo puede accederse a un megabyte de memoria y, además, basándose en la segmentación por lo que, en la práctica, tan sólo son visibles en un momento dado, para una instrucción, 64 kbytes de memoria. A pesar de ello, en este modo real podemos distinguir dos modos de direccionamiento distintos: el modelo plano de 64 kbytes y el modelo segmentado.

El primero de ellos, el modelo plano de 64 kbytes, es característico de las aplicaciones ejecutables en formato COM del DOS.

Dicho formato, como ya se describiese en un capítulo previo, limita todo el programa, código, datos y pila, a un máximo de 64 kbytes, por lo que el valor de los registros de segmento se inicializa en la puesta en marcha del programa y, hasta su finalización, permanecen estáticos.

Es posible acceder a toda la memoria disponible mediante los registros de desplazamiento, como BX, sin tener que preocuparse de segmentos.

Las limitaciones del modelo plano de 64 kbytes son obvias, ya que no podemos acceder a más cantidad de memoria que esa. Cuando se necesita más memoria pero se dispone de un procesador 8086/8088, o es imperativo operar en modo real, la alternativa está en el uso de la segmentación.

La aplicación puede utilizar un segmento para el código, otro para la pila y múltiples segmentos de datos, rompiendo la barrera de los 64 kbytes y llegando hasta el megabyte.

Todos los procesadores posteriores al 8086/8088, desde el 80286 hasta los actuales Intel Core y AMD Phenom, cuentan, por compatibilidad, con el modo real citado antes. De hecho, y por mantener esa compatibilidad, cuando se inician estos procesadores activan el modo real y se comportan como un 8086 pero más rápido.

Por último, todos los procesadores de 32 bits de la familia x86, desde el 80386 en adelante, cuentan con un modo de funcionamiento denominado *modo protegido*. En él, estos procesadores dejan de comportarse como un 8086 y permiten al acceso a todas las posibilidades específicas de los 32 bits, entre ellas un modelo de memoria plano de 32 bits mediante el cual puede accederse hasta a 4 gigabytes de memoria, simplemente con la capacidad de direccionamiento de 32 bits de cualquier registro.

En este modelo de memoria no existen los segmentos, aunque sí que se emplean los registros de segmento con otros fines, concretamente para alojar unos descriptores que determinan la posición y naturaleza de cada sección lógica de la memoria.

Como es lógico pensar, los modelos de memoria *planos*, tanto el de 64 kbytes como el de 32 bits, son más simples, a la hora de direccionar datos, que el modelo segmentado. Éste, sin embargo, ha convivido con los programadores desde la aparición del PC hasta nuestros días, si bien cada vez se utiliza menos.

Nota

Los microprocesadores tipo X64, como los Core y Athlon 64, disponen de un modo de funcionamiento adicional denominado *Long*. Básicamente es un modelo plano como el de 32 bits, pero con una capacidad de direccionamiento de 64 bits.

Registros de uso general

La capacidad de direccionar, de componer una dirección que actúe como puntero a una cierta celdilla de memoria, tiene utilidad a la hora de escribir o leer datos que van desde la memoria al procesador o viceversa. En la mayoría de estas operaciones siempre se ve implicado alguno de los registros de propósito general que ya conoció básicamente en un capítulo previo.

Los microprocesadores anteriores al 80386 no cuentan con registros de 32 bits, por lo que EAX, EBX, ECX y EDX no existirían, pero sí AX, BX, CX, DX, AH, AL, BH, BL, CH, CL, DH y DL.

En una instrucción en ensamblador, tal y como veremos después, podemos utilizar los registros completos o bien parte de ellos. Para leer una doble palabra, por ejemplo, se utilizará EAX, pero si el dato a leer es una palabra usaremos entonces AX y si es un byte AL o AH.

Imagine los registros del procesador como variables que se almacenan en su interior, cada una de ellas con un tamaño fijo. AH, AL, BH, BL, CH, CL, DH y DL serían variables con capacidad para 8 bits, AX, BX, ex y DX para 16 bits y EAX, EBX, ECX y EDX para 32.

A pesar de ser registros de propósito general, y por tanto poder utilizarse en una variedad de situaciones, cada uno de ellos tiene un papel afín:

- AX: La A viene de *acumulador*, siendo el registro que se utiliza en la mayoría de operaciones aritméticas. Si bien éstas pueden efectuarse con otros registros, el micro tiene optimizaciones para éste en particular con dichas operaciones.
- BX: La B viene de *base*, empleándose como desplazamiento, conjuntamente con el registro de segmento DS, para acceder a direcciones de memoria.
- CX: La C viene de *contador*, utilizándose en la repetición de conjuntos de sentencias para contabilizar el número de ciclos y operaciones similares.
- DX: La D viene de *dato*, y se utiliza en ciertas operaciones de una forma conjunta con AX.

En lugar de AX, registro de 16 bits, también pueden emplearse AH, AL o EAX, dependiendo de la operación que esté efectuándose. Lo mismo es aplicable para los demás registros.

Asignación de valores

Al poner en marcha un programa, invocándolo desde el sistema operativo, ciertos registros tendrán valores concretos, predeterminados por la cabecera del ejecutable o el propio sistema, mientras que otros pueden contener valores aleatorios o indeterminados. En cualquier caso, e independientemente del programa que estemos desarrollando, una de las primeras necesidades que nos surgirá será la de asignar valores, ya sea a registros o a celdillas de memoria.

La instrucción ensamblador que nos interesa es mov, capaz de transferir información desde la memoria a los registros y viceversa, así como entre registros. El formato de esta instrucción es el siguiente:

```
mov destino,origen
```

El destino puede ser un registro o una posición de memoria, mientras que origen será un valor inmediato, un registro o una posición de memoria. Existen, no obstante, ciertas limitaciones en cuanto a las combinaciones posibles.

En el capítulo dedicado a los modos de direccionamiento podrá encontrar información más detallada sobre las posibles combinaciones de destino y origen y la forma en que operan.

Un principio básico, que hemos de tener en cuenta a la hora de emplear esta instrucción, es que destino y origen deben ser datos del mismo tamaño. No es posible, por ejemplo, asignar a AH un valor de 32 bits. Si destino es AX, por poner otro ejemplo, el origen no puede ser DH o EDX, ya que el tamaño de estos registros no concuerda con el de AX.

Valores inmediatos

En el ejemplo escrito en el segundo capítulo, al tratar las bases de numeración, asignábamos a un registro distintos valores.

Éstos no están contenidos en un registro ni se toman de una cierta celdilla de memoria, sino que forman parte de la instrucción ensamblador. Por esto se les llama valores inmediatos.

El destino de un valor inmediato puede ser un registro, lo más habitual, o bien una posición en memoria, pero no otro valor inmediato. Es decir, no podemos escribir una sentencia como:

```
mov 10, 24
```

En ella, 10 es un valor inmediato, no la dirección de una celdilla de memoria ni un registro, por lo que no es posible la asignación. Sí son válidas estas dos sentencias:

```
mov ah, 24
mov [10], 24
```

La primera asigna el valor inmediato al registro ah, mientras que la segunda lo introduce en la celdilla de memoria correspondiente a la dirección DS: 10.

riOi3

La notación mov [10] , 24 es incompleta, ya que el ensamblador no tiene forma de saber el tamaño del dato a transferir. Cuando en una instrucción mov uno de los operandos es un registro, se asume que el tamaño de la transferencia es el del registro. En ésta, sin embargo, tenemos una dirección de memoria y un valor inmediato.

Hay que tener en cuenta que este tipo de asignación no está permitida en ciertos casos. No es posible, por ejemplo, asignar un valor inmediato a un registro de segmento:

```
mov ds, ObSOÜh
```

Esta sentencia provocaría un error de ensamblado. La solución pasa, como probablemente habrá deducido a partir del código de ejemplos previos, por llevar el valor inmediato a un registro de propósito general y, desde ahí, al registro de segmento:

```
mov ax, 0b800h
mov ds, ax
```

El valor inmediato puede contener operaciones aritméticas, siempre que el valor resultante sea una constante. La operación, en realidad, se calcula durante el ensamblado, de tal forma que en el código ejecutable lo que aparece es el resultado y no la operación.

Asignación entre registros

Los valores inmediatos se recuperan de la memoria conjuntamente con las instrucciones a medida que éstas se ejecutan, por lo que no es necesario un acceso adicional a memoria para recuperarlos. Cuando la instrucción se ejecuta, el valor inmediato ya se encuentra en el interior del procesador.

Otro tanto ocurre cuando el valor a transferir se encuentra alojado en un registro y el destino es otro registro. En este caso el código producido por la instrucción ensamblador

indica el destino y el origen, no existiendo valores independientes y siendo, por tanto, la transferencia más rápida que puede ejecutarse.

Siempre que los dos registros, destino y origen, sean del mismo tamaño, es posible prácticamente cualquier asignación. Podemos asignar un registro de segmento a uno de propósito general o viceversa, así como asignar registros de propósito general a otros del mismo tipo. Hay, no obstante, ciertas excepciones. No es factible, por ejemplo, la alteración directa de registros que, como IP, determinan la localización del código a ejecutar.

En el mismo punto anterior puede ver un ejemplo de asignación entre registros, actuando DS como destino y AX como origen. En este caso ambos registros son de 16 bits, por lo que la transferencia se ejecuta sin problemas. No podríamos, por el contrario, sustituir AX por AH o AL, ya que en ese caso el tamaño de los operandos no coincidiría.

Lectura de datos de la memoria

En el interior del microprocesador sólo es posible mantener una pequeña cantidad de información, alojada en los registros de propósito general, por lo que es indispensable almacenar todos los datos en memoria, generalmente en una sección o segmento específicos, recuperándolos cuando se necesitan. En estos casos el destino de la transferencia sería un registro del procesador, mientras que el origen siempre sería la dirección de una celdilla de memoria.

La dirección de la celdilla a leer, o de la primera a leer en caso de que se vaya a recuperar más de un byte, puede facilitarse como una constante o bien obteniéndola de un registro. En cualquier caso, esa dirección actuaría como desplazamiento en caso de estar usando el modelo de memoria segmentado.

A la hora de leer datos de la memoria, y en general cuando se hagan transferencias de datos, hay que poner un especial cuidado con la notación empleada. Suponga que quiere recuperar, en el registro AH, el byte almacenado en la celdilla 100 de memoria. Si escribiésemos la sentencia siguiente, lo que asignaríamos a AH sería el valor 100, no el contenido de la celdilla de memoria 100.

```
mov ah, 100
```

Cuando lo que necesitamos es interpretar un cierto valor como puntero a una celdilla de memoria, es preciso introducirlo entre corchetes. Por ejemplo:

```
mov ah, [100]
```

En vez de facilitar la dirección como una constante, lo cual es adecuado sólo si conocemos la posición de memoria donde se encuentra el dato a leer y, además, dicha posición es fija, también podemos usar el contenido de un registro. Éste, como es lógico, se habrá establecido previamente tras las comprobaciones necesarias. Un ejemplo podría ser:

```
mov bx, 100
```

```
mov ah, [bx]
```

La ventaja es que el valor de BX puede haberse obtenido a partir de un cálculo o una referencia marcada con una etiqueta, según veremos después.

En ambos casos, tanto si la dirección es una constante como si se aloja en un registro, se asume que la dirección estará compuesta de un registro de segmento más la dirección indicada. Los dos ejemplos anteriores, asumiendo que se emplea el modelo de memoria segmentado, accederían a la dirección DS : 100 y DS : BX, respectivamente, ya que el registro DS es el que se asume por defecto con direcciones constantes y el registro BX. De tener el segmento en otro registro, distinto a DS, sería necesario indicarlo explícitamente, por ejemplo:

```
mov ah, [es:bx]
```

Caso en el que se usaría como selector de segmento el valor de ES en vez del de DS.

Escritura de datos en la memoria

Como puede suponer, para escribir datos en la memoria hay, básicamente, que intercambiar destino y origen. El destino sería una dirección de memoria, ya sea constante o alojada en un registro, mientras que el origen podría ser tanto un valor inmediato como el contenido de un registro. Como en el caso anterior, no hay que olvidar los corchetes cuando lo que se quiere es emplear un operando, en este caso el destino, como puntero a una celdilla de memoria.

Si el origen es un registro, éste determinará el tamaño del operando y, por tanto, la cantidad de información a transferir a la memoria. En caso de ser un valor inmediato, sin embargo, esa información no está disponible, siendo necesario indicarla de manera explícita. La forma de hacerlo dependerá del ensamblador usado. Con NASM, por ejemplo, la notación sería como la siguiente:

```
mov word [bx], 24          ?*
```

Observe la palabra `word` justo detrás de la instrucción `mov`. Ella es la que indica el tamaño del operador y, además de `word`, también podría ser `byte` y `dword`. En este caso se indica que el valor de origen, 24, se alojará como una palabra, ocupando dos bytes, en la dirección de memoria apuntada por `bx`.

Si utilizáramos MASM, tras el indicador de tamaño habría que añadir `ptr`, quedando la instrucción así:

```
mov word ptr [bx], 24
```

En caso de que el valor esté almacenado en un registro, por ejemplo AX, sería redundante indicar el tamaño del valor a transferir:

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

Siendo suficiente con:

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

Recuerde que la sentencia anterior no es equivalente a:

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

El primer caso es una asignación entre registros, concretamente se transfiere una copia del valor de AX a BX. En el segundo, BX recibe la palabra alojada en la dirección de memoria que indica AX, mientras que la tercera sentencia es, supuestamente, una transferencia desde una celdilla de memoria a otra, pero se trata de una combinación no válida que generaría un error de ensamblado.

Definición de datos en el programa

Leer y escribir en celdillas de memoria aleatoriamente, sin saber exactamente qué es lo que hay en esas celdillas y para qué utiliza el sistema su contenido, no es una práctica aconsejable. De hecho, podemos alterar el funcionamiento del programa o el ordenador e, incluso, bloquearlo.

En la mayoría de los casos, las posiciones de memoria empleadas por los programas corresponden a su propio segmento de datos, en el que se habrán definido previamente los elementos que se necesiten.

En cierta manera, esta definición sería similar a la declaración de variables de los lenguajes de alto nivel, si bien lo único que se indica es el tamaño del dato y su valor inicial, sin tipo ya que los tipos no existen en lenguaje ensamblador.

Campos simples

La sintaxis para definir campos en el segmento de datos depende en parte del ensamblador usado, si bien casi siempre guardan cierta similitud. Ciftemosnos a NASM, que es el que hemos usado y vamos a seguir utilizando en la mayoría de los ejemplos, dicha sintaxis sería:

```
identificador dbtdwId valor
```

El identificador es una etiqueta o nombre que vamos a asignar a esa posición en el segmento de datos, facilitando así las referencias posteriores. La mayoría de los ensambladores, entre ellos NASM, no distinguen entre mayúsculas y minúsculas, por lo que puede usar cualquier combinación en los identificadores tanto en la definición como en las posteriores referencias.

A continuación aparecerá uno de los tres indicadores: db (*define byte*), dw (*define word*) o dd (*define double word*), especificando el tamaño asociado al identificador. Esto hará que el ensamblador reserve en el segmento de datos uno, dos o cuatro bytes, de tal manera que el siguiente identificador que se defina estaría en la celdilla de memoria siguiente, dos celdillas más adelante o cuatro, según el caso.

Por último, facilitaremos el valor inicial que se almacenará en ese espacio. Podemos usar números o caracteres delimitados entre comillas simples, siempre sabiendo que lo que se almacenará será el código numérico de los caracteres, es decir, su representación en memoria.

A continuación tiene tres ejemplos en los que se define un identificador para acceder a un byte, otro para una palabra y un tercero para una doble palabra:

```
Asterisco db '*'
Posición dw 3280
Valor32 dd 0
```

Conjuntos de campos

Cuando se necesitan múltiples campos de un mismo tamaño y asociados entre sí, lo que en otros lenguajes de programación se conoce como matrices, arreglos o vectores, tenemos varias opciones. Por una parte, podemos asociar cualquiera de los indicadores utilizados en el punto anterior, db, dw o dd, con el repetidor times de NASM. Éste precede al indicador y va seguido de un entero que comunica el número de veces que se repetirá. Por ejemplo:

```
Caracteres times 2b6 db '?'
```

Esta línea de código reservaría 256 bytes de memoria, ya que hemos usado db, introduciendo en cada uno de ellos el carácter ? y asociando al identificador Caracteres la dirección del primero de ellos.

Si no necesitamos dar un valor inicial a cada uno de los bytes reservados, podemos usar la sintaxis siguiente:

```
Caracteres resb 256
```

Además de resb, para reservar bytes, también podemos utilizar resw y resd para reservar un cierto número de palabras o dobles palabras.

Nota

En otros ensambladores, como MASM y TASM, no existe ni times ni resb/resw/resd, utilizándose, en su lugar, la sintaxis db 256 dup (?) para reservar el mismo espacio de 256 bytes. Si se le desea dar un valor inicial, basta con sustituir el carácter ? que aparece entre paréntesis por dicho valor, que será un número o un carácter entrecomillado.

Referencias al segmento de datos

Una vez definidos los campos que se precisen, para acceder a ellos desde el código del programa es necesario componer una dirección. El primer paso será asignar a un registro de segmento, generalmente DS, la dirección del segmento de datos, lo cual podemos hacer de diferentes maneras dependiendo del ensamblador utilizado.

La mayoría de los ensambladores contemplan el uso de la palabra seg para obtener el segmento correspondiente a una variable, de tal forma que podríamos hacer lo siguiente:

```
mov ax, seg Caracteres  
mov ds, ax
```

Al definir un segmento, en el caso de NASM con la palabra segment, normalmente se asigna a éste un nombre. Ese identificador, siempre asumiendo que trabajamos con NASM, tiene atributo de segmento, por lo que podríamos hacer también lo que se muestra a continuación:

```
mov ax, Datos ; Datos seria el nombre del segmento  
mov ds, ax
```

Esta sintaxis, sin embargo, no es posible en otros ensambladores.

Teniendo el registro de segmento adecuadamente establecido, el acceso al área de memoria se haría con la sintaxis ya explicada en puntos previos. Debe recordar siempre la diferencia que hay entre:

```
mov ax, Caracteres
```

Que asigna al registro AX la dirección que ocupa el identificador Caracteres en el segmento de datos, con:

```
mov ax, [Caracteres]
```

En cuyo caso, lo que se asigna a AX es el contenido de la celdilla cuya dirección de memoria apunta el identificador Caracteres.

Un ejemplo

Para finalizar este capítulo, y poner en práctica algunas de las posibilidades de la instrucción mov, nos serviremos de un pequeño ejemplo. Lo compilaremos con NASM en DOS y habrá que ejecutarlo en dicho sistema en modo texto o, en su defecto, en una ventana de consola de Windows o un emulador DOS en Linux.

El objetivo del programa es mostrar en la pantalla de texto dos caracteres, en diferentes posiciones y con colores distintos. Para ello es necesario saber en qué dirección comienza el área de memoria dedicada al contenido de la pantalla, así como la estructura con la que se almacena dicho contenido.

Todos los ordenadores que cuentan con un adaptador de gráficos que permite textos en color, y ejecutan el sistema operativo DOS, colocan el contenido de la pantalla, caracteres y atributos, en el segmento B800h. La notación hexadecimal es la más habitual a la hora de hablar de segmentos. A partir de la dirección 0 de dicho segmento existe una palabra por cada posición de pantalla, conteniendo uno de los bytes el código de carácter y el otro el atributo asociado: color de fondo y tinta.

Operando en el habitual modo de 25 líneas por 80 columnas cada una, obtendríamos que cada línea ocuparía 160 bytes, 2 por posición, y que la pantalla completa serían 4000 bytes. De esos bytes, el primero, en la posición 0 del segmento, contendría el código del carácter correspondiente a la columna 1 de la línea 1, mientras que el byte siguiente alojaría el atributo de dicho carácter. En el byte 2 tendríamos el código del carácter que aparece en la columna 2 de la misma columna, en el byte 4 el del carácter de la columna 3 y así sucesivamente.

Sabiendo que cada fila ocupa 160 bytes y cada columna 2 bytes, es fácil calcular dónde hay que introduce un valor para hacer aparecer un carácter en una cierta posición de la pantalla. La fórmula sería fila*160+columna*2 si fila y columna se cuentan partiendo de cero.

En cuanto al byte de atributo asociado a cada carácter, está estructurado en tres partes: color del carácter, en los bits 0 a 3; color de fondo, en los bits 4 a 6, e indicador de parpadeo, en el último bit. Codificándolo en hexadecimal, el dígito de la derecha indicaría el color del carácter, de 0 a F, y el de la izquierda el de fondo y el parpadeo.

Conociendo estos parámetros, observe el código mostrado a continuación correspondiente al programa de ejemplo.

```
; Definimos el segmento de datos
segment Datos

; definiendo varios campos
Asterisco db '*'
Blanco db 0f0h ; fondo blarrfj^-
Posición dw 3280 ; 1:20,c:40

; Segmento para la pila
segment Pila stack
    resb 256
Tnini oPila:

; Sgymynlo de código
segment Código
..start:

; inicializamos ds
mov ax, Datos
; para acceder a los datos
mov ds, ax
; preparamos el registro es
; para acceder al segmento
; donde está el contenido de
; la pantalla
```

/>
/ /

```

mov ax, 0b800h
mov es, ax

; recuperamos en AL el
; valor que hay en Asterisco
mov al,[Asterisco]

; en AH el color
mov ah,[Blanco]

; y en BX la posición
mov bx,[Posición]

; transferimos el contenido
; de AX a la dirección ES:BX
mov [es:bx], ax

; escribimos directamente en
la pantalla un valor inmediato
mov word I es : 5*160 i 3b*2 , Q0a41h

; salimos al sistema
mov ah, 4ch
int 21h

```

En el segmento de datos se han definido varios campos: `Asterisco` es un byte conteniendo un asterisco, `Blanco` otro byte conteniendo un atributo y `Posición` una palabra con el valor obtenido de la expresión $20*160 + 40*2$. A continuación se encuentra el segmento de pila, que no llegamos a utilizar.

Las primeras instrucciones se ocupan de preparar dos registros de segmento: DS y ES. El primero apuntará al segmento de datos del programa, mientras que al segundo asignamos el valor `0b800h`, consiguiendo así que apunte al área donde se almacena el contenido de la pantalla.

En los pasos siguientes se asigna a AL el valor del campo `Asterisco` y a AH el de `Blanco`. Esta asignación no es arbitraria, hay que tener en cuenta que AL será la parte del registro AX que se escriba en el primer byte de la palabra, mientras que AH lo hará en el siguiente.

Recuerde que los microprocesadores x86 utilizan la representación *little endian* descrita en el segundo capítulo, por lo que al escribir una palabra, como es el contenido del registro AX, el byte de menor peso se colocará en la primera celdilla de memoria (a la que corresponde una dirección más baja) y la de mayor peso en la siguiente.

Acto seguido recuperamos en BX la dirección correspondiente a la posición donde queremos poner el asterisco, obteniéndola del campo `Posición`. La visualización efectiva tiene lugar al ejecutarse la instrucción `mov [es : bx] , ax`.

Observe que hemos puesto ES : delante del registro BX. Esto es necesario porque, por defecto, al usar BX como puntero de una celdilla se asocia con el registro DS. Con esta notación indicamos explícitamente que el segmento está en ES.

La última sentencia, antes de las dos que devuelven el control al sistema, efectúa prácticamente el mismo trabajo de las anteriores pero de una forma mucho más directa. La dirección dentro del segmento se calcula como desplazamiento, mientras que el carácter y el atributo se facilitan como un valor inmediato.

Después de ensamblar y enlazar el programa, con NASM y ALINK según los pasos descritos en capítulos anteriores, la ejecución generaría un resultado como el de la figura 10.2. Fíjese en el asterisco negro sobre fondo blanco, en la parte inferior de la pantalla, y en la "A" que, aunque en la imagen no se aprecia, aparece en color verde sobre fondo negro.

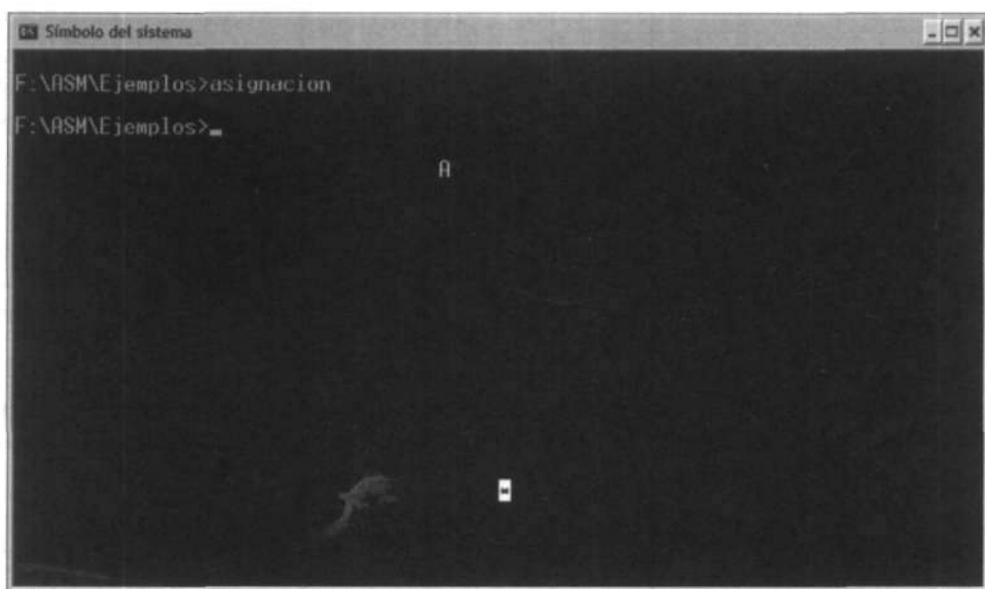


Figura 10.2. Resultado de la ejecución del programa.

Resumen

Aunque tan sólo hemos conocido una instrucción ensamblador, la instrucción mov, ésta nos ha servido para familiarizarnos algo más con aspectos como la capacidad de direccionamiento que tiene el procesador, los modelos de memoria que podemos encontrar en los procesadores de la familia x86 o los registros de segmento y algunos de los de propósito general.

De especial importancia resulta el mecanismo de segmentación utilizado en uno de los modelos de memoria, haciendo posible el acceso a un megabyte de memoria con ^

procesadores de 16 bits que, teóricamente, estarían limitados a 64 kbytes. La segmentación ha sido durante muchos años un quebradero de cabeza para los programadores, no siendo necesaria a la hora de crear aplicaciones para sistemas de 32 bits como Windows y Linux. Sigue siendo, por el contrario, totalmente imprescindible a la hora de operar en DOS.

Por último se han descrito algunas de las variantes posibles de la instrucción mov, transfiriendo datos entre registros, de la memoria al procesador y viceversa. El ejemplo aportado, no especialmente útil ni llamativo, puede servir como base, en posteriores capítulos, a medida que conozcamos otras instrucciones x86.

11

Depuración

Como programador, sabrá que el proceso de prueba y depuración de una aplicación es indispensable como parte del desarrollo, no sólo para detectar fallos puntuales que pudieran surgir en esa fase sino, además, como verificación de que el programa hace, efectivamente, lo que tiene que hacer. Ésta es una necesidad que se agudiza al programar en ensamblador, ya que cualquier proceso, por simple que pueda ser, requiere muchas instrucciones, siendo imprescindible comprobar cada paso y el resultado global.

En los capítulos siguientes vS "a conocer diversas instrucciones ensamblador que, por ejemplo, le permitirán efectuar cálculos y saltos de un punto a otro de un programa. En muchos casos, la única manera de ver el resultado del programa será observando los valores que toman los registros del procesador a medida que se ejecuta paso a paso, tarea para la cual precisaremos una herramienta de depuración.

Al estudiar un ejemplo de capítulos previos, concretamente el dedicado a la representación de información, utilizábamos DEBUG para desensamblar el código del programa y ver las instrucciones que contenía. Igualmente podríamos haberlas ejecutado paso a paso, viendo el contenido de los registros.

Para ello tan solo necesitamos conocer los comandos de di^ha herramienta, que facilita la visualización del contenido de los registros del microprocesador, el contenido de la zona de memoria que nos interese, etc.

Aunque las posibilidades de DEBUG son bastante básicas, con ellas tendremos suficiente en la mayoría de los casos. Es por ello que en este capítulo nos centraremos, principalmente, en el estudio de esta herramienta, teniendo también en cuenta que la mayor parte de los ejemplos desarrollados en el libro se ejecutarán sobre DOS. Existen, no obstante, otras herramientas de depuración, como GRDB (*Get Real Debugger*), para DOS,

o gdb, para Linux, con mayores posibilidades. También encontrará, en lugares como <http://www.thefreecountry.com/developercity/debuggers.shtml>, depuradores e interfaces gráficas para depuradores basados en texto, tanto para DOS como para Linux y Windows.

Algunas de las ventajas de esos otros depuradores, respecto a DEBUG, es que permiten establecer puntos de parada en ciertas posiciones de memoria, facilitando así la ejecución del programa hasta una cierta posición y, a partir de ahí, observar el estado de los registros o ejecutar paso a paso, según interese.

Puesta en marcha del depurador

Al igual que N ASM y ALTNK, el ensamblador y enlazador que estamos usando en los ejemplos desarrollados en capítulos previos, DEBUG es una herramienta que se utiliza desde la línea de comandos del DOS, o bien desde una ventana de consola de Windows. En cualquier caso, y asumiendo que nos encontraremos en el directorio del programa a depurar, lo primero que deberemos tener en cuenta es que DEBUG se encuentra en el camino de búsqueda del sistema, es decir, está en la variable PATH.

Si sabemos, en el momento de invocar a DEBUG, el nombre del ejecutable que vamos a depurar, podemos facilitarlo como parámetro. Hay que tener en cuenta, no obstante, que DEBUG es una aplicación DOS y, como tal, solamente reconoce nombres de archivo que estén formados por un nombre de un máximo de ocho caracteres y una extensión de tres. Esto es especialmente importante si estamos trabajando en alguna versión de Windows en una consola *dí* sistema, ya que se permiten nombres largos que, posteriormente, DEBUG no reconocería.

En cualquier caso, tanto si hemos facilitado el nombre del programa a ejecutar como si no, DEBUG mostrará el carácter indicador de que está esperando nuestras órdenes: un guión. Los comandos se componen siempre de un carácter que, opcionalmente, puede ir seguido de parámetros, según los casos. Para salir de DEBUG, volviendo a la línea de comandos de DOS, simplemente introduzca el comando q y pulse **Intro**.

Nombres de archivos DOS

Uno de los problemas que encontrará de forma habitual con DEBUG, especialmente si el sistema en el que lo usa opera con alguna versión de Windows y edita los archivos con el Bloc de notas usando nombres largos, es la imposibilidad de abrir esos archivos facilitando su nombre completo.

La solución, como se apuntaba antes, es emplear el nombre corto, compuesto de un máximo de 8 caracteres, un punto y una extensión.

El problema es que ese nombre corto no aparece por defecto con las habituales herramientas de Windows, ni tampoco al usar el comando DIR sin opciones. No obstante, basta con añadir la opción /x a dicho comando para obtener, como se aprecia en la parte superior de la figura 11.1, tanto los nombres largos como los cortos. Éstos se componen siempre de los primeros caracteres del nombre largo, un carácter ~ y un número. El carácter ~ puede obtenerlo manteniendo pulsada la tecla Alt mientras escribe 126 utilizando el área numérica o bien con la combinación AltGr-4.

```
Z:\Libros\Actuales\ProgramacionEnsamblador\CD-ROM\Ejemplos\10>dir /x
El volumen de la unidad Z es Datos
El n&umero de serie del volumen es: 7A4C-AC38

Directorio de Z:\Libros\Actuales\ProgramacionEnsamblador\CD-ROM\Ejemplos\10

16/04/2003 18:06    <DIR>
16/04/2003 18:06    <DIR>
05/11/2002 19:56      1.182 ASIGNA~1.ASM Asignacion.asm
05/11/2002 19:55      387 ASIGNA~1.EXE asignacion.exe
05/11/2002 19:55      195 ASIGNA~1.OBJ asignacion.obj
              3 archivos   1.764 bytes
              2 dirs   66.714.734.592 bytes libres

Z:\Libros\Actuales\ProgramacionEnsamblador\CD-ROM\Ejemplos\10>debug asignacion.e
xe
File not found

-q

Z:\Libros\Actuales\PROGRA~1\CD-ROM\Ejemplos\10>debug asigna~1.exe
-
```

Figura 11.1. DEBUG sólo acepta nombres cortos para los archivos.

La misma figura nos muestra la respuesta de DEBUG al intentar abrir el programa Asignacion.exe, desarrollado como ejemplo en el capítulo previo, usando el nombre largo y el corto. Observe cómo en el primer caso se indica la imposibilidad de encontrar el archivo.

Apertura desde DEBUG

Si hemos iniciado DEBUG sin facilitar parámetro alguno, en el momento en que deseemos iniciar la depuración de un programa habrá que recuperarlo desde el archivo en el que se encuentre. Para ello deberemos dar dos pasos:

- Facilitar el nombre del archivo, recordando que debe ser el nombre corto, utilizando el comando n. Bastará con introducir, tras el indicador de DEBUG, el comando n y, tras un espacio, el nombre del archivo.
- Recuperar el contenido del archivo empleando el comando 1.

En la figura 11.2 puede ver los dos pasos. Primero se utiliza el comando u para ver las instrucciones que hay en memoria, tras lo cual facilitamos el nombre del programa y lo recuperamos, repitiendo el comando u a fin de comprobar que ahora tenemos las instrucciones del programa escrito como ejemplo al final del capítulo previo.

```

Símbolo del sistema - debug
Z:\Libros\Actuales\PROGRA~1\CD-ROM\Ejemplos\10>debug
-u
17D0:0100 5A          POP    DX
17D0:0101 0000        ADD    [BX+SI], AL
17D0:0103 1E          PUSH   DS
17D0:0104 8800        MOV    [BX+SI], AL
17D0:0106 0000        ADD    [BX+SI], AL
17D0:0108 41          INC    CX
17D0:0109 53          PUSH   BX
17D0:010A 49          DEC    CX
17D0:010B 47          INC    DI
17D0:010C 4E          DEC    SI
17D0:010D 41          INC    CX
17D0:010E 7F31        JLE    0141
17D0:0110 CD20        INT    20
17D0:0112 FF9F009A    CALL   FAR [BX+9A00]
17D0:0116 F0          LOCK
17D0:0117 FE1D        CALL   FAR [DI]
17D0:0119 F0          LOCK
17D0:011A 4F          DEC    DI
17D0:011B 0334        ADD    SI,[SI]
17D0:011D 00BF1734    ADD    [BX+3417], BH
-n ASIGNA~1.exe
-1
-u
1802:0000 B8F117      MOV    AX,17F1
1802:0003 8ED8        MOV    DS,AX
1802:0005 B800B8      MOV    AX,B800
1802:0008 8EC0        MOV    ES,AX
1802:000A A00000      MOV    AL,[0000]
1802:000D 8A260100    MOV    AH,[0001]
1802:0011 8B1E0200    MOV    BX,[0002]
1802:0015 26          ES:
1802:0016 8907        MOV    [BX],AX
1802:0018 26          ES:
1802:0019 C7066603410A MOV    WORD PTR [0366],0A41
1802:001F B44C        MOV    AH,4C
-
```

Figura 11.2. Recuperamos el código del programa tras haber iniciado DEBUG.

Análisis del programa

Tras recuperar desde DEBUG el programa que pretendemos depurar, una de las primeras acciones que podemos llevar a cabo es el análisis del código de éste. Simplemente observando las sentencias, y el resto de datos que muestra DEBUG al desensamblarlas, podemos comprender muchas cosas. Ya sabe que el comando para desensamblar las instrucciones alojadas en memoria es u.

Por defecto, se desensamblan las primeras instrucciones a las que apunten las pareja de registros CS : IP. En la figura 11.2, concretamente en su parte inferior, puede ver que

no aparecen todas las instrucciones del programa, sino sólo las que hay en los primeros 32 bytes de memoria a partir de la dirección de IP. Para desensamblar el contenido de cualquier otra posición de memoria, puede interesarnos en algunos casos, no hay más que indicar dicha dirección tras el comando u.

Debe tener en cuenta que todos los datos numéricos introducidos en DEBUG se interpretan en hexadecimal. Al facilitar una dirección puede introducir simplemente un desplazamiento, que se emplearía con el registro de segmento que corresponda, o bien una dirección completa en la forma segmento:desplazamiento.

Si además de una dirección de inicio, como primer parámetro, facilitamos también una de fin, como segundo, el comando u desensamblará todas las instrucciones que haya entre ambas direcciones. Puede comprobarlo introduciendo, tras haber abierto el programa Asignacion.exe en DEBUG, el comando u 0 24. En este caso sí que verá todas las instrucciones del programa, además de una al final que no pertenece a éste.

Direcciones, instrucciones y código máquina

Al desensamblar las instrucciones que hay en una cierta porción de memoria, mediante el comando u, DEBUG lee bytes de código máquina y, a partir de ellos, reconstruye las instrucciones ensamblador. Hay que tener en cuenta que DEBUG no conoce el código fuente, que nosotros hemos escrito y alojado en un archivo .asm, y, por tanto, sí conoce ciertos elementos que se pierden durante el ensamblado, como son los comentarios y etiquetas.

X

Asumiendo que hemos iniciado DEBUG y abierto el programa Asignacion.exe, y teniendo el código ensamblador que habíamos escrito en el capítulo previo en pantalla, observemos el resultado que nos ofrece DEBUG al desensamblar ese código:

1802:0001 R8D615	MOV	AX,17F1
1802:0003 8ED8	MOV	DS,AX
1802:0005 B800B8	MOV	AX,B800
1802:0008 8ECÜ	MOV	ES,AX
1802:000A A00000	MOV	AL,[0000]
1B02:000D 8A260100	MOV	AH,[0001]
1802:0011 8B1E0200	MOV	BX,[0002]
1802:0015 26	ES :	N
1802:0016 8907	MOV	[BX],AX
1802:0018 26	ES :	
1802:0019 C70666034 1 0A	MOV	WORD PTR [0366],0A41
1802:001F B44C	MOV	AH,4C
1802:0021 CD21	INT	21

Podemos distinguir claramente tres columnas, que en el listado anterior se han distinguido de la siguiente manera:

- La primera, que aparece en cursiva, indica la dirección de memoria de la que se ha leído el código.
- En segundo lugar tenemos los bytes que hay a partir de esa zona de memoria, las instrucciones en código máquina ejecutable directamente por el procesador.
- Por último, en negrita, tenemos la conversión a ensamblador que efectúa DEBUG a partir del código máquina de la segunda columna.

Nota

El valor asignado inicialmente al registro es no será siempre el mismo, sino que variará según el punto en el que el sistema operativo decida cargar el programa. Lo mismo ocurrirá con el segmento de datos. No se preocupe, por tanto, si al abrir el ejecutable con la herramienta DEBUG en su sistema ver direcciones completamente distintas.

La dirección se expresa en el formato segmento:desplazamiento, siendo 18 02 el valor que tiene el registro CSy 0000 el desplazamiento dentro de ese segmento donde se encuentra la primera instrucción. El valor de CS se establece automáticamente al abrir el programa, al igual que el de IP que, en este caso, es 0 porque el programa es un ejecutable con múltiples segmentos y cuenta con un segmento específico para el código.

Al recuperar el primer byte existente en la dirección indicada por CS : IP, DEBUG descodifica para saber qué instrucción es y, consecuentemente, cuántos bytes adicionales le acompañan. Observe en la segunda columna que hay instrucciones de 1, 2, 3, 4 y hasta 6 bytes. Tras descodificar una instrucción completa, obteniendo la representación en ensamblador que aparece en la tercera columna, DEBUG avanza hasta la siguiente posición de memoria, tras incrementar su contador interno en el número de bytes leídos previamente, y repite el proceso.

Nota

Tras ejecutar un comando u seguido de una dirección, los sucesivos comandos u sin dirección que podamos introducir irán avanzando en la memoria, desensamblando bloques de instrucciones consecutivos.

Al iniciar el desensamblado es fundamental conocer con certeza la dirección de memoria en la que se encuentra la primera instrucción, ya que si se parte desde una dirección incorrecta la interpretación de las instrucciones será totalmente errónea por parte de DEBUG.

Puede comprobarlo introduciendo el comando u 1, solicitando así el desensamblado a partir de la dirección CS : 1. El problema es que lo que hay en esa dirección es el segundo byte de la instrucción mov ax, por lo que el resultado no será correcto:

```

1802:0001 D6          DB      D6
1802:0002 158ED8      ADC     AX,D88E
1802:0005 B800B8      MOV     AX,B800
1802:0008 8EC0         MOV     ES,AX
1802:000A A00000      MOV     AL,[0000]
1802:000D 8A260100    MOV     AII,[0001]
1802:0011 8B1E0200    MOV     BX,[0002]
1802:0015 26          ES:
1802:0016 8907         MOV    [BX],AX
1802:0018 26          ES:
1802:0019 C7066603410A MOV    WORD PTR [0366],0A41
1802:001F B44C         MOV    AH,4C

```

Como puede verse, las dos primeras instrucciones poco tienen que ver con el código real del programa. Al desensamblar, esta circunstancia no tiene mayor importancia porque es fácil darse cuenta del problema, pero debemos tener cuidado al emplear direcciones con otros comandos, porque podríamos intentar ejecutar instrucciones incorrectas.

Traducción de etiquetas

En el código ensamblador original, que puede ver al final del capítulo previo, usábamos diferentes etiquetas para hacer referencia al segmento de datos y unas cuantas variables. Como se decía antes, dichas etiquetas se pierden durante el ensamblado y, por ello, DEBUG no puede encontrarlas. En su lugar, lo que tenemos en las instrucciones son las direcciones en las que se tradujeron aquellas etiquetas.

Encontramos el primer caso en la primera instrucción del programa: mov ax, 17F1. Esa dirección es la del segmento de datos, que en el código ensamblador original introdujimos mediante la etiqueta Datos.

A continuación, tras la asignación al registro ES, encontramos tres sentencias que introducen en AL, AH y BX otros tantos valores. Estos se leen de las posiciones de memoria 0, 1 y 2 del segmento de datos bien en el código original usábamos las etiquetas Asterisco, Blanco y Posición. Dado que las dos primeras estaban definidas como datos de un byte, las inmediatamente siguientes se encuentran en direcciones consecutivas con un solo byte de diferencia.

Observe, por último, como el cálculo que efectuábamos en una de las últimas instrucciones, para introducir directamente en una posición de pantalla una letra A en color verde, aparece en el código como un valor constante. Lo que ha ocurrido es que el propio ensamblador, NASM en este caso, se ha ocupado de efectuar el cálculo aritmético para obtener el resultado, introduciéndolo como parte de la instrucción. Esta aparece en DEBUG con la sintaxis que se utilizaría en MASM, con la palabra PTR siguiendo a WORD.

Examen del contenido de datos

Como acabamos de ver, el programa que hemos desensamblado hace uso de cierta información que recupera del segmento de datos. En instrucciones como mov ah, [0001] sabemos que AH tomará el dato, un byte, que esté alojado en la dirección DS: 0001, y

también sabemos que, previamente, a DS se ha asignado el valor 17F1. Por tanto, AH contendrá el byte que haya en la celdilla de memoria 17F1:0001 una vez que el programa se haya puesto en marcha.

Para poder ver el contenido de una cierta porción de memoria, tanto en hexadecimal como en representación ASCII, usaremos el comando d. Al igual que el comando u, podemos facilitar tanto una dirección de inicio como una de fin, utilizándose por defecto el contenido del registro CS como segmento.

Si introduce, tras iniciar DEBUG y abrir el archivo que estamos usando como ejemplo, el comando d sin parámetros, obtendrá el resultado que puede verse en el centro de la figura 11.3. Hay que tener en cuenta que CS apunta al segmento de código, por ello lo que vemos son los bytes de código máquina.

The screenshot shows the DEBUG command-line interface. The title bar reads "Símbolo del sistema - debug asigna-1.exe". The command line at the top says "Z:\Libros\Actuales\PROGRA~1\CD-ROM\Ejemplos\10>debug asigna-1.exe". Below the command line, there are three sections of output:

- u**: Displays assembly code. The first few lines are:


```
1802:0000 B8F117      MOV     AX,17F1
1802:0003 8ED8      MOV     DS,AX
1802:0005 B800B8      MOV     AX,B800
1802:0008 8EC0      MOV     ES,AX
1802:000A A00000      MOV     AL,[0000]
1802:000B 8A260100    MOV     AH,[0001]
1802:0011 8B1E0200    MOV     BX,[0002]
1802:0015 26          ES:
1802:0016 8907      MOV     [BX],AX
1802:0018 26          ES:
1802:0019 C7066603410A MOV     WORD PTR [0366],0A41
1802:001F B44C      MOV     AH,4C
```
- d**: Displays a memory dump. The first few lines are:


```
1802:0000  B8 F1 17 8E D8 B8 00 B8-8E C0 A0 00 00 8A 26 01 .....&..&..F.A.
1802:0010  00 8B 1E 02 00 26 89 07-26 C7 06 66 03 41 0A B4 .....L.I.....
1802:0020  4C CD 21 00 00 00 00 00-00 00 00 00 00 00 00 00
1802:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
1802:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
1802:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
1802:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
1802:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```
- d 17F1:0**: Displays a memory dump starting at address 17F1:0000. The first few lines are:


```
17F1:0000  2A F0 D0 0C 00 00 00 00-00 00 00 00 00 00 00 00 *
17F1:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
17F1:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
17F1:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
17F1:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
17F1:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
17F1:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
17F1:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Figura 11.3. Examinamos el contenido de varias porciones de memoria.

Usando el comando d 17F1:0, como se aprecia en la parte inferior de la misma figura, examinamos el área de datos que se había definido en el programa. Es fácil distinguir, en el área de la derecha, el asterisco que habíamos asignado al primer campo, mientras

que el resto de bytes corresponden al color negro sobre blanco: FO, y la posición donde se colocará el asterisco en pantalla: OCD0.

Nota

Es necesario reiterar de nuevo que al interpretar los datos contenidos en memoria debe tenerse en cuenta que las palabras, compuestas de dos bytes, se almacenan en formato LSB-MSB (*Least Significant Byte-Most Significant Byte*). Esto significa que el byte de mayor peso, en el caso del dato 17F1 sería 15, se almacena en la segunda celdilla de memoria, mientras que el byte menos significativo, D6 se aloja en la primera. Puede verlo en la misma figura 11.3, en la parte superior, al desensamblar la primera instrucción.

Estado inicial de los registros

Para completar nuestro escenario inicial, conociendo el contexto en el que va a ejecutarse el programa, además de las instrucciones y los datos, examinados en los puntos previos, también necesitaremos conocer el contenido de los registros del procesador. De éstos dependerá, por ejemplo, la instrucción que vaya a ejecutarse, el espacio de memoria que se use como pila, etc.

El comando que nos interesa, en este caso, es r que, en principio, usaremos sin parámetro alguno.

El resultado será como el que puede ver en la parte inferior de la figura 11.4. Aparecen todos los registros del microprocesador, en realidad los que conoce DEBUG como herramienta de depuración DOS de 16 bits, con su contenido, así como la instrucción que apunta actualmente la pareja CS : IP.

The screenshot shows a window titled "Símbolo del sistema - debug asigna-1.exe". The window displays assembly code and processor register values. The assembly code at the bottom is:

```
AX=0000 BX=FFFF CX=FF83 DX=0000 SP=0100 BP=0000 ST=0000 DI=0000
DS=17E1 ES=17E1 SS=17F2 CS=1802 IP=0000 NV UP EI PL NZ NA PO NC
1802:0000 B8F117      MOV    AX,17F1
```

The registers listed above the code are:

Register	Value
AX	0000
BX	FFFF
CX	FF83
DX	0000
SP	0100
BP	0000
ST	0000
DI	0000
DS	17E1
ES	17E1
SS	17F2
CS	1802
IP	0000
NV	
UP	
EI	
PL	
NZ	
NA	
PO	
NC	

Figura 11.4. Contenido inicial de los registros del procesador.

La primera fila muestra los registros de uso general: AX, BX, CX y DX; los registros puntero y base de pila: SP y BP, y los registros índices SI y DI. En la segunda fila aparecen los cuatro registros de segmento, el puntero de instrucción y, finalmente, el estado del registro de indicadores *oflags*. De éste nos ocuparemos posteriormente con mayor detalle.

Por ahora, tras abrir el programa y antes de ejecutar instrucción alguna, podemos ver que los registros de segmento DS y ES contienen el mismo valor, que no es el que apropiado para apuntar al segmento de datos; mientras que CS sí que apunta al segmento de código. SS, por su parte, también tiene un valor por defecto y, normalmente, tendríamos que hacer como con DS, asignándole el segmento que corresponda a la pila. Es algo que no hicimos en el programa de ejemplo porque, en realidad, no llegamos a emplear la pila para nada.

Ejecución paso a paso

Llegados a este punto conocemos las instrucciones que componen el programa, así como el contenido del área de datos y el estado inicial de los registros del procesador. Con esto podemos hacernos una composición o idea del estado previo a la ejecución. Lógicamente, también necesitamos saber qué es lo que se espera que haga el programa para, al ir ejecutándolo paso a paso, apreciar si su funcionamiento es o no el esperado.

Para ejecutar una instrucción, concretamente aquella a la que apuntan CS : IP, utilizaremos normalmente el comando p. Este ejecuta la instrucción y, de inmediato, muestra el estado en que quedan los registros y la instrucción que se ejecutaría a continuación. En la figura 11.5 puede ver cómo ha ido ejecutándose el programa, instrucción a instrucción, hasta llegar al final. En el momento en que se ejecuta la función 4C de la interrupción 21, mediante la instrucción int, aparece el mensaje que puede verse en la parte inferior de dicha figura.

Fíjese en la parte derecha de cada instrucción a medida que va utilizando el comando p para ejecutarlas. Si la instrucción efectúa un acceso a memoria, como en el caso de mov al, [0000], a la derecha se indica el contenido que hay en esa posición, independientemente de que vaya a leerse o escribirse.

El comando p se caracteriza por ejecutar instrucciones completas, incluso cuando éstas son llamadas a rutinas o interrupciones.

Por eso, al ejecutar la instrucción int 21 aparece directamente el mensaje que indica que el programa ha finalizado de forma normal.

Depuración de rutinas y BIOS

Si al ir ejecutando instrucciones, con el comando p, se encuentran instrucciones como cali e int, que transfieren el punto de ejecución a una rutina o el código de la BIOS, lo que se hace es ejecutar todo ese código hasta volver a la instrucción siguiente de la que estaba depurándose.

Ejecución hasta un cierto punto

El programa con el que estamos haciendo pruebas es muy sencillo, apenas se compone de una docena de instrucciones, por lo que ejecutarlo paso a paso no supone un mayor problema. En programas más extensos, sin embargo, lo normal será que deseemos ejecutar hasta un cierto punto, en el que suponemos que está el problema, procediendo a partir de ahí con los comandos explicados en los puntos previos.

La información fundamental, en este caso, es la dirección del programa hasta la que queremos ejecutar, dato que podemos obtener desensamblando, mediante el comando u, hasta encontrar el punto que nos interese. En ese momento usaríamos el comando g, al que facilitaríamos como parámetro la dirección antes anotada.

Notg t ^ ~~ ~~ "~~~ |

Si usa el comando g sin facilitar dirección alguna, se ejecutará todo el código desde la instrucción que indique CS: IP hasta el final del programa. También puede introducirse, opcionalmente, una dirección de inicio.

Observe la figura 11.6. En la parte superior hemos usado el comando u para desensamblar la mayor parte del programa. A continuación, mediante el comando g, se ha ejecutado hasta la dirección CS : 0015, tras lo cual aparece el estado de los registros y la siguiente instrucción a ejecutar. En ese momento los registros presentan el estado que deberían tener tras la ejecución de todo el código que hay hasta ese punto.

The screenshot shows a Windows application window titled 'Símbolo del sistema - debug asigna~1.exe'. The command line at the top reads: 'Z:\Libros\Actuales\PROGRA~1\CD-ROM\Ejemplos\10>debug asigna~1.exe -u'. The assembly code listed is:

```

Z:\Libros\Actuales\PROGRA~1\CD-ROM\Ejemplos\10>debug asigna~1.exe
-u
1802:0000 B8F117      MOV    AX,17F1
1802:0003 8ED8      MOV    DS,AX
1802:0005 B800B8      MOV    AX,B800
1802:0008 8EC0      MOV    ES,AX
1802:000A A00000      MOV    AL,[0000]
1802:000D 8A260100      MOV    AH,[0001]
1802:0011 8B1E0200      MOV    BX,[0002]
1802:0015 26          ES:
1802:0016 8907      MOV    [BX],AX
1802:0018 26          ES:
1802:0019 C7066603410A MOV    WORD PTR [0366],0A41
1802:001F B44C      MOV    AH,4C
-g 15

```

Below the assembly code, the register states are displayed:

```

AX=F02A  BX=0CD0  CX=FF83  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17F1  ES=B800  SS=17F2  CS=1802  IP=0015  NV UP EI PL NZ NA PO NC
1802:0015 26          ES:
1802:0016 8907      MOV    [BX],AX
                                         ES:0CD0=F020

```

Figura 11.6. Ejecutamos el programa hasta un cierto punto.

```

Z:\Libros\Actuales\PROGRA-1\CD-ROM\Ejemplos\10>debug asigna~1.exe
1802:0000 B8F117      MOV     AX,17F1
1802:0003 8ED8        MOV     DS,AX
1802:0005 B800B8        MOV     AX,B800
1802:0008 8EC0        MOV     ES,AX
1802:000A A00000        MOV     AL,[0000]
1802:000D 8A260100        MOV     AH,[0001]
1802:0011 8B1E0200        MOV     BX,[0002]
1802:0015 26          ES:
1802:0016 8907        MOV     [BX],AX
1802:0018 26          ES:
1802:0019 C7066603410A    MOV     WORD PTR [0366],0A41
1802:001F B44C        MOV     AH,4C

-g 15

AX=F02A  BX=0CD0  CX=FF83  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17F1  ES=B800  SS=17F2  CS=1802  IP=0015  NV UP EI PL NZ NA PO NC
1802:0015 26          ES:          A
1802:0016 8907        MOV     [BX],AX
-rax
AX F02A
:F041
-p

AX=F041  BX=0CD0  CX=FF83  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17F1  ES=B800  SS=17F2  CS=1802  IP=0018  NV UP EI PL NZ NA PO NC
1802:0018 26          ES:
1802:0019 C7066603410A    MOV     WORD PTR [0366],0A41
                           ES:0366=F042
-
```

Figura 11.7. Alteramos el contenido de un registro antes de ejecutar una instrucción.

En caso de indicar, tras el comando, tan sólo la dirección de desplazamiento, se asumirá que el segmento es el indicado por el registro DS. Podemos, no obstante, preceder el desplazamiento con cualquiera otro registro de segmento o, incluso, facilitar la dirección completa como si fuese una constante.

Suponga que quiere modificar el carácter que hay almacenado en el campo Asterisco del segmento de datos. Tras ejecutar las dos primeras instrucciones, para que DS apunte a dicho segmento, introducimos el comando e DS : 0 'A'. Examinando a continuación el principio del segmento de datos, como se ha hecho en la figura 11.8, puede apreciar el cambio. Al ejecutar el resto de las instrucciones verá que el programa muestra una letra A sobre fondo blanco en lugar del asterisco.

Para introducir múltiples valores, no hay más que facilitarlos en forma de lista, uno tras otro, separados por espacios. Pruebe, por ejemplo, a introducir el siguiente comando:

```
e b800:1680 'H' 70 'o' 70 '1' 70 'a' 60
```

Al pulsar **Intro** verá aparecer, al inicio de una de las últimas líneas de pantalla, la palabra "Hola" con las tres primeras letras con fondo blanco y la última con fondo verde. Los datos facilitados se introducen a razón de un byte por celdilla a partir de la dirección indicada que, en este caso, se encuentra en el segmento dedicado al contenido de la pantalla.

```

z:\Libros\Actuales\PROGRA~1\CD-ROM\Ejemplos\10>debug asigna~1.exe
-r
AX=0000 BX=FFFF CX=FF83 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 FS=17E1 SS=17F2 CS=1802 IP=0000 NV UP EI PL NZ NA PO NC
1802:0000 B8F117      MOV     AX,17F1
-p

AX=17F1 BX=FFFF CX=FF83 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F2 CS=1802 IP=0003 NV UP EI PL NZ NA PO NC
1802:0003 8ED8      MOV     DS,AX
-p

AX=17F1 BX=FFFF CX=FF83 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17F1 FS=17E1 SS=17F2 CS=1802 IP=0005 NV UP EI PL NZ NA PO NC
1802:0005 B80088      MOV     AX,B800
-d ds:0
17F1:0000 2A F0 D0 0C 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 *.....
17F1:0010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0030 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....

-e ds:0 'A'
-d ds:0
17F1:0000 41 F0 D0 0C 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 A.....
17F1:0010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0030 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....
17F1:0070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 .....

```

Figura 11.8. Modificación de los campos de datos.

En lugar de entregar la lista acalores, lo cual es adecuado sólo si deseamos modificar un conjunto de celdillas consecutivas, podemos introducir sólo la dirección tras el comando e. En ese caso aparecerá el contenido de la primera celdilla, quedando el cursor detrás de un punto a la espera de que introduzcamos un nuevo valor. Si queremos cambiar esa celdilla, escribimos el dato y sin pulsar **Intro**, avanzamos a la celdilla siguiente mediante la barra espaciadora. En caso de no querer modificarla, pulsamos directamente la barra. Así es posible ir recorriendo una zona de memoria y cambiar sólo algunas celdillas. La figura 11.9 muestra cómo se ha usado el comando e para ir modificando ciertas celdillas de la memoria de pantalla, invirtiendo el color de algunas posiciones sin alterar el carácter salvo en determinados casos. En cualquier momento puede pulsar la tecla **Intro** para finalizar el comando y volver al indicador habitual de DEBUG.

Nota

Además de la barra espaciadora, también puede usar la tecla - (el guion) para retroceder una posición, en lugar de avanzar.

The screenshot shows a Windows command-line interface window titled 'Símbolo del sistema - debug assign~1.exe'. The window displays assembly code and memory dump information.

```

Z:\Libros\Actuales\PROGRA~1\CD-ROM\Ejemplos\10>debug assign~1.exe
-u
1802:0000 B8F117    MOV     AX,1/F1
1802:0003 8ED8    MOV     DS,AX
1802:0005 B80088    MOV     AX,B800
1802:0008 8EC0    MOV     ES,AX
1802:000A A00000 [REDACTED] MOV     AL,[0000]
1802:000D 8A260100 MOV     AH,[0001]
1802:0011 8B1E0200 MOV     BX,[0002]
1802:0015 26        ES:
1802:0016 8907    MOV     [BX],AX
1802:0018 26        ES:
1802:0019 C7066603410A MOV     WORD PTR [0366],0A41
1802:001F B44C    MOV     AH,4C
-
-e b800:480
B800:0480 20.      F0.07   20.      F0.07   20.2a  F0.07   20.      F0.07
B800:0488 20.      F0.07   20.2a  F0.50   20.      F0.07   20.      F0.07
B800:0490 4D.      F0.07   20.      F0.07   20.2a  F0.07   20.      F0.07
-

```

Figura 11.9. Modificación interactiva del contenido de la memoria.

Ensamblar nuevas instrucciones

Con los comandos `r` y `e` podemos, en cierta forma, alterar la ejecución inicial del programa, al modificar el valor ^kílos registros o la memoria en un cierto instante. ¿Qué ocurre, sin embargo, si lo que hay que cambiar es la propia instrucción ensamblador? También en estos casos tenemos solución: el comando `a` de DEBUG.

Como otros comandos que hemos conocido, éste precisa un parámetro que le indique la posición de memoria a partir de la cual debe introducir el código máquina resultante del ensamblado de las instrucciones que vayan introduciéndose. Debe tener en cuenta, y esto es de especial importancia, que a medida que vaya ensamblando irán sobrescribiéndose las instrucciones que hubiese en esa zona de memoria, por el que el funcionamiento del programa puede verse completamente alterado.

Imagine que quiere cambiar la instrucción del programa con la que mostramos una letra A en pantalla, sustituyéndola por las que sean necesarias para poner un segundo asterisco justo a la derecha del segundo. Para ello tendremos que incrementar la dirección de memoria que se encuentra en BX dos veces, para saltar los dos bytes que ocupa una posición de pantalla, repitiendo la asignación de AX. Esto puede sustituir también las dos instrucciones finales del programa, fundamentales para que el regreso a la línea de comandos del sistema se produzca sin problemas. Por ello tendremos que volver a introducirlas también.

En la figura 11.10 puede ver tres partes:

- En la superior está parte del código del programa, obtenido con el comando u, antes de modificarlo.
- La zona central, destacada del resto, muestra cómo usar el comando a para introducir las sentencias que necesitábamos.
- El área inferior corresponde al nuevo código del programa y su ejecución. Observe los dos asteriscos más o menos en el centro de la pantalla.

Este mismo sistema puede utilizarse para probar un conjunto no muy grande de instrucciones de una forma rápida, sin tener que pasar por el editor de texto, ensamblador y enlazador. Al ensamblar desde DEBUG recuerde que no puede utilizar etiquetas, tan sólo direcciones de memoria, y que la sintaxis no es la de NASM, sino más cercana a la que se emplearía con MASM.

The screenshot shows a Windows command-line interface window titled "Selecctionar Símbolo del sistema - debug asigna~1.lesje". The window displays assembly code in three sections:

- Top Section (-u):** Shows the original assembly code from memory address 1802:0000 to 1802:001F. It includes instructions like MOV AX,17F1, MOV DS,AX, MOV AX,B800, etc.
- Middle Section (-a 18):** Shows the assembly code starting at address 1802:0018. It contains instructions: inc bx, inc bx, es:, mov [bx],ax, mov ah,4c, int 21, and int 21. The word "es:" is highlighted in yellow.
- Bottom Section (-u cs:0 21):** Shows the modified assembly code from address 1802:0000 to 1802:0021. The changes include INC BX, INC BX, and INT 21 instead of the original instructions. The modified code is highlighted in yellow.

At the bottom of the window, the message "Program terminated normally" is displayed.

Figura 11.10. Alteramos el código del programa antes de ejecutarlo.

Otras posibilidades de DEBUG

A pesar de su espartana interfaz de usuario, DEBUG es un programa con bastantes posibilidades. Si bien con los comandos que hemos conocido tendremos suficiente para la mayoría de los casos, en ocasiones podemos precisar algo más. El propio programa puede facilitarle una referencia rápida de los comandos existentes, simplemente introduciendo el carácter ? y pulsando **Intro**, como se ha hecho en la figura 11.11.

```

Símbolo del sistema - debug
Z:\Libros\Actuales\PROGRA-1\CD-ROM\Ejemplos\10>debug
-?
assemble      A [address]
compare       C range address
dump          D [range]
enter         E address [list]
fill           F range list
go             G [=address] [addresses]
hex            H value1 value2
input          I port
load           L [address] [drive] [firstsector] [number]
move           M range address
name           N [pathname] [arglist]
output         O port byte
proceed        P [=address] [number]
quit           Q
register       R [register]
search         S range list
trace          T [=address] [value]
unassemble     U [range]
write          W [address] [drive] [firstsector] [number]
allocate expanded memory    XA [#pages]
deallocate expanded memory   XD [handle]
map expanded memory pages   XM [Lpage] [Ppage] [handle]
display expanded memory status XS
-
```

Figura 11.11. Ayuda de DEBUG.

En la tabla 11.1 se resume la función de cada uno de los comandos. Si necesitásemos alguno de ellos posteriormente, en los capítulos siguientes, en su momento se explicarían con mayor detalle.

Tabla 11.1. Comandos de DEBUG.

Comando	Función
A	Ensamblar instrucciones.
C	Comparar dos áreas de memoria.
D	Mostrar el contenido de las celdillas de memoria.
E	Modificar el contenido de las celdillas de memoria.
F	Rellenar zonas de memoria con un valor.
G	Ejecutar instrucciones hasta una posición o el final del programa.

Comando	Función
H	Efectuar sumas y restas de números.
I	Leer datos de un puerto de entrada/salida.
L	Recuperar el contenido de un archivo o sectores de disco.
M	Copiar datos de una zona de memoria a otra.
N	Establecer el nombre del archivo del que se va a leer o bien en el que se va a escribir.
O	Enviar un dato a un puerto de entrada/salida.
P	Ejecuta instrucciones paso a paso sin entrar en rutinas ni instrucciones.
Q	Sale de DEBUG.
R	Muestra/Modifica el contenido de los registros del procesador.
S	Busca en la memoria una determinada secuencia de bytes.
T	Ejecuta instrucciones paso a paso entrando en rutinas e interrupciones.
U	Desensambla instrucciones.
W	Escribe datos en un archivo o sectores de disco.

Resumen

La depuración de un programa escrito en ensamblador, como ha podido ver en este capítulo, no difiere en exceso del proceso típico de depuración que usaríamos en cualquier otro lenguaje. Las instrucciones, eso sí, son mucho más simples, de más bajo nivel, y esto puede hacer, en ocasiones, que perdamos la visión general del programa al centrarnos en los detalles de cada instrucción.

DBUG es una aplicación simple, pero sencilla al tiempo. Generalmente no necesitaremos un depurador más sofisticado para seguir la ejecución de la mayor parte de los ejercicios que vamos a desarrollar en los capítulos posteriores, salvo que empleemos características no reconocidas por DEBUG. Obviamente, si nuestra plataforma de trabajo principal es Linux, no DOS, tendremos que recurrir a un depurador para dicho sistema, como gdb.

12

Operaciones aritméticas

Hasta ahora hemos conocido las instrucciones ensamblador necesarias para llevar datos desde la memoria al procesador y viceversa, algo que nos permite asignar valores a ciertas posiciones de memoria o llevar datos de un punto a otro. En un capítulo previo vimos, por ejemplo, cómo usar dichas instrucciones para mostrar datos en pantalla.

Otra de las necesidades básicas, una vez que tenemos algunos datos en los registros, es la de efectuar sobre ellos algunas operaciones aritméticas. A diferencia de la mayoría de procesadores de 8 bits más populares, la familia x86 dispone de instrucciones para efectuar no sólo sumas y restas, sino también multiplicaciones y divisiones. Siempre hablamos de cálculos llevados a cabo en el interior del microprocesador y con números enteros. El 8086/8088, y sus primeros descendientes, no eran capaces de tratar con números que no fuesen enteros, es decir, con números en coma flotante. De esa tarea, cuando era necesaria, se ocupaba el coprocesador matemático, un segundo integrado, conocido como 8087/80287/80387 según la generación, que había que instalar en la placa del ordenador junto al procesador principal. A partir del 80486, y lógicamente todos los Pentium, Core y sucesores, ese coprocesador se encuentra integrado, de tal forma que es posible efectuar operaciones aritméticas más complejas empleando registros e instrucciones específicas para ese fin.

Suma de dos números

Comencemos por la operación aritmética más simple que podemos efectuar: la suma de dos números. Éstos pueden tener un byte, una palabra o una doble de palabra de tamaño, dependiendo del registro que actúe como receptor y de que tengamos activado

el conjunto de instrucciones de 32 bits que permite usar los registros EAX, EBX, EDX, etc. Lo mismo sería extensible para el trabajo con enteros de 64 bits.

La instrucción que nos permite sumar dos números es add que, al igual que mov, necesita dos parámetros: un destino y un valor a sumar a ese destino.

El resultado, por lo tanto, quedará en el primero de los dos parámetros que, lógicamente, debe ser un registro o bien una posición de memoria, pero nunca un valor inmediato. El número a sumar, entregado como segundo parámetro, sí puede ser un valor inmediato.

Que la suma sea de 8, 16, 32 ó 64 bits es algo que vendrá determinado por el destino, como ocurría con la instrucción mov. Si es un registro, su tamaño determinará el del segundo operando, mientras que si es una dirección de memoria sería necesario indicar explícitamente el tamaño como ya se ha visto en el quinto capítulo.

Aunque el destino de la operación de suma puede ser cualquiera de los registros de propósito general, el registro AX, EAX si la operación es de 32 bits o bien RAX si es de 64 bits, está optimizado para estas operaciones, por lo que su uso es más habitual que el de otros.

Puede hacer una comprobación simple introduciendo el código siguiente en un archivo de texto, ensamblándolo y siguiéndolo paso a paso con DEBUG.

```
segment Pila stack
resb 256

; Segmento de código
segment Código
..start:

    mov al, 10h ; Almacenamos 10h en AL
    add al, 20h ; y le sumamos 20h

    ; salimos al sistema
    mov ah, 4ch
    int 21h
```

Las dos sentencias que nos interesan en este momento son las que encontramos justo tras la etiqueta start. La primera de ellas asigna el valor 10h al registro AL, mientras que la siguiente suma a ese mismo registro el valor 2 Oh.

El resultado, como se decía antes, quedará en el operando usado como primer parámetro, AL en este caso.

Al ejecutar el programa paso a paso, como se ha hecho en la figura 12.1, observe el valor de AX, tanto el inicial como el posterior a la ejecución de las dos instrucciones mov y add. Como era de esperar, finalmente AL, el byte de menor peso de AX, contiene el valor 3 Oh.

Recuerde que el 1 Oh hexadecimal equivale al número 16 en decimal, mientras que 2 Oh es 32 y, por tanto, la suma resultante es 48, 3 Oh en hexadecimal.

The screenshot shows a Windows-style debugger window titled "Símbolo del sistema - debug suma.exe". The command line at the top says "D:\Ejemplos\12>debug suma.exe". The window displays assembly code and register states for three different additions:

- First addition: AX=0000 BX=FFFF CX=FF48 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000 DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC 1801:0000 B010 MOV AL,10
- Second addition: AX=0010 BX=FFFF CX=FF48 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000 DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0002 NV UP EI PL NZ NA PO NC 1801:0002 0420 ADD AL,20
- Third addition: AX=0030 BX=FFFF CX=FF48 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000 DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0004 NV UP EI PL NZ NA PE NC 1801:0004 B44C MOV AH,4C

Figura 12.1. Observamos el estado de los registros al ejecutar la operación de suma.

Desbordamiento y acarreo

Al efectuar una suma entre dos operandos, mediante la instrucción add, uno de los problemas que se plantea habitualmente es el desbordamiento del destino. En el código anterior, por ejemplo, se utiliza el registro AL como destino del resultado, y sabemos que se trata de un registro de 8 bits que, por tanto, no puede contener ningún número superior a 0 f f h, o 255 decimal. Si se suma un número que genere un resultado superior a ese límite, se produce un desbordamiento.

Suponga que el registro es como el cuentakilómetros de un coche, con sólo dos posiciones que pueden ir desde 0 hasta F cada una de ellas. Cuando se llega a FF se produciría la *vuelta al marcador*, mostrando éste de nuevo el valor 00. Si no tenemos esto en cuenta, podemos creer que el valor del marcador, del registro AL, es correcto, cayendo en un error.

Analice lo que ocurre cuando, usando la base decimal, suma dos números cualesquiera. Si suma 3 y 4, por ejemplo, el resultado es un solo dígito: 7. ¿Qué pasa, sin embargo, si suma 7 y 3? El dígito resultante es 0 y se produce un acarreo, el habitual *y me llevo una*, convirtiéndose en un nuevo dígito que aparece a la izquierda del anterior.

En los microprocesadores ocurre algo similar y, al efectuar una suma que provoca el desbordamiento del destino, se activa un indicador de acarreo. Éste se encuentra, junto con otros indicadores, en el registro de estado, registro de indicadores o registro de banderas del microprocesador, conocido habitualmente como registro *deflags*.

Dicho registro se compone de 16 bits, 32 bits o 64 bits, según los casos, indicando cada uno de ellos el estado de uno de los indicadores. El bit que nos interesa puede estar a 1, estado CY (*Carry*) o bien a 0, NC (*No Carry*), según se haya o no producido un acarreo, respectivamente.

Nota

El indicador de acarreo, en el registro de indicadores, es conocido habitualmente como *carry*.

Podemos comprobar fácilmente el estado del indicador de acarreo desde DEBUG. Antes sustituya el valor que se asigna inicialmente a AL, y el que se suma posteriormente, por otros mayores, por ejemplo 210 y 130, asegurando así que se producirá el desbordamiento. A continuación abra el ejecutable con DEBUG y ejecute paso a paso. Observe como, tras ejecutar la instrucción add, el indicador NC pasa a CY (véase figura 12.2). Fíjese también en el resultado que ha quedado en AL.

```

Símbolo del sistema : debug_bitaca-1.exe
D:\Ejemplos\12>nasm -f obj bitacarreo.asm
D:\Ejemplos\12>alink bitacarreo
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file bitacarreo.obj
matched Externs
matched ComDefs

D:\Ejemplos\12>debug bitaca-1.exe
-r
AX=0000 BX=FFFF CX=FF48 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC
1801:0000 B0D2          MOV    AL,D2
-p

AX=00D2 BX=FFFF CX=FF48 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0002 NV UP EI PL NZ NA PO NC
1801:0002 0482          ADD    AL,82
-p

AX=0054 BX=FFFF CX=FF48 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0004 OV UP EI PL NZ NA PO CY
1801:0004 B44C          MOV    AH,4C
-
```

Estado del indicador de acarreo

Figura 12.2. Al ejecutar la suma se produce el acarreo.

Suma con acarreo

La instrucción add tiene en cuenta, a la hora de efectuar una suma, sólo los dos operandos que intervienen en ella. *

Existe otra instrucción de suma, adc, que además tiene también en cuenta el estado del indicador de acarreo, sumando uno más en caso de que dicho indicador se encuentre activo.

Modifique el programa usado como ejemplo en el punto anterior, añadiendo tras la instrucción add la siguiente sentencia:

```
adc ah, 0
```

Lo que hacemos es sumar el número 0 al registro AH pero teniendo en cuenta el indicador de acarreo, de tal forma que se sumará también en caso de que la suma previa lo haya activado.

Siguiendo de nuevo el programa paso a paso, con DEBUG, verá que al ejecutar esa instrucción en AX queda el resultado correcto, 154h, además de borrarse el indicador de acarreo ya que esta nueva suma no ha provocado desbordamiento.

```
D:\Ejemplos\12>nasm -f obj acarreo.asm
D:\Ejemplos\12>alink acarreo
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file acarreo.obj
matched Externs
matched ComDefs

D:\Ejemplos\12>debug acarreo.exe
-r
AX=0000  BX=FFFF  CX=FF4B  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=0000  NV UP EI PL NZ NA PO NC
1801:0000 B0D2      MOV     AL ,D2
-p

AX=00D2  BX=FFFF  CX=FF4B  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=0002  NV UP EI PL NZ NA PO NC
1801:0002 0482    ADD     AL ,82
-p

AX=0054  BX=FFFF  CX=FF4B  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=0004  OV UP EI PL NZ NA PO CY
1801:0004 80D400   ADC     AH,00
-p

AX=0154  BX=FFFF  CX=FF4B  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=0007  NV UP EI PL NZ NA PO NC
1801:0007 B44C      MOV     AH,4C
-
```

Figura 12.3. La suma con acarreo nos facilita en AX el resultado correcto.

Si convierte 154h a decimal verá que el resultado es 340, la suma de los valores 210 y 130 que introdujimos en el registro AL y posteriormente sumamos. No olvide nunca que DEBUG facilita todos los números en hexadecimal.

Lógicamente, no tiene sentido intentar una suma que sabemos que puede exceder la capacidad de un registro de 8 bits y usar el acarreo cuando tenemos a nuestra disposición registros de 16 bits. Podríamos introducir el valor 210 en AX, sumarle 130 y obtener directamente el resultado correcto, sin necesidad de operaciones adicionales. ¿Qué ocurre si quiere sumar dos números de 32 bits y no puede emplear registros de dicho tamaño, por ejemplo en una aplicación corriente de DOS que no puede acceder a los registros EAX, EBX, etc.? Es aquí donde encontraremos la aplicación de la suma con acarreo.

Sumas de 32 bits con registros de 16

Al ensamblar código ejecutable para DOS, indistintamente del procesador que tenga el sistema, estamos limitados a emplear registros de 16 bits. ¿Cómo podríamos sumar entonces los números 240.000 y 300.000, por poner un ejemplo? Lo primero que tenemos que hacer es dividir esos dos números, que son dobles palabras, en dos palabras separadas, sumándolas después de forma independiente pero teniendo en cuenta el acarreo de una a otra.

Los registros AX y DX son de 16 bits, pero supongamos que podemos unirlos en uno solo, un hipotético registro AXDX, de 32 bits. AX contendría la palabra de más peso o más significativa, mientras que DX alojaría la palabra de menor peso. Esto significa que cuando DX pase de 65535, volverá a ser 0 y se incrementará en 1 el valor de AX. Es lo que hemos visto que sucedía con AL y AH en el ejemplo previo.

Ya que cada palabra, cada registro de 16 bits, puede contener 65536 valores diferentes, dividimos los números 240.000 y 300.000 entre 65536, quedándonos con un cociente, que sería la palabra de mayor peso, y un resto, que sería la de menor peso. El resultado sería el de la tabla 12.1.

Tabla 12.1. División de números de 32 bits en dos palabras.

240.000	3	43.392 (A98 Oh)
300.000	4	37.856 (93E0h)

Alojaríamos, por tanto, el valor 3 en el registro AX, que es el de mayor peso, y el valor 43392 en DX. A continuación, para efectuar la suma, tendríamos que dar dos pasos: sumar las dos palabras de menor peso y, teniendo en cuenta el posible acarreo, sumar las de

mayor peso. El resultado quedaría en los mismos registros AX y DX. El código necesario sería el siguiente:

```
mov ax, 3
mov dx, 43392
add dx, 37856
adc ax, 4
```

Como en casos anteriores, emplee DEBUG para seguir la ejecución del programa y observar el resultado que, si todo va bien, debería ser como el de la figura 12.4.

The screenshot shows a Windows command-line interface window titled "Símbolo del sistema - debug acarre~1.exe". The window displays the following text:

```
D:\Ejemplos\12>nasm -f obj acarre~1.asm
D:\Ejemplos\12>alink acarre~1
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved.

Loading file acarre~1.obj
matched Externs
matched ComDefs

D:\Ejemplos\12>debug acarre~1.exe
-r
AX=0000 BX=FFFF CX=FF51 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC
1801:0000 B80300      MOV     AX,0003
-p

AX=0003 BX=FFFF CX=FF51 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0003 NV UP EI PL NZ NA PO NC
1801:0003 BA80A9      MOV     DX,A980
-p

AX=0003 BX=FFFF CX=FF51 DX=A980 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0006 NV UP EI PL NZ NA PO NC
1801:0006 81C2E093    ADD     DX,93E0
-p

AX=0003 BX=FFFF CX=FF51 DX=3D60 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=000A OV UP EI PL NZ NA PE CY
1801:000A 150400    ADC     AX,0004
-p

AX=0008 BX=FFFF CX=FF51 DX=3D60 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=000D NV UP EI PL NZ NA PO NC
1801:000D B44C      MOV     AH,4C
-
```

Figura 12.4. Usamos dos registros de 16 bits para sumar dos números de 32 bits.

Al sumar el valor 37856 a DX se produce un desbordamiento, activándose el indicador de acarreo. Esto provoca que la instrucción adc sume a AX no sólo el valor 4, sino también ese acarreo.

El resultado es el valor 8 en AX y 15.712 en üX (3D60 en hexadecimal). Dado que AX y DX no son un solo registro real de 32 bits, sino dos de 16, es necesario multiplicar el valor de AX por 65.536 y sumarlo al de DX para hallar el resultado: 540.000.

Nota

De manera análoga, en las aplicaciones de 32 bits es posible usar dos registros para operar con números de 64 bits, o incluso cuatro registros para hacerlo con números de 128 bits. Tan sólo hay que incrementar el número de sumas con acarreo a ejecutar.

Restar un número de otro

La operación de resta de dos números, el segundo del primero que actúa, a su vez, como destino, se efectúa mediante la instrucción sub. El mecanismo es similar al descrito en los puntos previos para la suma, produciéndose un acarreo si fuese necesario y existiendo una instrucción de suma con acarreo: sbb.

Al igual que ocurre con la suma con acarreo, la resta con acarreo tiene sentido en casos en los que los números a tomar como operandos exceden la capacidad de los registros. Por ejemplo, suponga que quiere restar el número 240.000 del número 300.000, por tomar los mismos dos que habíamos usado antes. Introduciremos el segundo, 300.000, en los registros AX y DX, procediendo a continuación a restar las palabras de menor peso y, después, las de mayor peso teniendo en cuenta el acarreo. El código sería el siguiente:

```
mov ax, 4
mov dx, 37856
sub dx, 43392
sbb ax, 3
```

En la figura 12.5 puede ver que al restar el número 43392 de DX, que contiene el valor 37856, se activa el acarreo. Por ello al restar el número 3 del contenido de AX éste se queda en cero, al restarse también acarreo. El resultado es el valor que queda en DX: 60.000.

Multiplicar dos números

A diferencia de la suma y la resta, operaciones en las cuales indicamos de manera explícita los operandos, al efectuar una multiplicación, mediante la instrucción muí, uno de los operandos, así como el destino del resultado, están implícitos, de tal forma que solamente hay que facilitar el segundo operando, el multiplicador. Éste puede ser tanto un registro como una referencia a un dato alojado en memoria, pero no un valor inmediato.

En caso de que el multiplicador, facilitado como único parámetro a la instrucción muí, sea de 8 bits, el primer operando, el multiplicando, será el contenido del registro AL, quedando el resultado en AX. Si, por el contrario, el multiplicador es de 16 bits, el parámetro implícito será AX y el resultado quedaría en la pareja de registros DX: AX, siendo el primero el que contendría la palabra de mayor peso.

```

D:\Ejemplos\12>nasm -f obj resta.asm
D:\Ejemplos\12>alink resta
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file resta.obj
matched Externs
matched ComDefs

D:\Ejemplos\12>debug resta.exe
-r
AX=0000  BX=FFFF  CX=FF51  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=0000  NV UP EI PL NZ NA PO NC
1801:0000 B80400      MOV     AX,0004
-p

AX=0004  BX=FFFF  CX=FF51  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=0003  NV UP EI PL NZ NA PO NC
1801:0003 BAE093      MOV     DX,93L0
-p

AX=0004  BX=FFFF  CX=FF51  DX=93E0  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=0006  NV UP EI PL NZ NA PO NC
1801:0006 81EA80A9     SUB    DX,A980
-p

AX=0004  BX=FFFF  CX=FF51  DX=EA60  SP=0100  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=000A  NV UP EI NG NZ NA PE CY
1801:000A 1D0300      SBB    AX,0003
-p

AX=0000  BX=FFFF  CX=FF51  DX=EA60  SP=0100  BP=0000  ST=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1801  IP=000D  NV UP EI PL ZR NA PE NC
1801:000D B44C        MOV    AH,4C
-p

```

Figura 12.5. Efectuamos una resta de 32 bits con dos registros de 16.

Puede efectuar una prueba simple, para comprobar esta nueva instrucción, escribiendo y ensamblando el código siguiente:

```

segment1. Pil^ stsnk
rcob 256

; Segmento de código
segment Código
.start:

mov al, 30
mov bl, 15
muí bl

; salimos al sistema
mov ah, 4ch
int 21h

```

Como en casos anteriores, emplee la herramienta DEBUG (o el depurador que prefiere si es otro) para ejecutar paso a paso y comprobar cómo al final, tras la ejecución de la instrucción muí, queda en AX el resultado: 1C2h, que es 450 convertido a decimal.

La instrucción muí efectúa la multiplicación asumiendo que los operandos no tienen signo, lo cual provocaría un resultado erróneo en caso de que uno, o ambos, fuesen negativos. De ser éste el caso, en lugar de muí nos serviríamos de la instrucción imul. Dependiendo de que la multiplicación sea de 8 ó 16 bits, dicha instrucción asumirá que el bit 7 ó 15 es el de signo, teniéndolo en cuenta a la hora de efectuar el cálculo.

Para hacer pruebas con números negativos, asignándolos a los registros AL o BL del ejemplo anterior, no tiene necesidad de efectuar manualmente el complemento dos. Puede, directamente, asignar valores como -30 ó -15, dejando la tarea de conversión al ensamblador, en este caso NASM. Si multiplica -30 por -15, dado que ambos son negativos, el resultado obtenido debería ser exactamente el mismo: 1C2h. Si hace la multiplicación con la instrucción muí verá que esto no es así, porque el bit de signo no se interpreta como tal. Cambiando muí por imul, haga la prueba, el resultado sí será correcto.

Operando en el modo de 32 bits, también pueden multiplicarse números de ese tamaño. En este caso el operando implícito sería EAX y el resultado quedaría en la pareja EDX:EAX.

Dividir un número entre otro

La última operación aritmética posible, la división, se lleva a cabo mediante las instrucciones dive y idiv, dependiendo de que deba o no tenerse en cuenta el signo. Como ocurre con la multiplicación, uno de los operandos, concretamente el dividendo, es un parámetro implícito, debiendo facilitarse un solo dato adicional: el divisor. Éste será un registro o una referencia a memoria, dependiendo de su tamaño que el dividendo sea AX, DX : AX o F.nx: F.AX. El cociente de la división quedará en EAX, AX o AL, dependiendo de que la división fuese de 32, 16 u 8 bits, respectivamente. El resto, por su parte, quedaría en EDX, DX o AH. Como puede observar, el método es análogo al de la multiplicación.

Puede hacer una prueba con el código mostrado a continuación:

```

segment Pila stack
    resb 2fe

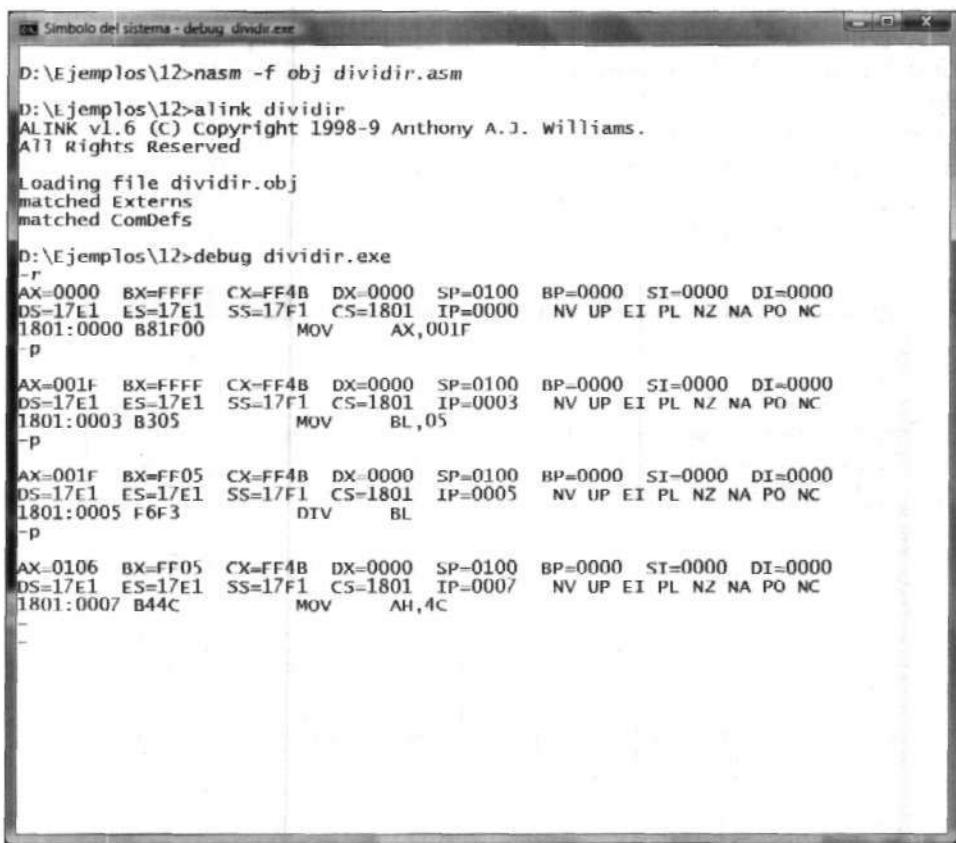
; Segmento de código
segment Código
.start:

    mov ax, 31
    mov bl, 5
    div bl

; salimos al sistema
    mov ah, 4ch
    int 21h

```

Al seguir su ejecución paso a paso (véase la figura 12.6) podrá ver que, tras ejecutar la instrucción div, tenemos el valor 6 en AL (cociente) y el valor 1 en AH (resto).



The screenshot shows a terminal window titled "Símbolo del sistema - debug dividir.exe". The command entered is "D:\Ejemplos\12>nasm -f obj dividir.asm". The output shows the assembly code being assembled:

```
D:\Ejemplos\12>alink dividir
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file dividir.obj
matched Externs
matched ComDefs

D:\Ejemplos\12>debug dividir.exe
-r
AX=0000 BX=FFFF CX=FF4B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC
1801:0000 B81F00    MOV     AX,001F
-p

AX=001F BX=FFFF CX=FF4B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0003 NV UP EI PL NZ NA PO NC
1801:0003 B305    MOV     BL,05
-p

AX=001F BX=FF05 CX=FF4B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0005 NV UP EI PL NZ NA PO NC
1801:0005 F6F3    DIV     BL
-p

AX=0106 BX=FF05 CX=FF4B DX=0000 SP=0100 BP=0000 ST=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0007 NV UP EI PL NZ NA PO NC
1801:0007 B44C    MOV     AH,4C
-
-
```

Figura 12.6. Tras la división obtenemos un cociente y un resto en los registros AL y AH.

Incrementos y reducciones

Al codificar aplicaciones en lenguaje ensamblador, que es nuestro objetivo en este libro, dos de las operaciones que se efectúan de manera más habitual consisten en incrementar y reducir el valor de un cierto registro o valor almacenado en una posición de

memoria. Puesto que sabemos sumar y restar, mediante las instrucciones add y sub, no tendríamos mayor problema en sumarle o restarle 1 a cualquier operando.

Dichas operaciones, como acaba de decirse, son tan habituales que existen instrucciones específicas para ejecutarlas. Éstas son más efectivas que sumar o restar una unidad, generando un código más compacto y rápido ya que no hay necesidad de facilitar un operando adicional: el número 1 para incrementar o reducir.

Para incrementar el valor de un operando, ya sea registro o contenido de una posición de memoria, utilizaremos la instrucción inc. Ésta precisa como único parámetro el operando sobre el que va a actuarse. Para reducir el valor en una unidad, en lugar de incrementar, la instrucción a usar sería dec.

Al actuar sobre un cierto valor límite, al incrementar o reducir es posible que se produzca un desbordamiento. Suponga que tiene en AL el valor 255 y ejecuta una instrucción inc al1. En dicho registro quedaría el valor 0. El caso inverso se daría si tuviese el valor 0 y ejecutase la instrucción dec al1. Cuando se produce esta situación, el procesador activa un indicador de acarreo auxiliar o secundario. En DEBUG aparece como NA, cuando está desactivado, o AC, al activarse. En la figura 12.7 puede verlo tanto al reducir como al incrementar el valor de un registro provocando ese desbordamiento.

```

Símbolo del sistema - debug incdec.exe
All Rights Reserved

Loading file incdec.obj
matched Externs
matched ComDefs

D:\Ejemplos\12>debug incdec.exe
-r
AX=0000 BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC
1801:0000 B401          MOV    AH,01
-p

AX=0100 BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0002 NV UP EI PL NZ NA PO NC
1801:0002 FECC          DEC    AH
-p

AX=0000 BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0004 NV UP EI PL ZR NA PE NC
1801:0004 FECC          DEC    AH
-p

AX=FF00 BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 FS=17E1 SS=17F1 CS=1801 IP=0006 NV UP EI NG NZ AC PE NC
1801:0006 B8FEFF        MOV    AX,FFFF
-p

AX=FFE1 BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0009 NV UP EI NG NZ AC PE NC
1801:0009 40            INC    AX
-p

AX=FFFF BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=000A NV UP EI NG NZ NA PE NC
1801:000A 40            INC    AX
-p

AX=0000 BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=000B NV UP EI PL ZR AC PE NC

```

Figura 12.7. Estados del indicador de acarreo auxiliar al incrementar y reducir valores de un registro.

Aritmética BCD

Al efectuar cualquiera de las operaciones aritméticas antes descritas, asumimos siempre que estamos usando aritmética binaria. Es decir, el contenido de los registros se emplea como si fuese un conjunto de 8, 16, 32 ó 64 bits, según los casos, siendo necesario convertir a y desde la base decimal, trabajo que, en parte, queda oculto tanto por el ensamblador como por el depurador.

Esa conversión, no obstante, sí resultaría necesaria en caso de que necesitásemos mostrar los resultados en pantalla.

Suponga, por ejemplo, que tiene en el registro AL el valor 65. Para mostrarlos en pantalla, por ejemplo accediendo directamente a la memoria de texto como hemos hecho en ejemplos de capítulos previos, tendríamos que obtener a partir de ese dato el código del carácter '6' y el del carácter '5', ya que si asignásemos directamente el valor AL a una posición de pantalla lo que obtendríamos sería una letra 'A', a la que corresponde el código 65.

Existe una alternativa a la aritmética binaria: la aritmética BCD (*Binary Codea Decimal*) o aritmética decimal codificada en binario. Ésta consiste en emplear sólo los cuatro bits inferiores de cada byte, representando con ellos los números 0 a 9. Mediante este sistema, podríamos almacenar en el registro AX cualquier número comprendido entre 0 y 99, almacenando el primero de los dígitos en AH y el segundo en AL.

Nota

Recuerde que los dígitos 0 a 9 son los mismos para la numeración decimal y hexadecIMAL, y también que cada dígito de una cifra hexadecimal ocupa exactamente un *nibble*, por lo que puede utilizarse ésta última para asignar directamente un dígito a AL y olio a AH con una sola instrucción mov al registro AX.

Números BCD empaquetados y sin empaquetar

Cuando se emplea sólo un *nibble* de cada byte para almacenar un dígito, según se acaba de describir, se habla de números BCD sin empaquetar. Ésta es la forma más fácil de operar con números decimales y no tener que estar haciendo conversiones de forma continua.

A cambio, sin embargo, se desaprovecha una gran cantidad de espacio. Como se ha dicho antes, con los registros AL y AH podríamos representar los números 0 a 99 en BCD, mientras que con aritmética binaria pueden representarse 65536 valores distintos, aunque a costa de las conversiones desde y a decimal.

Un paso intermedio es el uso de números BCD empaquetados. Esto quiere decir que el byte se divide en dos *nibble*?, utilizando cada uno de ellos para representar un dígito numérico.

De esa forma, sólo con el registro AL podríamos representar los números del 0 al 99, mientras que con AX iríamos desde el 0 hasta el 9999. Seguimos perdiendo capacidad respecto a la aritmética binaria, pero en menor medida.

El inconveniente de los números BCD empaquetados es que, como puede suponer, es necesario combinar y separar los *nibbles* que representan a cada dígito cada vez que, por ejemplo, queremos mostrarlos en pantalla. A la hora de asignar un número, hay que tener en cuenta esa necesaria separación en dos *nibbles*.

Así, para almacenar el número 12 en el registro AL, como BCD empaquetado, no podríamos simplemente ejecutar la instrucción mov a1, 12. Esto introduciría el valor binario 00001100 en AL, en lugar de 00010010.

En la tabla 12.2 puede ver más claramente la diferencia entre codificar en decimal, hacerlo en BCD empaquetado y sin empaquetar, asumiendo siempre que estamos usando los registros AL y AH.

Tabla 12.2. Representación de números codificados en binario y BCD.

Codificación	Decimal	Hexadecimal	Binario (<i>nibble/nibble</i>)
Binaria	12	0C	AL=0000 1100
BCD sin empaquetar	12	0102	AH=0000 0001 AL=0000 0010
BCD empaquetado	12	12	AL=0001 0010
BCD sin empaquetar	2173	02010703	AH=0000 0111 AL=0000 0011
BCD empaquetado	2173	2173	AH=0010 0001 AL=0111 0011

Fijémonos, para empezar, en las tres representaciones hexadecimales del número 12. En la primera, la representación binaria, el número corresponde al dígito C hexadecimal, equivalente al 12 decimal.

Al usar la codificación BCD sin empaquetar, sin embargo, cada dígito debe ocupar 8 bits (cuatro para el dígito y otros cuatro que quedan siempre a cero) y, por ello, la representación es 0102. Por último tenemos el BCD empaquetado, en el que cada *nibble* es un dígito y el 12 decimal equivale al 12 hexadecimal. Recuerde que en este caso no debe efectuar la conversión de ese 12 hexadecimal a decimal usando el mecanismo descrito en un capítulo previo, sino simplemente tomar los dígitos directamente como decimales, al fin y al cabo es un número BCD (*Decimal codificado en binario*).

¿Qué ocurre cuando queremos representar un número más grande, por ejemplo el 2173? Sin empaquetar, cada dígito necesita un byte, por ello la representación hexadecimal, en la que cada dígito equivale exactamente a un *nibble*, se compone de los mismo dígitos precedidos de ceros.

Al pasar a binario, puesto que es un número de cuatro dígitos y cada uno de ellos necesita un byte, está claro que necesitaríamos cuatro registros. En la tabla 12.2 se han mostrado los valores de AH y AL, siendo precisos dos más.

Por último tenemos el mismo número en BCD empaquetado, caso en el que cabe sin problemas en el registro AX.

Suma de números BCD

Ahora que ya sabemos qué es un número BCD, veamos cómo podemos operar con ellos ya que existen algunas peculiaridades. Las instrucciones para sumar, restar, multiplicar y dividir son exactamente las mismas, pero, como puede suponer, el resultado no aparece directamente en BCD en los registros afectados. Pruebe, simplemente, a ejecutar el código siguiente:

```
mov al, 5  
add al, 7
```

Observe el contenido que queda en AL, ejecutando el programa desde DEBUG. Verá que AX contiene el valor 000c o, lo que es lo mismo, 00000000 00001100 en binario. Salta a la vista que es necesario un ajuste posterior a la suma, que diferirá dependiendo de que optemos por operar con BCD empaquetado o sin empaquetar. Por ello existen dos instrucciones para llevar a cabo dicho ajuste: aaa y daa. La primera de ellas opera sobre números BCD sin empaquetar y la segunda cuando son números BCD empaquetados.

En cualquier caso, tanto aaa como daa asumen que el resultado de la suma BCD, un byte, ha quedado en el registro AL. Al efectuar el ajuste, estas instrucciones tienen en cuenta no sólo el valor que contiene AL sino también ciertos indicadores, como el de acarreo, y también pueden activar esos mismos indicadores en caso de que el ajuste provoque un desbordamiento. De esta manera se facilita la suma de sucesiones de dígitos BCD, que pueden ir recuperándose de la memoria, sumándose y devolviéndose a una localización de memoria.

Introduzca en un archivo el código mostrado a continuación, con la misma suma seguida de los dos tipos de ajuste, para apreciar los diferentes resultados que obtendría:

```
segment Pila stack  
    resh 256  
  
; Seqmento de código  
segment Código  
.start:  
  
    mov al, 5  
    add al,7  
    aaa ; ajuste BCD no empaquetado  
  
    mov ax, 0 ; Poner AX a 0  
  
    mov al, 5  
    add al, 7  
    daa ; ajuste BCD empaquetado  
  
    ; salimos al sistema  
    mov ah, <3dh  
    int 21h
```

Las dos sumas generan exactamente el mismo resultado, alojando en el registro AX, según puede verse desde DEBUG, el valor 000C, que es el 12 decimal. Tras ejecutar la

instrucción aaa, el número se convierte en BCD desempaquetado, quedando el 1 en el registro AH y el 2 en el registro AL. Puede verlo en la figura 12.8, en la que, tras desensamblar el código para obtener las direcciones, se ha ejecutado directamente hasta las instrucciones aaa y daa.

Al ajustar el mismo resultado con la instrucción da a, asumiendo que es un número BCD empaquetado, puede ver en la misma figura que los dos dígitos, formando el número 12, quedan en AL, mientras que AH permanece a cero. Esto, como se ha dicho antes, facilitaría el trabajo con números BCD mayores.

The screenshot shows a debugger window titled "Simbolo del sistema - debug bcd.exe". The assembly code is as follows:

```

AX=0000 BX=FFFF CX=FF54 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=000B NV UP EI PL ZR AC PE NC
1801:000B B009      MOV     AL ,09
-
-q

D:\Ejemplos\12>debug bcd.exe
-u
1801:0000 B005      MOV     AL ,05
1801:0002 0407      ADD     AL ,07
1801:0004 37        AAA
1801:0005 B80000    MOV     AX,0000
1801:0008 B005      MOV     AL ,05
1801:000A 0407      ADD     AL ,07
1801:000C 27        DAA
1801:000D B86501    MOV     AX,0165
1801:0010 054300    ADD     AX,0043
1801:0013 27        DAA
1801:0014 B00400    ADC     AH,00
1801:0017 B44C      MOV     AH,4C
1801:0019 CD21      INT     21
1801:001B 1D13C5    SBB     AX,C513
1801:001E 1F        POP     DS
1801:001F 3E        DS:
1801:0020 06        PUSH    ES
-g 5

AX=0102 BX=FFFF CX=FF5B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0005 NV UP EI PL NZ AC PE CY
1801:0005 B80000    MOV     AX,0000
-g d

AX=0012 BX=FFFF CX=FF5B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17F1 SS=17F1 CS=1801 IP=000D NV UP FI PL NZ AC PE NC
1801:000D B86501    MOV     AX,0165

```

Figura 12.8. Ajustes BCD de la suma.

Empleando números BCD empaquetados, y teniendo en cuenta el uso del indicador de acarreo, podemos efectuar operaciones como la siguiente:

```

mov ax, 165h
add ax, 43h
da a
ade ah,0

```

Queremos sumar los números decimales 165 y 43, codificándolos como BCD empaquetados. En lugar de tratar cada *nibble* por separado, ya que sabemos que en hexadecimal cada dígito corresponde a un *nibble* usamos esa base de numeración para introducir directamente los valores en los registros AH y AL.

El ajuste, efectuado con la instrucción daa, actúa sobre el registro AL y, en caso de que el valor de éste sea superior a 9, activaría el indicador de acarreo. Por eso sumamos con acarreo al registro AH, para recoger ese posible incremento.

Como puede verse, el resultado inicial de la suma (véase la figura 12.9) es 01A8, que se convierte en 0108 al ejecutar la instrucción daa, activándose, como se aprecia en la parte derecha, el indicador de acarreo. Esto hace que se incremente el valor de AH y obtengamos finalmente el resultado correcto: 208. Podría cambiar los dos números por otros dos cualesquiera y observar el resultado.

The screenshot shows a Windows-style debugger window titled "Símbolo del sistema - debug bcd.exe". The assembly code is as follows:

```
AX=0012 BX=FFFF CX=FF5B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=000D NV UP EI PL NZ AC PE NC
1801:0000  B86501    MOV     AX,0165
-p

AX=0165 BX=FFFF CX=FF5B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0010 NV UP EI PL NZ AC PE NC
1801:0010  054300    ADD     AX,0043
-p

AX=01A8 BX=FFFF CX=FF5B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0013 NV UP EI PL NZ NA PO NC
1801:0013  27        DAA
-p

AX=0108 BX=FFFF CX=FF5B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0014 NV UP EI PL NZ NA PO CY
1801:0014  80D400    ADC     AH,00
-p

AX=0208 BX=FFFF CX=FF5B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0017 NV UP EI PL NZ NA PO NC
1801:0017  B44C      MOV     AH,4C
-
-
-

```

Figura 12.9. Suma de números BCD empaquetados con acarreo.

Nota

Recuerde que las instrucciones de ajuste BCD siempre actúan sobre el registro AL y, de manera secundaria, en el registro de indicadores. No puede usarlas, por tanto, si el resultado de la suma lo tiene almacenado en otro registro.

Otras operaciones con números BCD

Hn puntos previos ha conocido, aparte de las instrucciones de suma, otras para efectuar restas, multiplicaciones y divisiones. Todas esas operaciones pueden llevarse a cabo con números BCD sin empaquetar y, en el caso de la resta, también con números BCD empaquetados. El ajuste de los resultados obtenidos, sin embargo, no puede llevarse

a cabo con aaa ni daa, sino que existen instrucciones específicas para cada una de las operaciones. Éstas son:

- aas: Ajuste BCD sin empaquetar tras una resta.
- das: Ajuste BCD empaquetado tras una resta.
- aam: Ajuste BCD sin empaquetar tras la multiplicación.
- aad: Ajuste BCD sin empaquetar antes de la división.

La utilización de aas y das es básicamente el mismo que hemos visto en el punto previo para aaa y daa, respectivamente. La obvia diferencia es que sigue a una instrucción sub o sbc, no a una suma. Actúan sobre el registro AL y pueden activar los indicadores de acarreo principal y auxiliar.

Mediante la instrucción aam se ajusta el contenido del registro AX tras efectuar una multiplicación con dos operandos de 8 bits. El mecanismo, como se aprecia en el ejemplo siguiente, es también similar al empicado con la suma.

```
segment Pila stack
    resb 2S6

    ; Segmento de código
    segment Código
..start:

    mov al, fi ; multiplicamos
    mov bl, 3 ; 8 por 3
    muí bl

    aam      ; ajuste BCD

    ; salimos al sistema
    mov ah, 4ch
    int 21h
```

La figura 12.10 muestra el contenido del registro AX antes y después de efectuar el ajuste. Como puede verse, el número se separa en dos dígitos BCD desempaquetados, quedando el primero en AH y el segundo en AL, como es habitual en las demás operaciones antes efectuadas. A diferencia de lo que ocurre con el resto de operaciones, el ajuste BCD para la división se debe efectuar antes de la propia división, y no después. El dividendo, en BCD desempaquetado, se introducirá en el registro AX, tras lo cual debe ejecutarse la instrucción aad. Realizada la división, AL contendrá el cociente y AL I el resto, como en cualquier otra división. Compruébelo con el código siguiente, en el que se divide 15 entre 7. Observe el estado del registro en la figura 12.11: AL contiene 2, el cociente, y AH almacena 1, el resto.

```
mov ax, 0105h
mov bl, 7

aad    ; Ajuste BCD
div bl
```

D:\Ejemplos\12>debug MulBCD.exe
-q
D:\Ejemplos\12>debug MulBCD.exe
-r
AX=0000 BX=FFFF CX=FF4C DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC
1801:0000 B008 MOV AL,08
-p
AX=0008 BX=FFFF CX=FF4C DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0002 NV UP EI PL NZ NA PO NC
1801:0002 B303 MOV BL,03
-p
AX=0008 BX=FF03 CX=FF4C DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0004 NV UP EI PL NZ NA PO NC
1801:0004 F6E3 MUL BL
-p
AX=0018 BX=FF03 CX=FF4C DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0006 NV UP EI PL NZ NA PO NC
1801:0006 D40A AAM
-p
AX=0204 BX=FF03 CX=FF4C DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0008 NV UP EI PL NZ NA PO NC
1801:0008 B44C MOV AH,4C

Figura 12.10. Observamos el ajuste del resultado de la multiplicación.

D:\Ejemplos\12>debug divbcd.exe
-r
AX=0000 BX=FFFF CX=FF4D DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17F1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC
1801:0000 B80501 MOV AX,0105
-p
AX=0105 BX=FFFF CX=FF4D DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0003 NV UP EI PL NZ NA PO NC
1801:0003 B307 MOV BL,07
-p
AX=0105 BX=FF07 CX=FF4D DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0005 NV UP EI PL NZ NA PO NC
1801:0005 D50A AAD
-p
AX=000F BX=FF07 CX=FF4D DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0007 NV UP EI PL NZ NA PE NC
1801:0007 F6F3 DIV BL
-p
AX=0102 BX=FF07 CX=FF4D DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0009 NV UP EI PL NZ NA PE NC
1801:0009 B44C MOV AH,4C

Figura 12.11. El ajuste se efectúa antes de ejecutar la división.

Negativos, palabras y dobles palabras

Como se ha visto en alguno de los ejemplos de los puntos previos, para introducir un número negativo en un registro o una posición de memoria no es necesario efectuar manualmente el complemento a dos, ya que de ello se ocupa el propio ensamblador.

Puede darse el caso, sin embargo, de que tengamos un número positivo y necesitemos convertirlo en negativo. Para ello, como ya sabe, deberíamos invertir el estado de cada uno de los bits del valor y, a continuación, incrementarlo en una unidad.

La instrucción neg, aplicado a un operando que puede ser un registro o una dirección de memoria, se encarga precisamente de efectuar esa conversión. Si lo aplica dos veces al mismo operando, obtendría de nuevo el valor original, es decir, puede tanto convertir un número negativo en positivo como hacer la operación complementaria.

Introduzca el código siguiente en un archivo, ensámblelo y, por último, ejecútelo paso a paso con DEBUG. Fíjese en el contenido de AX tras ejecutar cada una de las instrucciones neg.

```
segment Pila stack
    resb 256

; Segmento de código
segment Código
..start:

    mov ax, 5
    neg ax

    neg ax

; salimos al sistema
    mov ah, 4ch
    int 21h
```

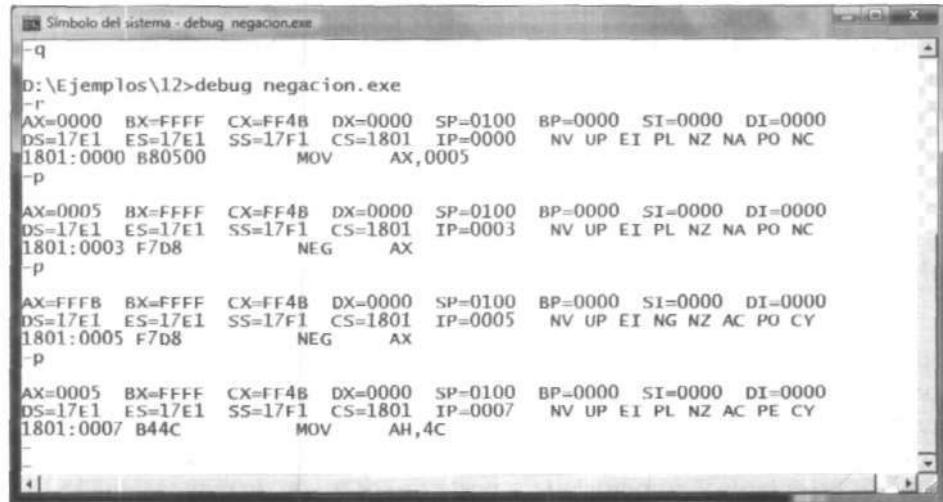
Puede utilizar la instrucción neg, como se ha dicho antes, sobre cualquier operando y de cualquier tamaño, siempre que sea un registro o un valor alojado en memoria. No se puede emplear, por tanto, con valores inmediatos.

Fíjese de nuevo en la figura 12.12, poniendo especial atención al área del registro de indicadores. Al asignar el valor a AX el indicador de signo, que puede tomar los estados PL/NG, nos comunica que el número es positivo. Tras ejecutar el primer neg, sin embargo, su estado cambia de PL a NG, indicando que ahora el número es negativo.

Al operar con números positivos, en los que todos los bits son utilizados para representar el valor, nada nos impide usar como una palabra lo que antes era un byte. Podemos, por ejemplo, obtener un valor en AL como resultado de una suma y, a continuación, asignar 0 a AH para usar el resultado de la suma haciendo referencia a AX. Esto es necesario en ciertas situaciones en las que se precisa un registro de un cierto tamaño, y no de otro.

Si el número tiene signo, por el contrario, la conversión de byte a palabra no resultará tan fácil. No podríamos, sin más, poner a 0 AH y dejar el valor en AL para usarlos conjuntamente, como AX, ya que el bit de signo, que en AL se encuentra en el séptimo bit, debería pasar al decimoquinto de AX y, además, los demás bits de AH deberían tomar

un valor u otro dependiendo de que el bit de signo esté activo o no, es decir, según el número sea negativo o positivo.



The screenshot shows a Windows command-line interface window titled "Símbolo del sistema - debug negacion.exe". The command entered is "D:\Ejemplos\12>debug negacion.exe". The assembly code displayed is:

```
D:\Ejemplos\12>debug negacion.exe
-r
AX=0000 BX=FFFF CX=FF4B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0000 NV UP EI PL NZ NA PO NC
1801:0000 B80500    MOV     AX,0005
-p
AX=0005 BX=FFFF CX=FF4B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0003 NV UP EI PL NZ NA PO NC
1801:0003 F7D8    NEG     AX
-p
AX=FFFF BX=FFFF CX=FF4B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0005 NV UP EI NG NZ AC PO CY
1801:0005 F7D8    NEG     AX
-p
AX=0005 BX=FFFF CX=FF4B DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=17E1 ES=17E1 SS=17F1 CS=1801 IP=0007 NV UP EI PL NZ AC PE CY
1801:0007 B44C    MOV     AH,4C
-p
```

Figura 12.12. Una doble negación deja el valor en su estado original.

Para evitar tener que comprobar el estado del bit de signo y actuar en consecuencia, podemos emplear la instrucción cbw. Ésta se ocupa de tomar el valor de 8 bits que hay en AL y convertirlo en un valor de 16 bits, que quedaría en AX, extendiendo el bit de signo. Si el valor original es de 16 bits, una palabra, y queremos ampliarlo a 32, una doble palabra, utilizaremos la instrucción cwd. Esta deja el resultado en la pareja de registros DX: AX, conteniendo AX la palabra de menor peso y DX la de mayor. Pruebe simplemente, a introducir un valor en AL, positivo o negativo, y ejecutar después las instrucciones cbw y cwd para ampliarlo a palabra y doble palabra, viendo el resultado.

Uso de la unidad de punto flotante

Los actuales microprocesadores, todos los Pentium, Core, sucesores y compatibles, se caracterizan por incorporar en el mismo circuito integrado en el que se encuentra el procesador principal una unidad adicional para operaciones en punto flotante, lo que se conoce como FPU (*Floating Point Unit*). Ésta es una característica con la que no contaban los 8086/8088/80186/80286/80386, siendo necesario adquirir dicha FPU por separado, era un integrado que se llama igual que el microprocesador principal pero cambiando el 6 final por un 7. Tal como se indicaba al inicio del capítulo, esto cambió con la aparición del 80486, primer microprocesador de la familia Intel que integraba el *coprocesador matemático*, o FPU, en el propio circuito del micro.

Actualmente es difícil contar con un ordenador que sea inferior a un 80486, e incluso a un Pentium o similar, por lo que podemos asumir que la FPU siempre está presente y

hacer uso de ella. En los sistemas más antiguos, con 80386 o un procesador previo, esta suposición no podía hacerse.

Mediante la FPU es posible efectuar cálculos mucho más complejos que los llevados a cabo con las instrucciones aritméticas explicadas en los puntos previos, además con números, tanto enteros como con parte decimal, de mayor tamaño. Su uso, no obstante, cuenta con una cierta complejidad, ya que la FPU dispone de un conjunto de registros propios e instrucciones específicas para su uso. Por ello, lo que encontrará a continuación no pasa de ser una breve introducción al aprovechamiento de las posibilidades de la FPU del procesador.

Registros de la FPU

En capítulos previos ha conocido algunos de los registros con que cuentan los procesadores de la familia x86, principalmente registros de segmento, de propósito general, puntero de instrucción y registro de indicadores. Todos estos registros son utilizados por el núcleo de la CPU para ir recuperando y ejecutando instrucciones, pero resultan insuficientes a la hora de efectuar ciertas operaciones matemáticas, especialmente con números no enteros y números que, a pesar de serlo, son de mayor tamaño.

Esta es la razón de que la FPU cuente con su propio conjunto de registros, específico para el almacenamiento y manipulación de números con gran precisión. Para efectuar cualquier operación matemática, lo habitual es que al menos uno de los operandos, en ocasiones los dos, se encuentre en esos registros.

La FPU cuenta con ocho registros dispuestos en forma de pila y denominados como se aprecia en la figura 12.13. Al primero de ellos, ST (0), se hace referencia habitualmente como ST (*Stack Top*, Cabecera de pila).

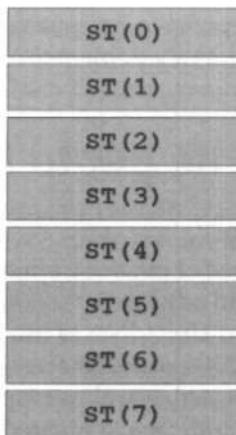


Figura 12.13. Registros de la FPU.

Cada uno de estos registros tiene una capacidad de 80 bits, pudiendo alojar varios tipos de datos diferentes tanto enteros como con parte decimal (véase el capítulo dedicado

a la representación de información). Las instrucciones que actúan sobre la FPU pueden introducir valores en la pila de registros, operar sobre ellos y extraer valores. Cuando se introduce un nuevo valor éste va a parar a ST(0), que pasa su valor a ST(1), éste a ST(2) y así sucesivamente hasta ST(7), que tomará el valor del anterior para perder el suyo. Al efectuar la mayoría de las operaciones el resultado se almacena también en ST(0), desplazando si es necesario toda la pila de registros.

Tipos de datos

Mediante la FPU podemos operar sobre tres tipos numéricos: enteros, BCD y reales. A escala interna, no obstante, la FPU efectúa todos los cálculos como si los números fueran reales, números decimales en coma flotante que se componen, como ya sabe, de un signo, una mantisa y un exponente.

Los números enteros pueden ser de 16, 32 ó 64 bits, todos ellos con signo. Mediante los enteros de 64 bits, en codificación binaria, es posible trabajar con grandes números enteros, con signo, sin necesidad de emplear varios registros. Los campos para almacenar en memoria estos números podemos definirlos mediante dw, dd y dq, como haríamos con cualquier otro número.

Ya sabemos que cualquier número en binario, a la hora de ser mostrado en pantalla o solicitado al usuario por teclado, debe convertirse a y desde la base decimal. También sabemos que los números BCD nos ahorrarán esa conversión. La FPU nos permite trabajar con números BCD de 80 bits, toda la capacidad del registro, usando 9 bytes, de los 10 que caben en sus registros, para alojar 18 dígitos BCD empaquetados. El último byte, concretamente su último bit, se deja para señalizar el signo del número.

En comparación con el uso de aritmética BCD desempaquetada, en la que sólo podemos representar los valores 0 a 99 en el registro AX, mediante un registro de la FPU, siempre con BCD empaquetado, podemos representar los números -999.999.999.999.999.999 a 999.999.999.999.999.999, operando sobre ellos sin ningún problema de ajuste posterior ni nada parecido.

En cuanto a los tipos en coma flotante, o reales, también existen tres tamaños: 32 bits, 64 bits y 80 bits. El último bit, como siempre, se utiliza para indicar el signo, dividiéndose los bits restantes en dos partes: una mantisa que contiene los dígitos significativos, por una parte, y un exponente, por otra.

Introducción de datos en la FPU

El primer paso a dar siempre que deseemos efectuar cálculos empleando la FPU, consiste en alojar en sus registros el operando u operandos que se necesiten. Para ello existen varias instrucciones, dependiendo del tipo de dato de que se trate. Las posibilidades son:

- fId: Puede tomar como operando un número real o bien uno de los registros de la FPU. Lo que hace es introducirlo en ST.

- **f i Id:** Introduce en ST un número entero, facilitado como argumento tras esta instrucción.
- **f bld:** Introduce en ST un número BCD empaquetado que debe facilitarse como argumento.

En los tres casos lo que se entrega a la instrucción no es un valor inmediato, el valor en sí, sino una dirección de memoria donde se encuentra dicho dato. La operación almacena el valor en ST y, actuando como una pila, todos los demás se desplazarían hacia abajo.

La introducción de ciertos valores, más habituales, puede realizarse directamente con instrucciones específicas, sin necesidad de utilizar las anteriores. Algunas de ellas son **fldz**, **fldl** y **fldpi**. La primera de ellas introduce en ST el valor 0, la segunda el valor 1 y la tercera el valor del número PI.

Ejecución de operaciones

Una vez que tenemos los operandos en los registros de la FPU, el paso siguiente será actuar sobre ellos efectuando el cálculo que nos interese. Además de la suma, resta, multiplicación y división, operaciones que contempla el núcleo del procesador, tenemos a nuestra disposición otras más complejas, como la potenciación, raíz cuadrada, senos, tangentes, etc. Algunas de las instrucciones cuentan con distintas versiones, como le ocurre a **f Id**, dependiendo de que los operandos sean enteros, reales o BCD. Aparte de las instrucciones para ejecutar cálculos, también existen otras para efectuar comparaciones. De forma similar al procesador, el coprocesador o FPU cuenta con un registro de indicadores de estado, que pueden comprobarse para saber la situación posterior a la ejecución de una cierta operación. Algunas de las instrucciones aritméticas disponibles son:

- **f abs:** Haya el valor absoluto del dato que se encuentra en ST, dejándolo en el mismo registro.
- **f add:** Suma dos operandos, que pueden ser dos registros de la FPU o un dato externo que se suma a ST. Como en los demás casos, el resultado siempre queda en ST.
- **f chs:** Invierte el signo del número que haya en ST.
- **feos:** Haya el coseno del valor almacenado en ST, que debe estar en radianes.
- **f div:** Divide un operando entre otro, facilitando cociente y resto.
- **f muí:** Multiplica los dos operandos, que pueden ser dos registros o bien un operando externo y ST como operando implícito.
- **frndint:** Redondea un número decimal a número entero.
- **fseal:** Eleva el número 2 al exponente indicado, aplicándole un multiplicador.
- **fsqrt:** Haya la raíz cuadrada del valor que contiene ST.
- **fsub:** Resta un operando de otro.

Recuperación de datos de la FPU

La mayoría de las operaciones de la FPU dejan el resultado en la parte alta de la pila de registros, es decir, en ST. Para poder usarlos en la aplicación, ya sea mostrándolos en pantalla o utilizándolos en algún proceso, necesitaremos recuperarlos. Esta acción puede implicar o no la extracción del dato de la pila, de tal forma que todos los inferiores subirán un nivel.

Al igual que ocurre con las instrucciones vistas antes para la introducción de datos en los registros, existen instrucciones de extracción para cada uno de los tipos de dato posibles. La instrucción base es f st, aplicable a números reales. En caso de que no tan sólo deseemos recuperar el valor, en la posición que indiquemos de la memoria, sino también extraerlo de la pila del coprocesador, usaremos f stp en su lugar.

Para recuperar valores enteros usaremos f ist o f istp, dependiendo de que además deseemos extraerlo de la pila. Por último, para los números BCD sólo existe la instrucción fbstp que, como puede suponer, recupera el valor y lo extrae de la pila.

Un sencillo ejemplo

Ahora que conocemos los pasos básicos para hacer uso de la FPU o coprocesador matemático, vamos a servirnos de un sencillo ejemplo para ponerlos en práctica. Dicho ejemplo efectuará dos operaciones: una multiplicación y una raíz cuadrada. Los valores se recuperarán en un campo, previamente declarado en el segmento de datos, para poder examinarlos desde DEBUG.

El código del programa, ampliamente comentado, es el siguiente:

```
segment Pila stack
    resb 256

    ; Segmento de datos
    segment Datos
Multiplicando dw 3 ; Multiplicaremos
Multiplicador dw 5 ; 3 por 5
Dato dw 256 ; y hallaremos la raiz cuadrada de 256

; para almacenar los resultados
Resultado dw 0

    ; Segmento de código
    segment Código
..start:

    ; Hacemos que Dí apunte al
    ; segmento de datos
    mov ax, Datos
    mov ds, ax

    ; introducimos en ST el primer
    ; operando de la multiplicación
    fild word [Multiplicando]
```

```

; y multiplicamos por el segundo
fimul word [Multiplicador]

; extraemos el resultado de ST
fistp word [Resultado]

; Introducimos en ST el Dato
fld word [Dato]

fsqrt ; para hallar su raíz cuadrada

; recuperamos el resultado
fistp word [Resultado]

; salimos al sistema
mov ah, 4ch
int 21h

```

Ejecute el programa en dos partes, primero la multiplicación y después la raíz cuadrada. Es lo que se ha hecho en la figura 12.14.

The screenshot shows a Windows command-line interface window titled "Símbolo del sistema - debug fpu.exe". The command entered is "D:\Ejemplos\12>debug fpu.exe". The assembly code is displayed in two columns:

1802:0000 B80118	MOV AX,1801
1802:0003 8ED8	MOV DS,AX
1802:0005 DF060000	FILD WORD PTR [0000]
1802:0009 DE0E0200	FIMUL WORD PTR [0002]
1802:000D DF1E0600	FISTP WORD PTR [0006]
1802:0011 DF060400	FILD WORD PTR [0004]
1802:0015 D9FA	FSQRT
1802:0017 DF1E0600	FISTP WORD PTR [0006]
1802:001B B44C	MOV AH,4C
1802:001D CD21	INT 21
1802:001F 75EF	JNZ 0010

Below the assembly code, the CPU registers are shown:

AX=1801 BX=FFFF CX=FF7F DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=1801 ES=17E1 SS=17F1 CS=1802 IP=0011 NV UP EI PL NZ NA PO NC
1802:0011 DF060400 FILD WORD PTR [0004] DS:0004=01
00

Registers (ds:0):

1801:0000 03 00 05 00 00 01 0F 00-00 00 00 00 00 00 00 00 00 00
1801:0010 B8 01 18 8E D8 DF 06 00-00 DE 0E 02 00 DF 1E 06
1801:0020 00 DF 06 04 00 D9 FA DF-1E 06 00 B4 4C CD 21 75L.lu
1801:0030 EF AD 13 37 9B 70 52 AF-66 AE BB F8 37 BA DF 1E	...7.pR.F..7...
1801:0040 6B 58 A0 C8 65 C0 B1 4F-8D 41 DE 5E 44 CB 62 83	kx..e..O.A.^D.b.
1801:0050 B8 B5 A2 E3 71 7D 55 5E-79 3E 72 AC 7C 57 A7 6C	...q]U^y>r. W.1
1801:0060 53 94 BE 36 2F 0C 40 3D-23 E8 DC C6 43 3F E6 EE	5..6/.@#=..C?..
1801:0070 5D 4A 3A CE 72 65 56 C0-54 C4 72 9B 5F 90 24 9D]J:.rev.T.r._\$.

Registers (ds:1):

1801:0000 03 00 05 00 00 01 10 00-00 00 00 00 00 00 00 00 00 00
1801:0010 B8 01 18 8E D8 DF 06 00-00 DE 0E 02 00 DF 1E 06
1801:0020 00 DF 06 04 00 D9 FA DF-1E 06 00 B4 4C CD 21 75L.lu
1801:0030 EF AD 13 37 9B 70 52 AF-66 AE BB F8 37 BA DF 1E	...7.pR.F..7...
1801:0040 6B 58 A0 C8 65 C0 B1 4F-8D 41 DE 5E 44 CB 62 83	kx..e..O.A.^D.b.
1801:0050 B8 B5 A2 E3 71 7D 55 5E-79 3E 72 AC 7C 57 A7 6C	...q]U^y>r. W.1
1801:0060 53 94 BE 36 2F 0C 40 3D-23 E8 DC C6 43 3F E6 EE	5..6/.@#=..C?..
1801:0070 5D 4A 3A CE 72 65 56 C0-54 C4 72 9B 5F 90 24 9D]J:.rev.T.r._\$.

Figura 12.14. Proceso de ejecución y examen de los resultados generados por la FPU.

Tras efectuar la multiplicación, y ejecutar la instrucción que extrae el resultado a memoria, usamos el comando d de DEBUG para ver el contenido del segmento de datos. Los dos primeros bytes corresponden al campo Multiplicando, los dos siguientes a Multiplicador, los dos siguientes a Dato y los dos siguientes, que son los que nos interesan, forman el campo Resultado.

Recuerde que una palabra, al ser almacenada en memoria, se divide en dos bytes que se almacenan en orden inverso. El valor 0F 00, por tanto, habría que interpretarlo como 00 0F, es decir, 15, resultado de multiplicar el valor de 5 por 3. Obviamente, puede probar con números mucho más grandes, definiendo los operandos, y el receptor del resultado, como dobles o cuádruples palabras.

Tras ejecutar la instrucción fsqrt, que halla la raíz cuadrada del valor previamente introducido en ST, vemos que Resultado contiene el valor 0010, que es 16. Si lo comprueba, 16 por 16 son 256, por tanto 16 es la raíz cuadrada de 256.

Resumen

Como ha podido ver en este capítulo, el lenguaje ensamblador de la familia x86 cuenta con instrucciones para efectuar las operaciones aritméticas básicas sobre números enteros, así como una FPU, o coprocesador matemático, que amplía las posibilidades a otras operaciones y tipos de datos adicionales, como números BCD grandes y números reales.

Ta mayor parte de las aplicaciones no precisan cálculos complejos, por lo que el uso del coprocesador matemático es menos habitual que el de las instrucciones propias del procesador. Hay que tener en cuenta, no obstante, que las mismas operaciones matemáticas, incluidas sumas, restas, multiplicaciones y divisiones, se ejecutan con más rapidez en el coprocesador matemático.

En capítulos posteriores nos serviremos de las instrucciones que hemos conocido en éste para algunos ejemplos, especialmente una vez sepamos cómo codificar bucles y condicionales, lo cual nos permitirá la implementación de lógicas con mayor complejidad.

13

Condicionales

Los ejemplos de capítulos previos, muy sencillos, se han ejecutado siempre de principio a fin. Esto es normal al realizar pequeñas pruebas, pero en aplicaciones reales no se ejecuta todo el código del programa cada vez que éste se inicia. Dependiendo de ciertas condiciones, algunas sentencias pueden ejecutarse o no. Para verificar esas condiciones, y proceder a la ejecución de unos bloques u otros de sentencias, necesitaremos conocer algunos elementos nuevos. El primer paso será profundizar en algunos bits adicionales del registro de indicadores, introducido en un capítulo previo, al tratar la arquitectura de los microprocesadores x86, y que usábamos por primera vez en el capítulo anterior para poder gestionar adecuadamente los acarreos de algunas operaciones aritméticas. Aprenderemos a comparar los valores de los registros, o ciertos bits de éstos, actuándose sobre el registro de indicadores pero sin perder los valores de los registros que actúen como operandos. Partiendo del estado de los bits del registro de indicadores, conoceremos ciertas instrucciones que transfieren la ejecución a un cierto punto del programa.

Por último, en cuanto a este capítulo, conocerá una serie de instrucciones que facilitan la manipulación individual de bits, lo cual puede ser muy interesante en ciertos casos. Recuperando el registro de indicadores en un registro de propósito general, a través de la pila, obtendremos más flexibilidad a la hora de comprobar ciertas situaciones.

El registro de indicadores

El registro de indicadores *oflags* se compone de 16 bits, 32 en el caso de los procesadores 80386 y posteriores, utilizados para señalizar ciertas situaciones. En el capítulo previo, por ejemplo, vimos cómo se activaban los bits de acarreo y acarreo auxiliar en

el momento en que una cierta operación, como puede ser la suma, causaba un desbordamiento del destino.

Cada uno de los bits del registro de indicadores es conocido normalmente mediante una letra, aparte de tener un nombre indicativo de su significado. Teniendo en cuenta su orden en el registro, desde el bit 0, el que está más a la derecha, hasta el 15, recordemos brevemente cuál es la posición, denominación y finalidad de cada uno de ellos:

- **Bit 0 - CF:** Indicador de acarreo. Cuando está a 1 indica que la última operación aritmética, típicamente una suma o resta, ha producido un acarreo.
- **Bit 1-1:** Primero de los bits del registro de indicadores que no tiene uso o su uso está reservado. Éste concretamente se encuentra siempre a 1.
- **Bit 2 - PF:** Indicador de paridad. Se pone a 1 cuando el resultado de la última operación efectuada cuenta con un número par de bits. Como su propio nombre indica, es útil para comprobar la paridad de un cierto dato, es decir, comprobar si ese dato tiene un número par o impar de bits.
- **Bit 3-0:** Bit sin uso. Se encuentra siempre a cero.
- **Bit 4 - AF:** Indicador auxiliar de acarreo. Se pone a 1 cuando se produce un acarreo en una operación BCD, según pudo verse en el capítulo previo.
- **Bit 5 - ?:** Bit sin uso: Se encuentra siempre a cero.
- **Bit 6 - ZF:** Indicador de cero. Se pone a 1 cuando el resultado de la última operación efectuada ha sido 0. Como verá después, es un bit fundamental a la hora de comparar valores.
- **Bit 7 - SF:** Indicador de signo. Se pone a 1 cuando el resultado de la última operación tiene a 1 el bit de signo. Lo usábamos en el capítulo previo para comprobar si un número era o no negativo.
- **Bit 8 - TF:** Indicador de interrupción. Cuando está puesto a 1, el procesador genera una interrupción a cada instrucción ejecutada. La utilizan los depuradores para ir ejecutando paso a paso los programas.
- **Bit 9 - IF:** Indicador de activación de interrupciones. A medida que va ejecutando las instrucciones de un programa el procesador puede recibir interrupciones, generadas por el hardware, que devíen su atención a otro punto, por ejemplo recogiendo las pulsaciones del teclado. Esta atención a las interrupciones puede desactivarse poniendo a 0 este bit que, generalmente, está a 1. Para manipular su estado se usan las instrucciones `cli`, que lo pone a 0, y `sti`, que lo pone a 1.
- **Bit 10 - DF:** Indicador de dirección. Este bit determina si ciertas instrucciones que transfieren bloques de datos lo hacen en dirección ascendente o descendente. Existe una instrucción para activar este bit: `std`, y otra para desactivarlo: `cld`.
- **Bit 11 - OF:** Indicador de desbordamiento. Se pone a 1 cuando el resultado de una operación es demasiado grande para el destino especificado.

- Bits 12-15: No tienen aplicación en procesadores anteriores al 80386, si bien en dicho procesador indican si el modo de trabajo es el modo virtual 8086 o si está ejecutándose una tarea anidada.

Además de las instrucciones que se ha mencionado, y que permiten activar o desactivar los indicadores de dirección e interrupciones, existen tres más relacionadas con el registro de indicadores.

Las tres actúan sobre un mismo bit: el de acarreo. Con la instrucción `ele` lo pondremos a cero, con `ste` lo pondremos a 1 y mediante `eme` lo invertimos.

Obtención y restauración del registro de indicadores

El microprocesador cuenta con instrucciones que permiten actuar sobre indicadores concretos, como el de dirección o activación de interrupciones, pero no existe una instrucción genérica para comprobar o modificar cualquier indicador, ni tampoco instrucciones específicas para todos ellos.

Existe, no obstante, la posibilidad de obtener todos los indicadores del registro usando la pila como intermediario.

La pila es un espacio de almacenamiento temporal, indispensable en cualquier aplicación que efectúe llamadas a rutinas o emplee las instrucciones `push/pop` para guardar el contenido de cualquier registro a fin de recuperarlo posteriormente. Hasta ahora no nos hemos preocupado especialmente de la pila, aunque hemos definido un segmento de pila en todos nuestros programas ya que es algo obligatorio en el formato ejecutable EXE del DOS.

Al igual que con el segmento de datos y el registro `ds`, antes de usar cualquier operación con la pila es necesario inicializar el registro de segmento, `ss` en este caso, así como el puntero de pila: `sp`. Dado que la pila es una estructura de datos que crece en orden inverso, desde la dirección más alta hasta la más baja, el valor inicial del registro `sp` deberá ser la dirección del final de la pila.

Teniendo la pila adecuadamente configurada, para almacenar en ella el contenido del registro de indicadores utilizaremos la instrucción `pushf`. Ésta no precisa parámetro alguno, limitándose a introducir el valor actual del registro de indicadores en la dirección `SS : SP` y reduciendo el valor del registro `SP` en previsión del almacenamiento de un nuevo valor.

Una vez que tenemos los indicadores en la pila, podemos recuperarlos en cualquier registro de propósito general mediante la instrucción `pop`, por ejemplo ejecutando la sentencia `pop ax`. En este caso `AX` contendrá una copia del registro de indicadores, siendo fácil determinar el estado de cada uno de los bits.

El valor, almacenado en `AX`, puede modificarse como cualquier otro, aunque en este caso lo habitual es poner a 1 o a 0 un cierto bit, usando para ello las instrucciones lógicas que conocerá más adelante en este mismo capítulo. Para restaurar el valor en el registro de indicadores, haciendo que éste represente el estado que nos interese, primero alojaríamos el valor de `AX` en la pila y, a continuación, lo recuperaríamos en el registro de indicadores mediante la instrucción `popf`.

Veamos en la práctica cómo usar las instrucciones pushf y popf, introduciendo el código siguiente en un archivo de texto y ensamblándolo a continuación.

```

segment Pila stack
    resb 64
InicioPila:

; Segmento de código
segment Código
..start:

; Inicializamos los registros
; relacionados con la pila
mov ax, Pila
mov ss, ax
mov sp, InicioPila

pushf ; guardamos registro de indicadores
pop dx ; recuperando en AX
or ax, 1 ; activamos el bit 0
push ax ; guardamos AX en la pila
popf ; y devolvemos al registro de
; indicadores

; salimos al sistema
mov ah, 4ch
Int 21h

```

Recuperamos los indicadores en el registro AX, según los pasos antes descritos, modificando el bit 0, indicador de acarreo, mediante la instrucción lógica or. Concretamente ponemos dicho bit a 1, activando el señalizador de acarreo. Esto provocará que al recuperar los bits en el registro de indicadores, mediante popf, el indicador de acarreo esté activo, como si hubiésemos provocado un acarreo mediante una suma. Puede verlo ejecutando el programa paso a paso desde DEBUG y, de paso, observar el valor que toma AX. Si lo convierte a binario, y usando como referencia la lista de bits comentada en el punto anterior, podrá saber el estado de cada uno de los indicadores.

Un método más directo para recuperar el contenido del registro de estado, en realidad únicamente de los 8 bits de menor peso, es el que ofrecen las instruccioneslahf y sahf. La primera toma esos 8 bits de menor peso del registro de estado y los copia en el registro AH, donde pueden ser examinados y manipulados según convenga. Con la instrucción sahf, complementaria de la anterior, se transfiere el contenido de AH a los 8 bits de menor peso del registro de estado.

Esto nos permite implementar el mismo programa anterior con estas tres simples instrucciones:

```

lahf ; Obtenemos 8 bits de menor peso de flags en AH
or ah, 1 ; activamos el bit 0
sahf ; y devolvemos al registro do indicadores

```

Como se aprecia en la figura 13.2, el resultado que se obtiene es exactamente el mismo ya que el bit que nos interesa modificar está en el byte bajo del registro de estado.

```
Simbolo del sistema - debug flags.exe

AX=1874 BX=FFFF CX=FEA3 DX=0000 SP=003E BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=0009 NV UP EI PL NZ NA PO NC
1878:0009 58          POP     AX
-p

AX=3202 BX=FFFF CX=FEA3 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=000A NV UP ET PL NZ NA PO NC
1878:000A 0D0100      OR      AX,0001
-p

AX=3203 BX=FFFF CX=FEA3 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=000B NV UP EI PL NZ NA PE NC
1878:000D 50          PUSH    AX
-p

AX=3203 BX=FFFF CX=FEA3 DX=0000 SP=003E BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=000E NV UP EI PL NZ NA PE NC
1878:000E 9D          POPF
-p

AX=3203 BX=FFFF CX=FEA3 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=000F NV UP EI PL NZ NA PO CY
1878:000F B44C        MOV     AH,4C
-0011 0010 0000 0011
```

Figura 13.1. Al recuperar el registro de indicadores de la pila se activa el señalizador de acarreo.

```
Simbolo del sistema - debug flags2.exe

D:\Ejemplos\13>nasm -f obj flags2.asm
D:\Ejemplos\13>alink flags2
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file flags2.obj
matched Externs
matched ComDefs

D:\Ejemplos\13>debug flags2.exe
-p

AX=0000 BX=FFFF CX=FEA1 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=0000 NV UP EI PL NZ NA PO NC
1878:0000 B87418      MOV     AX,18/4
-p

AX=1874 BX=FFFF CX=FEA1 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=0003 NV UP EI PL NZ NA PO NC
1878:0003 8ED0        MOV     SS,AX
-p

AX=1874 BX=FFFF CX=FEA1 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=0008 NV UP EI PL NZ NA PO NC
1878:0008 9F          LAHF
-p

AX=0274 BX=FFFF CX=FEA1 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=0009 NV UP EI PL NZ NA PO NC
1878:0009 80cc01      OR      AH,01
-p

AX=0374 BX=FFFF CX=FEA1 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=000C NV UP EI PL NZ NA PE NC
1878:000C 9E          SAHF
-p

AX=0374 BX=FFFF CX=FEA1 DX=0000 SP=0040 BP=0000 SI=0000 DI=0000
DS=1864 ES=1864 SS=1874 CS=1878 IP=000D NV UP EI PL NZ NA PO CY
1878:000D B44C        MOV     AH,4C
-
```

Figura 13.2. Modificación del registro de estado con lahf/sahf.

Comparación de valores

Los procesadores x86 disponen de una sola instrucción para comparar dos valores, instrucción que, realmente, lo que hace es una operación aritmética que, sin producir un resultado, afecta a los bits del registro de indicadores como si se hubiese ejecutado realmente. El único fin de dicha instrucción, por tanto, es activar o desactivar ciertos indicadores, simplemente.

La instrucción en cuestión es cmp. Precisa, como las instrucciones add o sub, dos operandos: un destino y un origen. Lo que hace es restar el origen del destino, como la instrucción sub, pero sin llegar a almacenar el resultado de la operación. Es decir, la única diferencia entre sub ax, bx y cmp ax, bx es que la primera almacena el resultado en AX, mientras que la segunda no. Los indicadores afectados, sin embargo, serán exactamente los mismos.

¿Qué utilidad tiene la activación de unos ciertos indicadores? La respuesta la encontrará al conocer las instrucciones de bifurcación, un gran conjunto de órdenes que se encargan de comprobar uno o más de esos indicadores y, según su estado, desviar la ejecución a otro punto. Vamos a ir conociendo algunas de ellas en los puntos siguientes, junto con pequeños ejemplos de uso.

Igualdad y desigualdad

La mayor parte de los condicionales que se codifican en los programas persiguen comprobar si dos valores concretos son o no iguales. Cuando se comparan dos valores mediante cmp, restando el segundo del primero, si el resultado es la activación del indicador ZF esto querrá decir que el resultado ha sido cero. ¿Cuándo se obtiene el valor cero al restar un valor de otro? Efectivamente, cuando ambos son el mismo valor. La activación del bit ZF, por tanto, indicaría esa igualdad, mientras que su no activación indicaría desigualdad.

Para saltar a un cierto punto cuando el indicador de cero esté activado tenemos dos instrucciones: j z y je. Son equivalentes y, como las demás instrucciones de salto, necesitan como parámetro la dirección a la se quiere desviar la ejecución. Ésta puede ser una etiqueta previamente definida, una dirección almacenada en un cierto campo o en uno de los registros, según interese.

En caso de que se desee desviar la ejecución cuando el indicador esté a cero, en lugar de a uno, utilizaríamos las instrucciones j nz y j ne que, al igual que las anteriores, son equivalentes, pudiendo utilizarse de manera indistinta.

Para desviar la ejecución a un cierto punto de manera incondicional, sin examinar ningún indicador, puede usar la instrucción j mp que, al igual que las ya citadas, necesita un parámetro que indique la dirección a partir de la cual se procederá a ejecutar.

Conociendo las instrucciones cmp, j z y jmp, veamos cómo podemos usarla para, por ejemplo, examinar el color de un cierto carácter de la pantalla y, dependiendo de ello, invertirlo. El código del programa sería el siguiente:

```
segment Pila stack
    resb 64
TnicioPila:
    ; Segmento de código
    segment Código
..start:
    ; preparamos DS y BX
    ; para acceder al atributo
    ; de la fila 12 columna 40
    ; de la pantalla de texto
    mov ax, 0b800h
    mov ds, ax
    mov bx, 160*12+40*2+1

    ; comprobamos si el atributo
    ; es blanco sobre blanco
    cmp byte [bx], 07h
    ; de ser así saltamos
    jz FondoNegro

    ; en caso contrario establecemos
    ; el atributo por defecto
    mov byte [bx], 07h

    ; y saltamos al punto de salida
    jmp Salir

FondoNegro:
    ; invertimos los colores
    mov byte [bx], 70h

Salir:
    ; salimos al sistema
    mov ah, 4ch
    int 21h
```

Comenzamos preparando los registros DS y BX a fin de poder, mediante ellos, acceder a la posición de pantalla que nos interesa. Acto seguido, usamos la instrucción cmp para comprobar si la celdilla contiene el atributo correspondiente al blanco sobre negro. De ser así, saltamos a la etiqueta FondoNegro y ejecutamos la instrucción mov que invierte dicho atributo, devolviendo a continuación el control al sistema. De no cumplirse la condición, la celdilla contendrá un atributo distinto, la ejecución seguirá su curso, ejecutándose otra instrucción mov que le devolverá el atributo por defecto.

Observe que a continuación hemos introducido una instrucción jmp que desvía la ejecución hasta las instrucciones de salida. De no hacerlo así, se ejecutaría nuevamente la instrucción mov que establece el atributo invertido, por lo que éste sería el que quedaría siempre en pantalla.

La familia de instrucciones de salto, de las cuales forman parte `jz`, `jnz`, `je`, `jne` y `jmp`, representa uno de los mecanismos que tenemos los programadores en ensamblador para modificar el contenido del registro IP, asignándole la dirección a partir de la cual se desee ejecutar.

Tras ensamblar el programa, y borrar la pantalla con la orden `cls`, ejecútelo varias veces. Observe el carácter que aparece como un recuadro blanco y después desaparece, alternando el estado a cada ejecución.

Esto será así siempre que ejecute el programa en DOS, una ventana DOS de algunos sistemas o un emulador de dicho sistema operativo, pero en la ventana Símbolo de sistema de las últimas versiones de Windows, como es el caso de Windows Vista, el efecto posiblemente no será el deseado ya que la ventana no emula un sistema DOS real y la memoria de pantalla no se encuentra donde debería.

Si intenta seguir la ejecución del programa paso a paso con DEBUG, como ya ha hecho en capítulos previos, se encontrará con que ciertas instrucciones no son interpretadas correctamente. Así, el salto condicional `jz` FondoNegro aparece como múltiples instrucciones que poco tienen que ver.

Pruebe a utilizar el depurador GRDB que se comentó en un capítulo anterior, lo encontrará en el CD-ROM que acompaña a este libro. Su funcionamiento es muy similar al de DEBUG, con la diferencia de que reconoce registros de 32 bits e instrucciones más avanzadas.

En la figura 13.3 puede observar una imagen de GRDB tras desensamblar el código del programa.

```
Get Real Debugger Version 9.4 Copyright (c) 1997-2007 David Lindauer (LADSoft)
GRDB comes with ABSOLUTELY NO WARRANTY, for details type '?g'
This is free software, and you are welcome to redistribute it
under certain conditions: type '?gr' for details

History enabled
eax:00000000 ebx:0000FFFF ecx:0000FESC edx:00000000 esi:00000000 edi:00000000
ebp:00000000 esp:00000040 eip:00000000 flag:00000202 MU UF EI PL NZ NC PD MC
ds:1130 es:1130 fs:1130 gs:1130 ss:1140 cs:1151
1151:0000        mov     ax,B800
Size: 0000005C
...
1151:0000 8D 00 00    mov     ax,B800
1151:0003 8B 00      mov     ds,ax
1151:0005 8B 07      mov     bx,B701
1151:0008 39 07      cmp     byte [bx],07
1151:000B 7E 00      jz    0015
1151:000F 7E 07      jz    B701,07
1151:0012 03 00      add    ah,00
1151:0015 07 70      jnp   B701,70
1151:0018 4C          mov    ah,4C
1151:001A C9 21      int    21
1151:001C 8D 00      add    bx,sil,al
1151:001E 8D 00      add    bx,sil,al
```

Figura 13.3. La herramienta GRDB funciona de manera similar a DEBUG.

Menor y mayor que

Las comprobaciones no siempre son de igualdad y, en ocasiones, puede interesar saber si un cierto valor es mayor o menor que otro dado. ¿Cómo se puede saber si un número es menor que otro efectuando una resta que, al fin y al cabo, es lo que hace la instrucción cmp? La respuesta está en los indicadores de signo, acarreo y desbordamiento, dependiendo de que se opere con números positivos o negativos.

Aunque existen instrucciones de salto asociadas a cada uno de los bits citados: signo, acarreo y desbordamiento, para saber si un número es mayor o menor que otro tendríamos que comprobar, en ocasiones, más de un indicador, lo cual nos llevaría a hacer múltiples saltos encadenados. Para evitarlo, usaremos instrucciones de salto capaces de comprobar varios bits del registro de indicadores.

En caso de que operemos con números sin signo, las instrucciones que nos interesan son ja y jb. La primera de ellas salta en caso de que el primer operando sea mayor que el segundo, mientras que jb lo hace en el caso contrario, cuando el primer operando es menor que el segundo. Observe que estas instrucciones no contemplan el caso de igualdad, por lo que ninguna de ellas saltaría si los dos operandos fuesen iguales. Podemos usar las instrucciones jae y jbe en caso de que deseemos comprobar si el primer operando es mayor o igual que el segundo o menor o igual que el segundo, respectivamente.

¿Y si los operandos no son números sin signo? Entonces tendremos que usar las instrucciones jg, jl, jge y jle, equivalentes a ja, jb, jae y jbe, respectivamente, pero capaces de operar con números con signo.

Todas las instrucciones anteriores saltan en caso de que la condición se cumpla. Existe una instrucción complementaria para cada una de ellas, con el formato jnX, que salta si la condición no se cumple. Por ejemplo, jna salta si el primer operando no es mayor que el segundo, por lo que sería equivalente a jbe, que salta si es menor o igual.

Vamos a servirnos de otro ejemplo breve para ver en la práctica cómo usar algunas de las instrucciones que acabamos de conocer. Imagine que tiene que comprobar si un cierto carácter que hay en la pantalla es una letra mayúscula y, en caso afirmativo, convertirla en minúscula. Las letras de la A a la Z tienen códigos consecutivos, como las minúsculas, y la diferencia entre el código de cualquier minúscula y su mayúscula correspondiente es 32. Por ejemplo, la A mayúscula tiene el código 65, mientras que la minúscula tiene el código 97. Sabiendo esto, el código de nuestro programa sería el siguiente:

```
segment Pila stack
    resb 64
InicioPila:
; Segmento de código
segment Código
```

```

..start:
; preparamos DS y BX
; para acceder al carácter
; de la fila 12 columna 2
; de la pantalla de texto
mov ax, 0b800h
i<ov ds, ax
mov bx, 160*12+1*2

; Recuperamos el carácter en AL
mov al, [bx]

cmp al, 'A' ; Comparamos con la A
jb Salir ; si es inferior saltamos
cmp al, 'Z' ; Comparamos con la Z
ja Salir ; si es superior saltamos

; Convertimos a minúscula
add al, 32 ; sumando 32
mov [bx], al ; y escribiendo en pantalla

Salir:
; en cualquier caso modificamos el atributo
inc bx
; para resaltar el carácter tanto si se ha
; cambiado como si no
mov byte [bx], 0fh

; salimos al sistema
mov ah, 4ch
int 21h

```

Tras determinar la posición de memoria del carácter que vamos a convertir, el primer paso es comprobar que, en efecto, se trata de una letra mayúscula, ya que de lo contrario al sumarle 32 podríamos convertir cualquier otro carácter en otro sin significado. Dicha comprobación está compuesta de dos pasos, como puede verse. Comprobamos si el carácter es menor que la A mayúscula o mayor que la Z mayúscula. En cualquiera de esos casos se salta a la etiqueta Salir, evitando así la ejecución de las dos instrucciones siguientes que son las que suman 32 al código y lo devuelven a la pantalla.

En la figura 13.4 puede ver un ejemplo de ejecución de este programa, se ha ejecutado varias veces tras obtener un directorio, y se indican dos de los caracteres que el programa ha cambiado. Lógicamente, para que el programa pueda efectuar la conversión la pantalla deberá tener algún contenido y, en cualquier caso, tendrá que probar varias veces.

Instrucciones de manipulación de bits

La familia de procesadores x86 carece de instrucciones específicas para la manipulación individual de bits, exceptuando las de rotación que conocerá de inmediato. Si queremos activar o desactivar, poner a 1 o 0, un cierto bit, tendremos que hacerlo a través de operaciones lógicas y aplicando máscaras que modifiquen sólo el bit que nos interese.

Letras
convertidas

```
C:\>
C:\>
C:\>cd \nasm\in
C:\N13>dir /w
Directory of C:\N13\.
[.]
BITS386.ASM     BITS386.EXE     BITS386.OBJ     AND.EXE      AND.OBJ
BITSBCD.OBJ     FLAGS.ASM      FLAGS.EXE      BITSBCD.ASM  BITSBCD.EXE
FLAGS2.EXE       FLAGS2.OBJ     GRUB.DPT      FLAGS.OBJ   FLAGS2.ASM
FLAGS2.OBJ       MAYMIN.ASM    MAYMIN.EXE    GRUBLD.DAT  GRUBLD.DAT
MAYMIN.ASM      8,592 Bytes.
2 File(s)        110,540,880 Bytes free.
C:\N13>namain
C:\>
C:\>
C:\>namain
C:\>
```

Figura 13.4. Al ejecutar varias veces el programa se convierten algunos caracteres a minúscula.

Al programar en ensamblador, manipulando registros y valores del área de datos del sistema o puertos de entrada / salida, un solo bit puede ser muy significativo. Por ello su manipulación, y comprobación individual, resulta de tanto interés.

Activación de bits individuales

La activación de un cierto bit es una operación que hemos llevado a cabo en uno de los ejemplos previos, concretamente al poner a 1 el indicador de acarreo. La instrucción a usar es or, siendo dos los parámetros necesarios: el registro o dirección de memoria que contiene el dato a manipular, en primer lugar, y una máscara que determinará los bits a poner a 1, en segundo. El tamaño de la máscara deberá ser el mismo que el del operando que va a manipularse. Suponiendo que el dato estuviese en el registro AL, la máscara se compondría de 8 bits. De éstos, sólo estarían a 1 aquellos que ocupasen la posición del bit que deseásemos poner a 1 en el destino. La máscara 01001000, por ejemplo, pondría a 1 los bits 3 y 6 del destino sin afectar a los demás.

La operación lógica or produce un o si los dos operandos son o, y un i en todos los demás casos. Los bits que dejemos a o en la máscara, por tanto, quedarán como estuviesen en el dato original.

Mediante esta operación lógica podríamos, por ejemplo, tomar el byte de atributo de un carácter que hay en pantalla y, sin cambiar el color, hacerlo más brillante, simplemente poniendo a 1 el cuarto bit.

Desactivación de bits individuales

Es una operación análoga a la de activación de los bits, pero usando la instrucción and, en lugar de or, e invirtiendo la máscara, de tal forma que estarían a 0 los bits cuya posición en el dato deseemos desactivar, dejando todos los demás bits a 1. Si antes usábamos la máscara 01001000 para poner a 1 los bits 3 y 6, la máscara 10110111, utilizada con la instrucción and, pondría a 0 esos mismos bits.

Además de para desactivar ciertos bits, esta instrucción puede ser útil también a efectos de comprobación. Suponga, por ejemplo, que quiere saber si el tercer bit de un valor que tenemos en AL está o no a 1. Ejecutando la instrucción and al, 00000100, pondríamos a cero todos los bits a excepción de ese. El resultado sería que en AL quedaría el valor 0, si el bit estaba a cero, o el valor 4, si estaba a 1. En el primer caso, si quedase un 0 en AL, el indicador Z del registro de indicadores se activaría.

Veamos, con el siguiente programa, cómo usar la instrucción and para comprobar el estado de un bit y la instrucción or para activar ese bit. El cuerpo del programa se ejecutará dos veces si, inicialmente, el campo Dato tiene su tercer bit a 0. En caso contrario, puede comprobarlo modificando el programa, se indicará que el bit ya está a 1 y, por tanto, no se ejecutarán las instrucciones que hay tras la etiqueta EstaACero.

```

segment Datos

; Dato cuyo tercer bit vamos a comprobar
Dato db 10100010b
; Mensajes para notificar el estado
MsgActivo db 'El bit se encuentra a 1', 13, 10, '$'
MsgNoActivo db 'El bit se encuentra a 0', 13, 10, '$'

segment Pila stack
resb 64

InicioPila:

; Segmento de código
segment Código
.start:

; Preparamos DS para acceder
; al segmento de datos
mov ax, Datos
mov ds, ax

; recuperamos el dato en AL
mov al, [Dato]

Comprueba:
; y ponemos todos sus bits a 0
; menos el tercero
and al, 00000100b

; Si el resultado es 0 significa
; que el tercer bit estaba a 0
jz EstaACero

```

```

; en caso contrario mostrar
; que estaba a 1
mov dx, MsgActivo
mov ah, 9 ; imprimir el mensaje
int 21h

jmp Salir ; terminar

EstaACero: ; el bit está a 0
    mov dx, MsgNoActivo
    mov ah, 9 ; lo indicamos
    int 21h

    ; lo ponemos a 1
    or al, 00000100b
    ; y volvemos a comprobar
    jmp Comprueba

Salir:
    ; salimos al sistema
    mov ah, 4ch
    int 21h

```

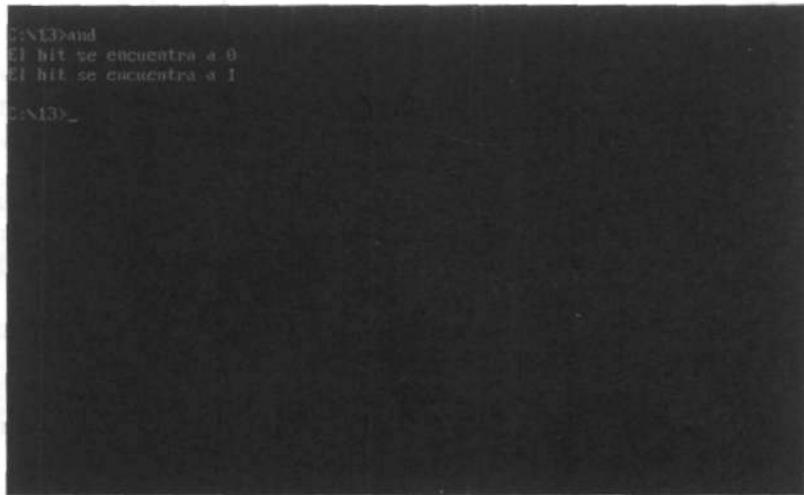


Figura 13.5. El programa comprueba el estado del bit y lo modifica.

Otras operaciones lógicas

Además de las instrucciones or y and, existen dos más capaces de efectuar operaciones lógicas sobre los bits de un dato. Una de ellas es xor, que actúa sobre los bits del operando ejecutando una operación "o" exclusiva. En el resultado estarán a 0 todos aquellos bits cuyo valor coincidan entre el dato original y la máscara, ya se encuentren a 0 o a 1, quedando a 1 los que difieran entre máscara y dato de origen. Si se ejecuta una operación xor de un registro sobre sí mismo, por ejemplo xor ax, ax, el resultado

es que el registro queda a cero, ya que todos los bits de destino y origen coinciden. Es un modo eficiente de poner a cero el contenido de un registro, más rápido que el típico mov ax, 0.

La otra operación lógica disponible es la negación de todos los bits, efectuada por la instrucción not. Ésta toma un solo parámetro: el operando sobre el que va a aplicarse la operación. El resultado es la puesta a 0 de los bits que están a 1 y viceversa.

Nota

Si utiliza dos veces la instrucción not sobre un mismo operando, éste quedará con su valor original. Una doble negación, desde el punto de vista lógico, no tiene efecto sobre los bits a los que se aplica.

Comprobación de bits individuales

Ya hemos visto cómo mediante la instrucción and es posible comprobar el estado de ciertos bits de un dato, aunque a costa de perderlo puesto que dicha instrucción efectúa una operación a partir de la cual se obtiene un resultado, igual que una suma o una resta. La alternativa es el uso de la instrucción test, equivalente a and con la única diferencia de que no almacena el resultado, de igual forma que cmp es equivalente a sub pero sin modificar el destino.

Puede modificar el programa de ejemplo del punto anterior, tras haberlo ejecutado paso a paso, para apreciar cómo la instrucción and altera el valor de AL mientras que test no, a pesar de lo cual el registro de indicadores refleja el mismo estado, como si la operación se hubiese producido.

Como se indicaba antes, los procesadores x86 no disponen de instrucciones específicas para la manipulación y comprobación de bits individuales.

En realidad esto es una verdad a medias, cierta para los 8086/8088/80186/80286 pero no a partir del 80386. Éste, y todos sus sucesores, cuentan con las instrucciones bt, bts, btr y btc.

Las cuatro precisan dos parámetros: un operando sobre el que actuar y un valor que indique el bit a comprobar, activar, desactivar o invertir, respectivamente.

A diferencia de las anteriores instrucciones lógicas, éstas propias de los 80386 sólo pueden utilizarse sobre operandos de 16 y 32 bits, por lo que no podríamos actuar, por ejemplo, sobre el registro AL, pero sí sobre AX. La instrucción bt ax, 2, por ejemplo, llevaría una copia del tercer bit del registro AX al bit de acarreo del registro de indicadores. Utilizando las instrucciones je y jnc desviaríamos la ejecución en caso de que el bit estuviese o no a 1, respectivamente.

Tomando como base el último de los ejemplos que hemos escrito, podríamos modificarlo dejándolo como se muestra a continuación. Hemos sustituido la instrucción and por bt, lógicamente adaptando también la sintaxis, y la instrucción or por la instrucción bts, para poner el bit a 1.

Si lo ejecuta, siempre que sea sobre un equipo con procesador 80386 o posterior, obtendrá exactamente el mismo resultado.

```
segment Datos

; Dato cuyo tercer bit vamos a comprobar
Dato db 10100010b
; Mensajes para notificar el estado
MsgActivo db 'El bit se encuentra a 1', 13, 10, '5'
MsgNoActivo db 'El bit se encuentra a 0',13,10,'$'

segment Pila stack
resb 64
InicioPila:

; Segmento de código
segment Código
.start:

; Preparamos DS para acceder
; al segmento de datos
mov ax, Datos
mov ds, ax

; recuperamos el dato en AL
mov al, [Dato]

Comprueba:
; comprobamos el tercer bit
bt ax, 2

; Si no se ha activado el acarreo
jnc EstaACero

; en caso contrario mostrar
; que estaba a 1
mov dx, MsgActivo
mov ah, 9 ; imprimir el mensaje
int 21h

jmp Salir ; terminar

EstaACero: ; el bit está a 0
mov dx, MsgNoActivo
mov ah, 9 ; lo indicamos
int 21h

; lo ponemos a 1
bts ax, 2
; y volvemos a comprobar
jmp Comprueba

Salir:
; salimos al sistema
mov ah, 4ch
int 21h
```

Rotación y desplazamiento de bits

Los bits de un número entero pueden rotarse de forma cíclica hacia la izquierda o la derecha, haciendo que el bit que sale por un extremo entre por el opuesto, o bien utilizando el indicador de acarreo como bit auxiliar, con lo que se consigue una rotación a 9 ó 17 bits, según el tamaño del operando.

Las instrucciones disponibles son las siguientes:

- **rcl:** Rota los bits hacia la izquierda a través del indicador de acarreo.
- **rcr:** Rota los bits hacia la derecha a través del indicador de acarreo.
- **rol:** Rota los bits hacia la izquierda, sin participación del indicador de acarreo.
- **ror:** Rota los bits hacia la derecha, sin participación del indicador de acarreo.

Las cuatro instrucciones necesitan dos parámetros: el operando cuyos bits se van a rotar, en primer lugar, y un entero indicado el número de posiciones a rotar. Este entero puede ser un valor inmediato si sólo se va a rotar un solo bit, en caso contrario debe introducirse el número deseado en el registro CL, utilizando este registro como segundo parámetro. En la figura 13.6 puede observar representadas gráficamente las cuatro operaciones.

En los procesadores 80286 y posteriores, el segundo parámetro de todas las instrucciones de rotación y desplazamiento puede ser un valor inmediato incluso cuando no es 1, mientras que en los micros previos era obligatorio usar CL en esos casos.

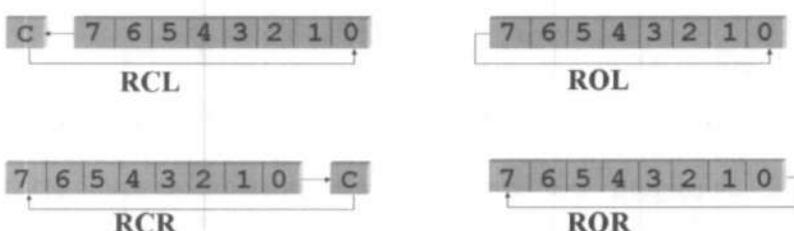


Figura 13.6. Operaciones de rotación de bits.

A las de rotación hay que añadir las instrucciones de desplazamiento, que se caracterizan porque el bit que sale por un extremo no entra por el opuesto, sino que se va rellenando con ceros. Estas instrucciones son shl y shr, siendo su sintaxis similar a la de las anteriores.

Mediante las instrucciones de desplazamiento podríamos, por ejemplo, componer y descomponer números BCD empaquetados. Suponga que tiene en AX el número BCD

desempaquetado 15, almacenándose, por tanto, el 1 en AH y el 5 en AL. Quiere empaquetarlo para que los dos dígitos queden en AL y, después de efectuar una operación, volver a desempaquetarlos.

Eso es, precisamente, lo que el código mostrado a continuación:

```
segment Pila stack
    resb 64

; Segmento de código
segment Código
..start:
    ; introducimos el número
    ; desempaquetado en AH y AL
    mov ah, 1
    mov al, 5

    ; desplazamos el contenido
    ; de AH cuatro bits a la izquierda
    shi ah, 4
    ; y lo unimos con AL
    or al, ah

    ; Hacemos una suma
    add al, 21h

    ; llevamos AL a AH
    mov ah, al
    ; para quedarnos con los
    ; cuatro bits superiores
    shr ah, 4
    ; que eliminamos de AL
    and al, 00001111b

Salir:
    ; salimos al sistema
    mov ah, 4ch
    int 21h
```

Tenemos en el registro AL un valor que ocupa el *nibble* inferior, y queremos añadir en el *nibble* superior el valor que hay en AH. Para ello desplazamos el contenido de AH cuatro bits a la izquierda, pasando el valor del *nibble* inferior al superior, y a continuación lo unimos con el contenido de AL mediante una operación lógica or. Se asume que el *nibble* superior de AL está a cero. De esta forma hemos convertido un BCD desempaquetado en un BCD empacado.

Tras efectuar la operación que interese, en este caso una simple suma, procedemos a efectuar el proceso inverso.

En este caso los dos dígitos BCD se encuentran en AL. Copiamos el valor en AH y, a continuación, lo desplazamos cuatro posiciones a la derecha, de tal forma que el *nibble* superior se convierte en el inferior y éste se pierde.

Para eliminar el *nibble* superior de AL, una vez que ya lo tenemos en AH, basta una operación lógica and con los cuatro bits a cero.

Ejecute el programa paso a paso, mejor con GRDB que con DEBUG ya que éste no reconoce algunas de las instrucciones que hemos usado, y observe el contenido de AX a medida que empaqueta y desempaque el número.

Resumen

Todas las instrucciones que hemos conocido en este capítulo nos permiten crear programas mucho más complejos que los de capítulos anteriores, al ser posible la ejecución condicional de porciones de código dependiendo de la evaluación de expresiones. También ha conocido de manera más detallada todos los bits del registro de indicadores, la instrucción de salto incondicional jmp y un conjunto de instrucciones para manipulación, comprobación, rotación y desplazamiento de bits.

Ahora que contamos con lo que podrían considerarse las estructuras condicionales del lenguaje ensamblador x86, el paso siguiente será conocer las estructuras de repetición. A ellas dedicaremos el siguiente capítulo y, como podrá ver, nos facilitarán el desarrollo de programas bastante más útiles y demostrativos, al poder repetir una cierta operación sobre multitud de datos.

14

Buckles

Son muchas las ocasiones en las que un programa necesita efectuar una determinada operación no sobre un dato individual, sino sobre listas de datos compuestas de cientos o miles de elementos. Lógicamente, no se codificará un conjunto de instrucciones para cada uno de esos elementos, algo prácticamente imposible, sino que se utilizará una estructura de repetición clásica: el bucle.

Un bucle es un conjunto de sentencias, en este caso una o más instrucciones ensamblador, que se ejecutan múltiples veces, por regla general dependiendo de un contador o alguna otra condición. Con un bucle, por ejemplo, podríamos procesar todos los caracteres contenidos en un archivo, sin saber de antemano cuántos son.

En este capítulo vamos a conocer algunas posibilidades a la hora de codificar bucles, tanto empleando instrucciones que ya sabemos usar como sirviéndonos de otras nuevas que nos facilitarán aún más el trabajo.

Bucles con saltos condicionales

Al final del capítulo previo escribíamos un programa, cuyo fin era indicar el estado de un cierto bit en un dato, que ejecutaba dos veces parte de las instrucciones. Para ello contábamos con un salto condicional, que volvía a un punto anterior del programa en caso de que se cumpliese una cierta situación.

Es posible usar esta técnica para repetir no dos veces, sino todas las que sean necesarias un determinado conjunto de sentencias.

Suponga que quiere mostrar en pantalla una tabla de caracteres, asignando el valor 0 a la celdilla que corresponde a la primera línea y columna, el valor 1 al siguiente y así sucesivamente hasta completar los 255 caracteres de la tabla ASCII extendida propia de DOS. Necesitaríamos que un registro, pongamos por caso AL, contuviese el código del carácter a mostrar, código que se incrementaría a cada ciclo del bucle.

La repetición del código se efectuaría mediante un salto condicional, por ejemplo mientras no se activase el bit ZF del registro de indicadores.

Nda -i^ ^gB^gg^IU^ ^fIj Hg

Si incrementa el valor de AL y el registro contiene en ese momento el valor 255, se volverá al valor 0 y se activará el indicador de cero. Comprobando dicho indicador, por tanto, sabremos que hemos completado todos los valores que puede tomar el registro.

El código necesario para mostrar esa tabla de caracteres sería el siguiente:

```
segment Pila stack

; Segmento de código
segment Código
..start:
; DS apuntará al segmento
; de pantalla
mov ax, 0b800h
mov ds, ax

; ponemos a 0 BX para
; acceder a la primera posición
xor bx, bx

; ponemos a 0 AL, para
; mostrar el primer carácter
xor al, al

Bucle:
; mostramos el carácter en pantalla
mov [bx], al

; avanzamos al atributo
inc bx
; y lo establecemos
mov byte [bx], 70h

; avanzamos a la posición siguiente
inc bx
; avanzamos al siguiente carácter
inc al

; si AL no es 0 saltamos
jnz Bucle
```

Salir:

```
; salimos al sistema  
mov ah, 4ch  
int 21h
```

Observe que no se usa en ningún momento la instrucción cmp. Una sentencia cmp al, 0 justo antes del j nz Bucle sería totalmente redundante, ya que la propia instrucción inc al se encarga de activar los indicadores apropiados, que es lo único que conseguimos con la instrucción cmp. En la figura 14.1 puede apreciarse, en la parte superior, el resultado que genera el programa.

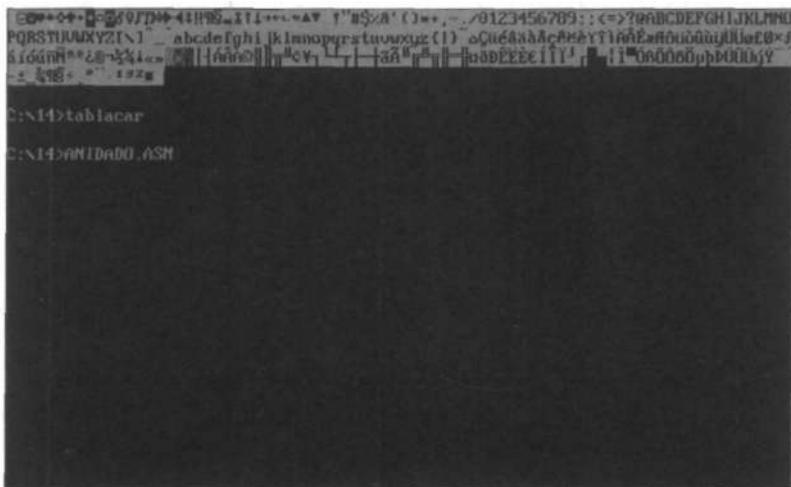


Figura 14.1. La tabla de caracteres en la parte superior de la pantalla.

En este ejemplo, el registro AL hace las veces de contador, recorriendo los valores van desde el 0 hasta el 255. Igualmente podríamos usar un registro de 16 bits, consiguiendo así la ejecución de bucles más largos. De esta forma podría, por ejemplo, recorrer todo el contenido de la pantalla cambiando todas las letras minúsculas por mayúsculas. A continuación puede ver el código de un programa que hace precisamente eso.

```
segment Pila stack  
resb 64  
  
; Segmento de código  
segment Código  
.start:  
    ; DS apuntará al segmento  
    ; de pantalla  
    mov ax, 0b800h  
    mov ds, ax  
  
    ; ponemos a 0 BX para  
    ; acceder a la primera posición  
    xor bx, bx
```

```
; ponemos en el registro CX
; el número de caracteres
; que debemos inspeccionar
mov ex, 80*25
```

Bucle:

```
; recuperamos un carácter
mov al, [bx]
; comprobamos que sea una
; letra minúscula
cmp al, 'a'
jb NoCambiar
cmp al, 'z'
ja NoCambiar

; es una letra minúscula
; y la convertimos a mayúscula
sub al, 32
; devolvéndola a la pantalla
mov [bx], al
```

NoCambiar:

```
; incrementaremos BX para avanzar
; al siguiente carácter
inc bx
inc bx

; tenemos un carácter menos a
; inspeccionar, por lo que
; reducimos el valor de CX
dec ex

; si CX no es 0 saltamos
jnz Bucle
```

Salir:

```
; salimos al sistema
mov ah, 4ch
int 21h
```

La figura 14.2 muestra la ejecución del programa tras haber obtenido en pantalla el contenido del archivo minmay.asm mediante la orden type. Observe que todo el texto se ha convertido a mayúsculas.

Una vez más, se ha empleado un registro, en este caso CX, como si fuese un contador, en esta ocasión funcionando en orden inverso. Pruebe a modificar el bucle para conseguir que no sea necesario el uso del registro CX, vigilando, en su lugar, el valor de BX, que deberá ser 3998 como límite. En ese caso sí necesitará emplear la instrucción cmp.

Otra posibilidad, a la hora de implementar un bucle, consiste en usar la instrucción j cxz. Ésta salta a la dirección indicada sólo si el valor que contiene el registro CX es 0.

```
NOCAMBIAR:  
    ; INCREMENTAMOS BX PARA AVANZAR  
    ; AL SIGUIENTE CARACTER  
    INC BX  
    INC BX  
  
    ; TENEMOS UN CARACTER MENOS A  
    ; INSPECCIONAR, POR LO QUE  
    ; REDUCIMOS EL VALOR DE CX  
    DEC CX  
  
    ; SI CX NO ES 0 SALIMOS  
    JNZ BUCLE  
  
    LOOP BUCLE  
  
SALIR:  
    ; SALIMOS AL SISTEMA  
    MOV AH, 4CH  
    INT 21H  
  
C:\DAM\INMAY  
C:\DAM\
```

Figura 14.2. Todas las letras minúsculas que hay en pantalla se convierten a mayúsculas.

Instrucciones para implementar bucles

Siempre que los bucles que necesitemos codificar estén controlados por un contador, sabiéndose de antemano cuántas veces debe repetirse la ejecución de las sentencias, podemos servirnos de las instrucciones específicas con que cuenta el procesador para controlar este tipo de lógica. En realidad se trata de una sola instrucción, loop, que cuenta con dos variantes adicionales.

La instrucción loop necesita, como todas las de salto condicional e incondicional, la dirección a la que debe transferir la ejecución. Una limitación, importante, de esta instrucción es que dicha dirección es obligatoriamente relativa y de 8 bits, de tal forma que el punto de destino puede estar 128 bytes antes o 127 bytes después de la propia instrucción loop. Esto significa que para bucles compuestos de un bloque de instrucciones importante, de más de 128 bytes, no puede utilizarse loop.

Al ejecutar la instrucción loop se dan dos pasos: primero se reduce en una unidad el valor del registro CX y, a continuación, se efectúa un salto condicional basándose en el estado del bit ZF del registro de indicadores. De esta forma, se salta a la dirección indicada siempre que CX no haya llegado a ser cero. Es, por tanto, una abreviación del bucle que hemos implementado en el segundo ejemplo del punto previo, ni más ni menos. Puede comprobarlo sustituyendo las sentencias dec ex y j nz Bucle que hay tras la etiqueta NoCambiar por un simple loop Bucle. El programa, al ejecutarse, genera exactamente el mismo resultado.

Las dos variantes de loop son loope y loopne, que, aparte de si CX es o no 0, también comprueban previamente si el bit ZF del registro de indicadores está activo. De esta manera es posible preceder la instrucción loop de una comparación u operación lógica o aritmética que actúe sobre dicho indicador.

Las instrucciones loop, loope y loopne, debido a sus limitaciones y un peor rendimiento que los saltos condicionales, caen en desuso, hasta tal punto que microprocesadores posteriores al 8086 no han incorporado mejora alguna en estas instrucciones.

Casos concretos

Indistintamente de que usemos las instrucciones de salto condicional o bien la instrucción loop que acaba de comentarse, a la hora de codificar bucles podemos encontrarnos habitualmente con algunos casos típicos. Por una parte tenemos los bucles simples controlados mediante un contador, a los que puede aplicarse la instrucción loop, y aquellos cuyo fin depende de algún otro tipo de condición, empleándose normalmente algún salto condicional que examine un bit del registro de indicadores. Por otra, tenemos bucles con condiciones compuestas, por ejemplo con un contador y una validación; bucles que tienen en su interior otros bucles, lo que se conoce como bucles anidados, y, por último, bucles que, básicamente, se limitan a transferir información de un lugar de la memoria a otro. En los puntos siguientes vamos a analizar esos tres casos concretos de bucle, con ejemplos sencillos pero que podrá usar en muchas otras situaciones para resolver una necesidad que surja en un momento dado.

Bucles con condición compuesta

Se trata de bucles que, por una parte, deben ejecutarse un cierto número de veces, por lo que emplean un contador, y, por otra, necesitan poner fin al proceso de repetición en caso de que se cumpla o no una condición adicional.

Suponga, por ejemplo, que necesita convertir todos los caracteres de una cadena, alojada en una cierta posición de memoria, de minúsculas a mayúsculas, estableciendo un límite de N caracteres y, al tiempo, comprobando si se encuentra un carácter que indique el fin de la cadena. Por una parte tendría que establecer la longitud máxima a convertir en un registro que actúe como contador, generalmente CX, restándole uno a cada ciclo y comprobando cuando es cero. Además, no obstante, habría que comprobar si el carácter que está procesándose es el que indica el fin de la cadena. Modifique el segundo programa usado como ejemplo en este capítulo, añadiendo las sentencias siguientes tras el doble incremento del registro BX. Si ejecuta ahora el programa, comprobará que convierte las minúsculas a mayúsculas que haya en pantalla hasta encontrar el carácter >, bastante habitual ya que forma parte del indicador de espera usado por DOS.

```
; Si el carácter era >
cmp al, '>'
; terminamos
je Salir
```

```

; NOCAMBIAR:
; INCREMENTAMOS BX PARA AVANZAR
; AL SIGUIENTE CARÁCTER
INC BX
INC BX

; SI EL CARÁCTER ERA >
cmp al, '>'
; terminamos
je Salir

loop Bucle

Salir:
; salimos al sistema
mov ah, 4ch
int 21h

C:\>ninay
C:\>

```

Figura 14.3. La conversión se detiene al encontrar el carácter >.

Como puede observar, la solución es tan simple como añadir la comparación y el salto condicional antes de la instrucción loop del bucle, o las instrucciones dec y jz si estamos usando ese sistema en lugar de loop.

Antes se apuntaba la existencia de las instrucciones loope y loopne, mediante las cuales podríamos abreviar el código evitando la disminución explícita de ex y los dos saltos condicionales. En este ejemplo concreto, puesto que queremos que el bucle siga ejecutándose mientras no se encuentre el carácter >, usaríamos la instrucción loopne.

El programa siguiente genera el mismo resultado, convirtiendo en mayúsculas las minúsculas que haya en pantalla hasta encontrar el carácter >, pero usar loopne. Las instrucciones sobrantes, respecto a la versión anterior del mismo ejemplo, aparecen ahora como comentarios, para que pueda apreciar mejor la diferencia.

```

segment Pila stack
resb 64

; Segmento de código
segment Código
..start:
; DS apuntará al segmento
; de pantalla
mov ax, 0b800h
mov ds, ax

; ponemos a 0 BX para
; acceder a la primera posición
xor bx, bx

; ponemos en el registro CX
; el número de caracteres
; que debemos inspeccionar
mov ex, 80*25

```

```

Bucle:
; recuperamos un carácter
mov al, [bx]

; comprobamos que sea una
; letra minúscula
cmp al, 'a'
jb NoCambiar
cmp al, 'z'
ja NoCambiar

; es una letra minúscula
; y la convertimos a mayúscula
sub al, 32
; devolviéndola a la pantalla
mov [bx], al

NoCambiar:
; incrementamos BX para avanzar
; al siguiente carácter
inc bx
inc bx

; Si el carácter era >
cmp al, '>'

; terminamos
;je Salir

; tenemos un carácter menos a
; inspeccionar, por lo que
; reducimos el valor de CX
;dec ex

; si CX no es 0 saltamos
;jnz Bucle

loopne Rucie
Salir:
; salimos al sistema
mov ah, 4ch
int 21h

```

Bucles anidados

El número de registros con que cuenta el procesador es limitado, lo cual en ocasiones puede plantear problemas. La instrucción `loop`, y sus derivadas, sólo sirven para crear bucles en los que el registro CX actúa como contador, no siendo aplicable a otros registros. Cabe la posibilidad, ciertamente, de usar DX, BX o incluso AX como contador, reduciendo su valor manualmente y saltando de forma condicional. En cualquier caso, sería complicado tener varios bucles, uno dentro de otro, usando distintos registros como contadores, ya que nos quedaríamos sin registros para poder efectuar otras operaciones.

Partamos, como en casos anteriores, de un supuesto práctico. Imagine que quiere dibujar un recuadro en pantalla, ocupando un determinado número de filas y columnas. Decide usar el registro DL para contener la línea y DH para la columna. Los registros AX y BX los necesitará para, mediante multiplicaciones y sumas, calcular la dirección de pantalla correspondiente a esa fila y columna. Tan sólo nos queda libre CX, pero necesitamos dos bucles: uno para recorrer las líneas y, dentro de él, otro para recorrer las columnas de cada línea.

La solución pasa por utilizar un almacenamiento temporal para algunos de los registros, por ejemplo el que actúa como contador. Además, si pretendemos usar la instrucción loop para codificar los bucles tendrá necesariamente que ser CX el registro que se guarde y recupere cuando sea necesario, ya que no puede alojar simultáneamente dos contadores, el del bucle exterior y el interior. El almacenamiento temporal más habitual, aunque existen otras opciones, es la pila. Mediante la instrucción push podemos insertar en ella cualquier registro de 16 bits, recuperándolo posteriormente mediante la instrucción pop. Recuerde que los valores de la pila se recuperan en orden inverso, de tal forma que si guarda tres registros en este orden:

```
push ax  
push bx  
push cx
```

Para recuperar los valores en los mismos registros tendríamos que ejecutar las sentencias siguientes:

```
pop cx  
pop bx  
pop ax
```

Teniendo esto en cuenta, observe el código del siguiente programa. Al ejecutarlo genera el resultado que puede verse en la figura 14.4: un recuadro que ocupa ocho filas por 20 columnas. Las tres sentencias que hay tras la etiqueta BucleLineas, así como las tres que hay justo antes de la etiqueta Salir, se ejecutarán ocho veces. Por cada una de ellas, las instrucciones que hay tras BucleColumnas se ejecutarán veinte veces, es decir, 160 veces en total. Fíjese en cómo preservamos el contador del bucle exterior, guardando CX en la pila, recuperándolo cada vez que se llega al final de la ejecución del bucle interior.

```
segment Pila stack  
    resb 64  
FinPila:  
  
; Segmento de código  
segment Código  
.start:  
  
; Preparamos la pila  
mov ax, Pila  
mov ss, ax  
mov sp, FinPila
```

```

; DS apuntará al segmento
; de pantalla
mov ax, 0b800h
mov ds, ax

mov di, 8 ; primera línea
mov ex, 8 ; número de líneas

BucleLineas:
    ; guardamos CX en la pila
    push ex

    mov dh, 25 ; primera columna
    mov ex, 20 ; número de columnas

BucleColumnas:
    ; Calculamos la posición
    mov al, 160 ; bytes por línea
    muí di ; por número de línea

    mov bx, ax ; guardamos en BX

    mov al, 2 ; bytes por columna
    muí dh ; por número de columna

    add bx, ax ; sumamos a bx

    ; ponemos un carácter en esa posición
    mov «ord [bx], 070feh

    inc dh ; incrementamos la columna

loop BucleColumnas ; y repetimos

    ; al finalizar todas las columnas
    ; de una línea

    pop ex ; recuperamos contador líneas

    inc di ; pasamos a línea siguiente

loop BucleLineas ; y repetimos

Salir:
    ; salimos al sistema
    mov ah, 4oh
    int 21h

```

Utilizando la misma técnica podría crear tres, cuatro o más bucles anidados, según las necesidades que tuviese en cada programa.

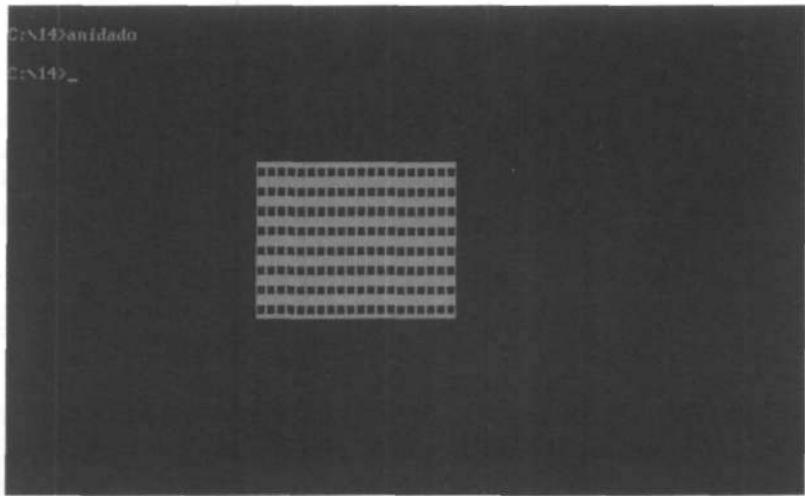


Figura 14.4. Aspecto del recuadro mostrado por el programa.

Transferencia de datos

El tercer caso típico, como se apuntaba antes, es el de los bucles cuyo núcleo se centra en la transferencia de datos de un punto a otro de la memoria. Con lo que sabemos hasta ahora, la instrucción mov y la codificación de un bucle, no tendríamos problema para hacerlo, aunque el tema puede complicarse si, por ejemplo, los datos a copiar se encuentran en un segmento distinto del que contiene el área receptora de los datos.

Imagine que quiere tomar el contenido de la pantalla y almacenarlo en un espacio que, a tal efecto, ha definido en el programa. Tendría que establecer un registro de segmento, por ejemplo DS, con la dirección del área de pantalla, y otro, ES, apuntando al segmento de datos. A continuación necesitaría dos registros más que actuarían como índices, por ejemplo BX asociado a DS y DX asociado a ES. Como en casos anteriores, CX actuaría a modo de contador. A partir de ahí podríamos ir recuperando bytes o palabras del origen y copiándolos en el destino, incrementando índices, reduciendo el contador y repitiendo hasta terminar con todo el contenido.

Además de los registros de propósito general, que hemos utilizado con más o menos frecuencia en los ejemplos previos, contamos con dos más, SI y DI, que hasta ahora no se han utilizado en la práctica. Aunque pueden servir para trabajos genéricos, como el resto, son los más apropiados cuando se necesita un índice de origen (SI, *Source Index*) y otro de destino (DI, *Destination Index*). El primero va asociado automáticamente con el registro de segmento DS, y el segundo con ES, no siendo necesario indicar de manera explícita el registro de segmento como sí sería necesario, por ejemplo, si deseásemos usar el registro BX como índice sobre un segmento ES. La ventaja de utilizar los registros SI y DI como índices es que podemos reducir la sentencia:

```
mov byte [es:di], [ds:si]
```

A un simple:

movsb

Esta instrucción copia un byte desde la dirección a la que apunta SI a la dirección que indica DI. Además, también puede incrementar o reducir automáticamente esos registros, con lo que nos ahorraremos dos operaciones más. En caso de que el indicador de dirección, que conocíó en un capítulo previo, esté a 0, movsb incrementará los registros SI y DI.

Si el indicador está a 1, el valor de dichos registros se reducirá. Recuerde que puede activar y desactivar el indicador de dirección mediante las instrucciones std y cid, respectivamente.

Nota

En caso de que desee mover palabras en lugar de bytes, utilice la instrucción movsw en sustitución de movsb.

Conociendo esta nueva instrucción, y la aplicación de los registros SI y DI, podríamos crear un programa como el siguiente, basado en la idea de transferir el contenido de la pantalla a una zona de memoria del programa. Este ejemplo hace precisamente eso, preserva el contenido de la pantalla, a continuación la llena de asteriscos en negro sobre fondo blanco y, finalmente, recupera el contenido efectuando la transferencia a la inversa, desde la variable a la memoria de pantalla.

El código, como se ve a continuación, incluye todos los comentarios necesarios para su comprensión.

```

segment Datos
; Reservamos espacio para poder
; guardar el contenido de la pantalla
Pantalla resb 25*80*2

segment Pila stack
resb 64
FinPila:

; Segmento de código
segment Código
..start:

; Preparamos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; DS:SI apunta a la pantalla
mov ax, 0b800h
mov ds, ax
xor si, si

```

```
; ES:DI apunta a nuestra variable  
mov ax, Datos  
mov es, ax  
mov cli, Pantalla  
  
; Vamos a copiar 4000 bytes  
mov ex, 4000  
cid ; incrementar automáticamente SI y DI
```

Guarda:

```
; movemos el contenido de la celdilla  
; apuntada por SI a la que indica DI  
movsb  
loop Guarda ; repetir  
  
; Llenamos la pantalla de asteriscos  
  
mov ex, 2000 ; 2000 caracteres  
xor bx, bx ; desde el principio de  
; pantalla  
mov al, '*' ; asteriscos  
mov ah, 70h ; en video inverso
```

Llena:

```
; introducimos carácter y atributo  
mov [bx], ax  
inc bx ; pasamos a la posición siguiente  
inc bx  
loop Llena ; y repetimos hasta el final  
  
; Invertimos ES y DS  
mov ax, ds  
mov bx, es  
mov ds, bx  
mov es, ax  
  
; preparamos los Índices  
xor di, di  
mov si, Pantalla  
  
; para restaurar 4000 bytes  
mov ex, 4000
```

Restaura:

```
movsb  
loop Restaura
```

Salir:

```
; salimos al sistema  
mov ah, 4ch  
int 21h
```

Si, una vez ensamblado, ejecuta el programa, verá que, aparentemente, no ocurre nada. En realidad la memoria de pantalla ha perdido su contenido, se ha llenado de asteriscos y lo ha recuperado posteriormente, pero todo ha ocurrido tan rápido que ni

siquiera llegamos a verlo en pantalla. Sin embargo, si ejecuta el programa paso a paso desde DEBUG, podrá ver que los asteriscos aparecen (véase la figura 14.5) y después desaparecen.

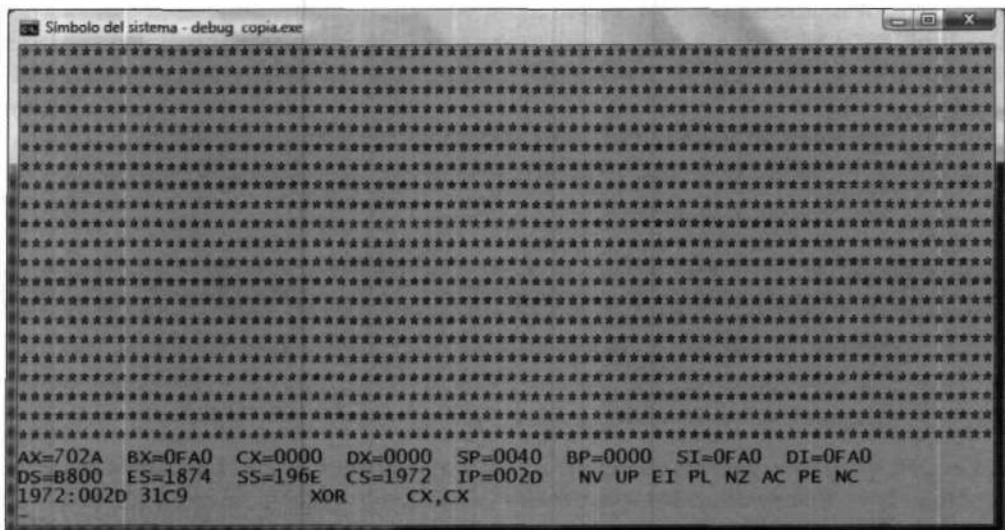


Figura 14.5. Al ejecutar la primera parte del programa, desde DEBUG, podemos ver los asteriscos.

Para ver el efecto ejecutando el programa normalmente, sin ejecutarlo paso a paso, podríamos intentar varias técnicas. Con los conocimientos que tenemos ahora mismo, una de ellas sería introducir un bucle de espera justo después de la sentencia `loop Llena`:

```
        xor ex, ex    ; Ponemos CX a cero  
Espera:  
        loop Espera  ; para repetir 65536 veces
```

En procesadores antiguos esto podría causar un leve retardo, pero con los actuales, ejecutando miles de millones de instrucciones por segundo, dicho bucle no representa nada en el tiempo. Optaremos por otro método igualmente sencillo, pero más efectivo, consistente en esperar a que se pulse una tecla tras llenar la pantalla de asteriscos. Ponga detrás de la citada sentencia las líneas siguientes:

```
; esperamos la pulsación de una tecla  
xor ah, ah  
int. 16h
```

Lo que hacemos es invocar a una de las interrupciones de la BIOS, la 16h, compuesta de múltiples servicios relacionados con el teclado. El servicio 0, que indicamos en AH, espera hasta que se pulse una tecla.

Si ejecuta ahora el programa, observando primero el contenido que hay en la pantalla, verá que ésta se llena de asteriscos y, tras pulsar una tecla, vuelve a su estado inicial.

Resumen

La posibilidad de codificar bucles, mediante saltos condicionales y el conjunto de instrucciones `loop`, nos permite crear aplicaciones mucho más interesantes, como las escritas a modo de ejemplo en este capítulo. Aunque nos hemos centrado en el aspecto visual de los ejemplos, usando las mismas técnicas podría, por ejemplo, tomar una lista de números y sumarlos mediante un bucle, o tomar un gráfico y rotar sus puntos. Conoce todas las instrucciones necesarias, es tan sólo cuestión de imaginación.

Además de la instrucción `loop`, en este capítulo también ha conocido otras que pueden resultar muy útiles, como las de inserción y extracción datos de la pila o bien la instrucción `movsb` para llevar datos de un punto a otro de la memoria mediante los registros `SI` y `DI`.

En el siguiente capítulo aprenderá a estructurar el código que, hasta ahora, siempre se ha ejecutado de manera más o menos secuencial, a pesar de los saltos que se hayan introducido. Esto nos permitirá implementar tareas más funcionales, al dividirlas en partes simples que después podemos unir.

15

Estructuración del código

Como programados estará acostumbrado a estructurar el código de sus aplicaciones en distintos módulos físicos, componentes, bibliotecas de clases, procedimientos independientes o recursos similares. La división en pequeñas porciones funcionales es, en ocasiones, la única alternativa para resolver un cierto problema, especialmente cuando éste es de cierta complejidad. La técnica conocida como *Divide y vencerás* es una de las más conocidas y, cuando se usan lenguajes de bajo nivel, también la más útil.

Al trabajar con un lenguaje ensamblador la necesidad de la estructuración es, si cabe, aún más vital, ya que cualquier tarea, por sencilla que parezca, puede requerir una gran lista de sentencias ejecutables. No hay que olvidar que estamos trabajando a un nivel muy bajo, más que cualquier lenguaje al que esté acostumbrado.

El objetivo de este capítulo es mostrarle los recursos que tiene a su disposición, trabajando en ensamblador, para estructurar su código. Alguno de estos recursos lo tendrá siempre a su disposición, al estar compuesto de instrucciones que el microprocesador puede ejecutar directamente, mientras que otros dependerán del ensamblador que esté utilizando, ya que será él quien, mediante un proceso previo, tenga que interpretar.

Procedimientos

Los lenguajes de alto nivel, al menos la mayoría de ellos, permiten codificar bloques de sentencias a los que se asigna un nombre, de tal forma que para ejecutar dichas sentencias, desde cualquier punto de la aplicación, basta con introducir ese identificador

y, en caso necesario, facilitar los parámetros apropiados. A esos bloques de sentencias, dependiendo del lenguaje, se les conoce como procedimientos, funciones o métodos, según los casos.

En ensamblador, como ya sabe, se opera con direcciones, a pesar de que éstas pueden estar representadas mediante etiquetas que hacen más fácil nuestro trabajo. La transferencia *mágica* de parámetros, por la que unos valores facilitados tras un identificador aparecen como argumentos en el procedimiento invocado, es algo que no existe en ensamblador. Sí tenemos a nuestra disposición, no obstante, una instrucción capaz de saltar a una dirección dada guardando el actual valor del registro IP, de tal forma que ésta puede recuperarse posteriormente para volver y continuar la ejecución por el punto en el que se interrumpió.

Nota

En los lenguajes de alto nivel, la transferencia de los parámetros desde un punto dado hacia un procedimiento queda a cargo del compilador. Éste puede usar diversos recursos, desde determinados registros del microprocesador hasta la propia pila del programa.

Llamada a un procedimiento

La instrucción que nos permite invocar a un procedimiento es `call`. Ésta toma como parámetro un valor inmediato, la dirección a la que debe transferir el control, o bien indirecto a través de un registro o un campo que contendría la dirección a la que tiene que apuntar IP a partir de ese momento.

Aparentemente `call` es similar a la instrucción `jmp`, ya que transfiere la ejecución a una dirección de forma incondicional. Pero existen diferencias notables, representadas esquemáticamente en la figura 15.1.

Para comenzar, antes de que se produzca el salto a la dirección indicada, `call` guarda en la pila el valor actual del registro IP, en caso de que el salto sea a una dirección dentro del mismo segmento que indica CS, o bien los valores de CS e IP, si el salto es a una dirección de otro segmento. El primer tipo de salto se denomina *near* o cercano, mientras que al segundo se le conoce como salto *far* o lejano. En el primero se necesitan 16 bits, sólo la dirección relativa al segmento que indica CS, y en el segundo los 32 bits precisos para dar valor a CS e IP tras guardarlos en la pila. La figura 15.1 representa un salto cercano, en el que se guarda en la pila únicamente el contenido de IP.

Una vez que se ha guardado el valor de IP, o de CS e IP si es llamada lejana, se ejecutan las instrucciones que haya en la dirección indicada. Éstas pueden efectuar, como es lógico, cualquier tarea, siempre que tengamos en cuenta que al llegar al final, justo antes de devolver el control, la pila debe encontrarse en la misma situación en la que estaba al principio. Si hemos introducido valores, éstos deben extraerse.

Lo importante es que el puntero de la pila, en el momento de volver, esté apuntando a los valores que introdujo en ella la instrucción `call`.

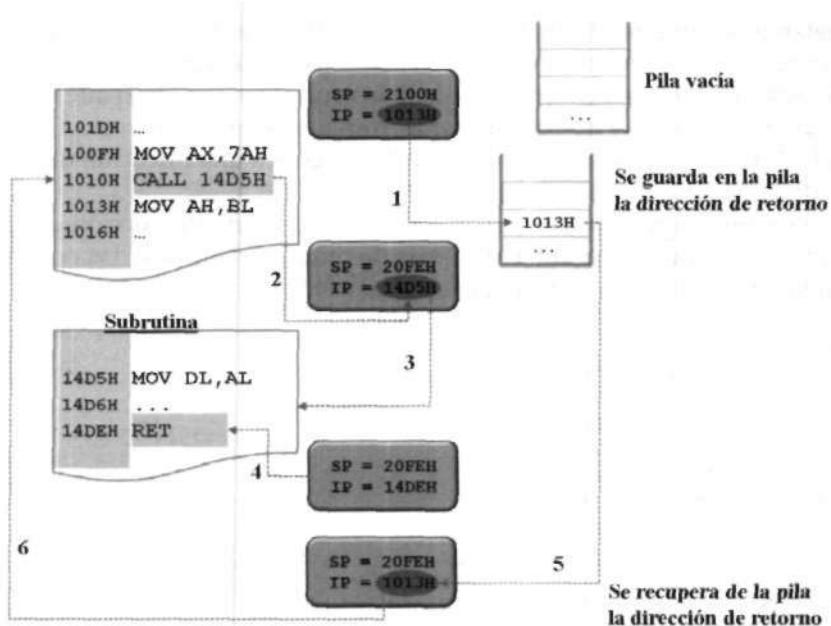


Figura 15.1. Esquema de ejecución de las instrucciones CALL y RET.

Retorno de un procedimiento

Si la instrucción `call` guarda el actual puntero de instrucción, incluyendo el segmento si es necesario, y transfiere la ejecución al procedimiento, la instrucción `ret` tiene el efecto inverso, recuperándose la dirección de la pila, devolviéndola a CS e IP, que se había incrementando antes de la llamada para proceder a ejecutar la instrucción que sigue al `call` que invocó al procedimiento.

La instrucción `ret`, por tanto, también es un salto incondicional, similar a `jmp`, con la diferencia de que la dirección a la que saltará debe estar almacenada en la pila, generalmente por una llamada previa a `call`. Teniendo esto en cuenta, entenderá lo importante que es que SS: SP conserven los valores que tenían al iniciar la ejecución del procedimiento, porque, de no ser así, la instrucción `ret` podría transferir la ejecución a un punto incorrecto y causar un fallo en la ejecución de la aplicación.

¿Cómo sabrá la instrucción `ret` si debe recuperar de la pila sólo el valor de IP o también el de CS? La respuesta es que depende de cómo se haya definido el procedimiento al que se invocó con `call`, en caso de que el ensamblador cuente con macros que faciliten la tarea de escribir procedimientos. MASM y TASM cuentan con un preprocesador que, a partir de ciertas palabras clave como `proc` y `procnear`, se encargan de generar las instrucciones ensamblador adecuadas. Aunque hay disponibles macros para NASM que efectúan un trabajo similar, su uso no es tan habitual. Por eso, al efectuar la llamada mediante `call` es necesario indicar si la dirección implica sólo a IP o también a CS, ocurriendo lo mismo al volver del procedimiento.

Existen dos variantes de la instrucción ret, llamadas retn y retf, utilizándose la primera para volver recuperando sólo IP de la pila y la segunda CS e IP. En cuanto a las llamadas, no hay varias versiones de la instrucción cali, sino que es el parámetro, la dirección a la que va a invocarse, la que implica si se guardará sólo IP o también CS. En cualquier caso, puede indicarse que la llamada es a una dirección de otro segmento poniendo la palabra f ar tras la instrucción cali.

En el ejemplo siguiente puede ver cómo se utiliza una instrucción cali para invocar a uno de tres procedimientos o rutinas distintas, cada una de las cuales dispone de su propia instrucción ret para devolver el control.

```

segment Datos
; Direcciones de varias etiquetas
; con distintas rutinas
Rutinas dw Procesol, Proceso2, Procesos

; Datos de una de las rutinas
Mensaje db 'Primer proceso?'

; Mensaje de error si se pulsa un
; número distinto a 0, 1 ó 2
PulsacionIncorrecta db 'Pulsa 0, 1 ó 2$'

segment Pila stack
resb 64
FinPila:

; Segmento de código
segment Código
.start:

; Preparamos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; DS apunta al segmento de datos
mov ax, Datos
mov ds, ax

xor ah, ah ; esperamos una tecla
int 16h

; comprobamos que la tecla
; sea 0, 1 ó 2
cmp al, '0'
jb Incorrecto
cmp al, '2'
ja Incorrecto

; Convertir el carácter en
; un número
xor ah, ah
sub al> '0'

```

```
; lleva el valor a BX  
; multiplicándolo por 2  
snl ax, 1  
mov bx, ax
```

```
; invocamos a la rutina  
cali [Rutinas+bx]  
; y finalizamos  
jmp Salir
```

Incorrecto: ; mostrar mensaje de error

```
mov dx, PulsacionIncorrecta  
mov ah, 9  
int 21h
```

Salir:

```
; salimos al sistema  
mov ah, 4ch  
int 21h
```

Procesol: ; Primera rutina
; mostramos el mensaje

```
mov dx, Mensaje  
mov ah, 9  
int 21h
```

```
ret ; y devolvemos el control
```

Proceso2: ; Segunda rutina
; Mostramos un carácter en pantalla

```
mov ax, 0b800h  
mov es, ax  
mov bx, 12*160+80  
mov word [es:bx], 0370fh
```

```
ret ; y devolvemos el control
```

Proceso?: ; Tercera rutina

```
ret ; simplemente devuelve el control
```

Tras configurar adecuadamente los registros de segmento DS y SS, esperamos la pulsación de una tecla que deberá ser 0, 1 ó 2. El servicio 0 de la interrupción 16h devuelve el código ASCII de la tecla pulsada en el registro AL, por eso lo siguiente que hacemos es comprobar si su valor se encuentra entre 0 y 2, saltando en caso de que no sea así a la etiqueta **Incorrecto**. En ella se mostrará un mensaje indicativo y se finalizará el programa.

Asumiendo que el carácter es uno de los que se espera, lo que tenemos en AL es un código ASCII, por ejemplo el valor 49 correspondiente al carácter '1'. Si restamos a ese dato el código del carácter 0, lo que nos queda es un número comprendido entre 0 y 2, que nos indica la rutina que debemos ejecutar.

Las direcciones de los procedimientos se encuentran a partir del campo **Rutinas** y, puesto que cada dirección ocupa una palabra, multiplicamos por 2 (simplemente

desplazando un bit a la izquierda el contenido del registro) y lo basamos a BX, registro que vamos a emplear como desplazamiento en el argumento que sigue a cali.

Esto provocará la ejecución de las sentencias que hay tras las etiquetas Procesol, Proceso2 o Proceso3, según el caso, que siempre terminarán por volver a la sentencia que sigue a cali.

Nota

En este caso los procedimientos son sólo tres y, además, relativamente simples, pero esta técnica es útil cuando las posibilidades son muchas, evitando múltiples decisiones sucesivas.

Salvaguarda de los registros

En el ejemplo anterior, el cometido de los procedimientos es bastante simple pero, a pesar de ello, se modifican varios registros, como AX, ES y BX, en el caso de Proceso2. ¿Qué ocurre si el código que ha llamado al procedimiento tenía valores útiles en dichos registros? Suponga que antes de llegar al cali se hubiesen asignado valores a algunos de esos registros y que, al volver del procedimiento, se pretende utilizarlos, sin advertir que han sido modificados.

Tomemos el ejemplo del capítulo previo que llenaba la pantalla de asteriscos, restaurándola después, y dividámoslo en tres procedimientos, estructurando así el código y haciendo más fácil su comprensión.

Quedaría como sigue:

```

segment Datos
; Reservamos espacio para poder
; guardar el contenido de la pantalla
Pantalla resb 25*80*2

segment Pila stack
    resb 64
FinPila:

        ; Segmento de código
        segment Código
..start:
        ; Preparamos la pila
        mov ax, Pila
        mov ss, ax
        mov sp, FinPila

        ; Configuramos DS
        ; para acceder a la pantalla
        mov ax, 0b800h
        mov ds, ax

```

```
; Llenamos la pantalla de asteriscos
mov ex, 2000 ; 2000 caracteres
xor bx, bx ; desde el principio de pantalla
mov al, '*' ; asteriscos
mov ah, 70h ; en video inverso

; Guardamos el contenido
; de la pantalla
cali GuardaPantalla

Llena:
; introducimos caracter y atributo
mov [bx], ax
inc bx ; pasamos a la posición siguiente
inc bx
loop Llena ; y repetimos hasta el final

; esperamos la pulsación de una tecla
xor ah, ah
int 16h

; restauramos el contenido de la pantalla
cali RecuperaPantalla

Salir:
; salimos al sistema
mov ah, 4ch
int 21h

; Procedimiento para guardar
; el contenido de la pantalla
; en la variable Pantalla

GuardaPantalla:
; DS:SI apunta a la pantalla
mov ax, 0b800h
mov ds, ax
xor si, si

; ES:DI apunta a nuestra variable
mov ax, Datos
mov es, ax
mov di, Pantalla

; efectuamos la transferencia
cali TransfiereDatos

; y devolvemos el control
re L

;
; Procedimiento para restaurar
; el contenido de la pantalla
; desde la variable Pantalla
```

RecuperaPantalla:

```
; DS:SI apunta a la variable
mov ax, Datos
mov ds, ax
mov si, Pantalla

; ES:DI apunta a la pantalla
mov ax, 0b800h
mov es, ax
xor di, di

; efectuamos la transferencia
cali TransiereDatos

; y devolvemos el control
ret
```

; Procedimiento que efectúa
; la transferencia de datos

TransiereDatos:

```
; Vamos a copiar 4000 bytes
mov ex, 4000
cid ; incrementar automáticamente SI y DI
```

BucleO:

```
; movemos el contenido de la celdilla
; apuntada por SI a la que indica DI
movsb
loop BucleO ; repetir

ret ; devolvemos el control
```

Hemos separado el proceso de copiar el contenido de la pantalla a la memoria de nuestro programa en un procedimiento, el de restauración en otro y, por último, codificamos un tercero que es el que se encarga de la transferencia real de los datos, evitando la repetición de este código en los dos anteriores. El cuerpo principal del programa, que tiene como objetivo llenar la pantalla de asteriscos y después devolver a su estado inicial, prepara sus registros, a continuación llama a GuardaPantalla, llena la memoria de asteriscos y, tras esperar la pulsación de una tecla, llama a RecuperaPantalla para restaurar el contenido.

Aparentemente todo es perfecto pero, si ejecuta el programa, verá que el efecto no es el deseado. La pantalla en realidad se llena de otro carácter (véase la figura 15.2) y no aparece en negro sobre blanco.

Si ejecuta el programa paso a paso, desde DEBUG o GRDB, descubrirá rápidamente dónde está el problema. Antes de llamar a GuardaPantalla el registro AX contiene el carácter y el atributo a introducir en pantalla, pero en ese procedimiento el registro AX se ve alterado.

Por norma, todo procedimiento que vaya a modificar un determinado registro debe preservarlo antes, restaurándolo antes de devolver el control. De esta forma su trabajo

se efectuará de manera transparente, sin afectar al comportamiento del programa que utilice la rutina. El recurso preferente para almacenamiento temporal, como ya sabe, es la pila.

Ya conoce las instrucciones push y pop, mediante las cuales puede almacenar y recuperar los registros de manera individual.

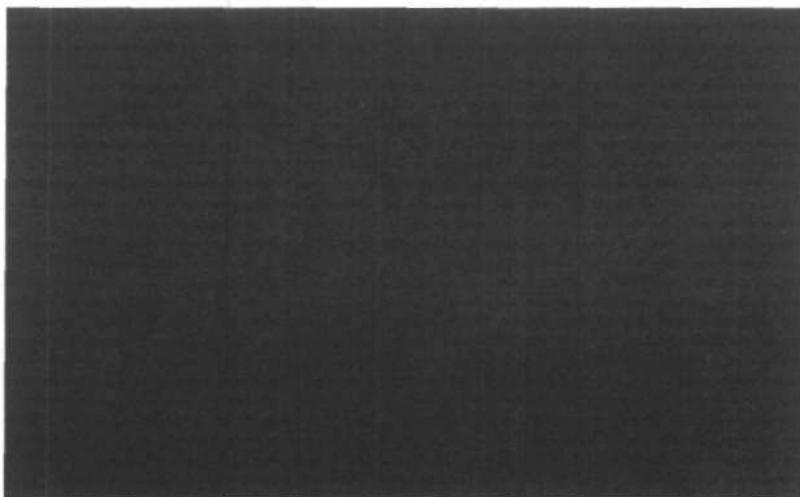


Figura 15.2. En lugar de asteriscos la pantalla se llena con otro símbolo.

Modifique el programa añadiendo al inicio del primer procedimiento, justo detrás de la etiqueta GuardaPantalla, las cuatro sentencias siguientes:

```
push ax  
push ds  
push es  
push si
```

Con ellas guardamos los registros que vamos a utilizar en el código de la rutina. Justo antes de la instrucción ret, deberemos insertar las sentencias que se muestran a continuación:

```
pop si  
pop es  
pop ds  
pop ax
```

De esta manera los registros quedarán, al salir de la subrutina, como estaban inicialmente y el programa funcionará tal y como se esperaba. Recuerde que los registros deben recuperarse en orden inverso y que es de vital importancia no olvidar nada en la pila. Si nos faltase un pop de un registro, al ejecutar la instrucción ret ésta recuperaría ese valor y lo asumiría como la dirección a la que debe saltar, probablemente causando un problema en el programa.

Si en un procedimiento va a modificar la práctica totalidad de los registros de propósito general, en lugar de usar múltiples instrucciones push, para guardarlos, y pop, para recuperarlos, puede servirse de las instrucciones pushaypopa. La primera guarda en la pila los registros AX, ex, DX, BX, SP, BP, SI y Di, en ese orden concreto, mientras que la segunda los restaura en orden inverso. Estas instrucciones están disponibles siempre que el procesador sea un 80286 o posterior.

Transferencia de parámetros

Los procedimientos que hemos codificado en el ejemplo anterior efectúan siempre el mismo trabajo, indistintamente del punto desde el que se les invoque o el número de veces que se ejecuten. La funcionalidad de un procedimiento puede ser mucho más flexible en caso de que se adapte sobre la base de los parámetros que reciba. No es lo mismo un procedimiento que dibuje un recuadro de dimensiones fijas en una posición fija que otro al que pueda indicársele tanto el tamaño como la situación en pantalla. Obviamente, este segundo procedimiento será útil en muchas más situaciones que el primero.

Tras la instrucción `call` tan sólo puede facilitarse la dirección del procedimiento al que va a invocarse, pero ningún parámetro adicional. Si deseamos transferir argumentos desde el programa principal a las rutinas, por tanto, tendremos que recurrir a algún otro sistema, como ya se indicase antes, el uso de ciertos registros o bien la pila. En caso de optarse por la pila, hay que tener en cuenta que ésta también contiene la dirección de retorno y que dicha dirección debe encontrarse ahí en el momento en que se ejecute la instrucción `ret`.

Para la devolución de parámetros cabe decir otro tanto, pudiendo utilizarse tanto registros como la pila, o bien áreas de datos comunes bien conocidas, el equivalente a las variables globales utilizadas en los lenguajes de alto nivel.

Una rutina de espera

Vamos a ver en la práctica cómo una rutina puede tomar y devolver parámetros construyendo un procedimiento que nos permite programar esperas de un tiempo determinado. En realidad, el procedimiento estará formado de dos partes: una rutina que obtendrá el tiempo y lo devolverá como resultado en el registro BX, por una parte, y otra que se servirá de la anterior para esperar el número de segundos que se indique en el registro CX, por otra.

Lo primero que necesitaremos, lógicamente, es saber cómo calcular una espera de N segundos. Para ello podemos servirnos del reloj con que cuentan todos los PC actuales, un pequeño integrado, alimentado por una batería, con el que nos podemos comunicar mediante una serie de puertos de entrada/salida. El microprocesador cuenta con dos

instrucciones que facilita la escritura y lectura de valores en dichos puertos, por lo que tan sólo necesitamos saber de qué puertos se trata y qué significado tienen los valores.

Instrucciones de E/S

Los puertos de entrada/salida, E/S para abreviar, representan el mecanismo mediante el cual el microprocesador puede comunicarse con dispositivos externos. Algunos de esos puertos son de salida o escritura, otros de entrada o lectura y algunos contemplan tanto la escritura como la lectura. Las instrucciones para efectuar esas operaciones son `out` e `in`, respectivamente.

Mediante la instrucción `out` se envía el contenido del registro AL o AX, el que se entregue como segundo parámetro, al puerto de E/S indicado como primer argumento. Éste puede ser un valor inmediato, en caso de que el puerto se encuentre entre 0 y 255, o bien facilitarse en el registro DX, si está entre 256 y 65535. El efecto de la instrucción dependerá, lógicamente, del puerto en el que se escriba y el valor que se escriba.

Nota

No pruebe a escribir valores en puertos aleatorios sin saber a qué corresponden, ya que puede alterar seriamente el funcionamiento de su ordenador.

La instrucción complementaria a `out`, que es `in`, toma exactamente los mismos parámetros, pero en orden inverso. Tendremos que facilitar en primer lugar, por tanto, el registro AL o AX, dependiendo del tamaño del dato a recuperar, y en segundo el número de puerto que, como en el caso anterior, puede ser un valor inmediato o bien el registro DX, según que el puerto sea o no superior al 255.

Comunicación con el reloj del sistema

Como se decía antes, el reloj de tiempo real que hay en el sistema es un integrado mediante el cual podemos comunicarnos a través de una serie de puertos. Para el objetivo que perseguimos, nos basta con conocer dos de ellos: el 7 Oh y el 7 Ih. El primero es de escritura y el segundo de lectura. Este reloj tiene un área de datos en la que almacena la fecha y hora actuales, un byte para cada porción de la fecha y la hora, así como la hora de alarma. En la posición 0 de esa área se encuentran los segundos, en la 2 los minutos y en la 4 las horas correspondientes a la hora actual.

Para poder recuperar el contenido de una de dichas áreas de memoria, leyendo del puerto 7 lh, antes tenemos que indicar la posición a leer, escribiéndola en el puerto 7 Oh. A fin de recuperar los segundos, por ejemplo, la secuencia de instrucciones a ejecutar sería la siguiente:

```
xor al, al  
out 70h, al  
in al, 71h
```

Tras la instrucción `in`, tendríamos en el registro AL los segundos correspondientes a la hora actual. Cambiando el valor de AL, antes de la instrucción `out`, podríamos leer cualquier otra de las posiciones de memoria del reloj.

Código de la rutina

Ahora que ya sabemos cómo recuperar datos del reloj del sistema, codificar la rutina que espere durante el número de segundos indicados en el registro CX no será difícil. Comenzaremos implementando una subrutina del procedimiento principal, cuyo fin será obtener el minuto y segundo de la hora actual expresado en segundos, es decir, multiplicando los minutos por 60 y sumándole los segundos. El valor se devolverá como parámetro en el registro BX. Éste es el código de dicha rutina.

```
; Procedimiento que obtiene
; los minutos y segundos del
; reloj, los convierte a
; segundos y devuelve en BX
;
SegundosActual:
    push ax ; guardamos AX
    ;
    ; queremos leer los minutos
    mov al, 2
    out 70h, al
    in al, 71h
    ;
    ; los multiplicamos por 60
    mov bl, 60
    muí bl
    ;
    ; y guardamos en BX
    mov bx, ax
    ;
    ; queremos leer los segundos
    xor al, al
    out 70h, al
    in al, 71h
    ;
    ; los sumamos
    add bx, ax
    ;
    pop ax ; y volvemos
    ret
```

Siguiendo los comentarios introducidos, el código es fácil de interpretar. Se compone, básicamente, de cuatro pasos: recuperación del minuto que indica el reloj, multiplicación por 60 de dicho dato, recuperación del segundo del reloj y suma al resultado obtenido antes.

Para estas operaciones es necesario usar el registro AX, por eso se guarda al inicio de la rutina y restaura justo antes de salir. En BX quedará el dato útil.

Disponiendo de esta rutina, centrémonos ahora en la codificación del procedimiento principal. Éste deberá obtener el tiempo actual en segundos, tal y como se lo entrega la rutina SegundosActual, sumarle el número de segundos que debe esperar, recibido en CX, y, a partir de ahí, quedar a la espera hasta que el valor que es devuelto por SegundosActual sea mayor que el obtenido de esa suma. El código del procedimiento Espera es el siguiente:

```
; Procedimiento que provoca  
; una espera de N segundos
```

```
; Espera recibir en CX el  
; número de segundos
```

Espera:

```
pusha ; guardar registros  
  
; obtenemos el número  
; de minutos y segundos que  
; indica el reloj ahora  
cali SegundosActual  
; lo movemos a BX  
mov ax, bx  
; y le sumamos los segundos  
; a esperar  
add ax, ex
```

Buclel:

```
; vigilamos los minutos y  
; segundos del reloj  
cali SegundosActual  
; viendo si ya se ha completado  
; la espera  
emp ax, bx  
; volviendo al bucle  
; de no ser así  
ja Buclel  
  
popa ; recuperamos registros  
ret ; y volvemos
```

Como puede ver, el procedimiento provoca la espera sin modificar ninguno de los registros implicados en el proceso, dado que los preserva todos al inicio y recupera justo antes de volver.

Un ejemplo de uso

Puede comprobar el funcionamiento de la rutina de espera con un programa muy sencillo, simplemente invocándola y devolviendo el control al sistema. Verá que, al ejecutar el programa, el indicador del sistema tarda en volver a aparecer exactamente los segundos que hubiese indicado. Su uso, no obstante, puede permitirnos efectos algo más vistosos, como el del ejemplo propuesto a continuación.

En este ejemplo vamos a tener, aparte de las rutinas Espera y SegundosActual, varias más: GuardaPantalla, RecuperaPantalla y Transfiere, que ya conoce de un ejemplo anterior, y DibujaRecuadro, un procedimiento que dibujará el recuadro que usábamos en ejemplos del capítulo previo en una posición determinada entre-gada en DH y DL.

La parte central del programa hará uso de todas estas rutinas para simular el mo-vimiento del recuadro por la pantalla. Para ello comenzará guardando su contenido, tras lo cual entrará en un bucle que restablecerá la pantalla, dibujará el recuadro en una determinada posición y esperar un par de segundos. Tras esto modificará las coordena-das, incrementándolas, y volverá a repetir el proceso unas cuantas veces. Terminado el bucle se restaurará la pantalla y el recuadro, por tanto, desaparecerá. El efecto aparen-te es que el recuadro se desplaza hacia el área inferior derecha de la pantalla, un efecto que sólo podrá apreciar ejecutando el programa en su sistema.

El código del programa, al que tiene que añadir al final el de las dos rutinas del punto previo, es el que puede verse a continuación. Se facilitan las demás rutinas ya que han experimentado algunos cambios respecto a las versiones iniciales.

```

segment Datos
; Reservamos espacio para poder
; guardar el contenido de la pantalla
Pantalla resb 25*80*2

segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
..start:

; Preparamos la pila
mov ax, Pila
mov aa, ax
mov op, FinPila

; Guardamos el contenido
; de la pantalla
cali GuardaPantalla

; establecemos la primera fila
; y columna
mov di, 2
mov dh, 10

; vamos a repetir 15 veces
mov ex, 15

Movimiento:
: recuperamos la pantalla
cali RecuperaPantalla
; y dibujamos el recuadro
cali DibujaRecuadro

```

```
; guardamos el contador
push ex

; esperamos 2 segundos
mov nx, 7
cali Espera

; recuperamos el contador
pop ex

; actualizamos la posición
inc di
inc dh
inc dh

; repetimos
loop Movimiento

; restauramos el contenido de la pantalla
cali RecuperaPantalla
```

Salir:

```
; salimos al sistema
mov ah, Ich
int 21h
```

```
; Procedimiento para guardar
; el contenido de la pantalla
; en la variable Pantalla


---


```

GuardaPantalla:

```
pusha ; conservamos los registros
push ds
push es

; DS:SI apunta a la pantalla
mov ax, UbüOUh
mov ds, ax
xor si, si

; ES:Di apunta a nuestra variable
mov ax, Datos
mov es, ax
mov di, Pantalla

; efectuamos la transferencia
call TransfiereDato8

; restaurar registros
pop es
pop ds
popa

; y devolvemos el control
ret
```

```

; Procedimiento para restaurar
; el contenido de la pantalla
; desde la variable Pantalla
;-----  

RecuperaPantalla:
    pusha ; guardar registros
    push ds
    push es

    ; DS:SI apunta a la variable
    mov ax, Datos
    mov ds, ax
    mov si, Pantalla

    ; ES:DI apunta a la pantalla
    mov ax, ObSOOh
    mov es, ax
    xor di, di

    ; efectuamos la transferencia
    cali TransfiereDatos

    ; restaurar registros
    pop es
    pop ds
    popa

    ; y devolvemos el control
    ret

; Procedimiento que efectúa
; la transferencia de datos

TransfiereDatos:
    ; Vamos a copiar 4000 bytes
    mov ex, 4000
    cid ; incrementar automáticamente SI y DI

BucleO:
    ; movemos el contenido de la celdilla
    ; apuntada por SI a la que indica DI
    movsb
    loop BucleO ; repetir
    ret ; devolvemos el control

; Procedimiento que dibuja el
; cuadrado en pantalla

; Espera recibir en DL,DH la
; posición en que debe ponerlo

DibujaRecuadro:
    pusha,- guardamos registros
    push ds

```

```

; DS apuntará al segmento
; de pantalla
mov ax, 0b800h
mov ds, ax

mov ex, 8 ; número de líneas

BucleLineas:
    push dx ; guardamos la posición

    ; guardamos CX en la pila
    push ex

    mov ex, 20 ; número de columnas

BucleColumnnaa:
    ; Calculamos la posición
    mov al, 160 ; bytes por línea
    muí di ; por número de línea

    mov bx, ax ; guardamos en BX

    mov al, 2 ; bytes por columna
    muí dh ; por número de columna

    add bx, ax ; sumamos a bx

    ; ponemos un carácter en esa posición
    mov word [bx], 070feh

    inc dh ; incrementamos la columna

    loop BucleColumnnas ; y repetimos

    ; al finalizar todas las columnas
    ; de una linea

    pop ex ; recuperamos contador líneas

    pop dx ; recuperamos la posición
    inc di ; y pasamos a línea siguiente

    loop BucleLineas ; y repetimos

    pop ds ; recuperamos registros
    popa

    ret ; volvemos

```

Haga pruebas modificando el valor asignado al registro CX antes de llamar al procedimiento Espera, así podrá ver cómo el recuadro se mueve más o menos rápido debido al retardo introducido por dicha rutina.

Macros

Los procedimientos o subrutinas, como se indicaba al inicio del capítulo, son un recurso que nos ofrece el propio lenguaje ensamblador de los x86, indistintamente de las herramientas que utilicemos para ensamblar y enlazar el código. Éstas, de forma adicional, pueden ofrecer otros recursos que hagan más fácil la codificación, siendo uno de los más usados las macros.

Una macro es similar a un procedimiento o función, pudiendo ser tan simple como una sola línea o tan compleja como se deseé. La diferencia entre una macro y un procedimiento, como los que acabamos de conocer, es que las macros son procesadas por el ensamblador, por ejemplo NASM, momento en el que se sustituyen por las instrucciones apropiadas. Éstas pasan a formar parte del programa que se ensambla para producir el código ejecutable.

La sintaxis empleada para definir y usar macros depende de cada ensamblador concreto, de tal forma que las macros de MASM, por ejemplo, no pueden, por regla general, usarse directamente en NASM o viceversa.

Dado que NASM puede utilizarse en distintos sistemas operativos, vamos a centrarnos en la creación de macros con esta herramienta, evitando así tener que usar diferentes sintaxis en Linux, DOS o Windows, lo que nos obligaría a tener diferentes versiones de las mismas macros.

Macros simples

Una macro sencilla, que ocupe una sola línea, puede crearse mediante la directiva `%define`. Tras ésta facilitaremos el identificador de la macro, introduciendo entre paréntesis los parámetros en caso de que éstos existieran. A continuación iría el desarrollo de la macro, en el que podemos usar los operadores reconocidos por el ensamblador, símbolos, registros, etiquetas, etc.

En el programa siguiente puede ver cómo se define una macro, llamada `Posición`, que recibe dos parámetros y los usa para calcular la posición de memoria que corresponde a unas coordenadas dadas. Más adelante, se usa esta macro para colocar un carácter en distintas posiciones de pantalla, sin tener que ejecutar cálculo alguno, simplemente indicando la posición, columna y fila, donde se quiere poner.

```
; Macro para calcular la posición
; de memoria correspondiente a
; una cierta columna y fila
%define Posición(x,y) y*160+x*2

segment Pila stack
    resw 512
FinPila:

I Segmento de código
segment Código
```

```

.start:
; DS apuntará a la pantalla
mov ax, 0b800h
mov ds, ax

; preparamos el carácter
mov ax, 7041h

; lo colocamos en varias
; posiciones
mov [Posición(35, 8)], ax
mov [Posición(60,12)], ax
mov [Posición(10,20)], ax

; salimos al sistema
mov ah, 4ch
int 21h

```

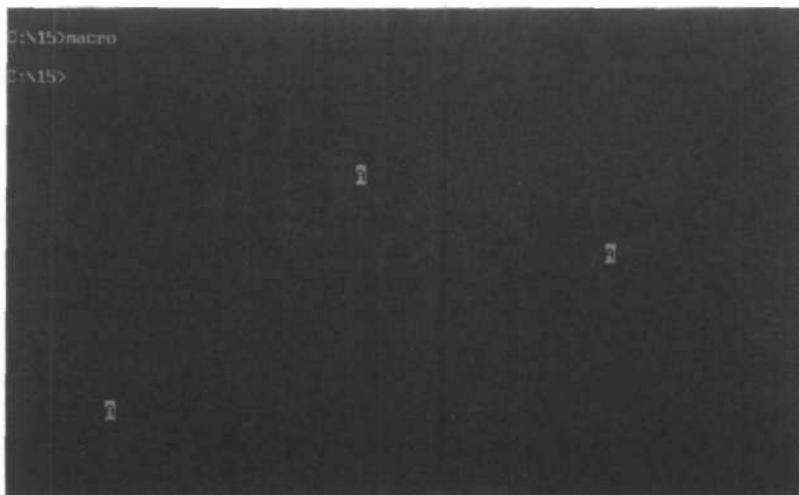


Figura 15.3. El carácter aparece en las tres posiciones indicadas.

Expansión de la macro

Los ensambladores como NASM, MASM y TASM son conocidos como macroensambladores, ya que, además de ensamblar generando código máquina, también efectúan un proceso, independiente y previo al ensamblado, mediante el cual expande las macros introduciendo en el archivo las sentencias que generan. Tras la expansión, el macroensamblador ya dispone del código que deberá ser ensamblado. A la parte del macroensamblador que se encarga de procesar y expandir las macros se le conoce como *preprocesador*, ya que efectúa su trabajo antes que el ensamblador.

Por defecto, al invocar a NASM se ejecuta el preprocesado y después el ensamblado, generando un archivo de código temporal intermedio que no llegamos a ver. Podemos,

no obstante, indicar a NASM que deseamos que efectúe sólo el preprocesado o bien sólo el ensamblado. Suponiendo que tiene el código del ejemplo anterior almacenada en el archivo macro.asm, introduzca la orden nasm -e macro.asm en la línea de comandos del sistema. Deberá obtener un resultado similar al que aparece en la figura 15.4. Como puede ver, ya no existe la macro Posición y, donde aparecía, se ha sustituido por el cálculo adecuado.

```
D:\Ejemplos\15>nasm -e macro.asm
%line 1+1 macro.asm

[segment Pila stack]
resw 512
FinPila:

[segment Código]
.start:
    mov ax, 0b800h
    mov ds, ax

    mov ax, 7041h

    mov [8*160+35*2], ax
    mov [12*160+60*2], ax
    mov [20*160+10*2], ax

    mov ah, 4ch
    int 21h

D:\Ejemplos\15>_
```

Figura 15.4. Aspecto del código tras haberse preprocesado.

Puede redirigir la salida de la consola a un archivo, mediante el carácter >, para posteriormente ensamblarlo sin necesidad de preprocesso.

Macros complejas

En caso de que la macro que necesitamos no pueda resumirse en una línea, podemos usar las directivas %macro y fendmacro para delimitar el inicio y fin de la macro. Tras %macro indicaríamos el nombre de la macro y, opcionalmente, el número de parámetros que tendrá. En este caso no hay paréntesis ni nombres para los parámetros, simplemente un número entero que indica cuántos parámetros se esperan.

La definición de la macro se compondrá, como en el caso de % de fine, de los elementos habituales que podríamos usar en cualquier punto del programa. Además, es posible introducir los parámetros que se hayan facilitado haciendo referencia a ellos con la notación %n, donde n sería el número del parámetro, comprendido entre 1 y el número indicado en la cabecera.

Mediante las macros es posible simplificar la introducción repetitiva de secuencias cortas de instrucciones sin tener que aislarlas en un procedimiento. Por ejemplo:

```
%macro MuestraCaracter 4
    mov bx, %1*2+%2*160
    mov byte [bx], %3
    inc bx
    mov byte [bx], %4
%endmacro
```

Esta macro recibe cuatro parámetros: columna, fila, carácter y atributo. Asumiendo que DS está apuntando al segmento de la memoria de pantalla, lo que hace la macro es introducir el carácter indicado en la posición que se especifica con los dos primeros parámetros. Podría invocar a esta macro de la siguiente manera:

```
MuestraCaracter 40, 10, 'A', 70h
MuestraCaracter 40, 15, '*', 60h
```

Debe tener en cuenta que una macro se expande en el código cada vez que es invocada. Un programa que utilizase la macro `MuestraCaracter`, por lo tanto, se compondría de la secuencia de cuatro instrucciones de la macro cada vez que ésta sea utilizada, a diferencia de los procedimientos vistos antes que sólo generan código una vez.

El uso de las macros no precisa llamadas ni retornos, no hay que utilizar las instrucciones `call` y `ret`, por lo que para funciones breves puede ser un método más eficiente que la codificación de un procedimiento.

Si el proceso es complejo, por ejemplo la rutina de espera creada anteriormente, el uso de las macros no es aconsejable, ya que el tamaño del programa crecería notablemente a medida que se llame a la macro en el programa. Recuerde que cada vez que aparezca en el código el identificador `MuestraCaracter` éste será sustituido por las instrucciones `mov` e `inc` que escriben el carácter en pantalla.

Archivos de macros y procedimientos

Las instrucciones y directivas que hemos conocido en este capítulo tienen el objetivo de ayudarnos a estructurar nuestro código, y una forma de conseguir esa estructuración consiste en introducir en archivos independientes todas aquellas rutinas y macros que, potencialmente, puedan ser útiles en distintos programas. De esta manera es posible reutilizar parte de nuestro código, como haríamos en cualquier lenguaje de alto nivel.

Por convención, los archivos que alojan rutinas o macros en lenguaje ensamblador tienen la extensión `.inc`, aunque realmente esto es totalmente opcional. Para incluir esos

módulos en los correspondientes a los programas donde vayan a utilizarse, no hay más que emplear la directiva %include, facilitando el nombre del archivo entre comillas. Por ejemplo:

```
%include "Pantalla.inc"
```

Asumiendo que tuviésemos un archivo, llamado Pantalla.inc, con rutinas y macros útiles para el acceso a pantalla, la sentencia anterior incluiría todo su código en el módulo actual.

Resumen

Para crear aplicaciones de una cierta complejidad es indispensable poder estructurar el código, tanto desde el punto de vista físico, dividiéndolo en múltiples módulos, como lógico, mediante macros y procedimientos que podamos invocar cuando sea necesario. Este capítulo nos ha servido para conocer las instrucciones cali y ret, relativas a la implementación de rutinas o procedimientos, así como las directivas %define, %macro e %include.

Tras este nuevo paso, en el próximo capítulo seguiremos avanzando en la simplificación del código de nuestros programas, en esta ocasión mediante las instrucciones de repetición con que cuentan los procesadores x86. Gracias a ellas podremos ahorrarnos algunos bucles, haciendo más breves y rápidos determinados procesos.

Una de las tareas que más tiempo consume en las aplicaciones, especialmente en aquellas que manejan grandes cantidades de datos de cualquier tipo, es la transferencia de secuencias de bytes de una zona a otra, la búsqueda de datos en esos bloques de memoria, su comparación, etc. En ejemplos de capítulos previos hemos efectuado operaciones de este tipo, por ejemplo a la hora de guardar el contenido de la pantalla o llenar un área de ésa con un cierto carácter. Hasta ahora, si queríamos introducir el contenido de AX en una determinada posición de memoria usábamos una instrucción mov. Para llevar el contenido de cualquier zona a un registro, otra instrucción mov. Si necesitamos desplazar un dato desde una zona de memoria a otra, dos instrucciones mov a menos que hubiésemos configurado adecuadamente los registros necesarios para utilizar la instrucción movs. La familia de microprocesadores x86 dispone de un conjunto de instrucciones que facilitan el trabajo con secuencias de bytes o cadenas, introduciendo un cierto valor en memoria, comparándolo con otro valor, llevándolo de un punto a otro, etc. Aunque pueden resultar útiles por sí mismas, lo más interesante de estas instrucciones es que pueden situarse tras un prefijo que automatiza su repetición. Esto, como verá de inmediato, nos ahorrará la codificación manual de muchos bucles.

Orígenes, destinos e incrementos

Todas las instrucciones que vamos a conocer en los puntos siguientes precisan una dirección de origen, de la que van a tomar datos; en ocasiones una dirección de destino, en la que van a almacenarlos; un recipiente en el que se almacenarán los datos leídos o contendrá el dato a escribir y, finalmente, un señalizador que indique si los registros

que indican las direcciones de origen y destino deben incrementarse o bien reducirse a cada paso.

La dirección de origen, aquella en la que residen los datos que van a recuperarse o que se van a inspeccionar, siempre vendrá implícita en los registros DS : SI. De forma análoga, la dirección de destino, en caso de que la instrucción que vaya a usarse la requiera, se facilitará en los registros ES : DI. Es algo que sabe puesto que lo vio en un ejemplo de un capítulo previo, al copiar el contenido de la memoria de pantalla a un campo interno del programa. Aquellas instrucciones que recuperan un dato o lo almacenan, requieren el uso del registro AL, AX o EAX, según los casos. Además, el bit de dirección del registro de indicadores debe establecerse adecuadamente, antes de usar la instrucción en que estemos interesados, para optar entre el incremento o la reducción automática a cada paso. Recuerde que tiene a su disposición las instrucciones cid y std para poner a 0 o a 1 dicho indicador.

Una vez configurados todos estos aspectos: dirección de origen, dirección de fin, contenido del registro AL/AX e indicador de dirección, estaremos en disposición de comenzar a recuperar, escribir, copiar o inspeccionar la secuencia de bytes, según los casos.

Recuperación y almacenamiento de datos

Las dos primeras instrucciones que vamos a conocer nos permiten ir recuperando datos de una secuencia de bytes o palabras, obteniéndolas individualmente en el registro AL/AX, y llenar una secuencia con el contenido que tenga ese mismo registro en su versión de 8 ó 16 bits, según nos interese. En caso de que vayamos a ir leyendo una secuencia de bytes o palabras para manipularlos, algo que hemos hecho en ejemplos previos al, por ejemplo, convertir en mayúsculas las minúsculas que aparecían en pantalla, podemos obtener un código más eficiente utilizando la instrucción lods en sustitución del correspondiente mov con su ñ c asociado.

Existen dos variantes de esta instrucción: 1 odsb y 1 odsw, que indican de forma explícita si van a ir recuperándose bytes o palabras, respectivamente. En caso de utilizarse lods, será preciso facilitar un parámetro cuyo único fin es facilitarle al ensamblador la información que necesita para saber si debe leer bytes o palabras.

Antes de utilizar lods, o una de sus variantes, deberemos asignar a DS : SI la dirección donde comienza la secuencia de datos a leer. Además, el bit de dirección tiene que estar adecuadamente establecido para incrementar o reducir automáticamente el valor SI en cada llamada a lods. Ese incremento o reducción será de uno o dos bytes, dependiendo de que se obtenga un byte o una palabra.

Además de las formas lods, lodsb y lodsw, los microprocesadores 80386 y posteriores también pueden usar la instrucción lodsd para recuperar en el registro EAX el dato apuntado por DS: ESI, siendo éste una doble palabra: cuatro bytes. Operando

en modo de 64 bits también existe la posibilidad de leer una cuádruple palabra, ocho bytes, con la instrucción lodsq. En este caso la dirección se toma de DS : RSI y el dato queda en el registro RAX, la versión de 64 bits del acumulador. Dependiendo del bit de dirección, el contenido de ESI o RSI se incrementará/reducirá adecuadamente, según se indicaba antes.

Suponga, por ejemplo, que quiere codificar un programa que recorra el contenido actual de la pantalla y cuente el número de letras mayúsculas que hay. En vez de usar, como en otros ejemplos, el registro BX para contener el desplazamiento e ir incrementando el valor de dicho registro a cada ciclo del bucle, podríamos configurar adecuadamente los registros DS : SI, así como el indicador de dirección, utilizando la instrucción lods w para ir recuperando en AX cada pareja de carácter/atributo. El código quedaría así:

```
segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
..start:

    ; Preparamos la pila
    mov ax, Pila
    mov ^fi, ax
    mov sp, FinPila

    ; DS:SI apuntan al inicio
    ; de la memoria de pantalla
    mov ax, 0b800h
    mov ds, ax
    xor si, si

    ; DX nos servirá para ir
    ; contando las mayúsculas
    xor dx, dx

    ; tendremos que recorrer
    ; 200Q posiciones de 2 bytes
    mov ex, 2000

    ; incrementando el
    ; valor de SI
    cid

Bucle:
    ; recuperaremos una palabra
    lodsw

    ; comprobamos si el carácter
    ; es una letra mayúscula
    cmp al, 'A'
    jb NoMayuscula
```

```

        cmp al, 'Z'
        ja NoMayuscula

        ; de ser así incrementamos DX
        inc dx

NoMayuscula:
        ; en cualquier caso repetimos
        ; el proceso en toda la pantalla
        loop Bucle

        ; Salimos al sistema
        mov ah, 4ch
        int 21h

```

Para poder ver el resultado, puesto que éste queda en DX, tendrá que ejecutar el programa desde DEBUG o GRDB y observar su contenido.

Conversión de binario a decimal

En muchos programas, como en el caso anterior, tras efectuar una serie de cálculos nos encontraremos con un resultado, un número almacenado en binario en un registro, que necesitamos mostrar al usuario, por ejemplo escribiéndolo por la consola. Si introducimos el valor de AL directamente en la memoria de pantalla, sin embargo, no veremos el número propiamente dicho, sino un carácter que tiene como código asociado el byte que indica AL. Es necesario, por tanto, efectuar una conversión de binario a decimal, generando una cadena de caracteres a partir del valor contenido en un registro.

Con los conocimientos que hemos adquirido hasta ahora podemos escribir una rutina que haga precisamente eso. Supongamos que el número a convertir vendrá dado en AL, mientras que en ES : DI se indicará la dirección del último dígito, de forma que se irá reduciendo DI hasta introducir los tres posibles. El proceso, básicamente, consistirá en una división sucesiva por 10 quedándonos con el resto, devuelto en AH, hasta que el cociente, contenido en AL, no sea mayor que 9. El código del procedimiento es el siguiente:

```

; Este procedimiento convierte
; el valor de AL en una cadena
; de tres caracteres

; Entrada: AL = número a convertir
;           ES:DI = destino cadena

EnteroCadena:
        ; establecemos valor inicial
        mov byte [di], '0'

        ; comprobamos si AL es cero
        or rfl, al
        ; de ser así, no hay más
        ; que hacer
        jz FinConversion

```

```

push bx ; guardamos bx
; y establecemos el divisor
mov bl, 10

Bucle0:
; vamos dividiendo por 10
div bl

; quedándonos con el resto
; que convertimos a ASCII
add ah, '0'
; y guardamos
mov [di], ah
; retrocediendo al dígito anterior
dec di

; eliminamos el contenido
; de AH para quedarnos con
; el cociente de AL.
xor ah, ah

; si el cociente es mayor que 9
cmp al, 9
; seguimos dividiendo
iaRuc1o0

; en caso contrario guardamos
add al, '0'
mov [di], al

pop bx ; recuperamos BX

```

```

FinConversion:
ret

```

Para comprobar el funcionamiento de esta rutina deberá hacer algunos cambios en el programa de ejemplo del punto previo.

Para comenzar, añadiremos al principio la declaración del segmento de datos con la definición de una cadena de caracteres:

```

segment Datos
; Mensaje informativo
Mensaje db 'mayúsculas$'

```

Observe que hay varios espacios en blanco, concretamente cinco, antes de la palabra mayúsculas. A continuación insertaremos, antes de las sentencias que devuelven el control al sistema, el código siguiente:

```

; Preparamos DS y ES
; apuntando al segmento
;- de datos
mov ax, Datos
mov es, ax
mov ds, ax

```

```

; DI indica el punto
; donde se insertará
; el último dígito
mov di, Mensaje+3

; debemos facilitar el número
; a convertir en AL, y lo tenemos
; en DX
mov ax, dx

; convertimos
cali EnteroCadena

; mostramos el mensaje por pantalla
mov dx, Mensaje
mov ah, 9
int 21h

```

Hecho este cambio, al ejecutar el programa obtendremos una respuesta similar a la que se aprecia en la figura 16.1. A pesar de que el programa no recoge parámetros de la línea de comandos, nada nos impide, tal y como se ha hecho en dicha figura, introducir una secuencia de caracteres para que, al estar en pantalla, el programa los cuente.

The screenshot shows a DOS terminal window with the following text:

```

:C:\>contar
01 mayúsculas
C:\>

:C:\>contar ABCDEFGH
11 mayúsculas
C:\>

:C:\>DIR /W
Directory of C:\>
C:\>          . . .
BÚSQUEDA.ASM   BÚSQUEDA.EXE    BÚSQUEDA.OBJ      BÚSQUEDA.EXE    BÚSQUEDA.OBJ
CONTAR.OBJ     GRAB.OPT       LETRAS.ASM      CONTAR.ASM     CONTAR.EXE
MOVIL.ASM      MOVIL.EXE      MOVIL.OBJ       LETRAS.ASM     LETRAS.OBJ
17 File(s)        29,159 Bytes.
2 Dir(s)         110,540,000 Bytes free.

:C:\>contar
176 mayúsculas
C:\>

```

Figura 16.1. El programa nos indica el número de letras mayúsculas que encuentra.

La rutina ofrecida como ejemplo asume que el dato a dividir es de 8 bits, pero puede modificar esto introduciendo un valor de 16 bits en AX, al tiempo que pone a 0 el registro DX, y usando como divisor el registro DX en lugar de BL, aunque con el mismo valor 10. En la comprobación de fin, en lugar de verificar que AL no es mayor que 9 usaríamos AX.

Almacenamiento de valores

La instrucción complementaria de lods es stos que, como puede suponer, toma el valor que existe en AL o AX y lo introduce en la dirección indicada por ES : DI, incrementando o reduciendo el valor de este último registro según indique el bit de dirección del registro de indicadores.

Existen dos variantes de esta instrucción: stosb y stosw. La primera escribe el valor de AL en ES: DI e incrementa o reduce en una unidad la dirección de DI, mientras que la segunda escribe el valor de AX e incrementa o reduce en dos unidades la dirección de DI. También están disponibles las versiones que trabajan con dobles y cuádruples palabras, en microprocesadores de 32 y 64 bits, respectivamente.

Suponga que necesita en un programa llenar la pantalla de espacios, borrando todo su contenido, pero dejando el color de fondo en verde. Sustituyendo la instrucción mov a la que recurriríamos como primera solución, así como el incremento manual de la dirección base, por la instrucción stosw, el programa quedaría así:

```
segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
..start:

    ; Preparamos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; ES:DI apuntan al inicio
    ; de la memoria de pantalla
    mov ax, 0b800h
    mov es, ax
    xor di, di

    ; Carácter y número de ciclos
    mov ax, 6020h
    mov ex, 2000

    ; incrementando el
    ; valor de DI
cid

Bucle:
    . escribimos una palabra
    stosw

loop Bucle

    ; Salimos al sistema
    mov ah, 4ch
    int 21h
```

El número de instrucciones que hay en el interior del bucle, entre la etiqueta Bucle y la instrucción loop, es menor, por lo que el código generado será más eficiente.

Repetición automática de la operación

Además de incrementar o reducir automáticamente la dirección de origen o destino, según se trate de lods o stos, una ventaja adicional de estas instrucciones es que pueden repetir automáticamente la operación que ejecutan sin necesidad de codificar bucle alguno. Para ello basta con indicar el número de ciclos, como es habitual en el registro CX, y a continuación preceder la instrucción con la palabra rep. De esta forma el ejemplo del punto anterior quedaría aún más simple, como se ve a continuación:

```

segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
..start:

; Preparamos la pila
mov ax, Pila
mov 38, ax
mov sp, FinPila

; ES:DI apuntan al inicio
; de la memoria de pantalla
mov ax, 0b800h
mov es, ax
xor di, di

; Carácter y número de ciclos
mov ax, 6020h
mov ex, 2000

; incrementando el
; valor de DI
cid

; llenamos la pantalla
rep stosw

; Salimos al sistema
mov ah, 4ch
int 21h

```

El resultado obtenido, si ensambla el programa y lo comprueba, es exactamente el mismo pero más rápido, una circunstancia que resulta imposible de apreciar a simple vista ya que la cantidad de información manipulada es muy pequeña.

De uso habitual con la instrucción stos, y las demás instrucciones que conoceremos en este capítulo para operar sobre secuencias de bytes, el prefijo rep no resulta útil con

`lod`s ya que esa instrucción lee un dato y lo almacena en AL/AX. Si repetimos el proceso, los valores que va tomando dicho registro se pierden a cada ciclo.

Mediante `rep stosw` podríamos optimizar, en el programa del ejemplo anterior que movía un recuadro por pantalla, el código que se encarga, precisamente, de dibujar ese recuadro, automatizando la inserción de todas las columnas de cada fila en lugar de usar un bucle anidado.

Transferencia de una secuencia de datos

La tercera instrucción relacionada con la manipulación de secuencias ya la conocemos: `movs`. Cuenta con las variantes `movsb` y `movsw`. La primera transfiere un byte desde DS : SI a ES : DI, incrementando SI y DI en una unidad o reduciendo en caso de que el bit de dirección así lo indique. La segunda opera con palabras en vez de bytes. A éstas hay que agregar las instrucciones `movsd` y `movsq`, equivalentes a las anteriores pero operando sobre dobles palabras y cuádruples palabras.

Al igual que `stos`, `movs` puede precederse con la palabra `rep`, haciendo así innecesaria la codificación manual de un bucle.

Revise el código del procedimiento `GuardaPantalla` de uno de los ejemplos del capítulo previo. Tal como recordará, hacía uso de una rutina adicional para transferir los bytes desde la memoria de pantalla a un campo previamente definido en el programa. Podemos eliminar esa rutina, que implementaba un bucle, dejando `GuardaPantalla` como sigue:

```
GuardaPantalla:  
    pusha ; conservamos los registros  
    push ds  
    push es  
  
    ; DF:SI apunta a la pantalla  
    mov ax, 0b800h  
    mov ds, ax  
    xor si, si  
  
    ; ES:DI apunta a nuestra variable  
    mov ax, Datos  
    mov es, ax  
    mov di, Pantalla  
  
    ; copiamos el origen  
    ; en el destino  
    mov ex, 2000  
    cid  
    rep movsw
```

```

; restaurar registros
pop es
pop ds
popa

; y devolvemos el control
ret

```

De manera análoga podría modificar RecuperaPantalla, eliminando por lo tanto TransfiereDatos, y modificar la lógica de dibujo del recuadro para que use stos, cambiando el procedimiento Dibuj aRecuadro. El resultado será un código más breve y eficiente que el que tiene ahora mismo.

Búsqueda de un dato

Una de las tareas que más tiempo consumen, a pesar de que se codifiquen en ensamblador, es la búsqueda de un dato en una secuencia, por ejemplo un carácter en una cadena de caracteres. En principio podríamos pensar en ir recuperando byte a byte o palabra a palabra, mediante la instrucción lods, y efectuando una comparación con el dato que se busca, usando la instrucción emp, debiendo construir, necesariamente, un bucle que se repita hasta encontrar el dato o llegar al final de la cadena.

La alternativa consiste en usar la instrucción seas que, como las anteriores, cuenta con dos versiones: scasb y scasw, en los procesadores de 16 bits, a las que se suman scasd para 32 bits y scasq para 64 bits.

Lo que hace esta instrucción es comparar el byte o palabra a la que apunta ES : DI con el contenido de AL, AX, EAX o RAX, según el caso. El resultado es la activación de los bits correspondientes en el registro de indicadores.

Aunque puede utilizarse de forma independiente, en el interior de un bucle, esta instrucción es especialmente interesante cuando se usa conjuntamente con los prefijos de repetición repe y repne. El primero repite la operación de comparación mientras el contenido de AL/AX coincide con el dato apuntado por ES : DI, mientras que el segundo hace lo contrario.

Una sentencia repe seas, por tanto, nos permitiría buscar es una secuencia de datos que se repiten el primer dato que es diferente, mientras que con repne se as buscaríamos el primer dato que coincide con el facilitado en AL/AX. Cuando se da esa condición, la búsqueda se interrumpe a pesar de que CX no haya llegado a ser cero, quedando la pareja ES: DI apuntando al dato que se buscaba. Si CX es 0, significará que no se ha encontrado una coincidencia.

Observe cómo el programa siguiente utiliza la sentencia repne scasw para ir localizando las letras A en blanco sobre fondo negro que aparecen en la pantalla. Cada vez que repne interrumpe la búsqueda comprobamos si ha sido porque ha llegado al final, caso en el que CX contendrá el valor 0, o bien, de lo contrario, porque ha encontrado una coincidencia. En el primer caso terminamos, convirtiendo el número en una cadena para mostrarlo por la consola. En el segundo incrementamos DX y volvemos a la sentencia repne scasw.

```

        segment Datos
; Mensaje informativo
Mensaje db '      letras "A"$'

        segment Pila stack
        resw 512

FinPila:

; Segmento de código
segment Código
..start:

        ; Preparamos la pila
        mov ax, Pila
        mov ss, ax
        mov sp, FinPila

        ; ES:DI apuntan al inicio
        ; de la memoria de pantalla
        mov ax, Qb8QQh
        mov es, ax
        xor di, di

        ; DX nos servirá para ir
        ; contando
        xor dx, dx

        ; tendremos que recorrer
        ; 2000 posiciones de ? bytes
        mov ex, ' 2000

        ; incrementando ei
        ; valor de DI
cid

        ; Buscamos letras 'A'
        ; en blanco sobre negro
        mov ax, 07 41h

Rucie:
repne scasw ; buscamos

        ; si no hemos encontrado
        or ex, ex
        jz Fin ; terminamos

        ; incrementamos DX
        inc dx
        jmp Bucle ; y seguimos

Fin:
        ; Preparamos DS y ES
        ; apuntando al segmento
        ; de datos
        mov ax, Datos
        mov es, ax
        mov ds, ax

```

```

; DI indica el punto
; donde se insertará
; el último dígito
iiow di, Mensaje+3

; debemos facilitar el número
; a convertir en AL, y lo tenemos
; en DX
mov ax, dx

; convertimos
cali EnteroCadena

; mostramos el mensaje por pantalla
mov dx, Mensaje
mov ah, 9
int 21h

; Salimos al sistema
mov ah, 4ch
int 21h

```

El procedimiento EnteroCadena es el mismo que habíamos creado previamente en otro ejemplo, sin ningún campo, por lo que no se muestra como parte del código.

```

C:\>letrasa
0 letras "A"
C:\>

C:\>letrasa
01 letras "A"
C:\>

C:\>letrasa AAAAAAAAAAAAAAAAAAAAAA
33 letras "A"
C:\>

```

Figura 16.2. El programa cuenta el número de letras A que hay en pantalla.

Comparación de cadenas

La última instrucción que vamos a conocer, relativa a la manipulación de secuencias de bytes o palabras, es emps que, como puede suponer, es análoga a emp, con la diferencia de que no compara un solo byte o palabra sino una secuencia completa de ellos.

Existen las variantes crapsbycmpsw, según nos interese comparar byte a byte o palabra a palabra, así como las versiones cmpsd y cmpsq, para comparar dobles y cuádruples palabras. La primera cadena estará en la dirección indicada en DS : SI y la segunda en ES : DI. El bit de dirección del registro de indicadores determinará si esas direcciones se incrementan o reducen tras cada comparación.

Aunque, como movs, podemos usar esta instrucción de manera aislada, adquiere su mayor sentido cuando se emplea conjuntamente con los prefijos de repetición repe y repne que ha conocido en el punto previo. De esta manera se puede interrumpir cuando se encuentre la primera diferencia, o coincidencia, entre ambas cadenas.

Como es habitual, el registro CX determinará el número de ciclos que se repetirá la operación. Al comparar dos cadenas siempre debe indicarse la longitud de la más corta, en caso de que su tamaño no coincida. Si va a comparar dos cadenas de caracteres, o de números de 8 bits, use siempre cmpsb. En caso de que vaya a comparar números de 16 bits, utilice cmpsw.

Veamos cómo usar esta última instrucción mediante un nuevo ejemplo que, en este caso, comparará dos cadenas de caracteres y nos indicará por consola si son o no iguales y cuántos caracteres se han comparado finalmente. El código del programa completo es el que se muestra a continuación:

```
segment Datos
; Cadenas a comparar
Cadena1 db 'Comparación de cadenas',0

; dependiendo de que esté o no
; definido el símbolo IGUALES
Ufdef IGUALES
Cadena2 db 'Comparación de cadenas',0
%else
Cadena2 db 'Comparación de dos cadenas', 0
*endif

; Mensajes a mostrar
General db '    caracteres comparados.$'
SonIguales db 'Las cadenas son iguales. $'
NoSonIguales db 'Las cadenas no son iguales. $'

segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
..start:

; Preparamos la pila
mov ax, Pila
mov ss, ax
mov sp, E'inPila

; DS y ES apunta al segmento de datos
mov ax, Datos
```

```

mov ds, ax
mov es, ax

; primera cadena
mov di, Cadenal
; calculamos la longitud
cali Longitud

; la guardamos en DX
mov dx, ex

; segunda cadena
mov di, Cadena2
; calculamos la longitud
cali Longitud

; vemos si la longitud de
; la primera cadena es
; menor que la de la segunda
emp dx, ex

; de no ser asi saltamos
; al proceso de comparación
ja Compara

; en caso contrario
; intercambiamos DX y CX
; para quedarnos en CX con
; la longitud más corta
xchg ex, dx

```

Compara:

```

push ex ; guardamos la longitud

; apuntamos las cadenas a comparar
mov si, Cadenal
mov di, Cadena2

cid ; incrementar

repe empsb ; comparamos

; ¿se ha comparado entera?
or ex, ex
jz Iguales ; saltar

; en caso contrario apuntar
; DX al mensaje de desigualdad
mov dx, NoSonIguales

; y saltar al final del proceso
jmp MuestraMensaje

```

Iguales:

```

; las cadenas son iguales
mov dx, SonIguales

```

```
MuestraMensaje:
    ; convertimos AL a cadena
    mov di,General+2
    pop ax ; recuperamos longitud
    cali EnteroCadena

    mov ah, 9 ; mostramos el

    ; y la indicación general
    mov dx, General
    mov ah, 9
    int 21h

Fin:
    ; Salimos al sistema
    mov ah, 4ch
    int 21h

; tiste procedimiento toma en
; EStDI una cadena terminada
; con nulo, al estilo de C,
; y facilita la longitud en CX

Longitud:
    push ax ; guardamos AX

    ; asumimos longitud
    ; máxima
    mov cx,0ffffh

    ; buscamos el fin de la
    ; cadena
    xor al, al

    cid

    ; buscar
    repne scasb

    ; calcular la longitud
    mov ax, 0ffffh
    sub ax, ex

    ; ponerla en CX
    mov ex, ax

    pop ax ; restauramos ax

    ret ; y volvemos

; Este procedimiento convierte
; el valor de AL en una cadena
; de tres caracteres
```

```
; Entrada: AL = número a convertir
;           ES:DI = destino cadena
```

EnteroCadena:

```
; establecemos valor inicial
mov byte [di], '0'
```

```
; comprobamos si AL es cero
or al, al
; de ser asi, no hay más
; que hacer
jz FinConversion
```

```
push bx ; guardamos bx
; y establecemos el divisor
mov bl, 10
```

BucleO:

```
; vamos dividiendo por 10
div bl
```

```
; quedándonos con el resto
; que convertimos a ASCII
add ah, '0'
; y guardamos
mov [di], ah
; retrocediendo al dígito anterior
dec di
```

```
; eliminamos el contenido
; de AH paia quedarnos con
; el cociente de AL
xor ah, ah
```

```
; si el cociente es mayor que 9
cmp al, 9
; seguimos dividiendo
ja BucleO
```

```
; en caso contrario guardamos
add al, '0'
mov [di], al
```

```
pop bx ; recuperamos BX
```

FinConversion:

```
ret
```

Como verá, existen algunos elementos nuevos que merecen ser tratados con algo más de detalle. Para empezar, al inicio del código aparecen las directivas %ifdef, %else y %endif. Éstas son propias de NASM, aunque otros ensambladores cuentan con construcciones similares, y facilitan el ensamblado condicional.

La sentencia que hay tras el %ifdef, que en este caso es la definición de un campo asociado a una cadena de caracteres, sólo será ensamblada en caso de que esté definido

el símbolo IGUALES. Éste puede definirse con la directiva %def ine, en el propio código, o bien en el momento de ensamblar, mediante la opción -d de NASM. Esta última posibilidad es interesante ya que permite obtener un ejecutable distinto sin necesidad de tener que manipular el código fuente.

En resumen, si está definido el símbolo IGUALES los campos Cadena1 y Cadena2 serán dos cadenas de caracteres exactamente iguales, mientras que en caso contrario serán diferentes incluso en longitud. Observe que hemos puesto un byte nulo, con el valor 0, al final de cada cadena. Éste es un mecanismo habitual para indicar el final de una cadena de caracteres en ciertos lenguajes, como es el caso de C y C++.

A continuación, tras configurar adecuadamente los registros de segmento, usamos el procedimiento Longitud para obtener el número de caracteres de cada cadena. Esta rutina recibe como parámetro, en ES : DI, la dirección de una cadena cualquiera y devuelve en CX su longitud. Se asume que la cadena tendrá como máximo 65535 caracteres, buscándose, mediante la instrucción scasb, la primera aparición del carácter 0, indicador del final. Cuando se encuentre, CX contendrá 65535 menos el número de caracteres recorridos, por lo que una simple diferencia nos dará la longitud.

Una vez tenemos la longitud de las dos cadenas, una en DX y otra en CX, comparamos para ver cuál es la menor, ya que ésa será la que utilicemos para efectuar la comparación. Observe que, en caso necesario, se intercambia el valor de los registros citados mediante la instrucción xchg que, hasta ahora, no habíamos usado nunca.

Por último, se comparan las dos cadenas y, dependiendo de que sean iguales o no, se muestra un mensaje u otro. En cualquier caso, se convierte el número de caracteres comparados en una cadena y se muestra en la consola a título informativo. En la figura 16.3 se puede ver el resultado del programa sin definir el símbolo IGUALES y tras definirlo. Fíjese en que el mensaje es distinto.

```
D:\Ejemplos\16>busqueda
Las cadenas no son iguales. 23 caracteres comparados.
D:\Ejemplos\16>
D:\Ejemplos\16>
D:\Ejemplos\16>nasm -f obj -dIGUALES=True busqueda.asm
D:\Ejemplos\16>
D:\Ejemplos\16>alink busca...
ALINK v1.6 (C) Copyright 1998-9 Anthony A. J. Williams.
All Rights Reserved

Loading file busca...
matched Externs
matched ComDefs

D:\Ejemplos\16>
D:\Ejemplos\16>busqueda
Las cadenas son iguales. 23 caracteres comparados.
D:\Ejemplos\16>
D:\Ejemplos\16>
```

Figura 16.3. Probamos el programa sin definir el símbolo y tras haberlo definido.

Resumen

Como hemos podido ver en este capítulo, los procesadores x86 disponen de una serie de instrucciones específicas para operar sobre secuencias de bytes y palabras, facilitando la recuperación, almacenamiento, transferencia, búsqueda y comparación, de forma más eficiente, en cuanto a longitud de código y velocidad, que los bucles compuestos de instrucciones `mov` y `loop`.

Al finalizar este capítulo prácticamente conocemos todas las instrucciones comunes a todos los procesadores de la familia x86 a excepción de apenas media docena. Con lo que hemos aprendido podemos transferir datos desde el microprocesador hacia la memoria y viceversa, efectuar operaciones aritméticas y lógicas, evaluar condicionales, saltar de un punto a otro del programa, codificar bucles y operar sobre secuencias de datos, así como enviar y recibir datos de otros dispositivos comunicándonos a través de puertos de E/S.

Desde el próximo capítulo nos centraremos en el estudio de los recursos sobre los que podemos aplicar todas esas instrucciones ya que, hasta el momento, nos hemos centrado prácticamente en la manipulación del contenido de la pantalla en todos los ejemplos desarrollados.

17

La BIOS

Todos los programas que hemos desarrollado a modo de ejercicio, en los capítulos previos, han hecho uso principalmente de lo que podríamos denominar la capa de más bajo nivel a la hora de programar en ensamblador: las instrucciones que operan sobre los registros y mueven datos a y desde la memoria. Sobre esa capa existen otras que, al programar en ensamblador, pueden ahorrarnos mucho trabajo. Una de ellas es lo que se conoce normalmente como *la BIOS* (*Basic Input Output System*).

Sobre la BIOS se van superponiendo capas con servicios de más alto nivel, como los ofrecidos por DOS, Windows o Linux. Éstos, en la mayoría de las ocasiones, recurren a la BIOS o bien directamente al hardware para efectuar las operaciones que necesitan. Dependiendo de los servicios de que se trate, accederemos a ellos mediante interrupciones o bien con llamadas al estilo de las utilizadas para invocar a los procedimientos o rutinas que vimos en los capítulos precedentes.

Este capítulo le introducirá al uso de los servicios de la BIOS, unos servicios que siempre están ahí, a pesar de que no se haya iniciado sistema operativo alguno, puesto que residen en algunos de los circuitos integrados que hay alojados en la placa base del ordenador. Posteriormente, en otros capítulos, conocerá los servicios de más alto nivel de DOS, Windows y Linux.

¿Qué es la BIOS?

Un ordenador tipo PC está compuesto, básicamente, de un microprocesador, un banco de memoria y una serie de dispositivos externos, principalmente circuitos integrados, con los que se comunica mediante *buses* de datos y direcciones o puertos de entrada/salida.

Para desarrollar aplicaciones, usando el lenguaje que entiende el microprocesador, hay que apoyarse principalmente en las instrucciones que mueven datos a y desde la memoria y a y desde los dispositivos, con instrucciones como mov, in y out.

Cualquier operación, por básica que fuese, requeriría un volumen de trabajo considerable. Imagine lo que supone acceder a elementos como el disco duro, el adaptador de gráficos, el ratón o un adaptador de red usando exclusivamente las instrucciones in y out, enviando y recogiendo datos en bloques de 8 ó 16 bits. Obviamente, podríamos codificar rutinas genéricas para efectuar muchas operaciones comunes, como el cambio de modo de vídeo, la lectura de un sector de un disco o enviar un carácter a la impresora. Por suerte esto no es necesario, porque todas estas rutinas se encuentran ya definidas y conforman la BIOS.

La BIOS se aloja en una memoria tipo EPROM, un circuito integrado, que acompaña al resto de los elementos que existen en la placa principal del ordenador. Básicamente se trata de un programa que toma el control cuando el ordenador se pone en marcha, llevando a cabo todo el proceso de comprobación y detección de memoria y dispositivos, configurando un área de memoria con datos de control de los dispositivos y, finalmente, transfiriendo el control al programa de arranque que corresponda, normalmente alojado en un disco duro, disquete o CD-ROM.

Durante ese proceso de inicio, la BIOS configura en la memoria una serie de vectores de interrupción.

Éstos facilitan el acceso a los servicios que ofrece la propia BIOS, servicios que evitan que tengamos que comunicarnos directamente con cada dispositivo y, por tanto, tengamos que conocer todos los detalles sobre su funcionamiento. Al acceder a ellos a través de la BIOS nos aseguramos que nuestros programas sigan trabajando correctamente incluso aunque el hardware físico cambie, por ejemplo al sustituir un adaptador de vídeo por otro más moderno, ya que existe un estándar en cuanto a los servicios de la BIOS y la manera de usarlos.

Con el tiempo, algunos sistemas operativos han asumido competencias que antes eran exclusivas de la BIOS, como las funciones de E/S a disco o gestión del adaptador de vídeo. Es lo que ocurre, por ejemplo, con algunas versiones de Windows. En cualquier caso, el uso de la BIOS que va a describirse en este capítulo hace referencia siempre a aplicaciones que operan en modo real, nunca en modo protegido.

La gestión de interrupciones en modo protegido es diferente al empleado en modo real, por lo que desde sistemas que usan el modo protegido, todos los de 32 bits que funcionan sobre procesadores Pentium, Core o compatibles, no es posible el acceso directo a la BIOS. Existen, no obstante, alternativas a los servicios de la BIOS y caminos alternativos para llegar a ella.

Desde DOS, como sistema operativo de 16 bits que opera en modo real, no tenemos ninguna limitación en cuanto al acceso a los servicios de la BIOS. Es, por tanto, el sistema operativo ideal para comprobar su funcionamiento.

El mecanismo de interrupciones

La mayor parte de los servicios de la BIOS son accesibles a través de interrupciones, un mecanismo que no es exclusivo de la arquitectura de los PC, usándose ya en microprocesadores como el 8085 o el Z80. En prácticamente todos los programas de ejemplo que ha escrito hasta ahora ha usado la instrucción `int`, mediante la cual es posible ejecutar un servicio accesible mediante el sistema de interrupciones. Concretamente se ha usado para ejecutar un servicio de la interrupción `2 lh`, perteneciente al DOS, para devolver al control al sistema. En un capítulo previo aprendió a estructurar su código escribiendo procedimientos a los cuales invocaba mediante la instrucción `call` y de los que volvía con la instrucción `ret`. Para ello, necesita conocer la dirección exacta de la rutina a invocar, tanto si pertenece a su programa como si es un procedimiento externo. Esta dirección será sólo un desplazamiento, relativo al segmento de código actual, o bien estará compuesta por una dirección de segmento y un desplazamiento. Esta segunda forma es la habitual cuando va a invocarse a rutinas externas a un programa, ya estén ofrecidas por el sistema, un controlador de dispositivo o cualquier otro tipo de aplicación.

Teóricamente, para invocar a un servicio de la BIOS deberíamos saber el segmento y desplazamiento donde se encuentra su punto de entrada, es decir, la primera sentencia de ese servicio. Hay que tener en cuenta, sin embargo, que existen múltiples fabricantes de BIOS y que la implementación de esos servicios, como es lógico, difiere de unos casos a otros puesto que el hardware es distinto. Las direcciones físicas en las que se encuentra el código, consecuentemente, también podrían diferir.

La solución es la tabla de interrupciones, compuesta de 256 entradas de cuatro bytes cada una. Esta tabla se aloja en la dirección `0:0`, es decir, el primer kilobyte de memoria existente en el sistema en modo real. Cada uno de los vectores de interrupción está compuesto de una dirección de segmento y un desplazamiento, cuatro bytes en total, que apuntan a la dirección en la que se encuentra el punto de entrada a los servicios de dicha interrupción. De esta manera, cuando el procesador encuentra una sentencia `int n` en el código da los pasos siguientes:

- Guarda en la pila el contenido actual del registro de indicadores, el registro de segmento `CS` y el registro `IP`, facilitando así el regreso al código que estaba ejecutándose en ese momento.
- Se toma `n`, el número de interrupción indicado, y se multiplica por 4, encontrando así la dirección que corresponde en la tabla de vectores de interrupción (véase la figura 17.1). Si `n` es `2 lh`, por ejemplo, se multiplica 33 (valor equivalente en decimal) por 4 y se obtiene 132, sabiéndose que en las direcciones 132-135 del segmento 0 está el segmento y desplazamiento correspondientes a esta interrupción.
- Los valores recuperados de la tabla de interrupciones se introducen en `CS` e `IP`, pasando así a ejecutar el servicio solicitado.
- Cuando se termina de ejecutar el servicio, la instrucción `iret` se encarga de recuperar de la pila los registros `IP`, `CS` y de indicadores, devolviendo el control a la sentencia siguiente a la que invocó a la interrupción.

0:1020	Seg:Desp int 255	0:1023
0:1016	Seg:Desp int 254	0:1019
0:1012	Seg:Desp int 253	0:1015
...		...
...		...
...		...
...		...
...		...
...		...
...		...
...		...
...		...
...		...
...		...
...		...
...		...
0:16	Seg:Desp int 4	0:19
0:12	Seg:Desp int 3	0:15
0:8	Seg:Desp int 2	0:11
0:4	Seg:Desp int 1	0:7
0:0	Seg:Desp int 0	0:3

Figura 17.1. Estructura de la tabla de vectores de interrupción.

Nota

La tabla de vectores de interrupción está alojada, como se ha dicho, en el primer kilobyte de memoria, un área asignada a memoria RAM y, por lo tanto, esos vectores pueden ser modificados. Esto es lo que hace posible que los servicios de la BIOS, que se encuentran en ROM, establezcan los puntos de entrada a sus servicios, alterando el valor por defecto que almacenan los vectores. Incluso es posible, como verá en su momento, alterar esos vectores desde aplicaciones propias.

Cuando utilice la instrucción `int` para invocar a un servicio, por lo tanto, hágase a la idea de que lo que está haciendo es un `call` a una dirección que no conoce de antemano, una dirección que se encuentra alojada en una porción de memoria bien conocida que facilita una cierta independencia del hardware.

El área de parámetros de la BIOS

Como se decía anteriormente, la BIOS es un programa que se encarga de la puesta en marcha o iniciación del ordenador, aparte de ofrecer servicios de bajo nivel tanto al sistema operativo como a las aplicaciones. Este programa, alojado en memoria que sólo puede ser leída, necesita un área de lectura/escritura para almacenar parámetros que

no son constantes, como el tamaño de la memoria instalada en el sistema, el número de unidades de disco, el estado actual del teclado, etc. Para todos estos datos se reserva un área de memoria RAM, conocida como área de parámetros de la BIOS, que está justo detrás de la tabla de vectores, siendo su tamaño variable según la BIOS específica de que se trate.

La dirección de inicio de esta área de parámetros es, por tanto, 0:1024 o, si lo prefiere, 40h:0, que es equivalente a la anterior. Recuerde que el segmento se expresa en párrafos (16 bytes), que 1024 dividido entre 16 es 64 y que 64 en hexadecimal es 4 Oh. Asumiendo que utilizamos 4 Oh como segmento, la dirección de cada variable existente en el área de parámetros de la BIOS tendrá como referencia el desplazamiento 0.

Asumiendo que utilizamos 4 Oh como segmento, en la tabla 17.1 se enumera la dirección, tamaño y contenido de algunos elementos del área de parámetros de la BIOS. Tenga en cuenta que esta configuración se remonta a los tiempos de la aparición del PC por lo que, en la actualidad, puede no contener una información completamente exacta.

Tabla 17.1. Variables del área de parámetros de la BIOS.

Dirección	Tamaño	Contenido
0000h	8 bytes	Direcciones base de comunicación con los puertos serie, del COM1 al COM4.
0008h	8 bytes	Direcciones base de comunicación con los puertos paralelo, del LPT1 al LPT4.
0010h	1 palabra	Sus bits reflejan parte de la configuración del equipo, indicando el número de puertos serie, número de impresoras, unidades de disquete, modo inicial de vídeo, etc.
0013h	1 palabra	Número de bloques de 1 kbyte de memoria que hay instalados en el sistema. En los sistemas actuales siempre se indica 640 kilobytes, pero hay que tener en cuenta que en los años 80 los PC tenían como máximo esa memoria, partiendo desde 64 kilobytes.
0017h	1 byte	Cada bit, desde el 0 al 7, indica el estado de las teclas de mayúsculas derecha, mayúsculas izquierda, control, Alt, bloqueo de desplazamiento, bloqueo de números, bloqueo de mayúsculas e inserción.
0018h	1 byte	Extensión de la información aportada por el byte anterior, con el estado de las teclas control y Alt del lado izquierdo del teclado (que no existían originalmente), la tecla de pausa, etc.
001Ah	1 palabra	Dirección de la cabeza del buffer circular de teclado.
001Ch	1 palabra	Dirección de la cola del buffer circular de teclado.
001Eh	16 palabras	Buffer de teclado. Cada palabra contiene el código ASCII y código de identificación de cada tecla pulsada.
0049h	1 byte	Modo de vídeo actual.

Dirección	Tamaño	Contenido
004Ah	1 palabra	Número de columnas del modo de video actual.
004Ch	1 palabra	Contiene el número de bytes que necesita una página de pantalla en el modo de video actual, por ejemplo 4000 bytes en el modo de texto de 80x25 en color.
004Eh	1 palabra	Dirección relativa de la página actualmente visible dentro de la memoria de pantalla.
0050h	8 palabras	Cada palabra contiene la columna y linea donde se encuentra el cursor en cada página de video, asumiendo que como máximo existirán 8 páginas.
0062h	1 byte	Número de la página visible actualmente en pantalla.
006Ch	1 doble palabra	Número de ticks o pulsos de reloj generados desde que se conectó la máquina. Se producen 18,2 pulsos cada segundo.
0070h	1 byte	Indica si la doble palabra que contiene el número de pulsos ha sido desbordada.
0075h	1 byte	Número de discos duros existentes en el sistema.
0097h	1 byte	Sus bits indican el estado de los leds luminosos del teclado, como el fija mayúsculas o números.

Gran parte de la información disponible en la BIOS puede obtenerse por otros medios, principalmente a través de interrupciones. A pesar de que algunos de los datos indicados en la tabla son de lectura y escritura, no es recomendable que modifique un parámetro si no sabe bien de antemano qué está haciendo y el efecto que tendrá sobre el sistema.

Acceso a variables de la BIOS

Para acceder al área de parámetros de la BIOS utilizaremos la misma técnica que, por ejemplo, hemos empleado repetidamente para escribir o bien leer datos en la pantalla. Introduciremos el segmento, en este caso 4 Oh, en un registro de segmento, usando a continuación la dirección de la variable concreta que vamos a leer como desplazamiento. Existen alternativas, como la definición de segmentos en posiciones absolutas, es posible en MASM con el modificador at de la directiva segment y en NASM mediante la directiva absolute. Otra posibilidad consiste en almacenar en el área de datos punteros a las variables de la BIOS que interesen. Cada puntero se compondría de la dirección completa, segmento y desplazamiento, que después sería recuperado y almacenado en los registros correspondientes.

Observe el código del programa siguiente que, básicamente, se limita a leer el contenido de algunas de las variables del área de parámetros de la BIOS, concretamente la cantidad de memoria, el número de columnas en el modo de vídeo actual y el número de pulsos de reloj.

```

segment Pila stack
    resw 512
FinPila:

    segment Datos
        ; Direcciones de las variables
Memoria dw 13h,40h
Columnas dw 4Ah,40h
Pulsos dw 6Ch,40h

        ; Segmento de código
    segment Código
..start:

        ; Preparamos la pila
    mov ax, Pila
    mov ES, an
    mov sp, FinPila

        ; ES apunta al segmento de datos
    mov ax, Datos
    mov es, ax

        ; Cargamos en DS:SI el puntero
    lds si, [es:Memoria]
        ; y leemos la palabra
    lodsw

        ; Repetimos la operación
    lds si, [es:Columnas]
    lodsw

        ; Vamos a leer una doble palabra
    lds si, [es:Pulsos]
    lodsw
        ; dejándola en AX:DX
    mov dx, ax
    lodsw

        ; Salimos al sistema
    mov ah, 4ch
    int 21h

```

Fíjese, para empezar, en la forma en que se han definido los punteros a las distintas variables. Cada campo está compuesto de dos palabras: desplazamiento y segmento. Están dispuestos en orden inverso, el desplazamiento delante del segmento, porque las dobles palabras, que es lo que estamos componiendo realmente, se almacenan de esta forma en memoria, con la palabra más significativa, que es el segmento, detrás de la menos significativa.

Nuestra intención es ir leyendo los datos mediante la instrucción `lodsw` y, por lo tanto, los registros DS : SI deberán contener la dirección de la que va a leerse. En lugar de usar sendas instrucciones `mov`, para llevar los punteros a dichos registros, utilizamos

la instrucción `lds`. Ésta obtiene de la dirección indicada como parámetro un puntero compuesto de segmento y desplazamiento y la introduce, precisamente, en los registros DS:SI.

Existe una instrucción complementaria, `les`, que hace lo mismo que `lds` pero almacenando la dirección en la pareja de registros ES: DI.

Los datos simplemente se recuperan en AX, o en AX: DX en el caso del número de pulsos de reloj, sin hacer más con ellos. Deberá ejecutar el programa desde DEBUG o GRDB, por tanto, para poder leer los resultados.

```

Símbolo del sistema - debug param.exe
18B5:0018 AD          LODSW
-p
AX=0050  BX=0000  CX=0286  DX=0000  SP=0400  BP=0000  SI=004C  DI=0000
DS=0040  ES=18B4  SS=1874  CS=18B5  IP=0019  NV UP EI PL NZ NA PO NC
18B5:0019 26          ES:
18B5:001A C5360800    LDS      SI,[0008]           ES:0008=006C
-p
AX=0050  BX=0000  CX=0286  DX=0000  SP=0400  BP=0000  SI=006C  DI=0000
DS=0040  ES=18B4  SS=1874  CS=18B5  IP=001E  NV UP EI PL NZ NA PO NC
18B5:001E AD          LODSW
-p
AX=E67B  BX=0000  CX=0286  DX=0000  SP=0400  BP=0000  SI=006E  DI=0000
DS=0040  ES=18B4  SS=1874  CS=18B5  IP=001F  NV UP EI PL NZ NA PO NC
18B5:001F 89C2        MOV     DX,AX
-p
AX=E67B  BX=0000  CX=0286  DX=E67B  SP=0400  BP=0000  SI=006E  DI=0000
DS=0040  ES=18B4  SS=1874  CS=18B5  IP=0021  NV UP EI PL NZ NA PO NC
18B5:0021 AD          LODSW
-p
AX=0013  BX=0000  CX=0286  DX=E67B  SP=0400  BP=0000  SI=0070  DI=0000
DS=0040  ES=18B4  SS=1874  CS=18B5  IP=0022  NV UP EI PL NZ NA PO NC
18B5:0022 B44C        MOV     AH,4C
-
```

Figura 17.2. Los registros DX:AX contienen el valor de 32 bits que indica el número de pulsos.

Servicios de la BIOS

A pesar de que el área de parámetros de la BIOS puede resultar de cierto interés, lo más interesante, sin duda alguna, son los servicios que nos ofrece, servicios mediante los que podremos manipular la pantalla, acceder a los discos, enviar datos a la impresora, leer el teclado o comunicarnos mediante los puerto serie.

Hay cientos de funciones disponibles, estando agrupadas por categorías. De esta forma, disponemos de una interrupción que da paso a todos los servicios relacionados con

el acceso a la pantalla, otra para los servicios de disco, otra para los de impresora, etc. La selección del servicio, por regla general, se efectúa almacenando un cierto valor en el registro AH o AX, según los casos.

Cada servicio concreto precisará parámetros o los devolverá, usando para ello determinados registros del procesador, como hacíamos en nuestras propias rutinas de ejemplos en capítulos previos. En los puntos siguientes se describen de manera breve las interrupciones más útiles con servicios de la BIOS, algunas de las cuales conocerá con mayor detalle en los capítulos siguientes.

Acceso al adaptador de vídeo

Uno de los elementos de comunicación principales entre ordenador y usuarios es la pantalla, dispositivo de salida por excelencia que complementa las acciones de entrada de teclado y ratón. La imagen que vemos en pantalla se compone en memoria y transforma en una matriz de puntos que se actualiza constantemente por arte y magia del adaptador de vídeo, elemento que suele encontrarse en el sistema en un módulo o tarjeta de tipo PCI o, más recientemente, AGP y PCI Express. Es ese módulo el que aporta los servicios necesarios para establecer el modo de visualización, escribir caracteres o puntos, etc.

La interrupción para acceder a esos servicios es la 1 Oh, debiendo facilitarse en el registro AH o AX, según los casos, el número de servicio que se desea. Hay que tener en cuenta que existen unos servicios básicos, de aplicación general, mediante los cuales podemos establecer el modo de visualización, leer y escribir caracteres y puntos, determinar la apariencia y posición del cursor o desplazar una parte del contenido de la pantalla. Todos éstos tienen números de servicio de 8 bits y se seleccionan mediante el registro AH.

Por otra parte están los servicios propios de adaptadores de vídeo que han ido apareciendo con posterioridad al estándar de las BIOS de PC, como los servicios para VGA, VESA o los aún más específicos de cada adaptador concreto. Estos servicios también son accesibles mediante la interrupción 1Oh, empleando entonces el registro AX para su selección.

En el próximo capítulo conocerá algunos de los servicios de la interrupción 1 Oh. Por ahora, y sólo a modo de ejemplo, introduzca el código siguiente en un archivo, ensámbelo y ejecútelo. Verá que la pantalla se borra y el cursor vuelve a situarse en la esquina superior izquierda, como si se hubiese utilizado la orden cls del DOS. En el programa se emplean dos funciones de la interrupción 1 Oh: la función 2, que establece la posición del cursor, y la 9, que introduce un carácter y un atributo tantas veces como se indique a partir de la posición actual.

```
segment Pila stack
    resw 512

; Segmento de código
segment Código
.start:
    ; página 0
    xor bh, bh
```

```

; posición 0,0
xor dx, cx

; posicionamos el cursor
mov ah, 2
int 10h

; establecemos atributo
mov bl, 7
; carácter
mov al, ' '
; y número de caracteres
mov ex, 2 000

; llenamos la pantalla
mov ah, 9
int 10h

; Salimos al sistema
mov ah, 4ch
int 21h

```

Lectura del teclado

La mayoría de las aplicaciones que necesitan solicitar datos al usuario lo hacen a través del teclado, dispositivo que, conjuntamente con el ratón, representan las dos vías principales de entrada de información al sistema.

El teclado es uno de los elementos que menos ha evolucionado, comparativamente hablando con otros dispositivos, de ahí que los servicios disponibles sean prácticamente los mismos que había hace veinticinco años.

En el área de parámetros de la BIOS, según se vio antes, existe una zona de memoria reservada para alojar los caracteres, en realidad sus códigos, a medida que van siendo pulsados. Ese espacio o *buffer* es circular, contando con un puntero a la cabeza actual y otro a la cola. Aunque podríamos acceder directamente a ese *buffer* circular para recoger datos del teclado, generalmente nos resultará mucho más fácil hacerlo mediante la interrupción 16h. Ésta la hemos usado en algún ejemplo de los capítulos previos, como el que llenaba la pantalla de asteriscos para después restaurarla.

Los servicios de esta interrupción, mucho menos numerosos que los del adaptador de vídeo, se seleccionan siempre mediante el registro AH. Dedicaremos a ellos un capítulo posterior, aunque a continuación puede ver un ejemplo que utiliza el servicio 0 para ir recuperando caracteres y mostrando su código en la consola.

```

segment Pila stack
resw 512

segment Datos
CodASCII db '    ., 13, 10, '$'

; Segmento de código
segment Código

```

```

..start:
    ; DS y ES apuntan
    ; al segmento de datos
    mov ax, Datos
    mov es, ax
    mov ds, ax

Bucle:
    ; esperamos la pulsación
    ; de una tecla
    xor ah, ah
    int 16h

    ; si es ESC
    cmp al, 27
    ; terminar
    jz Fin

    ; en caso contrario
    mov di, CodASCII+2
    ; convertir a cadena
    cali EnteroCadena

    ; y mostrarlo en pantalla
    mov dx, CodASCII
    mov ah, 9
    int 21h

    ; eliminar el código
    ; para así poder
    ; introducir otro
    mov di, CodASCII
    mov al, ' '
    mov ex, 3
    cid
    rep srosri

    jmp Bucle ; repetir

Fin:
    ; Salimos al sistema
    mov ah, 4ch
    int 21h

```

Este programa utiliza la rutina EnteroCadena que habíamos creado en un capítulo previo, por lo que tendrá que añadir su código al final del archivo antes de poder ensamblarlo.

Configuración del sistema

Una de las tareas de la BIOS es determinar, durante la puesta en marcha del sistema, qué dispositivos hay presentes. Parte de esa información se aloja en el área de parámetros de la BIOS, pudiendo recuperarse también mediante la interrupción 11h.

Ésta cuenta con un solo servicio, por lo que no es necesario el uso del registro AH para seleccionarlo.

Este servicio, además, es realmente simple, porque se limita a devolver en AX un conjunto de bits con la información siguiente:

- **Bit 0:** Estará a 1 si hay disponible una o más unidades de disquetes, quedando a 0 en caso contrario.
- **Bit 1:** Indica si el sistema cuenta o no con un coprocesador matemático. Recuerde que en los sistemas previos al 80486 éste era opcional y se instalaba como un circuito integrado independiente del procesador principal.
- **Bits 4 y 5:** Contienen el modo de vídeo inicial, pudiendo ser 01 (40 columnas x 25 líneas en color), 10 (80 columnas x 25 líneas en color) o 11 (80 columnas x 25 columnas en monocromo).
- **Bits 6 y 7:** Con ellos podemos saber el número de unidades de disquete que existen en el sistema. Al valor de estos bits hay que sumar 1, de tal forma que puede indicar entre 1 y 4 unidades.
- **Bits 9,10 y 11:** Indican el número de puertos de comunicación RS-232 que hay en el ordenador.
- **Bit 12:** Si está a 1 indica que el sistema cuenta con un adaptador de juegos.
- **Bit 13:** Indica si hay o no un módem interno instalado.
- **Bits 14 y 15:** Número de puertos de impresora que hay en el sistema.

Basándonos en esta información, podemos construir un programa que, como el siguiente, muestre los datos de configuración por pantalla. En este caso nos hemos limitado a indicar el número de unidades de disquetes, número de puertos serie y de puertos de impresora, pero de manera análoga podría facilitarse el resto de la información, simplemente comprobando el estado de los distintos bits.

```

segment Pila stack
    reaw 51?

segment Datos
Disquetes db '    unidades de disquetes',13,10, '$'
PuertosSerie db '    puertos serie',13,10,'$'
PuertosImpresora db'    puertos impresora',13,10, '$'

; Segmento de código
segment Código
..start:

; DS y ES apuntan
; al segmento de datos
mov ax, Datos
mov es, ax
mov ds, ax

```

```
; obtenemos configuración
int 11h

; y la guardamos
push ax

; nos quedamos con los
; bits 6 y 7
and ax, 0C0h

; los desplazamos a AL
shr ax, 6

; incrementamos
inc al

; y convertimos en cadena
mov di, Disquetes+2
cali Enterocadcn

; mostramos en la consola
mov dx, Disquetes
mov ah, 9
int 21h

; recuperamos la
; configuración
pop ax

; y volvemos a guardarla
push ax

; nos quedamos con
; los bits 9, 10 y 11
and ax, 0E00h

; los desplazamos a AL
shr ax, 9

; convertimos a cadena
mov di, PuertosSerie+2
cali Enterocadena

; mostramos en la consola
mov dx, PuertosSerie
mov ah, 9
int 21h

; volvemos a recuperar
; la configuración
pop ax

; nos quedamos con
; los bits 14 y 15
and ax, 0C000h
```

```

; y los desplazamos a AL
shr ax, 14

; convertimos a cadena
mov di, PuertoImpresora+2
cali Enterocadena

; mostramos en la consola
mov dx, PuertoImpresora
mov ah, 9
int 21h

Fin:
; Salimos al sistema
mov ah, 4ch
int 21h

```

En la figura 17.3 puede observar el resultado que genera el programa en un equipo típico actual. La interrupción 11 h no tiene más aplicaciones, así que no volveremos sobre ella.

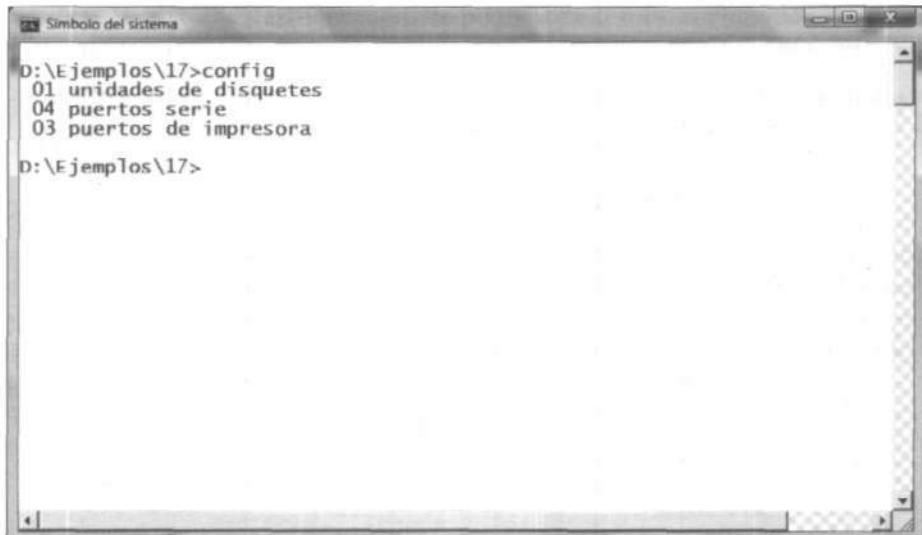


Figura 17.3. El programa nos indica parte de la configuración del equipo.

Nota

Otro método para conocer la configuración del ordenador consiste en leerla directamente de la memoria tipo CMOS que la almacena. Estos parámetros, en parte, se configuran desde la utilidad de edición de parámetros de la BIOS conocida popularmente como *setup*, accesible durante el inicio del ordenador.

Memoria disponible

Los primeros PC, como ya sabe, tenían una capacidad máxima de direccionamiento de un megabyte, cantidad que parecía inalcanzable en aquellos tiempos. En realidad los ordenadores que IBM, fabricante del PC original, puso en el mercado contaban en principio con sólo 64 kilobytes, una capacidad similar a la de muchos otros ordenadores de la época.

Del espacio total de direccionamiento, un megabyte como se ha dicho, parte estaba reservada para memoria ROM conteniendo servicios de diferentes dispositivos, de tal manera que, realmente, la cantidad máxima de memoria RAM que podía instalarse era de 640 kilobytes. Aun así, dicha cantidad se anunciaba más que suficiente para cualquier tipo de aplicación y, de hecho, durante años los ordenadores contaban con cantidades de memoria inferiores, como 256, 384 ó 512 kilobytes.

En este contexto es donde tiene sentido el servicio que ofrece la interrupción 12h, aunque actualmente no sea especialmente útil. Esta interrupción cuenta con un único servicio, no siendo necesaria la selección mediante AH. Tan sólo tenemos que ejecutar la sentencia int 12h para obtener, en el registro AX, el número de kilobytes de memoria existentes en lo que se denomina *área de memoria estándar o memoria baja*.

Los sistemas actuales cuentan normalmente con decenas de megabytes de memoria, de ahí que la interrupción 12h siempre devuelva en AX el mismo valor: 640 kilobytes. Puede comprobarlo sin necesidad siquiera de escribir un programa, simplemente iniciando DEBUG, introduciendo el comando a 100, la sentencia int 12 y, finalmente, ejecutándola directamente. Es lo que se ha hecho en la figura 17.4. Observe el valor de AX, 280h (en hexadecimal) que equivale a 640 en decimal.

The screenshot shows a window titled "Símbolo del sistema - debug". The command line shows "D:\Ejemplos\17>config" followed by configuration options: "01 unidades de disquetes", "04 puertos serie", and "03 puertos de impresora". Then, the command "D:\Ejemplos\17>debug" is entered, followed by "a 100", "int 12", and finally "-p". The output shows the state of registers before and after the interrupt:

Register	Value Before (17E9:0100)	Value After (17E9:0102)
AX	0000	0280
BX	0000	0000
CX	0000	0000
DX	0000	0000
SP	FFEE	FFEE
BP	0000	0000
SI	0000	0000
DI	0000	0000
DS	17E9	17E9
ES	17E9	17E9
SS	17E9	17E9
CS	17E9	17E9
IP	0100	0102
NV		
UP		
EI		
PL		
NZ		
NA		
PO		
NC		

The output also shows the instruction "MOV DH,56" was executed.

Figura 17.4. Ejecutamos desde DEBUG la interrupción 12h.

Las BIOS de los sistemas actuales cuentan con extensiones, nuevos servicios, que facilitan un mapa detallado de la configuración del sistema, incluyendo memoria reservada, no disponible, etc. Otra opción, como se indicaba en el punto previo, consiste en recuperar esta información de la memoria CMOS.

Acceso a unidades de disco

Los servicios de acceso a disco de la BIOS son, como cabría esperar, de bajo nivel, pero en este caso esa circunstancia quizás se note más que en otros servicios. La BIOS no entiende de archivos, directorios ni letras de unidad, tan sólo de unidades físicas y sectores de tamaño fijo cuya localización se describe según una cierta geometría. Como verá en su momento, los distintos sistemas operativos ofrecen servicios de más alto nivel para el almacenamiento y la recuperación de la información.

En este caso la interrupción es la 13h y, como en el caso de la interrupción 10h, la selección del servicio se efectúa mediante el registro AH o AX, según se necesite uno de los básicos o bien un servicio extendido de los añadidos con posterioridad. Mediante estos servicios es posible determinar los parámetros de un disco, leer y escribir sectores, formatearlos, etc.

Tenga en cuenta que al decir *disco* nos referimos tanto a disquetes como a discos duros, ya que la interrupción 13h es común para estos dispositivos a pesar de que, en realidad, se gestionan de forma distinta.

Los servicios originales de la BIOS para acceso a disco cuentan con serias limitaciones que, por ejemplo, le impiden el acceso a la cantidad de información que suelen contener las unidades actuales, de cientos de gigabytes, ya que fueron diseñados cuando los discos duros no eran habituales y los disponibles eran de 10, 20 ó 30 megabytes. Esto ha hecho necesaria la adición de extensiones que faciliten nuevos servicios.

El ejemplo que se muestra a continuación utiliza el servicio 8 de esta interrupción para obtener los parámetros, número de caras, sectores y pistas, de la primera unidad de disquetes.

En la figura 17.5 puede ver la información obtenida.

```

segment Pila stack
    resw 512
FinPila:

segment Datos
Sectores db '    sectores', 13, 10, '$'
Caras db '    caras', 13, 10, '$'
Cilindros db '    cilindros', 13, 10, '$'
MsgError db 'Se produce un error?'

; Segmento de código
segment Código

```

```
..start:

; Configuramos los registros
; de pila
mov ax, Pila
mov 55, ax
mov sp, FinPila

; DS y ES apuntan
; al segmento de datos
mov ax, Datos
mov es, ax
mov ds, ax
; obtener parámetros
mov ah, 8h
; del disquete
xor di, di
int 13h

,- si hay un error
je Error ; lo notificamos

; comenzamos con las caras
; que vienen en DH
mov al, dh
inc al ; incrementamos

; convertimos a cadena
mov di, Caras+2
cali EnteroCadena
; y las mostramos
mov dx, Caras
mov ah, 9
int 21h

; el número de sectores está
; , - en los bits 0 a 5 de CL
mov al, el
and al, 3Fh

; convertimos a cadena
mov di, Sectores+2
cali EnteroCadena
; y mostramos los sectores
mov dx, Sectores
mov ah, 9
int 21h

; el número de mayor cilindro
; está en el registro CH
mov al, ch
inc ai ; incrementamos

; lo convertimos
mov di, Cilindros+2
cali EnteroCadena
```

```

; y mostramos
mov dx, Cilindros
mov ah, 9
int 21h

Fin:
; Salimos al sistema
mov ah, 4ch
int 21h

Error: ; si se produce un error
; mostramos el mensaje
mov dx, MsgError
mov ah, 9
int 21h
jmp Fin ; y salimos

```

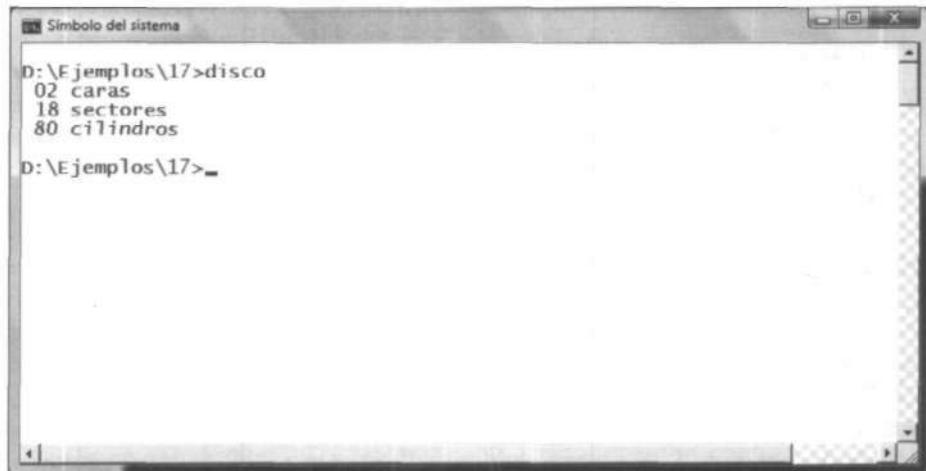


Figura 17.5. El programa muestra la estructura lógica de los disquetes.

Nota

Si el equipo donde esté trabajando no cuenta con una unidad de disquetes, algo cada vez más corriente, al ejecutar este programa obtendrá normalmente un error o información inadecuada, por ejemplo indicando que hay cero sectores, pistas, etc.

Puertos serie y paralelo

Si antes decíamos que el teclado era un dispositivo que había evolucionado muy poco desde su aparición en los primeros PC, y consecuentemente ocurría lo mismo con sus servicios, lo mismo cabría decir con los puertos de comunicación serie y paralelo,

también conocidos como RS-232 y Centronics, respectivamente. Se trata de dispositivos que, desde hace tiempo, vienen integrados en la propia placa base de los sistemas pero que, poco a poco, van dejando paso a soluciones más actuales, como por ejemplo los puertos USB.

Para acceder a una impresora, enviándole información, se utiliza la interrupción 17h. Ésta cuenta sólo con tres servicios: 0 para enviar un byte a la impresora, 1 para inicializarla y 2 para obtener su estado.

Como es habitual, el número de servicio se facilitará en AH, mientras que DX servirá para indicar el número del puerto de impresora a usar. El valor 0 indica el primer puerto y el 1 el segundo. El byte de estado, devuelto en AH, nos permite saber si la impresora está o no ocupada, si tiene papel, etc.

En cuanto a los puertos serie, la interrupción para acceder a ellos es la 14h. Los servicios disponibles son los enumerados en la tabla 17.2.

Aunque puede usar estos servicios si lo desea, la mayoría de sistemas actuales permiten utilizar los puertos serie como si fuesen archivos, facilitando así la tarea de envío y recepción de datos.

Tabla 17.2. Servicios de la interrupción 14h.

Servicio (AH)	Descripción
0	Establece los parámetros de configuración del puerto.
1	Envía un carácter.
2	Recibe un carácter.
3	Lee el estado del puerto.

El número de puerto se facilita en el registro DX, asumiéndose el valor 0 para el primer puerto, 1 para el segundo, 2 para el tercero y 3 para el cuarto, si bien no es habitual que existan más de dos.

El carácter a enviar o recibido estará en el registro AL, mientras que AH facilitará el estado, con el que sabremos si hay algún tipo de error, por ejemplo de expiración de tiempo de espera o paridad.

La configuración de puerto, mediante el servicio 0, se efectúa facilitando en el registro AL un byte que se divide en cuatro partes:

- **Bits 0 y 1:** Determinan si cada dato estará compuesto de siete (10) o bien ocho (11) bits.
- **Bit 2:** Indica si se usará uno (0) o dos (1) bits de parada.
- **Bits 3 y 4:** Establece la paridad, que será ninguna (0 0 ó 10), impar (01) o par (11).
- **Bits 5, 6 y 7:** Fijan la velocidad de comunicación, pudiendo ser 110 (000), 150 (0 01), 300 (010), 600(011), 1200 (100), 2400 (101), 4800 (110) o bien 9600 (111) baudios.

Fecha y hora

La última interrupción de interés de la BIOS, en cuanto a servicios para aplicaciones, es la interrupción 1Ah, mediante la cual es posible leer y escribir tanto el número de pulsos del reloj del sistema como la fecha y hora del reloj de tiempo real. Recuerde que en un capítulo previo leímos la hora actual comunicándonos directamente con el reloj, mediante instrucciones `in` y `out`. La existencia de esta interrupción hace innecesario ese trabajo.

En la tabla 17.3 se resumen los servicios de uso más habitual de esta interrupción. La fecha y la hora se facilitan en registros en formato BCD empaquetado, no en binario, por lo que su interpretación es bastante más sencilla.

Tabla 17.3. Servicios de la interrupción 1Ah.

Servicio (AH)	Descripción
0	Lee el contador de número de pulsos del sistema. Se devuelve en los registros CX:DX.
1	Establece el contador de pulsos del sistema.
2	Recupera la hora del reloj de tiempo real, entregando horas, minutos y segundos en CH, CL y DH.
3	Establece la hora del reloj de tiempo real, debiendo facilitarse los componentes en los mismos registros que el servicio anterior.
4	Recupera la fecha del reloj de tiempo real, entregando el día, mes, año y siglo en los registros DL, DH, CL y CH, respectivamente.
5	Establece la fecha del reloj de tiempo real, debiendo facilitarse los componentes en los mismos registros que el servicio anterior.

Conociendo estos servicios, no es difícil escribir un programa que nos muestre por la consola la fecha y hora actuales del sistema (véase la figura 17.6).

El código podría ser el siguiente:

```

segment Pila stack
    resw 512
FinPila:

segment Datos
; Cadena de datos a mostrar-
Fecha db 'Fecha: '
Dia db ' '
    db '/'
Mes db ' '
    db '/'
Ano db ' '
    db ' - '

```

```
Hora      db  'Hora: '
Horas     db  ' '
          db  ':'
Minutos   db  ' '
          db  ':'
Segundos  db  ' '
          db  '$'

; Segmento de código
segment Código
..start:

; Configuramos los registros
; de pila
mov ax, Pila
mov ssF, ax
mov sp, FinPila

; DS y ES apuntan
; al segmento de datos
mov ax, Datos
mov es, ax
mov ds, ax

; recuperamos la fecha
mov ah, 4
int 1Ah

; extraemos si dia
mov di, Dia
mov al, di
cali Convierte

; el mes
mov di, Mes
mov al, dh
cali Convierte

; y el año
mov d i, Año
mov al, ch
cali Convierte
mov al, el
cali Convierte

; recuperamos la hora
mov ah, 2
int 1Ah

; extraemos las horas
mov di, Horas
mov al, ch
cali Convierte

; los minutos
mov di, Minutos
```

```

    mov al, el
    cali Convierte

    ; y los segundos
    mov di, Segundos
    mov al, dh
    cali Convierte

    ; mostramos la información
    mov dx, Fecha
    mov ah, 9
    int 21h

Fin:
    ; Salimos al sistema
    mov ah, 4ch
    int 21h

; Esta, rutina recibe en AL
; un número RCD empaquetado
; y en DI el destino donde
; debe alojar los dos dígitos
; convertidos.

```

Conviente:

```

    ; guardamos el dato
    push ax

    ; nos quedamos con el
    ; primer digito, que
    ; viene en los bits 4-7
    shr al, 4

    ; convertimos a ASCII
    add al, '0'

    ; y guardamos
    stosb

    ; recuperamos el dato
    pop ax

    ; nos quedamos con el
    ; segundo digito, que
    ; viene en los bits 0-3
    and al,0fh

    ; convertimos a ASCII
    add al, '0'

    ; y guardamos
    stosb

;
    ret ; volver

```

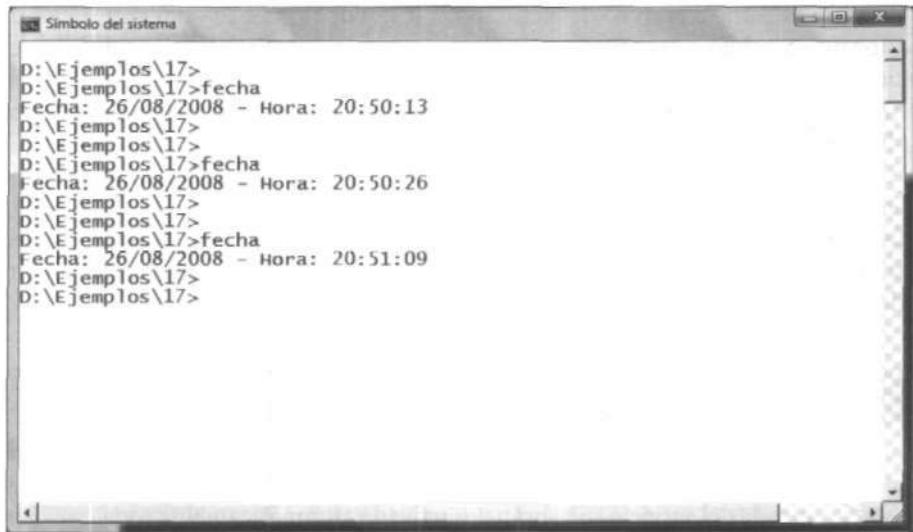


Figura 17.6. El programa muestra la fecha y hora actuales.

interrupciones hardware

Todas las interrupciones citadas hasta ahora entran en la categoría de *interrupciones software*, ya que se ejecutan a demanda de las aplicaciones mediante la instrucción int. Existe otra categoría, conocida como *interrupciones hardware*, que se ejecutan automáticamente cuando el procesador recibe una señal hardware.

Ésta puede estar generada por el teclado, una tarjeta de red o bien un puerto serie, en general cualquier dispositivo externo que requiera en un momento dado la atención del procesador.

El ordenador cuenta con un integrado encargado de gestionar las interrupciones hardware, el 8259 PIC descrito en un capítulo previo, facilitando al procesador un número que le permita saber qué código debe ejecutar. En realidad, para el procesador no hay mucha diferencia entre ejecutar una interrupción hardware o software, salvando su origen o procedencia. El reloj del sistema genera una interrupción 8, el teclado la interrupción 9 y así sucesivamente.

Los servicios que atienden a estas interrupciones hardware son conocidos como ISR (*Interrupt Service Routines*), encargándose de tareas vitales como la interpretación de los datos que envía el teclado y su almacenamiento en el área correspondiente. Por lo demás, no difieren de manera especial respecto a cualquier otro servicio de la BIOS.

Las interrupciones hardware se dividen en dos categorías: enmascarables y no enmascarables. Las primeras también se conocen como IRQ (*Interrupt Request*), mientras que las segundas se denominan NMI (*Non Maskable Interrupt*). En su momento conocí dos instrucciones, el i y sti, que actúan sobre un bit del registro de indicadores, poniéndolo a cero o a uno.

Cuando se pone a cero se desactivan las interrupciones, lo cual significa que el procesador no atenderá a las interrupciones enmascarables. Esto tiene sentido en ciertas situaciones, por ejemplo la modificación de un vector de interrupción, si bien el espacio de tiempo que están las interrupciones desactivadas debe ser siempre muy breve para no alterar el normal funcionamiento del ordenador.

Al ejecutar la orden `cli` se desactivan todas las interrupciones enmascarables, pero no las interrupciones NMI o no enmascarables.

Éstas no pueden desactivarse porque están asociadas a escenarios excepcionales, como un error de paridad de la memoria o situación similar, que comprometen la estabilidad del sistema.

La instrucción `int`, que hemos empleado repetidamente para invocar los servicios de diversas interrupciones, lleva implícita la ejecución de la instrucción `cli`, de tal forma que a la entrada del servicio se habrán desactivado automáticamente todas las interrupciones enmascarables. Por ello es habitual que al inicio del código de muchos servicios exista una instrucción `sti`, reactivando así las interrupciones.

Excepciones

Aún podemos encontrar una tercera categoría de interrupción: la formada por las excepciones. Una excepción es una señal hardware pero, a diferencia de las interrupciones hardware, se genera en el interior del propio procesador, no en un dispositivo externo. Señales de este tipo son las que se producen al intentar efectuar operaciones inválidas, como una división por cero, o bien cuando están activos ciertos indicadores del procesador. Los procesadores 386 y posteriores, por ejemplo, generan una excepción equivalente a `int 1` siempre que esté activo el indicador de ejecución paso a paso. Todos los procesadores de la familia x86 generan una excepción `int 0` cuando se intenta ejecutar una división por cero.

Si prueba este código, comprobará cómo por pantalla aparece el mensaje que se ve en la parte inferior de la figura 17.7. Elimine la instrucción `int 0` y, tras ensamblar el programa, vuelva a ejecutarlo. Verá que el resultado es exactamente el mismo, ya que al intentar ejecutar la instrucción `div bl` se encuentra que el divisor es 0.

```

segment Pila stack
    resw 512
    ; Segmento de código
segment Código
..start:

    ; provocamos la
    ; excepción 0
    int 0

```

```

; que se produce
; automáticamente al
; ejecutar una división
; por 0
mov ax, 10
xor bl, bl
div bl

; Salimos al sistema
mov ah, 4ch
int 21h

```

The screenshot shows a Windows command-line interface. The title bar says 'Símbolo del sistema'. The content of the window is as follows:

```

D:\Ejemplos\17>nasm -f obj int0.asm
D:\Ejemplos\17>alink int0
ALINK v1.6 (C) Copyright 1998-9 Anthony A. J. Williams.
All Rights Reserved

Loading file int0.obj
matched Externs
matched ComDefs

D:\Ejemplos\17>int0
Divide overflow
D:\Ejemplos\17>...

```

Figura 17.7. Resultado de provocar una excepción por división por cero.

Manipulación de los vectores de interrupción

La tabla de vectores de interrupción, cuya estructura ya conoce, se aloja en memoria RAM y, por tanto, puede modificarse. De hecho, es una técnica habitual al construir ciertos tipos de aplicaciones, por ejemplo programas que quedan residentes en memoria para extender las funciones del sistema operativo. El cambio de un vector de interrupción, no obstante, es una operación que podría calificarse como peligrosa, ya que normalmente son necesarias varias operaciones y entre una y otra podría generarse una interrupción que intentase, precisamente, acceder a ese vector que está a medio cambiar.

Existen diversas técnicas para evitar esta posibilidad, que podría causar la caída del sistema, comenzando por la desactivación de las interrupciones antes de iniciar el cambio del vector y reactivándolas una vez modificado. Si van a realizarse dos instrucciones mov para cambiar el vector, pondríamos la instrucción cli delante del primer mov y la instrucción sti tras el segundo.

Los vectores de interrupción deben contener un puntero a una dirección donde exista una rutina capaz de gestionar la situación. A la hora de cambiar un vector, por tanto, debe tener una dirección válida donde exista código preparado para tratar esa interrupción. Ese código no debe nunca descargarse de la memoria sin antes restaurar el vector de interrupción con el valor que contuviese anteriormente. Por ello es habitual que, antes de cambiar un vector de interrupción, se lea su valor y conserve para poder restablecerlo.

Nota

Entre los servicios del DOS existen dos que facilitan la lectura y escritura de vectores de interrupción, encargándose de esta operación de una forma segura.

Resumen

Según se ha visto en este capítulo, la BIOS es una capa que se superpone directamente al hardware del ordenador y facilita el acceso a él con una cierta independencia, evitando que sea preciso conocer detalles de bajo nivel de cada dispositivo. Esta capa se compone de código que gestiona la puesta en marcha del sistema, antes de que entre en acción el sistema operativo que haya instalado; un área de datos y un conjunto de servicios accesibles mediante interrupciones.

Hemos conocido algunas de las interrupciones software más interesantes, si bien de forma breve. Sobre algunas de ellas volveremos en capítulos posteriores, conociendo con mayor detalle los servicios para el acceso a vídeo o teclado.

También sabemos que existen interrupciones software, interrupciones hardware y excepciones, así como interrupciones enmascarables y no enmascarables.

En capítulos posteriores, al estudiar la creación de aplicaciones que quedan residentes en memoria, verá en la práctica cómo se modifica la tabla de vectores para gestionar con código propio la respuesta a ciertas interrupciones hardware.

18

**Servicios
de vídeo**

Una de las interrupciones más interesantes de la BIOS es, sin duda alguna, la que da acceso a los servicios de vídeo, de los cuales conocemos un pequeñísima parte. Mediante ellos podemos no sólo introducir una secuencia de caracteres o situar el cursor en una determinada posición, sino alterar el modo de visualización, estableciendo el que más nos interese; leer y escribir puntos en los modos de gráficos, manipular tablas o paletas de colores, etc. En realidad, las posibilidades de los actuales adaptadores de vídeo van mucho más allá de la simple generación de imágenes a partir del contenido de un área de memoria, contando con capacidades de aceleración para la construcción de polígonos, texturas, etc. Efectivamente, el estudio de todas estas posibilidades, y su programación mediante la BIOS, daría para escribir un libro completo, algo que queda totalmente fuera del alcance de este título.

A lo largo de este capítulo, por tanto, conocerá sólo algunos de los servicios básicos con que cuentan la mayoría de adaptadores de vídeo, sin entrar en detalles específicos de ninguno en particular. A pesar de todo, como tendrá ocasión de ver, las posibilidades son suficientes para la mayoría de tareas habituales si exceptuamos la creación de juegos con utilización intensiva de gráficos.

Detección del tipo de adaptador

Las posibilidades gráficas de un adaptador de vídeo, principalmente resolución y número de colores que puede mostrar, dependen directamente de su tipo. El primer estándar aparecido en este campo fue el conocido como CGA (*Computer Graphics Adapter*), con

una fuerte competencia de los adaptadores Hercules que, aunque en monocromo, ofrecía una mayor resolución, compartiendo temporalmente su existencia con otros adaptadores como el MDA (*Monochrome Display Adapter*) y el MCCA (*Multicolor Graphics Array*), ambos de IBM. En realidad MDA fue el primer tipo de adaptador utilizado por IBM en los primeros PC, pero CGA y Hercules tuvieron mucha mayor difusión ya que MDA sólo permitía el trabajo con texto, sin posibilidades gráficas.

Posteriormente aparecieron EGA (*Enhanced Graphics Adapter*) y VGA (*Video Graphics Array*), como estándares más difundidos. A causa de la competencia entre fabricantes de adaptadores de vídeo, cada uno de ellos fue añadiendo sus capacidades específicas. La VESA (*Video Electronics Standards Association*) promocionó entonces un nuevo estándar, conocido como SVGA (*Super VGA*), que contaba con mayores resoluciones, definiendo unas extensiones a los servicios de vídeo conocidas como VESA BIOS. Estándares aparecidos con posterioridad, como SXGA (*Super Extended Graphics Array*) y UXGA (*Ultra Extended Graphics Array*) siguen basándose en el estándar VGA, aunque mejorando tanto la resolución de puntos como de color, llegando a configuraciones de 1600x1200 puntos con 24 bits de color.

Dependiendo del adaptador que tengamos instalado en nuestro sistema, podremos usar o no determinados modos de visualización, de ahí que sea importante conocerlo. No obstante, todos los adaptadores de vídeo actuales contemplan, por compatibilidad, el uso de los modos CGA/EGA/VGA/SVGA, de tal forma que siempre que nos ajustemos a esos estándares no debemos tener problemas.

En cualquier caso, conocer el tipo de adaptador con que contamos es una tarea muy simple. Basta con asignar el valor 1AO0h al registro AX e invocar a la interrupción 10h. El valor devuelto en BL, uno de los enumerados en la tabla 18.1, nos indicará qué adaptador de vídeo tiene el sistema.

Tabla 18.1. Valores devueltos en BL por el servicio 1A00h.

Valor	Tipo de adaptador
0	No hay adaptador de vídeo.
1	MDA.
2	CGA.
4	EGA color.
5	EGA monocromo.
6	PGC (<i>Professional Graphics Controller</i>).
7	VGA monocromo.
8	VGA color.
10	MCGA color.
11	MCGA monocromo.

Si escribe un sencillo programa, que se limite a llamar a este servicio de la interrupción 10h con dos instrucciones como éstas, verá que obtiene el valor 8 (véase la figura 18.1). Es lo habitual en equipos actuales, ya que todos son compatibles con el estándar VGA.

```
; obtenemos información  
; del adaptador  
mov ax, 1AOOh  
int 10h
```

```
D:\Ejemplos\18>debug adapta~1.exe  
-q  
D:\Ejemplos\18>debug adapta~1.exe  
-r  
AX=0000  BX=0000  CX=0261  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000  
DS=1864  ES=1864  SS=1874  CS=1884  IP=0000  NV UP EI PL NZ NA PO NC  
1884:0000  B87418  MOV     AX,1874  
-p  
AX=1874  BX=0000  CX=0261  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000  
DS=1864  ES=1864  SS=1874  CS=1884  IP=0003  NV UP EI PL NZ NA PO NC  
1884:0003  8ED0    MOV     SS,AX  
-p  
AX=1874  BX=0000  CX=0261  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000  
DS=1864  ES=1864  SS=1874  CS=1884  IP=0008  NV UP EI PL NZ NA PO NC  
1884:0008  B8001A  MOV     AX,1A00  
-p  
AX=1A00  BX=0000  CX=0261  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000  
DS=1864  ES=1864  SS=1874  CS=1884  IP=000B  NV UP EI PL NZ NA PO NC  
1884:000B  CD10    INT     10  
-p  
AX=001A  BX=0008  CX=0261  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000  
DS=1864  ES=1864  SS=1874  CS=1884  IP=000D  NV UP EI PL NZ NA PO NC  
1884:000D  B44C    MOV     AH,4C
```

Figura 18.1. Observamos el valor obtenido en BL.

El servicio 1A0 Oh de la interrupción 1 Oh sólo está disponible en los adaptadores posteriores a la aparición de los estándares MCGA y VGA, por lo que si lo usa en un sistema antiguo no obtendrá un valor fiable. Para saber si el servicio 1AO Oh está o no disponible, invóquelo y después compruebe que el registro AL contiene el valor 1 Ah. De no ser así es que el adaptador es anterior, es decir, un CGA/MDA/EGA o Hercules.

Modos de visualización

Como acaba de decirse, dependiendo del adaptador con que cuente el sistema podremos utilizar o no determinados modos de visualización. Un adaptador Hercules, por ejemplo, no puede usar los modos de baja resolución en color de un adaptador CGA, ni viceversa. Por eso es interesante conocer qué modos existen y, en particular, en qué adaptadores pueden utilizarse.

Un modo de vídeo determina si el contenido de la memoria dedicada a imagen se interpreta como texto o como gráficos, establece la configuración de esa memoria y, en el caso de los modos gráficos, la resolución en puntos y el número de colores. En las tablas 18.2 a 18.6 se recogen estos datos relativos a los modos de vídeo de los adaptadores MDA, CGA, Hercules, EGA y VGA. Los adaptadores actuales contemplan el uso prácticamente de todos esos modos, aparte de otros como los enumerados en la tabla 18.7.

Tabla 18.2. Modos disponibles en un adaptador MDA.

Código	Tipo	Resolución	Colores	Dirección	Páginas
7	Texto	80x25	Mono	0B000h	1

Tabla 18.3. Modos disponibles en un adaptador CGA.

Código	Tipo	Resolución	Colores	Dirección	Páginas
0	Texto	40x25	16	0B800h	8
1	Texto	40x25	16	0B800h	8
2	Texto	80x25	16	0B800h	4
3	Texto	80x25	16	0B800h	4
4	Gráfico	320x200	4	0B800h	1
5	Gráfico	320x200	4	0B800h	1
6	Gráfico	640x200	2	0B800h	1

Tabla 18.4. Modos disponibles en un adaptador Hercules.

Código	Tipo	Resolución	Colores	Dirección	Páginas
7	Texto	80x25	Mono	0B000h	1
-	Gráfico	720x348	Mono	0B000h	1

Tabla 18.5. Modos disponibles en un adaptador EGA.

Código	Tipo	Resolución	Colores	Dirección	Páginas
13	Gráfico	320x200	16	0A000h	8
14	Gráfico	640x200	16	0A000h	4
15	Gráfico	640x350	Mono	0A000h	2
16	Gráfico	640x350	16	0A000h	2

Tabla 18.6. Modos disponibles en un adaptador VGA.

Código	Tipo	Resolución	Colores	Dirección	Páginas
17	Gráfico	640x480	2	0A000h	1
18	Gráfico	640x480	16	0A000h	1
19	Gráfico	320x200	256	0A000h	1

Tabla 18.7. Modos adicionales en adaptadores compatibles con especificación VESA SVGA.

Código	Tipo	Resolución	Colores
100h	Gráfico	640x400	256
101h	Gráfico	640x480	256
102h	Gráfico	800x600	16
103h	Gráfico	800x600	256
104h	Gráfico	1024x768	16
105h	Gráfico	1024x768	256
106h	Gráfico	1280x1024	16
107h	Gráfico	1280x1024	256
108h	Texto	80x60	-
109h	Texto	132x25	-
10Ah	Texto	132x43	-
10Bh	Texto	132x50	-
10Ch	Texto	132x60	-
10Dh	Gráfico	320x200	32768
10Eh	Gráfico	320x200	65536
10Fh	Gráfico	320x200	16,7 millones
110h	Gráfico	640x480	32768
111h	Gráfico	640x480	65536
112h	Gráfico	640x480	16,7 millones
113h	Gráfico	800x600	32768
114h	Gráfico	800x600	65536
115h	Gráfico	800x600	16,7 millones

Código	Tipo	Resolución	Colores
116h	Gráfico	1024x768	32768
117h	Gráfico	1024x768	65536
118h	Gráfico	1024x768	16,7 millones
119h	Gráfico	1280x1024	32768
11Ah	Gráfico	1280x1024	65536
11Bh	Gráfico	1280x1024	16,7 millones
120h	Gráfico	1600x1200	256
121h	Gráfico	1600x1200	32768
122h	Gráfico	1600x1200	65536

Tenga en cuenta que el adaptador EGA contempla el uso de los modos CGA, y el VGA los de los dos anteriores, por lo que en las tablas respectivas se han incluido sólo los modos exclusivos de esos adaptadores.

De igual forma, en la tabla 18.7 aparecen sólo modos SVGA, pero también puede usar los modos CGA, EGA y VGA.

El número de páginas disponibles en cada modo es un dato que depende de la memoria disponible en el adaptador de vídeo, apareciendo en las tablas previas sólo a título orientativo. Actualmente todos los adaptadores incorporan 256,512 y hasta 1024 megabytes de memoria, muy lejos de los 4,16 ó 64 kilobytes con que contaban los adaptadores MDA, CGA y EGA originales.

Obtener y modificar el modo de visualización

Para poder operar en un cierto modo de vídeo, lo primero que necesitamos es establecerlo. También nos resultará útil conocer el modo de visualización actual, por ejemplo para devolver el ordenador a su estado previo una vez que el programa finalice su trabajo. Los servicios que nos interesan en este caso son el 0 Oh y el OFh.

El servicio OFh recupera el modo de vídeo actual, devolviendo su código en el registro AL. Además, también facilita en AH el número de columnas por línea y en BH el número de la página activa en este momento, en caso de que existan varias. El siguiente programa de ejemplo, que hace una vez más uso de la rutina EnteroCadena creada en un capítulo previo, puede ejecutarse desde la línea de comandos para obtener la información que se aprecia en la figura 18.2.

```
C:\>cd 18  
C:\18>modo  
El modo actual es 03  
La página actual es la 0  
C:\18>  
C:\18>  
C:\18>modo  
El modo actual es 03  
La página actual es la 0  
C:\18>  
C:\18>
```

Figura 18.2. El programa nos indica el modo de video y la página visible actualmente.

```
3egment Pila etack  
    resw 512  
FinPila:  
  
    segment Datos  
MsgModo db 'El modo actual es'  
Modo   db '    ',13, 10  
        db 'La página actual es la'  
Pagina db '      '$  
  
; Segmento de código  
segment Código  
  
.starL:  
    ; Configuramos la pila  
    mov ax, Pila  
    mov ss, ax  
    mov sp, FinPila  
  
    ; DS y ES apuntan al  
    ; segmento de datos  
    mov ax, Datos  
    mov ds, ax  
    mov es, ax  
  
    ; obtenemos información  
    ; del modo actual  
    mov ah, 0fh  
    int 10h  
  
    ; convertimos el modo  
    ; en cadena  
    mov di, Modo+2  
    cali ÉnteroCadena
```

```

; convertimos la página
; en cadena
mov al, bh
mov di, Página+2
cali EnteroCadena

; y mostramos el mensaje
mov dx, MsgModo
mov ah, 9
int 21h

; Salimos al sistema
mov ah, 4ch
int ?1h

```

II Uta

En el capítulo previo, al tratar el área de parámetros de la BIOS, se indicó la existencia de ciertas variables relacionadas con el modo de vídeo actual, de tal forma que podemos recuperar la misma información que nos ofrece el servicio OFh de la interrupción 1 Oh simplemente leyendo esa área de parámetros.

Para cambiar el modo de vídeo actual se utiliza el servicio OOh, que precisa como único parámetro el código del modo en el registro AL, pudiendo ser cualquiera de los vistos anteriormente en las tablas 18.2 a 18.6. Para los modos VESA existe un servicio específico, el 4F02h, que toma en BX el modo a establecer pudiendo ser cualquiera de los enumerados en la tabla 18.7 siempre que el adaptador pueda emplearlos.

El pequeño programa mostrado a continuación usa el servicio OOh para establecer el modo 19 (I3h en hexadecimal), existente en todas las VGA y, por tanto, en los adaptadores actuales. Si ejecuta el programa en una ventana de Windows, lo normal es que se pase a pantalla completa y no vea nada. Pulsando la combinación **Control-Intro** volverá al modo de texto normal. Trabajando en un equipo que realmente funcione con DOS, no en una ventana de consola de Windows, el efecto será que los caracteres aparecerán mucho más grandes (véase la figura 18.3). Además, cada punto puede manipularse de manera individual, como verá posteriormente. Puede volver al modo normal mediante el comando mode co80 del **DOS**.

```

segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
.start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

```

```
; Modo 320x200 con
; 256 colores
xor ah, ah
mov al, 13h
int 10h

; Salimos al sistema
mov ah, 4ch
int 21h
```

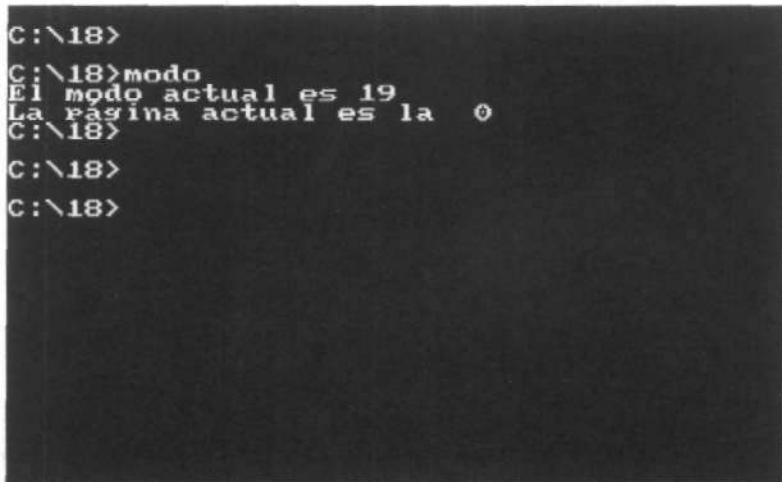


Figura 18.3. Aspecto de la pantalla en el modo de 320x200 con 256 colores.

Servicios para trabajar con texto

Una vez que conocemos los modos de visualización que tenemos a nuestra disposición, y la forma de activar el que más nos interese en cada momento, vamos a ir tratando los demás servicios de la interrupción 10h, comenzando por los relacionados con los modos de texto.

Al trabajar en los modos de texto la pantalla se divide en una cuadrícula, de L líneas por C columnas, dando lugar a LxC celdillas, cada una de las cuales puede contener un carácter y, según el modo, también un atributo.

El carácter es la representación de un código, normalmente de la tabla ASCII, mientras que el atributo, dependiendo del adaptador, puede indicar un color de fondo y uno de tinta o simplemente un parpadeo.

Los modos de texto cuentan con un cursor que normalmente está visible. Dicho cursor puede encontrarse en cualquiera de las celdillas, es decir, en una determinada línea y columna, indicando la posición en la que aparecerán los próximos datos a visualizar. El cursor puede adoptar diversas formas, así como mantenerse estático, parpadear o, incluso, desaparecer.

Dependiendo de la memoria con que cuente el adaptador, es posible que existan varias páginas de texto, lo cual hace posible la operación sobre un área no visible que aparece en el momento que interese. Existe una posición de cursor independiente para cada una de las páginas de texto.

Posición y aspecto del cursor

Relacionados con el cursor, su posición y aspecto, encontramos tres servicios en la interrupción 10h. Con ellos podemos alterar la posición, su aspecto y recuperar ambas informaciones. Comencemos por este último, el servicio 03h, que tan sólo precisa en el registro BH el número de página de cuyo cursor queremos recuperar información. La primera página, que es la visible por defecto, es la 0.

Los datos devueltos por el servicio 03h son los siguientes:

- CH: Contendrá la línea de inicio del cursor.
- CL: Contendrá la línea de fin del cursor.
- DH: Columna en la que se encuentra el cursor.
- DL: Línea en la que se encuentra el cursor.

El cursor está compuesto de un cierto número de líneas que depende del modo de vídeo en que nos encontramos. Esas líneas se numeran de arriba abajo, por ejemplo de 1 a 7 en el modo 3 en una CCA. El cursor por defecto aparece como una línea algo gruesa situada en la parte inferior de la línea en la que está trabajándose, teniendo, típicamente, un par de líneas de grosor. Las posiciones, devueltas en DH y DL, parten de 0, de tal forma que la primera columna y primera fila sería la 0,0. Al trabajar en modo de texto es más habitual contar de 1 en adelante, para lo cual no tenemos más que incrementar los valores devueltos por este servicio.

Para ver en la práctica el uso de este servicio vamos a escribir un programa informativo, al estilo de otros previos, que nos indicará la posición del cursor, su línea de inicio y de fin. El código es el siguiente:

```

segment Pila stack
    resw 512
FinPila:

        segment Datos
Mensaje    db 'El cursor está en la posición'
Columna   db ' ', '
Linea     db ' ', 13, 10
            db 'Comienza en la linea'
Curlnicio db ' '
            db ' y finaliza en la linea'
CurFin    db '$'

; Segmento de código
segment Código

```

```

..start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; DS y ES apuntan al
; segmento de datos
mov ax, Datos
mov ds, ax
mov es, ax

; obtenemos información
;- del cursor en la página 0
mov ah, 3
xor bh, bh
int 10h

; convertimos la columna
; en cadena
mov al, di
inc al
mov di, Columna+2
cali EnteroCadena

; convertimos la linea
; en cadena
mov al, dh
inc al
mov di, Linea+2
cali EnteroCadena

; convertimos la linea
; de inicio en cadena
mov al, ch
mov di, CurInicio+2
cali EnteroCadena

; convertimos la linea
; de fin en cadena
mov al, el
mov di, CurFin+2
cali EnteroCadena

; y mostramos el mensaje
mov dx, Mensaje
mov ah, 9
int 21h

; Salimos al sistema
mov ah, 4ch
int 21h

```

Al ejecutar el programa desde la línea de comandos, el cursor queda en el margen izquierdo de la pantalla y, por tanto, la columna siempre será la 1. La línea irá cambiando

a medida que pulse Intro. La línea de inicio y fin del cursor normalmente se mantendrán fijas.

Si con el servicio 03h obtenemos los datos del cursor, los servicios Olh y 02h nos permiten modificar el aspecto y posición, respectivamente, de este elemento. El primero necesita la línea de inicio y fin del cursor en los registros CH y CL, mientras que el segundo espera las nuevas coordenadas del cursor en DH y DL. Recuerde que el número de línea y columnas deben tomar como referencia el 0, no el 1.

Puede, utilizando el servicio Olh, crear dos programas muy sencillos, y prácticamente idénticos, que alteren el aspecto del cursor. Uno de ellos, podría llamarse curblo, estaría compuesto del código siguiente:

```
segment Pila stack
    resw 512
FinPila:

; Segmento de código
segmenl Código
..start:
    ; Configuramos la pila
    mov ax, Pila
    mov s8, ax
    mov sp, FinPila

    ; Establecemos el
    ; tamaño del cursor
    mov ah, 1
    mov ch, 1
    mov el, 7
    int 10h

    ; Salimos al sistema
    mov ah, 4ch
    int 21h
```

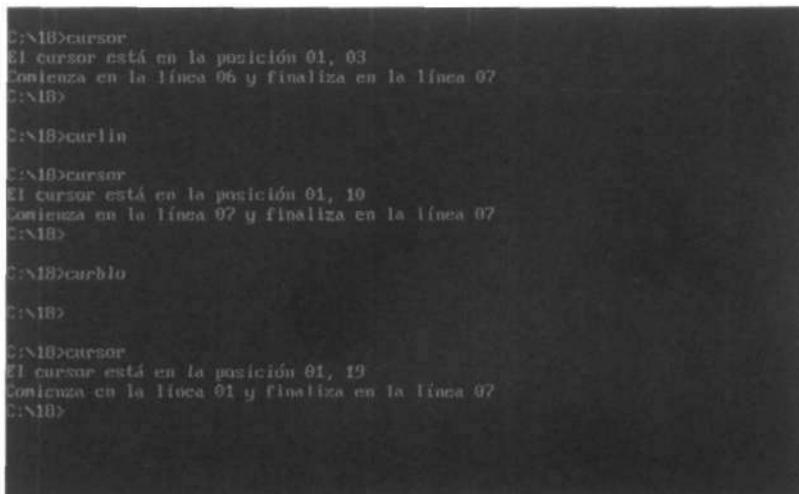
El segundo, que podría denominarse curlin, sería idéntico a éste salvo en que asignaría al registro CH el valor 7 en lugar del valor 1. Con estos dos programas, una vez ensamblados, podría seleccionar un cursor en forma de línea o de bloque. En la figura 18.4 puede ver cómo se han usado estos dos programas, así como el ejemplo anterior que nos informa de la posición y líneas del cursor.

Caracteres y atributos

La posición del cursor es un factor importante a la hora de usar los servicios de escritura y lectura de caracteres y atributos, ya que la operación de escritura siempre parte de la posición actual del cursor, mientras que la de lectura afecta exclusivamente al contenido de la posición en que se encuentra dicho elemento.

Los servicios disponibles para escribir datos en la pantalla son dos: 09h y OAh. La única diferencia es que el primero escribe un carácter y un atributo, mientras que el segundo introduce exclusivamente el carácter, dejando el atributo que contuviese dicha

celdilla de pantalla. El punto de partida de la operación de escritura es, como acaba de decirse, la que tenga actualmente el cursor, repitiéndose tantas veces como indique el contenido del registro ex.



The screenshot shows a DOS terminal window with the following text output:

```
C:\NIB>cursor  
El cursor está en la posición 01, 03  
Comienza en la línea 06 y finaliza en la línea 07  
C:\NIB>  
  
C:\NIB>curlin  
  
C:\NIB>cursor  
El cursor está en la posición 01, 19  
Comienza en la línea 07 y finaliza en la línea 07  
C:\NIB>  
  
C:\NIB>curblo  
  
C:\NIB>  
  
C:\NIB>cursor  
El cursor está en la posición 01, 19  
Comienza en la línea 01 y finaliza en la línea 07  
C:\NIB>
```

Figura 18.4. Alteramos el aspecto del cursor, que aparece como un bloque.

Ambos servicios precisan en el registro AL el código ASCII del carácter que va a escribirse, y en el registro BH la página en la que se escribirá. Es posible, por tanto, escribir en una página que no está visible actualmente. En el caso del servicio 09h, además deberá facilitarse el atributo en el registro BL. El atributo tiene la estructura que ya conoce, al haberlo usado múltiples veces para escribir directamente en pantalla.

Para escribir un carácter también puede utilizar el servicio 0Eh, similar al servicio 09h dado que recibe los mismos parámetros. La diferencia es que el servicio 0Eh escribe el carácter y hace avanzar el cursor, saltando de línea si es necesario.

Para recuperar la información contenida en una cierta posición de pantalla, carácter y atributo, puede utilizarse el servicio 08h. Éste tan sólo precisa el número de página de la que leer, en el registro BH, devolviendo el atributo en AH y el código ASCII del carácter en AL. En caso de que se quieran leer múltiples caracteres, es necesario ir actualizando la posición del cursor a medida que se invoca a este servicio.

A pesar de que estos servicios cuentan con la ventaja de que permiten leer y escribir en páginas que no están visibles, por regla general funcionan más lentamente que si accedemos directamente a la memoria de pantalla, como se ha hecho hasta ahora en ejemplos de capítulos previos. Puesto que conocemos la dirección del segmento donde

se encuentra esa área de memoria, dependiendo del modo en que nos encontremos, lo único necesario es saber cuánto espacio ocupa cada página para acceder directamente a las no visibles, ya que éstas se encuentran normalmente una tras otra.

El siguiente programa de ejemplo usa los servicios de lectura y escritura para copiar la tercera línea de pantalla en la decimosexta, carácter a carácter. Para ello colocamos el cursor en la línea de origen, leemos carácter y atributo, colocamos el cursor en la línea de destino y escribimos, incrementando la columna y repitiendo el proceso 80 veces.

```

segment Pila stack
    resw 512
FinPila:

        ; Segmento de código
segment Código
.start:
        ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

        ; partimos de la
        ; columna 0
    xor di, di

        ; vamos a leer y
        ; escribir de la
        ; página 0
    xor bh, bh

        ; 80 columnas en total
    mov ex, 80
Bucle:
        ; guardamos el contador
        ; del bucle
    push ex

        ; ponemos el cursor
        ; en la linea 2
    mov ah, 2
    mov dh, 2
    int 10h

        ; y leemos el carácter
        ; y atributo de esa posición
    mov ah, 8
    int 10h

        ; pasamos el atributo
        ; al registro BL
    mov bl, ah

        ; colocamos el cursor
        ; en la linea 15
    mov ah, 2

```

```

mov dh, 15
int 10h

; y escribimos el carácter
; y atributo antes leidos
mov ah, 9
mov ex, 1
int 10h

; pasamos a la
; columna siguiente
inc di

; recuperamos el
; contador
pop ex
; y repetimos
loop Bucle

; Salimos al sistema
mov ah, 4ch
int 21h

```

Como puede ver, el proceso es bastante largo si tenemos en cuenta que la mayor parte del código se repite 80 veces.

Asumiendo que el modo de visualización actual es el 3, el más usual al trabajar en modo texto, podríamos entonces conseguir exactamente el mismo resultado con el código siguiente:

```

#define Posición(x,y) y*160+x*2

segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código

..start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; ES y DS apuntarán
; a la pantalla
mov ax, 0b800h
mov ds, ax
mov es, ax

; establecemos en SI
; la linea de inicio
mov si, Posición(0,2)
; y en DI la de fin
mov di, Posición(0,15)

```

```
; número de caracteres  
mov ex, 80 ; acopiar  
cid  
  
; copiamos  
rep movsw  
  
; salimos al sistema  
mov ah, 4ch  
int 21h
```

A parte de ser más simple, este programa no repite un importante número de instrucciones y acceso a la BIOS, obteniendo un mejor rendimiento. Como puede ver, la mejor opción no siempre pasa por emplear los servicios de la BIOS.

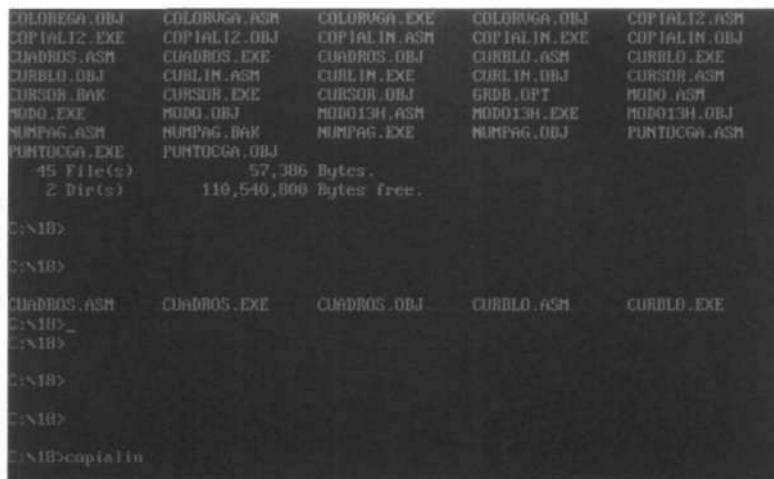


Figura 18.5. El programa copia la tercera línea en la decimosexta.

Cambio de la página activa

Dependiendo de la memoria con que cuente el adaptador de vídeo, al trabajar en modo texto contaremos con un número menor o mayor de páginas. El estándar CGA indica la existencia de 4 páginas al operar con 80 columnas y 8 si trabajamos con 40, pero los adaptadores posteriores, por ejemplo la VGA, contaban con una memoria mayor, por lo que podían permitir el uso de más páginas.

Para cambiar la página activa, aquella que está mostrándose en pantalla en un determinado momento, emplearemos el servicio 05h. Tan sólo es necesario facilitar un parámetro: el número de página a establecer como activa, en el registro AL. Al cambiar de página, especialmente operando desde la línea de comandos, la sensación será que la pantalla se ha borrado, ya que el indicador del sistema vuelve a aparecer. Sin embargo, lo que ocurre en realidad es que ahora está mostrándose otra porción de la memoria de vídeo.

Como no es fácil saber cuántas páginas como máximo permite un cierto adaptador, podemos usar un método como el mostrado en el ejemplo siguiente. En él se va cambiando de una página a otra y, tras cada cambio, se utiliza el servicio OFh para verificar que la página activa es realmente la que se supone que debía ser.

Recuerde que dicho servicio no sólo nos indica el modo de visualización actual, sino también la página activa.

```
segment Pila stack
    resw 512
FinPila:

segment Datos
Mensaje db 'Hay disponibles '
Número db '
    db ' páginas$'

; Segmento de código
segment Código
..start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; DS y ES apuntan al
; segmento de datos
mov ax, Datos
mov ds, ax
mov es, ax

; comenzamos con
; la primera página
xor al, al

,- comprobar como máximo
; 100 páginas
mov ex, 100

Bucle:
; establecemos la
; página indicada en AL
mov ah, 5
int 10h

push ax ; guardamos AL

; recuperamos información
; del modo actual
mov ah, Ofh
int 10h

pop ax ; recuperamos AL
; y vemos si coincide
emp al, bh
```

```

; de no ser así
; no hay más páginas
]ne NoHayMas

; en caso contrario
; incrementamos AL
inc al
; y repetimos el proceso
loop Bucle

NoHayMas:
; en este momento AL
; contiene el número
; de páginas

; lo convertimos a cadena
mov di, Numero+2
cali EnteroCadena

; restablecemos la
; página 0
xor al, al
mov ah, 5
int 10h

; y mostramos el mensaje
mov dx, Mensaje
mov ah, 9
int 21h

; Salimos al sistema
mov ah, 4ch
int 21h

```

Al ejecutar este programa en un sistema con una antigua VCA, operando realmente con DOS, se obtiene un resultado como el de la figura 18.6. En un sistema más moderno, incluso trabajando en una ventana de consola de Windows, el resultado es muy distinto. De hecho se indica el valor límite que hemos puesto en el programa, pero seguramente existen aún más páginas de texto.



Figura 18.6. Número de páginas en un sistema típico con VGA.

D:\Ejemplos\18>nasm -f obj numpag.asm
D:\Ejemplos\18>alink numpag
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved
Loading file numpag.obj
matched Externs
matched ComDefs
D:\Ejemplos\18>
D:\Ejemplos\18>
D:\Ejemplos\18>numpag
Hay disponibles 100 páginas
D:\Ejemplos\18>
D:\Ejemplos\18>

Figura 18.7. Número de páginas informadas en un sistema actual.

Desplazamiento del texto

Entre los servicios básicos para trabajo con texto encontramos, aparte de los ya citados en los puntos previos, dos más cuya finalidad es idéntica: desplazar una parte del texto verticalmente.

La diferencia es que el servicio 0 6h empuja el texto hacia arriba, mientras que el 0 7h lo hace hacia abajo.

Los parámetros, en ambos casos, son los mismos y se facilitarán en los registros siguientes:

- CH: Fila correspondiente a la esquina superior izquierda del recuadro de texto a desplazar.
- CL: Columna de la esquina superior izquierda del recuadro de desplazar.
- DH: Fila correspondiente a la esquina inferior derecha del recuadro de texto a desplazar.
- DL: Columna de la esquina inferior derecha del recuadro a desplazar.
- AL: Número de líneas a desplazar.
- BH: Atributo a introducir en las líneas en blanco que se inserten arriba o abajo, según el sentido del desplazamiento.

Como puede ver, no se indica en ningún parámetro la página sobre la que actuará el servicio, ya que éste siempre recae sobre la página que esté activa en ese momento.

Si desea borrar el contenido del recuadro indicado, asigne el valor o al registro AL.

El siguiente programa utiliza estos dos servicios para crear la composición que puede verse en la figura 18.8, compuesta de cuatro recuadros de diferentes colores.

```

segment Pila stack
    resw 512
FinPila:
; Segmento de código
segment Código
..start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila
; desde la posición 10,2
mov ex, 0109h
; hasta 33,10
mov dx, 0920h
xor al, al
; con fondo blanco
mov bh, 70h
mov ah, 6
int 10h ; borramos

; nuevo recuadro desde
; 49,2 hasta 73,10
mov el, 30h
mov di, 48h
mov bh, 60h
int 10h

; nuevo recuadro desde
; 10,13 hasta 33,21
mov ex, 0C09h
mov dx, 1420h
xor al, al
mov bh, 50h
mov ah, 6
int 10h

; nuevo recuadro desde
; 49,13 hasta 73,21
mov el, 30h
mov di, 48h
mov bh, 40h
int 10h

; Salimos al sistema
mov ah, 4ch
int 21h

```

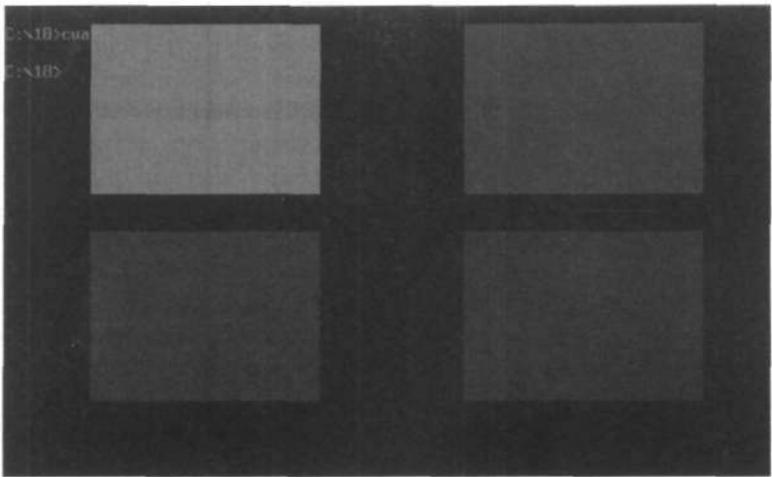


Figura 18.8. El programa muestra cuatro recuadros de color en pantalla.

Servicios para trabajar con gráficos

Algunos de los servicios que acabamos de conocer, aunque nos hemos centrado en el trabajo en modo de texto, pueden emplearse asimismo cuando el modo actual de visualización es un modo gráfico. En caso de que existan múltiples páginas, por ejemplo, puede utilizarse el servicio 05h para cambiar de una a otra.

Entre los servicios que podríamos catalogar como básicos de la interrupción 10h, aquellos disponibles para todos los adaptadores, tan sólo existen dos específicos para el trabajo en modos gráficos: el servicio OCh, que escribe un punto en pantalla, y el servicio ODh, que lo lee.

En caso de que el adaptador sea de tipo EGA contaremos con una serie de servicios adicionales, si es VGA tendremos a nuestra disposición otro grupo más, ocurriendo así con cada uno de los existentes.

Escritura y lectura de puntos

A diferencia de lo que ocurre en los modos de texto, en los cuales no se pueden manipular individualmente los puntos con los que se forman, por ejemplo, los caracteres, en los modos gráficos esto es, precisamente, lo más habitual. La BIOS tan sólo facilita servicios para efectuar las operaciones más básicas: darle un cierto color a un punto determinado o bien leer un punto para saber el color que tiene. No hay servicios para dibujar líneas, círculos ni nada parecido, solamente puntos individuales.

Para poder dar a un cierto punto el color que deseemos, mediante el servicio OCh de la interrupción 10h, tendremos que preparar una serie de parámetros en los registros siguientes:

- CX: Posición horizontal, columna, del punto a manipular.
- DX: Posición vertical, línea, del punto a manipular.
- BH: Número de página, en caso de que exista más de una.
- AL: Color a dar al punto.

Si lo que nos interesa es conocer el color actual de un punto, usaremos el servicio ODH en lugar del OCh facilitando los mismos parámetros en los registros CX, DX y BH, obteniéndose ese dato en el registro AL.

Utilizando el servicio OCh podríamos escribir un sencillo programa de ejemplo, como el siguiente, que dibujase tres líneas horizontales en pantalla, cada una de un color. En este caso utilizamos el modo 4, con una resolución de 320x200 puntos, porque sabemos que permite cuatro colores incluyendo el negro.

En la figura 18.9 se puede observar el resultado, aunque no apreciará el color de las distintas líneas.

```

segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
..start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; activamos el modo 4
    ; 320x200 puntos
    mov al, 4
    xor ah, ah
    int 10h

    ; vamos a dibujar la
    ; primera linea en
    ; la posición vertical 75
    mov dx, 75
    ; con el color 1
    mov al, 1
    ; en la página 0
    xor bh, bh
    ; preparamos el servicio
    mov ah, OCh

    ; dibujaremos tres lineas
    mov ex, 3

BucleO:
    ; guardamos el contador
    ; del bucle
    push ex

```

```
; para establecer la
; posición horizontal
mov ex, 200
Bucle1:
; dibujamos un punto
int 10h
; y repetimos hacia atrás
loop Bucle1

; incrementamos la
; posición vertical
add dx, 50
; y el color
inc al

; recuperamos el contador
; del bucle y repetimos
pop ex
loop Bucle0

; esperamos la pulsación
; de una tecla
xor ah, ah
int 16h

; volvemos al modo
; de vídeo de texto
mov al, 3
xor ah, ah
int 10h

; Salimos al sistema
mov ah, 4ch
int 21h
```

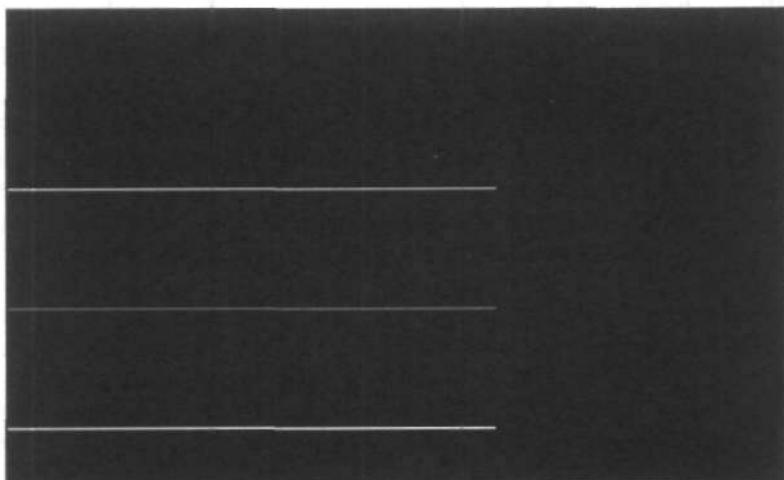


Figura 18.9. La pantalla muestra tres líneas de distintos colores.

El color en adaptadores CGA

Uno de los problemas que encontraremos a la hora de trabajar en modo gráfico, interpretando o estableciendo el color de los puntos, es, precisamente, cómo se define el color en cada adaptador y cada modo. En el anterior programa se han usado como color los números 1, 2 y 3, facilitados en AL. ¿Qué colores son esos? ¿Pueden utilizarse igualmente en los demás modos? La respuesta es que esos códigos dependen del modo en que operemos.

Los adaptadores CGA disponen de dos modos gráficos, como ya sabe, con resoluciones de 320x200 y 640x200. En este último tan sólo pueden utilizarse dos colores: blanco y negro, identificados por los códigos 0 y 1. Es fácil, por tanto, interpretar el estado de cada punto, ya que las posibilidades son muy limitadas.

En el modo de 320x200 puntos hay cuatro colores posibles, a los que corresponden los códigos 0, 1, 2 y 3. El 0 siempre es el color negro, mientras que los otros tres podrán ser turquesa, violeta y blanco, o bien, verde, rojo y amarillo, dependiendo de la paleta de color que se haya establecido.

No existen más posibilidades, si bien éstas parecen notables al compararse con el modo de 640x200 o bien las posibilidades de los adaptadores MDA y Hercules, que son monocromos.

Para activar una paleta u otra se utiliza el servicio OBh de la interrupción 10h, debiendo facilitarse en el registro BH el valor 1 y en BL el número de paleta. Este será 0, para tener verde, rojo y amarillo; o 1, para que los mismos códigos aparezcan como turquesa, violeta y blanco.

Nota

Los adaptadores actuales también contemplan el uso de estas paletas cuando se activa el modo de 320x200 con 4 colores por compatibilidad con las antiguas aplicaciones.

Tomando como base el ejemplo anterior, añada al archivo de código la rutina Espera, creada en un capítulo previo, y la que se muestra a continuación. Ésta alterna entre las dos paletas posibles diez veces, con una pausa de un segundo entre cambio y cambio. Inserte la sentencia call AlternaPaleta en el programa principal justo después del int 16h. De esta forma, al ejecutar el programa primero verá las tres líneas en la paleta por defecto y, tras pulsar una tecla, verá cómo esas líneas cambian de color al alternar la paleta.

```
; Esta rutina, asumiendo que
; nos encontramos en el modo
; de 320x200 con 4 colores,
; alterna la paleta actual
; varias veces con una pausa
; entre cambios.
```

```

AlteraPaleta:
; número del servicio
mov ah, 0Bh
; BH debe tener 1
mov bh, 1

; alternaremos entre
; las paletas 1 y 0
mov bl, 1
xor di, di

; 10 veces
mov ex, 10

BuclePaleta:
; establecemos la paleta
int luh

; guardamos el contador
push ex
; establecemos 1 segundo
mov ex, 1
; de espera
cal] Espera

; intercambiamos paleta
xchg di, bl

; recuperamos contador
pop ex
; y repetimos
loop BuclePaleta

ret ; volver

```

El color en adaptadores EGA

Los adaptadores EGA cuentan con varios modos de visualización que permiten 16 colores simultáneos, lo cual supone un importante incremento respecto al estándar CGA. Además, dichos 16 colores pueden ser tomados de una paleta de 64 posibles. En realidad no existe una paleta con una serie de colores predefinidos, como sí ocurre en la CGA, sino un conjunto de 16 registros de paleta que establecen las intensidades de rojo, verde y azul.

Cada uno de esos registros de paleta tiene un byte de tamaño, si bien sólo se emplean los seis bits de menor peso. Los bits 0, 1, 2 establecen el nivel intenso de azul, verde y rojo, respectivamente, mientras que los bits 3, 4 y 5 determinan el nivel no intenso de esos mismos colores y en el mismo orden.

Haciendo todas las combinaciones posibles tendríamos 64 colores distintos pero, puesto que únicamente existen 16 registros de paleta, tan sólo podemos tener 16 de manera simultánea.

La modificación de un registro de paleta, estando visible en pantalla el color que le corresponde, hará que el cambio se vea de manera inmediata, como ocurría al alternar la paleta de la CGA.

Los valores de color que es posible entregar al servicio para escribir un punto, por tanto, estarán comprendidos entre 0 y 15, haciendo referencia al color que indica uno de los 16 registros de la paleta. Éstos, inicialmente, tienen definidos los mismos colores existentes en los modos de texto del estándar CGA, pudiendo modificarse a través de dos servicios distintos.

En el caso de que deseemos alterar el color asociado a un cierto registro de paleta, el servicio que nos interesa es el 1000h. En realidad es el servicio 10h, que como siempre se introduce en AH, el cual cuenta con varios subservicios que pueden seleccionarse mediante el registro AL, de ahí que pueda asignarse el valor completo a AX. Los parámetros necesarios, en este caso concreto, son dos: el número de registro de paleta, en BL, y el valor a introducir en él, en BH.

Además de los registros de paleta asociados a los colores, tal y como ya se ha dicho 16 distintos que después pueden utilizarse para dibujar los puntos que interesen, también existe un registro adicional que establece el color del borde. Éste se fija mediante el servicio 1001h, facilitando el color en el registro BH.

Podemos establecer el valor de todos los registros de paleta, los 16 y el color de borde, en un solo paso mediante el servicio 1002h. Tan sólo es necesario un parámetro: una dirección, facilitada en ES : DX, a una tabla de 17 bytes, correspondiendo los 16 primeros a los valores que se introducirán en los registros de paleta y el último para el color de borde.

El siguiente programa utiliza el servicio 1002h para, tras haber dibujado 16 líneas con un color distinto cada uno, alterar los registros de paleta y conseguir tres tonos de azul, otros tantos de verde y otros de rojo, que aparecerán de derecha a izquierda. Los valores de la paleta, definidos al inicio, se han calculado de la siguiente forma:

- Los bits correspondientes al color azul son el 3, correspondiente al azul normal, y el 0, que activa el azul intenso. Los valores 8, 1 y 9 corresponden en binario a las combinaciones 001000, OOOO00yOO1001, de tal forma que activan el azul normal, el azul intenso y después ambos, consiguiendo un azul brillante.
- Los bits del color verde son el 4 y el 1 y, si convierte a binario los valores 16, 2 y 18, verá que hemos efectuado la misma operación anterior.
- Los bits del color rojo son el 5 y el 2, usando la misma técnica para conseguir tres tonos de rojo.
- El primer registro quedará con el valor 0, negro, al ser éste el registro que se usa para todos los puntos por defecto. Si le asigna cualquier otro valor verá que todo el fondo de la pantalla lo toma.

- Por último tenemos el índice correspondiente al color del borde, en el que se ha seleccionado el registro 1, es decir, un azul normal.

```

        segment Pila stack
        resw 512
FinPila:

        segment Datos
Colores db 0 ; color negro
        db 8,1,9 ; tres azules
        db 16,2,18 ; tres verdes
        db 32,4,36 ; y tres rojos
        db 5,7,15,36,42,60 ; otros colores
        db 1 ; color del borde

; Segmento de código
        segment Código
..start:
        ; Configurarnos la pila
        mov ax, Pila
        mov s s t ax
        mov sp, FinPila

        ; activamos el modo
        ; 640x350 con 16 colores
        mov al, 10h
        cali EstableceModo

        ; vamos a trazar
        ; 16 lineas
        mov ex, 16
Bucle0:
        ; calculamos la
        ; posición horizontal
        mov al,el
        mov di, 4 0
        muí di

        ; dejamos el resultado
        ; en DX
        mov dx, ax

        ; calculamos el color
        mov al, 16
        sub al, el

        ; y dibujamos una linea
        cali LineaVertical

        ; repetimos
        loop Bucle0

        ; esperamos la pulsación
        ; de una tecla
        xor ah, ah
        int 16h

```

```

; alteramos la paleta
; de color
cali ModificaPaleta

xor ah, ah
int 16h ; esperamos tecla

; volvemos al modo
; de video de texto
mov al, 3
cali EstableceModo

; Salimos al sistema
mov ah, 4ch
int 21h

; Esta rutina dibuja una
; linea vertical completa
; en la posicion horizontal
; facilitada en DX y el
; color indicado en AL

LineaVertical:
    pusha ; guardamos registros

    ; vamos a dibujar la
    ; linea desde 0 a 350
    mov ex, 34 9

    ; en la pagina 0
    xor bh, bh

    ; preparamos el servicio
    mov ah, 0Ch

BucleLO:
    ; El servicio espera las
    ; coordenadas en orden
    ; inverso
    xchg dx, ex

    ; dibujamos un punto
    int 10h

    ;- volvemos a tener el
    ; contador en CX
    xchg dx, ex

    ; y repetimos hacia atrás
    loop BucleLO

    ; recuperamos registros
    popa

    ret ; y volvemos

```

```

.
; Esta rutina establece el
; modo gráfico deseado, que
; se facilitará en AL

EstableceModo:
    ; Ponemos AH a cero
    xor ah, ah
    ; y establecemos el modo
    int 10h

    ret ; volver

; Esta rutina modifica los
; valores de los registros de
; paleta de color EGA

ModificaPaleta:
    pusha ; guardar registros

    ; ES:DX apunta al área
    ; en que se han definido
    ; los colores
    mov ax, Datos
    mov es, ax
    mov dx, Colores

    ; establecemos paleta
    mov ax, 1002h
    xor bh, bh
    int 10h

    ; recuperamos registros
    popa

    ret ; y volvemos

```

El programa calcula matemáticamente la posición horizontal donde dibujará cada línea, sirviéndose del contador, así como el color.

Después de dibujarlas, mediante una rutina codificada aparte, espera la pulsación de una tecla e invoca a **ModificaPaleta**, que se encargará de modificar los registros de paleta con los valores antes comentados. La figura 18.10 corresponde al aspecto de la pantalla antes de efectuar ese cambio.

Los servicios empleados para manipular los registros de paleta son una extensión añadida con la aparición de los adaptadores EGA y existente también en adaptadores posteriores, pero no puede acceder a ellos si el adaptador es de tipo CGA ya que la BIOS del mismo no cuenta con esas extensiones.

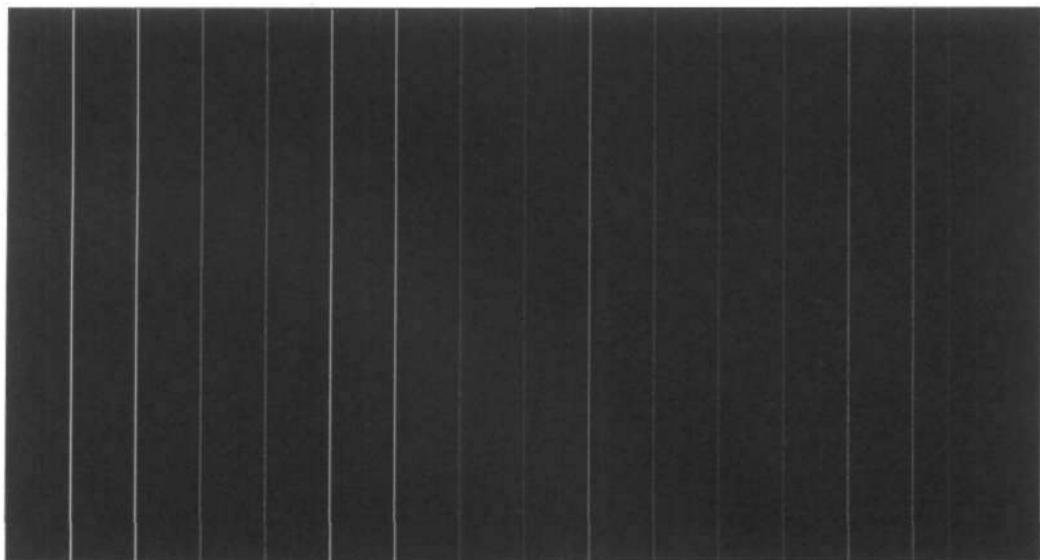


Figura 18.10. Trazamos 16 líneas de distinto color.

El color en adaptadores VGA

Los adaptadores de tipo VGA cuentan, por compatibilidad, con los mismos registros de paleta que los adaptadores EGA, pero el color realmente se genera a partir de un DAC (*Digital/Analog Converter*) que cuenta con 256 registros diferentes. Por eso al contar con una VGA es posible optar entre modos de 16 colores, como en EGA, o de 256 colores, mucho más llamativos.

Cada uno de los registros del DAC de una VGA se compone de tres bytes, si bien sólo son significativos los seis bits de menor peso de cada uno de ellos. El primero indica la intensidad de rojo, el segundo la de verde y el tercero la de azul. Ya que con seis bits tenemos 64 combinaciones posibles, de 0 a 63, podemos formar $64 \times 64 \times 64$ colores o, lo que es lo mismo, 2 elevado a 18, ya que en total tenemos 18 bits significativos.

Resumiendo: el DAC nos permite definir hasta 262.144 colores distintos, si bien al existir sólo 256 registros, tal como se ha dicho, ése es el número de colores simultáneos que es posible tener definidos.

Los bits de cada uno de los colores no tienen un significado especial, como sí ocurre con los registros de paleta de la EGA.

Por tanto, cuanto mayor sea el valor del byte de un cierto color significará más intensidad de dicho color, simplemente.

A la hora de dibujar puntos en una VGA, el máximo valor de color dependerá del modo empleado. Salvo el modo 13h, en el que se permiten 256 colores simultáneos, los demás funcionan como los de una EGA, siendo posible el uso de 16 colores. En ambos casos, eso sí, cada color puede estar compuesto por cualquier combinación de rojo, verde y azul en los límites ya explicados.

Lectura de los registros del DAC

A diferencia de los adaptadores EGA, en los cuales la BIOS sólo permite la modificación de los registros de paleta pero no su lectura, para las VGA existen dos servicios de lectura de los registros del DAC. El primero de ellos, servicio 1015h, nos permite leer un registro individual, para lo cual tendremos que entregar en el registro BL el número de dicho registro, estando comprendido entre 0 y 255. Los valores devueltos son los siguientes:

- DH: Intensidad de rojo.
- CH: Intensidad de verde.
- CL: Intensidad de azul.

En caso de que necesitemos leer múltiples registros DAC, en lugar del anterior usaremos el servicio 1017h. En este caso los parámetros necesarios son tres, introduciéndose en los registros indicados a continuación:

- BX: índice del primero registro del DAC a leer.
- CX: Número de registros a leer.
- ES: DX: Área de memoria donde se dejará la información.

La zona de memoria apuntada por ES : DX debe estar compuesta de, al menos, 3 bytes por cada elemento, es decir, el contenido de CX multiplicado por 3.

Nota

La lectura de la paleta de colores, alojada en los registros del DAC, puede sernos útil para generar ciertos efectos o, simplemente, con la idea de restaurar la paleta original una vez se haya finalizado el trabajo.

Modificación de los registros del DAC

Al igual que para la lectura, tenemos a nuestra disposición dos servicios de escritura de valores en los registros del DAC. Uno de ellos escribe un solo valor, mientras que el otro escribe un bloque de valores alojados en una cierta posición de memoria.

El servicio 1010h escribe en el registro indicado en BX los valores de rojo, verde y azul indicados en los registros DH, CH y CL. Dichos valores, como ya sabe, deben estar comprendidos entre 0 y 63.

El servicio 1012h es el encargado de escribir un bloque de valores. Los parámetros necesarios son los mismos que en el servicio 1017h, con la única diferencia de que en este caso los valores se escriben en lugar de leerse.

Efectos de color

Al emplear los modos de 16 colores de la VGA, sólo una parte de los registros DAC están activos en un determinado momento emulando así el comportamiento EGA. En los modos de 256 colores, por el contrario, es posible manipular los 256 registros del DAC afectando a los puntos que en ese momento haya en pantalla de manera inmediata.

Si anteriormente, al tratar los registros de paleta del adaptador EGA, vimos que era posible crear ciertos efectos, como varios tonos de un mismo color, dichos efectos son mucho más notables al tener disponibles no cuatro intensidades distintas por color sino sesenta y cuatro. Además, al poder mostrar en pantalla 256 colores distintos se pueden conseguir efectos realmente llamativos como degradados de color o fundidos a negro o a color. Tan sólo hay que manipular los valores alojados en los registros del DAC con un poco de imaginación, nada más.

El programa propuesto a continuación genera algunos efectos de color. Parte por dibujar 256 líneas verticales, tras activar el modo 320x200, cada una de ellas con un color distinto. A continuación guarda la paleta por defecto y genera otra, con cálculos matemáticos, compuesta de cuatro bloques de colores degradados. A continuación restablece la paleta original y la va desplazando hacia la izquierda, con un simple movimiento del contenido de los registros del DAC, dando la impresión visual de que las líneas se desplazan por la pantalla y van desapareciendo. Finalmente, se muestra de nuevo la paleta con colores degradados y se efectúa un fundido a negro, dando la impresión de que la imagen desaparece poco a poco de la pantalla.

Como puede observar, el código se ha dividido en múltiples rutinas a fin de facilitar su comprensión mediante el estudio individual de cada una de ellas. Se ha codificado una nueva rutina de espera, dado que la que teníamos hasta ahora era demasiado lenta al imponer como mínimo un segundo. En la figura 18.11 se muestra un momento de la ejecución del programa, aunque en ella no puede apreciarse el color ni los efectos de animación descritos.

```
; Macro que introduce las
; sentencias necesarias para
; esperar la pulsación de
; una tecla sin modificar AX
%macro EsperaTecla 0
    push ax
    xor ah, ah
    int 16h
    pop ax
%endmacro

    segment Pila stack
    resw 512
FinPila:

    segment Datos
; Reservamos espacio para
; dos paletas de color
Paletal resb 768
Paleta2 resb 768
```

```
; Un elemento que define
; el color negro
Negro db 0, 0, 0

        ; Segmento de código
        segment Código
..start:
        ; Configuramos la pila
        mov ax, Pila
        mov ss, ax
        mov sp, FinPila

        ; y los registros que
        ; apuntan a los datos
        mov ax, Datos
        mov ds, ax
        mov es, ax

        ; activamos el modo
        ; 320x200 con 256 colores
        mov al, 13h
        cali EstableceModo

        ; Dibujamos las
        ; 256 lineas
        cali DibujaLineas

        ; esperamos la pulsación
        ; de una tecla
        EsperaTecla

        ; Guardamos la paleta
        ; por defecto
        cali GuardaPaleta

        ; Establecemos una paleta
        ; de color con degradados
        cali PaletaDegradado

        EsperaTecla ; esperamos tecla

        ; Restauramos la paleta
        ; original
        cali RestauraPaleta

        EsperaTecla ; esperamos tecla

        ; efectuamos un
        ; desplazamiento de colores
        cali DesplazaColores

        EsperaTecla ; esperamos tecla

        ; Hacemos un fundido a negro
        cali FundidoNegro
```

```

; volvemos al modo
; de video de texto
mov al, 3
cali EstableceModo

; Salimos al sistema
mov ah, 4ch
int 21h

; Esta rutina dibuja las
; lineas en pantalla

DibujaLineas:
    pusha ; guardamos registros

    ; vamos a trazar
    ; 256 lineas
    mov ex, 255

BucleO:
    ; fijamos el color
    mov al, el
    ; y dibujamos una linea
    cali LineaVertical

    ; repetimos
    loop BucleO

    ; recuperamos registros
    popa

    ret ; y volvemos

; Esta rutina dibuja una
; linea vertical completa
; en la posición horizontal
; facilitada en CX y el
; color indicado en AL

LineaVertical:
    pusha ; guardamos registros

    ; llevamos la posición
    ; horizontal a DX
    mov dx, ex
    ; centramos en pantalla
    add dx, 30

    ;- vamos a dibujar la
    ; linea desde 0 a 199
    mov ex, 199
    ; en la página 0
    xor bh, bh
    ; preparamos el servicio
    mov ah, 0Ch

```

```
BucleLO:  
    ; El servicio espera las  
    ; coordenadas en orden  
    ; inverso  
    xchg dx, ex  
  
    ; dibujamos un punto  
    int 10h  
  
    ; volvemos a tener el  
    ; contador en CX  
    xchg dx, ex  
  
    ; y repetimos hacia atrás  
loop BucleLO  
  
    ; recuperaremos registros  
    popa  
  
    ret ; y volvemos  
  
; Esta rutina establece el  
; modo gráfico deseado, que  
; se facilitará en AL  
  
EstableceModo:  
    ; Ponemos AH a cero  
    xor ah, ah  
    ; y establecemos el modo  
    int 10h  
  
    ret ; volver  
  
; Esta rutina guarda la paleta  
; actual en Paleta2  
;  
_____  
GuardaPaleta:  
    pusha ; guardar registros  
  
    ; leemos la paleta  
    ; actual  
    mov dx, Paleta2  
    ; desde el elemento 0  
    xor bx, bx  
    ; 256 elementos  
    mov ex, 256  
  
    ; la obtenemos  
    mov ax, 1017h  
    int 10h  
  
    popa ; recuperamos  
  
    ret ; y volvemos
```

```

; Esta rutina genera y activa
; una paleta compuesta de
; degradados de color

PaletaDegradado:
    pusha ; guardamos registros

    ; DI apunta a la paleta
    ; que vamos a generar
    ; matemáticamente
    mov di, Paletal

    ; dividimos el proceso
    ; en 4 bloques
    mov ex, 4

    ; AL rojo, BL verde
    ;,- BH azul
    xor al, al
    xor bl, bl
    xor bh, bh

BucleP1:
    ; guardamos contador
    ; del bucle
    push ex

    ; vamos a generar 64 entradas
    mov ex, 64
    cid ; incrementar DI

BuclePO:
    ; almacenamos rojo
    stosb

    push ax
    mov al, bl
    stosb ; verde

    mov al, bh
    stosb ; y azul

    ; recuperamos rojo
    pop ax
    ; e incrementamos
    inc al

    ; repetimos bucle interno
    loop BuclePO

    ; ponemos a 0 el rojo
    xor al, al
    ; incrementamos verde
    add bh, 12
    ; y azul
    add bl, 4

```

```
; recuperamos contador
pop ex
; y repetimos bucle externo
loop BucleP1

; establecemos paleta
mov ax, 1012h
xor bx, bx
mov ex, 256
mov dx, Paletal
int 10h

popa ; recuperamos registros

ret ; y volvemos
```

```
; Esta rutina restablece la
; paleta de colores original
```

```
RestauraPaleta:
    pusha ; guardar registros

    ; restablecemos la
    ; paleta original
    mov ax, 1012h
    xor bx, bx
    mov ex, 256
    mov dx, Paleta2
    int 10h

    popa ; restaurar

    ret ; y volver
```

```
; Esta rutina rota los colores
; copiando los elementos de
; los registros del DAC
```

```
DesplazaColores:
    pusha ; guardamos registros

    ; Hay 256 lineas
    mov ex, 256
```

```
BucleP2:
    push ex ; guardamos contador

    ; apuntamos al inicio
    ; de la paleta
    mov di, Paleta2
    mov si, Paleta2+3
    ; compuesta de
    ; 7 68 bytes
    mov ex, 7 68
    rep movsb
```

```

; Activamos la paleta
; recién generada
mov ax, 1012h
mov bx, 1
mov ex, 255
mov dx, Paleta2
int 10h

; y esperamos una
; fracción de segundo
cali Espera

; recuperamos contador
pop ex

; y repetimos
loop BucleP2

; recuperamos registros
popa

ret ; y volvemos

; Esta rutina efectúa un
; fundido a negro de la
; Paletal

FundidoNegro:
    ; Número de valores
    ; para cada componente
    ; del color
    mov ex, 64

BucleFO:
    ; guardamos contador
    push ex

    ; la paleta tiene
    ; - 7 68 bytes
    mov ex, 7 68

    ; vamos a ir leyendo
    ; y escribiendo en
    ; la misma paleta
    mov si, Paletal
    mov di, Paletal

    ; dejamos DX preparado
    ; para establecerla
    mov dx, Paletal

BucleFl:
    ; leemos un byte
    mov al, [si]
    ; comprobamos si es 0
    or al, al

```

```

; en caso afirmativo
jz YaEsCero ; saltamos

; en caso contrario
dec al ; reducimos
; y sustituimos
mov [di], al
YaEsCero:
; avanzar en la paleta
inc si
inc di
; y repetir
loop BucleF1

; establecer la
; nueva paleta
mov ax, 1012h
xor bx, bx
mov ex, 25 6
int 10h

; y esperar un momento
cali Espera

; recuperaremos contador
pop ex
; y repetimos bucle exterior
loop BucleFO

ret ; volver

```

; Rutina que espera una fracción
; de segundo

Espera:

```

pusha ; guardar registros

; leer número de pulsos
xor ah, ah
int 1Ah

; copiar en BX
mov bx, dx
inc bx ; e incrementar

BucleEO:
; esperar hasta que
; haya transcurrido
int 1Ah
emp bx, dx

ja BucleEO

popa ; recuperar

ret ; y volver

```

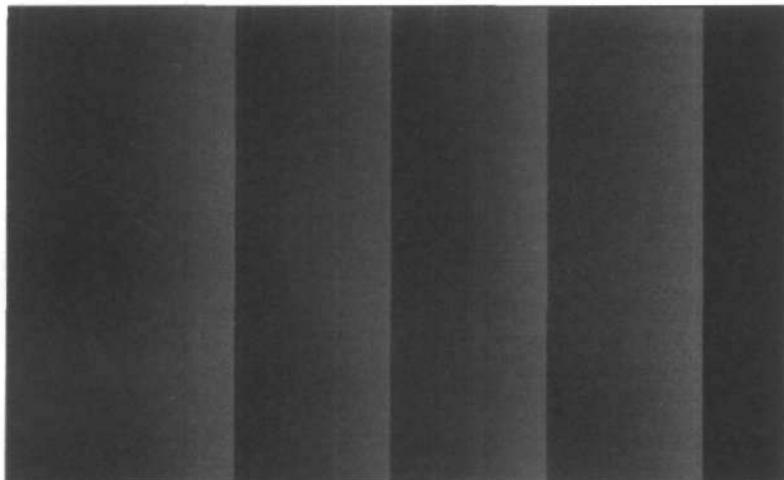


Figura 18.11. El programa muestra unos degradados de color.

Resumen

Los adaptadores gráficos son dispositivos bastante complejos, especialmente en la actualidad con la existencia de aceleradores gráficos con gran cantidad de memoria y capaces de operar con altas resoluciones y millones de colores. En este capítulo nos hemos centrado en el estudio de los servicios básicos de la BIOS para trabajar con esos dispositivos. A pesar de que se ha ido haciendo mención a modos de texto y gráficos de CGA, EGA y VGA, la mayoría de tarjetas gráficas actuales son capaces de comportarse como dichos adaptadores, por lo que podemos usarlos sin ningún problema.

En el capítulo siguiente conocerá los servicios relacionados con el teclado, un dispositivo mucho más sencillo pero tan imprescindible como la pantalla. Por ahora nos hemos limitado a esperar la pulsación de una tecla, pero los servicios de teclado pueden ofrecernos mucha más información.

19

Servicios de teclado

Mas allá de lo que es la selección de opciones predefinidas en menús y botones, que puede efectuarse con el ratón, el teclado es el dispositivo de entrada de datos por excelencia, al menos hasta que la interacción mediante voz se convierta en un recurso fiable y difundido. Poder conocer el estado y leer información proveniente del teclado es, consecuentemente, una necesidad en la mayoría de las aplicaciones.

En los ejemplos de capítulos previos hemos usado reiteradamente uno de los servicios de la interrupción de teclado, la 16h, con el fin de esperar una pulsación de tecla antes de continuar con la ejecución del programa. En este capítulo conocerá ese servicio con algo más detalle, así como los demás servicios relacionados con el teclado. Como se apuntase al final del capítulo previo, éste es un dispositivo mucho más simple que el adaptador de vídeo, de ahí que el trabajo con él resulte más sencillo y sean menos los servicios disponibles.

Recuperación de teclas pulsadas

La función principal del teclado es facilitar la introducción de datos en el sistema, datos que las aplicaciones pueden recuperar por diversos medios. Uno de ellos consiste en usar los distintos servicios que la BIOS ofrece para recuperar información sobre las teclas que van pulsándose. Podemos tanto recuperar una pulsación de tecla esperando a que ésta llegue, si el buffer de teclado está vacío, como explorar ese buffer para saber si hay o no información esperando a ser procesada. Esto último nos permitirá, por ejemplo, ejecutar otros procesos sin obviar la atención al teclado.

Dependiendo de que el teclado sea estándar, entendiendo como tal los primeros que aparecieron con sólo diez teclas de función muchas veces dispuestas en el margen izquierdo del teclado y sin teclas independientes de cursor, o extendido, categoría a la que pertenecen los actuales, también habrá que usar unos servicios u otros.

Teclados estándar

Si contamos con un teclado estándar, o bien con un teclado extendido pero no nos importa perder las pulsaciones de las teclas extendidas, podemos usar los servicios 0 Oh y 0 lh para recuperar información sobre las pulsaciones de tecla que hay pendientes de recoger. El primero, como ya sabe, recoge la información de una pulsación de tecla o, de estar vacío el buffer de teclado, espera a que ésta se produzca. El segundo servicio, por el contrario, comprueba si hay o no alguna información en dicho buffer, pero en ningún caso espera a que se pulse una tecla.

El servicio Olh activa el bit ZF del registro de indicadores en caso de que no haya ninguna tecla disponible, poniéndolo a 0 en caso contrario. En ambos servicios se devuelve, además, el código ASCII de la tecla pulsada, en el registro AL, y el código de búsqueda o *sean code* en el registro AH. Este código está fijado por el hardware y permite identificar de manera única cada una de las teclas existentes.

Los códigos ASCII son bastante conocidos por los programadores en general, algo que no ocurre con los códigos de búsqueda. Éstos, en ocasiones, son imprescindibles para detectar ciertas teclas que no tienen asociado un código ASCII, como es el caso de las teclas de función o de desplazamiento del cursor. En lugar de facilitarle una tabla con los códigos de búsqueda de todas las teclas del teclado, puede usar el programa siguiente para conocer tanto el código ASCII como el de búsqueda de cualquier tecla. No tiene más que ejecutarlo y pulsar las teclas que deseé, anotando sus códigos.

```

segment Pila stack
    resw 512
FinPila:

segment Datos
; Las variables siguientes se tomarán
; como una sola a la hora de imprimir
; ya que sólo hay un indicador de fin
Cadena db 'Código de búsqueda: '
Búsqueda db '123'
        db ' - Código ASCII: '
ASCII    db '123'
        db 13, 10, •$'

        i.
; Segmento de código
segment Código
..start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

```

```
; DS y ES apuntan al segmento
; que contiene los datos
mov ax, Datos
mov ds, ax
mov es, ax

; situamos el cursor
; al inicio de la
; pantalla para borrarla
xor dx, dx
mov ah, 2
int 10h

; introducimos 2000
; espacios en blanco
mov bl, 70h
mov al, ' '
xor bh, bh
mov ex, 2000
mov ah, 9
int 10h
```

Bucle:

```
; Eliminamos el contenido de
; Búsqueda y ASCII para
; recoger los códigos
cali Limpia

; esperamos una tecla
xor ah, ah
int 16h

; ¿Es Escape?
emp al, 27
; Si no es así saltar
jne Procede

; Salimos al sistema
mov ah, 4ch
int 21h
```

Procede:

```
; convertimos los
; códigos
cali Conver

; Ponemos el cursor en la
; línea 11 y columna 15
mov di, 15
mov dh, 11
mov ah, 2
int 10h

; mostramos la cadena
; con los códigos
```

```

mov dx, Cadena
mov ah, 9
int 21h

; repetimos el proceso
jmp Bucle

; Esta rutina tiene la finalidad
; de limpiar las dos partes
; variables del texto a imprimir

Limpia:
    ; ES:DI apunta a Búsqueda
    mov di, Búsqueda

    ; introducimos tres espacios
    mov al, ' '
    mov ex, 3
    rep stosb

    ; lo mismo para la
    ; variable ASCII
    mov di, ASCII
    mov ex, 3
    rep stosb

    ret ; volver

; Esta rutina convertirá el contenido
; de AX en dos cifras decimales y las
; almacenará en sus correspondientes
; posiciones. Se sirve de la rutina
; Conl que se encarga de convertir AH
; en una cadena ASCII decimal

```

```

Conver:
    push ax ; guardamos AX

    ; ES:DI apunta a Búsqueda
    mov di, Búsqueda

    cali Conl ; convertimos

    pop ax ; recupera los códigos
    ; para convertir AL
    mov ah, al

    ; y almacenarlo en ASCII
    mov di, ASCII

    ; convertir
    cali Conl

    ret ; volver

```

```
; Esta rutina recibirá un valor a  
; convertir en AH y una dirección  
; de destino en ES:DI
```

```
.
```

```
Conl:
```

```
; ¿Es AH menor que 100?  
cmp ah, 100  
; si es así salta  
jb Salto1  
  
; en caso contrario dividir  
; entre 100  
mov al, ah  
mov bl, 100  
xor ah, ah  
; el resultado serán  
; las centenas  
div bl  
  
; convertirlo a ASCII  
add al, '0'  
; y almacenarlo  
stosb  
  
xor al, al ; borrar AL
```

```
Salto1:
```

```
; ¿Es AH menor de 10?  
cmp ah, 10  
; si es así salta  
jb Salto0  
  
; en caso contrario  
; dividir entre 10  
mov al, ah  
mov bl, 10  
xor ah, ah  
; el resultado serán  
; las decenas  
div bl  
  
; convertirlo en ASCII  
add al, '0'  
; y almacenarlo  
stosb  
  
xor al, al ; borrar AL  
jmp Salto2 ; y saltar
```

```
Salto0:
```

```
,- aquí se llega si no  
; había decenas, por  
; lo que almacenamos  
; un 0 en esa posición  
mov al, '0'  
stosb
```

```

Salto2:
; Lo que queda en AH, tras
; las anteriores divisiones,
; son las unidades
mov al, ah
; convertir en ASCII
add al, '0'
; y almacenarlo
stosb

ret ; volver

```

Este programa le ofrece una rutina de conversión de binario a cadena con una implementación alternativa a la mostrada en un capítulo previo y que se basaba en la división sucesiva por 10.

Al ejecutar el programa la pantalla quedará en blanco. Pulse una tecla y verá aparecer, aproximadamente en el centro de la pantalla, su código ASCII y de búsqueda. Puede ir pulsando teclas y viendo los datos. Cuando pulse la tecla **Esc** el programa devolverá el control al sistema.

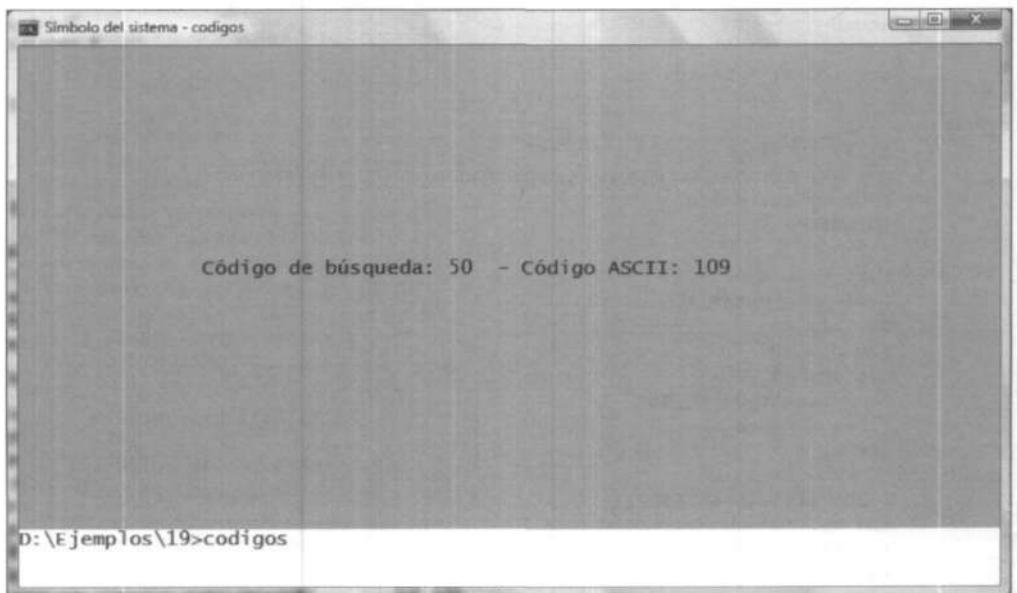


Figura 19.1. Aspecto del programa mostrando el código ASCII y de búsqueda de una tecla.

Una vez hemos localizado el código, ASCII o de búsqueda, de las teclas que nos interesen, podemos usarlos directamente sin ningún problema. El programa siguiente, algo más extenso que el anterior, detecta la pulsación de las teclas de movimiento del cursor a izquierda y derecha. Además, usa el servicio 0 lh para comprobar si se han pulsado o no, de tal forma que mientras no sea así la ejecución del programa continúa. El efecto es que el programa desplaza una pelota y hace una serie de comprobaciones, para hacerla

rebotar en los extremos y en una hipotética pala que hay en la parte inferior, sin por ello dejar de atender al teclado.

El código se ha dividido en múltiples macros y rutinas para facilitar su comprensión, empleándose un gran conjunto de las instrucciones y algunos servicios que ha conocido en todos los capítulos previos. Algunas rutinas emplean el indicador de acarreo para comunicar ciertas situaciones, como la pulsación de la tecla Esc la salida de la pelota por la parte inferior, al bucle principal del programa, de tal forma que éste puede centrarse en las acciones globales en lugar de en los detalles.

```
; Esta macro coloca el cursor
; «n el punto indicado sin
; modificar registros

%macro PosicionCursor 0
    push ax ; guardar AX
    push bx ; y BX

    ; Ponemos el cursor en la
    ; posición indicada
    xor bh, bh
    mov ah, 2
    int 10h

    ; restauramos registros
    pop bx
    pop ax
%endmacro

; Esta macro escribe el carácter
; indicada en la posición actual

%macro EscribeCaracter 2
    push ax ; guardamos
    push bx ; registros
    push ex

    ; establecemos
    ; carácter y atributo
    mov bl, 70h
    mov al, %1
    xor bh, bh
    mov ex, %2

    ; escribimos el carácter
    mov ah, 9
    int 10h

    ; recuperamos registros
    pop ex
    pop bx
    pop ax
```

^Rndmrico

```
; Constantes con los caracteres
Pelota equ '0'
Pala   equ 196

segment Pila stack
    resw 512
FinPila:

segment Datos
; Número máximo de pelotas
Mensaje    db 'Quedan '
NumPelotas db '3'
                db 'pelotas$'

; Segmento de código
segment Código
..start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; DS y ES apuntan al segmento
; que contiene los datos
mov ax, Datos
mov ds, ax
mov es, ax

; limpiamos la pantalla
cali LimpiaPantalla

; nos situamos en la esquina
; superior derecha
mov di, 60
xor dh, dh
PosicionCursor

; y mostramos el mensaje
mov dx, Mensaje
mov ah, 9
int 21h

; Bucle principal del programa
BuclePrincipal:
; ponemos la pala en pantalla

; obtenemos una columna aleatoria
; para la salida de la pelota
cali PosicionAleatoria

; colocamos el cursor
mov dh, 24
PosicionCursor

; y mostramos la pala
EscribeCaracter Pala, 3
```

```
; la pelota partirá de la fila 1
mov dh, 1

; factor de movimiento
; inicial para columna y fila
mov el, 1
mov ch, 1

; posición inicial
; de la pala
mov bl, di
mov bh, 24

BucleMovimiento:
    ; Examina el teclado
    cali ExaminaTeclado
    je Fin ; saltar si se pulsa ESC

    ; Actualiza la pala
    cali ActualizaPala

    ; colocamos el cursor
    PosicionCursor

    ; y mostramos la pelota
    EscribeCaracter Pelota, 1

    ; Examina el teclado
    cali ExaminaTeclado
    je Fin ; saltar si se pulsa ESC

    ; Actualiza pala
    cali ActualizaPala

    ; Esperamos una fracción
    ; de segundo
    cali Espera

    ; Examina el teclado
    cali ExaminaTeclado
    je Fin ; saltar si se pulsa ESC

    ; Actualiza la pala
    cali ActualizaPala

    ; eliminamos la pelota
    PosicionCursor
    EscribeCaracter ' ', 1

    ; Actualizamos la posición
    cali ActualizaPelota
    ; si se ha salido saltar
    je SiguientePelota

jmp BucleMovimiento ; continuar
```

```

; La pelota se ha salido por
; la parte inferior
SiguientePelota:
    ; reducimos el número de pelotas
    dec byte [NumPelotas]

    push dx , - guardamos posición

    ; limpiamos la pantalla
    cali LimpiaPantalla

    ; nos situamos en la esquina
    ; superior derecha
    mov di, 60
    xor dh, dh
    PosicionCursor

    ; y mostramos el mensaje
    mov dx, Mensaje
    mov ah, 9
    int 21h

    pop dx ; recuperamos posición

    ; comprobamos si es '0'
    cmp byte [NumPelotas], '0'
    ; si no es cero continuamos
    jnz BuclePrincipal

Fin:
    ; Salimos al sistema
    mov ah, 4ch
    int 21h

; Esta rutina se encargará de
; actualizar la posición de la
; pelota en pantalla.

; La posición actual es DL,DH
; Los desplazamientos están en CL,CH

ActualizaPelota:
    ; incrementamos las coordenadas
    add di, el
    add dh, ch

    ; comprobamos si estamos
    ; en la columna 79
    cmp di, 79
    ; de no ser asi saltar
    jne NoCol79

    ; en caso contrario
    mov el, -1

```

```
NoCol79:
; comprobar si estamos
; en la columna 0
or di, di
; de no ser asi saltar
jne NoCol0

; en caso contrario
mov el, 1

NoCol0:
; comprobamos si estamos
; en la linea 23
emp dh, 2 3
; de no ser asi saltar
jne NoT, i n2 3

; en caso contrario
; comprobar si la pala
; está justo debajo
emp bl, di
ja SeFue

; la pala tiene 3
; caracteres de ancho
add bl, 2
emp bl, di
jb SeFue

sub bl, 2

; rebota
mov ch, -1
ret

SeFue:
ste ; activar carry
ret ; y volver

NoLin23:
; comprobar si estamos
; en la linea 1
emp dh, 1
; de no ser asi saltar
jne NoLin1

; en caso contrario
mov ch, 1

NoLin1:
ret ; volver

; Esta rutina examina el teclado
; para ver si hay alguna tecla
; pulsada
```

ExaminaTeclado:

```

; comprobamos si hay
; alguna tecla pulsada
mov ah, 1
int 16h

; Si Z está a 1 es que no
jz NoHayTeclas

; de haberla la extraemos
xor ah, ah
int 16h

; comprobar si es Esc
cmp al/ 27
; de no ser asi saltar
jne NoEsESC

; en caso contrario
; activar el carry
stc

ret ; y volver

```

NoHayTeclas:

```
xor ax, ax
```

NoEsESC:

```

ele ; poner a 0 el carry
ret ; volver

```

; Esta rutina actualiza la posición
; de la pala según la tecla pulsada

ActualizaPala:

```

; ver si se ha pulsado
cmp ah, 75
; el cursor a izquierda
je Izquierda
; o el cursor a derecha
cmp ah, 77
je Derecha

```

Volver:

```
ret ; volver
```

Izquierda:

```

; si estamos en la columna 0
or bl, bl
jz Volver

; en caso contrario quitar
; de la posición actual
xchg di, bl
xchg dh, bh

```

```

PosicionCursor
EscribeCaracter ' ', 3

; y reducir la columna
dec di

; para volver a mostrar
PosicionCursor
EscribeCaracter Pala, 3

; dejar los registros
; como estaban
xchg di, bl
xchg dh, bh

ret ; y volver

Derecha:
; si estamos en la columna 77
cmp al, 77
jz Volver

; en caso contrario quitar
; de la posición actual
xchg di, bl
xchg dh, bh
PosicionCursor
EscribeCaracter ' ', 3

; e incrementar
inc di

; para volver a mostrar
PosicionCursor
EscribeCaracter Pala, 3

; dejar los registros
; como estaban
xchg di, bl
xchg dh, bh

ret ; y volver

; Esta rutina limpia la
; pantalla

LimpiaPantalla:
xor dx, dx
PosicionCursor
; establecemos
; carácter y atributo
mov bl, 70h
mov al, ' '
xor bh, bh
mov ex, 2000

```

```

; escribimos el carácter
mov ah, 9
int 10h

ret ; volvemos

; Esta rutina devuelve una
; posición horizontal pseudo-aleatoria
; en el registro DL

PosicionAleatoria:
    ; obtenemos el número
    ; actual de segundos
    cali SegundosActual

    ; llevamos a ax
    mov ax, bx
    ; y dividimos entre 80
    mov bl, 80
    div bl

    ; en AH tenemos un resto
    ; entre 0 y 79
    mov di, ah

    ret ; volvemos

; Procedimiento que obtiene
; los minutos y segundos del
; reloj, los convierte a
; segundos y devuelve en BX

SegundosActual:
    push ax ; guardamos AX

    ; queremos leer los minutos
    mov al, 2
    out 70h, al
    in al, 71h

    ; los multiplicamos por 60
    mov bl, 60
    muí bl
    ; y guardamos en BX
    mov bx, ax

    ; queremos leer los segundos
    xor al, al
    out 70h, al
    in al, 71h

    ; los sumamos
    xor ah, ah
    add bx, ax

```

```

pop ax ; y volvemos
ret

; Rutina que espera una fracción
; de segundo

Espera:
pusha ; guardar registros

; leer número de pulsos
xor ah, ah
int 1Ah

; copiar en BX
mov bx, dx
add bx, 2 ; e incrementar
BucleEO:
; esperar hasta que
; haya transcurrido
int 1Ah
emp bx, dx

ja BucleEO

popa ; recuperar

ret ; y volver

```

A parte del uso del indicador de acarreo, antes citado, el programa usa otras técnicas que pueden resultarle interesantes.

Observe que a pesar de la cantidad de parámetros que se manejan en el bucle principal del programa: posición de la pelota, posición de la paleta y dirección horizontal y vertical de la pelota, el único dato que se mantiene en memoria es el contador de pelotas, que es precisamente el que menos se utiliza. Todos los demás se mantienen en registros, facilitando así un acceso inmediato y, por tanto, una ejecución más rápida. Esto, a cambio, tiene el inconveniente de que en ciertos momentos es necesario intercambiar registros o guardarlos temporalmente en la pila.

Observe también cómo los desplazamientos que se aplican a la posición horizontal y vertical de la pelota, alojados en los registros CL y CH, pueden ser tanto positivos como negativos. Esto hace posible que la pelota se mueva de izquierda a derecha y de arriba a abajo pero también en sentido inverso, porque al sumar un número negativo lo que estamos haciendo es, realmente, restar.

El hecho de que en cada ciclo del bucle de movimiento se invoque tres veces a las rutinas que exploran el teclado y actualizan la pala, por sólo una a la rutina de movimiento de la pelota, contribuye a facilitar que el movimiento sea más rápido y dé tiempo a desplazarse al punto adecuado.

Hay que tener en cuenta que, en principio, la pelota sólo tiene que recorrer 23 líneas para salir de pantalla, mientras que la pala puede necesitar desplazarse hasta casi 80 posiciones.

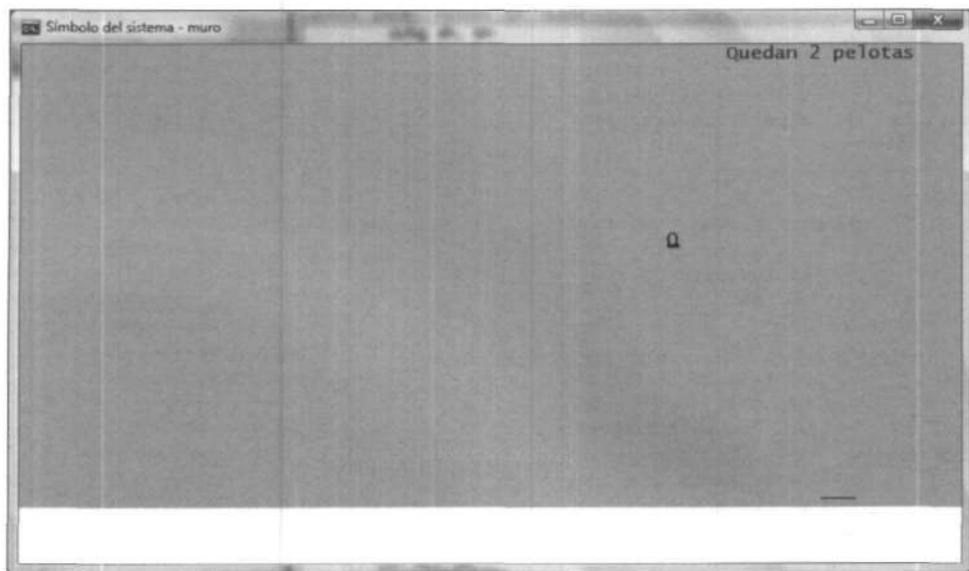


Figura 19.2. El programa en funcionamiento.

Nºa i^H^^^^BS^H

Fíjese que en la rutina ExaminaTeclado, invocada desde el bucle de movimiento del programa, se utiliza el servicio 0 lh de la interrupción 16h para comprobar si hay alguna pulsación de tecla y, en caso afirmativo, la extrae con el servicio OOh. De no hacerlo así, las siguientes pulsaciones se irían acumulando en el buffer de teclado hasta que éste se llenase, impidiendo el normal funcionamiento de la aplicación.

Teclados extendidos

Como se decía antes, todas las teclas extendidas de los teclados, incluyendo en éstas las de función, desplazamiento del cursor y las del área de edición, no cuentan con un código ASCII ya que son teclas que efectúan una cierta acción en muchos programas pero no se traducen en un carácter. En las BIOS originales los servicios OOh y Olh descartan el código de búsqueda de estas teclas, de tal forma que en programas como los anteriores no generarían acción alguna. Podría darse el caso, por ejemplo, de que el segundo de esos programas no detectase la pulsación de las teclas de desplazamiento del cursor y, por tanto, no pudiera moverse la pala que aparece en la parte inferior.

La alternativa consiste en emplear los servicios 1 Oh y 1 lh que, como puede suponer, son equivalentes a los servicios 0 0 h y 01 h con la única diferencia de que sí reconocen los códigos de búsqueda. Los parámetros devueltos son exactamente los mismos. Pruebe a sustituir las llamadas a los servicios originales, en los dos programas anteriores, por los

servicios 1 Oh y 1 lh, ensáblelos y ejecútelo de nuevo. El comportamiento, en equipos actuales, es exactamente el mismo.

Estado de teclas muertas y de doble estado

Además de teclas asociadas a caracteres y teclas de acción, los teclados también disponen de teclas muertas, como las teclas **Mayús**, **Control** y **Alt**, y teclas de doble estado, como **Bloq Mayús**, **Bloq Num** y **Bloq Despl**. Las primeras pueden estar o no pulsadas en un cierto momento en combinación con otras, no teniendo efecto alguno por sí solas. Las segundas pueden encontrarse en dos estados diferentes: activas o inactivas, según lo cual alteran el comportamiento por defecto de una parte del teclado. Los primeros teclados contaban con dos teclas **Mayús**, una tecla **Control** y una tecla **Alt**, mientras que los actuales disponen además, en la parte derecha, de una segunda tecla **Control** y una segunda tecla **Alt**. El estado de las primeras se obtiene mediante el servicio 02h y el de las segundas con el servicio 12h. No es necesario ningún parámetro de entrada y el estado de las distintas teclas se devuelve en AL o AH como un conjunto de bits. En el caso del servicio 02h la información devuelta en AL estará formada por los bits siguientes:

- **Bit 0:** A1 si está pulsada la tecla **Mayús** derecha.
- **Bit 1:** A1 si está pulsada la tecla **Mayús** izquierda.
- **Bit 2:** A1 si está pulsada la tecla **Control**.
- **Bit 3:** A1 si está pulsada la tecla **Alt**.
- **Bit 4:** A1 si está activa la tecla **Bloq Despl**.
- **Bit 5:** A1 si está activa la tecla **Bloq Num**.
- **Bit 6:** A1 si está activa la tecla **Bloq Mayús**.
- **Bit 7:** A1 si está pulsada la tecla **Insert**.

Por su parte, el servicio **12h** devuelve en AH el siguiente conjunto de bits.

- **Bit 0:** A1 si está pulsada la tecla **Control** izquierda.
- **Bit 1:** A1 si está pulsada la tecla **Alt** izquierda.
- **Bit 2:** A1 si está pulsada la tecla **Control** derecha.
- **Bit 3:** A1 si está pulsada la tecla **Alt** derecha.
- **Bit 4:** A1 si está activa la tecla **Bloq Despl**.
- **Bit 5:** A1 si está activa la tecla **Bloq Num**.
- **Bit 6:** A1 si está activa la tecla **Bloq Mayús**.
- **Bit 7:** A1 si está pulsada la tecla **PetSis**.

Nota

El servicio 12h devuelve en AL el mismo conjunto de indicadores que facilita el servicio 02h, de tal manera que mediante una llamada puede obtenerse el estado de todas las teclas.

El siguiente programa muestra cómo usar estos servicios para comprobar el estado de la mayor parte de las teclas muertas y de doble estado, representándolo gráficamente en la pantalla.

Para facilitar el trabajo se han definido al inicio dos macros y una tabla que contiene todos los datos necesarios para dibujar los recuadros, saber qué servicio hay que usar, qué máscara aplicar para comprobar el bit y cuál es el texto de la tecla. En la figura 19.3 puede verse el programa en funcionamiento.

```
; Esta macro coloca el cursor
; en el punto indicado sin
; modificar registros

%macro PosicionCursor 2
    push ax ; guardar AX
    push bx ; BX y DX
    push dx

    ; posición del cursor
    mov di, %1
    mov dh, %2

    ; Ponemos el cursor en la
    ; posición indicada
    xor bh, bh
    mov ah, 2
    int 10h

    ; restauramos registros
    pop dx
    pop bx
    pop ax
%endmacro

; Esta macro escribe el carácter
; indicado en la posición actual,
; tantas veces como diga el
; segundo parámetro y con el
; atributo del tercero
.

%macro EscribeCaracter 3
    push ax ; guardamos
    push bx ; registros
    push ex
```

```

; establecemos
; carácter y atributo
mov bl, %3
mov al, %1
xor bh, bh
mov ex, %2

; escribimos el carácter
mov ah, 9
int 10h
; recuperamos registros
pop ex
pop bx
pop ax
%endmacro

; Segmento de la pila
segment Pila stack
resw 512
FinPila:

; Segmento de datos
segment Datos

; Constantes que representan
; los caracteres para
; dibujar recuadros en
; modo de texto
EsqSI equ 218
EsqII equ 192
EsqSD equ 191
EsqID equ 217
Horz equ 196
Vert equ 179

; Constantes que identifican
; la posición de cada dato
; en la tabla siguiente
PosX equ 0
PosY equ 1
Servicio equ 2
Mascara equ 3
Longitud equ 4
Texto equ 5

; Tamaño, en bytes, de cada
; elemento de la tabla siguiente
TamanoTecla equ 15

; Tabla con 9 elementos, correspondientes
; a las 9 teclas a comprobar, conteniendo
; cada uno de ellos la posición X,Y donde
; se dibujará el recuadro, el servicio
; a usar, la máscara de bits a emplear
; para comprobar la tecla, la longitud del
; texto a mostrar y el texto propiamente dicho

```

```
Teclas db 5, 10, 2, 64, 10, 'Rioq Mayús'
      db 7, 15, 2, 2, 5,     'Mayús
      db 10, 20, 12h, 1, 7,   "Control
      db 20, 20, 12h, 2, 3,   'Alt
      db 50, 20, 12h, 8, 3,   'Alt
      db 60, 20, 12h, 4, 7,   'Control
      db 6b, 15, 2, 1, 5,     'Mayús
      db 45, 4, 2, 16, 10,   'Bloq Hespí'
      db 63, 4, 2, 32, 8,    'Bloq Núm •
```

Atributo db 0 ; para almacenar el atributo

```
; Segmento de código
segment Código
..start:
; Configuramos la pila
muv ¿ix, Pila
mov ss, ax
mov sp, FinPila

; DS apunta al segmento
; que contiene los datos
mov ax, Datos
mov ds, ax

; Limpiamos la pantalla
cali LimpiaPantalla

; y ocultamos el cursor
; para que no aparezca
; saltando por la pantalla
cali OcultaCursor

Bucle0: ; Bucle principal

; DS:BX apunta al primer
; elemento de la tabla
mov bx, Teclas
; hay 9 elementos
mov ex, 9

Bucle1: ; Bucle que recorre los elementos

; Comprobamos si está pulsada
; la tecla del elemento
cali CompruebaTecla

; dibujamos su marco
cali DibujaMarco

; e imprimimos su texto
cali ImprimeTexto

; pasar al siguiente
; elemento de la tabla
add'bx, TamanoTecla
```

```
; si se ha pulsado Esc
cali ExaminaTeclado

; terminamos
je Salir

loop Bucle1 ; repetir

; cuando terminemos con los
; 9 elementos volver al
; principio
jmp Bucle0

Salir: ; al salir

; volvemos a mostrar el cursor
cali RestauraCursor

; devolver el control al sistema
mov ah, 4ch
int 21h

; Esta rutina borra todo el
; contenido de la pantalla

LimpiaPantalla:
; Ponemos el cursor en la
; esquina superior izquierda
PosicionCursor 0, 0

; y escribimos 2000 espacios
EscribeCaracter ' ', 2000, 7

ret ; volver

; Esta rutina examina el teclado
; para ver si hay alguna tecla
; pulsada

ExaminaTeclado:
; comprobamos si hay
; alguna tecla pulsada
mov ah, 1
int 16h

; Si Z está a 1 es que no
jz NoHayTeclas

; de haberla la extraemos
xor ah, ah
int 16h

; comprobar si es Esc
emp al, 27
```

```
; de no ser así saltar
jne NoEsESC

; en caso contrario
; activar el carry
stc

ret , - y volver

NoHayTeclas:
    xor ax, ax

NoEsESC:
    ele ; poner a 0 el carry
    ret ; volver

; Esta rutina comprueba si está
; pulsada o activa una cierta
; tecla.
; DS:BX apunta al elemento de
; la tabla cuya tecla hay que
; comprobar

CompruebaTecla:
    ; obtenemos el servicio
    mov ah, [bx+Servicio]
    int 16h ; y lo invocamos

    ; si el servicio no es el 12
    emp byte [bx+Servicio], 12h
    jnc Continua ; saltamos

    ; en caso contrario intercambiamos
    ; para quedarnos con los bits en AL
    xchg ah, ai

Continua:
    ; usamos la máscara para
    ; comprobar el bit
    and al, [bx+Mascara]
    ; si está pulsada saltamos
    jnz Pulsada

    ; si no está pulsada dibujamos
    ; en blanco sobre negro
    mov byte [Atributo], 7

    ret ; volver

Pulsada:
    ; si la tecla está pulsada
    ; dibujamos en blanco sobre azul
    mov byte [Atributo], lfh

    ret ; volver
```

```
; Esta rutina dibujará el marco  
; según los parámetros del  
; elemento apuntado por BX
```

DibujaMarco:

```
pusha ; guardamos registros  
  
; recuperamos la posición  
; X, Y en AL,AH  
mov al, [bx+PosX]  
mov ah, [bx+PosY]  
  
; el atributo en DL  
mov di, [Atributo]  
; y la longitud en CX  
mov el, [bx+Longitud]  
xor ch, ch  
  
,• Nos ponemos en la esquina  
; superior izquierda  
PosicionCursor al, ah  
; y escribimos la esquina  
EscribeCaracter EsqSI, 1, di  
  
; pasamos a la linea siguiente  
inc ah  
; para dibujar la linea vertical  
PosicionCursor al, ah  
EscribeCaracter Vert, 1, di  
  
; y a la siguiente  
inc ah  
; para escribir la esquina inferior  
PosicionCursor al, ah  
EscribeCaracter EsqII, 1, di  
  
; pasamos a la siguiente columna  
inc al  
; y dibujamos la linea  
; horizontal inferior  
PosicionCursor al, ah  
EscribeCaracter Horz, ex, di  
  
; volvemos a la linea superior  
sub ah, 2  
; para dibujar el otro borde  
PosicionCursor al, ah  
EscribeCaracter Horz, ex, di  
  
; nos vamos a la última  
; columna del recuadro  
add al, el  
; para dibujar una esquina  
PosicionCursor al, ah  
EscribeCaracter EsqSD, 1, di
```

```
; línea siguiente
inc ah
; escribimos borde vertical
PosicionCursor al, ah
EscribeCaracter Vert, 1, di

; y en la línea siguiente
inc ah
; la esquina inferior derecha
PosicionCursor al, ah
EscribeCaracter EsqID, 1, di

popa ; recuperamos registros

ret ; y volvemos

; Esta rutina imprime el texto
; del elemento apuntado por BX

ImprimeTexto:
pusha ; guardamos registros

; obtenemos columna
mov di, [bx+PosX]
; y la incrementamos
inc di
; obtenemos línea
mov dh, [bx+PosY]
; y la incrementamos
inc dh
; obtenemos longitud en CX
mov el, [bx+Longitud]
xor ch, ch

; SI apuntará al texto
mov si, bx
add si, Texto
; tomamos el atributo en BL
mov bl, [Atributo]

Buclel0: ; Bucle de escritura

lodsb ; leemos un carácter
; colocamos el cursor
PosicionCursor di, dh
; y lo escribimos
EscribeCaracter al, 1, bl

; pasamos a la siguiente columna
inc di
loop Buclel0 ; y repetimos

popa ; recuperamos registros

ret ; volver
```

```
; Esta rutina oculta el cursor  
  
OcultaCursor:  
    ; activamos el bit 6  
    ; de ch  
    mov ch, 64  
    xor el, el  
  
    ; y usamos el servicio  
    ; para definir el aspecto  
    ; del cursor  
    mov ah, 1  
    int 10h  
  
    ret ; volver
```

```
; Esta rutina restaura el cursor
```

```
RestauraCursor:  
    ; devolvemos la apariencia  
    ; por defecto al cursor  
    mov ch, 6  
    mov el, 7  
    mov ah, 1  
    int 10h  
  
    ret ; volver
```

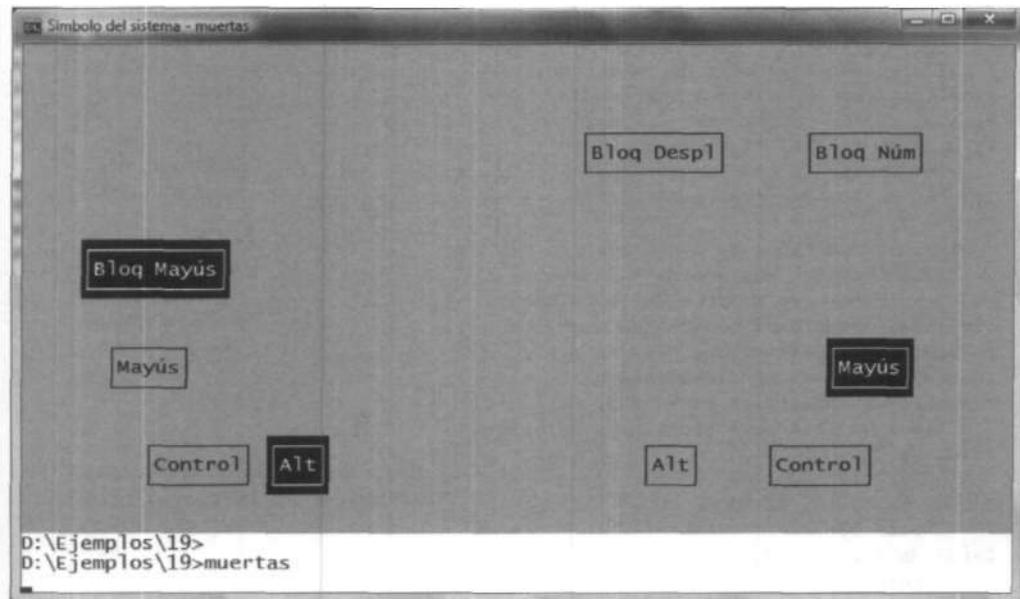


Figura 19.3. El programa muestra gráficamente el estado de las teclas.

Obtención de cadenas de caracteres

Ninguno de los servicios de la BIOS facilita la recuperación de más de un carácter, por lo que la solicitud de información más compleja por teclado, por ejemplo cadenas de caracteres, puede resultar un tanto tedioso. Por suerte, y aunque no pertenece a la BIOS, el DOS sí que cuenta con un servicio que simplifica dicha tarea que, de otra forma, deberíamos codificar manualmente.

Este servicio es el OAh de la interrupción 21h. Tan sólo se precisa un parámetro: la dirección, en los registros DS : DX, de un área de memoria que contendrá un primer byte indicando cuántos caracteres deben recogerse como máximo y, a continuación, una zona de libre que tenga tantos bytes como indica el primero más 1. Tras invocar al servicio, encontraremos en DS : DX+1 un byte que nos permitirá saber cuántos caracteres se han recuperado finalmente, caracteres que se encontrarán a partir de la siguiente posición de memoria.

Podríamos utilizar este servicio para recibir, como hace el programa siguiente, múltiples datos a través del teclado. En este caso el programa no hace nada con esos datos, pero podría escribirlos en un archivo en disco para recuperarlos posteriormente, o bien almacenarlos en una matriz en memoria. El código del programa es el siguiente:

```
; Segmento de la pila
segment Pila stack
    resw 512

FinPila:

; Segmento de datos
segment Datos

; Las siguientes cadenas de caracteres
; solicitarán los datos a introducir
Titulol db 10, 8, 'Nombre ....: $'
Titulo2 db 10, 10, 'Apellidos ..: $'
Titulo3 db 10, 12, 'Dirección ..: 5'
Titulo4 db 10, 14, 'Teléfono ..: 5'

; La siguiente tabla de datos servirá
; para solicitar cada uno de los datos.
; Cada elemento se compone de la columna
; y línea donde debe pedirse el dato, un
; byte con la máxima longitud del dato,
; otro byte donde se almacenará el
; número de caracteres pedidos y, por
;- último, un área para alojar los caracteres
Datol db 23, 8, 10, 0
    resb 10
Dato2 db 23, 10, 15, 0
    resb 15
Dato3 db 23, 12, 20, 0
    resb 20
Dato4 db 23, 14, 13, 0
    resb 13'
```

```
; Segmento de código
segment Código
..start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; DS apunta al segmento
    ; que contiene los datos
    mov ax, Datos
    mov ds, ax

    ; borramos el contenido
    ; de la pantalla
    cali LimpiaPantaila

    ; mostramos los textos
    cali MuestraTextos

    ; pide los datos
    cali PideDalos

    ; salir al sistema
    mov ah, 4ch
    int 21h

: Esta rutina pide los datos

PideDatos:
    ; DX apunta al primero
    mov dx, Dato1
    ; pedir el dato
    cali Entrada

    ; repetir
    mov dx, Dato2
    cali Entrada

    mov dx, Dato3
    cali Entrada

    mov dx, Dato4
    cali Entrada

    ret ; volver

; Esta rutina recibe en DX la
; dirección de uno de los elementos
; de la tabla de datos y lo
; solicita

Entrada:
    pusha ; guardar registros
```

```

; tomamos la dirección en BX
mov bx, dx
; e incrementamos DX para
; acceder a la longitud
inc dx
inc dx
push dx ; lo guardamos

; tomamos la columna
mov di, [bx]
; incrementamos la dirección
inc bx
; y leemos la linea
mov dh, [bx]
xor bh, bh
; colocamos el cursor
mov ah, 2
int 10h

; recuperamos la dirección
pop dx
; y la pasamos a bx
mov bx, dx

; tomamos la longitud en ex
mov el, [bx]
xor ch, ch
; quitamos el Intro final
dec ex
; carácter de subrayado
mov al, '_'
; atributo normal
mov bl, 7
xor bh, bh
; imprimimos la secuencia
; de subrayados
mov ah, 9
int 10h

; pedimos el dato apuntado
; por DS;DX
mov ah, 0Ah
int 21h

popa ; recuperar registros

ret ; volver

; Esta rutina muestra los textos

MuestraTextos:
; cuatro textos
mov ex, 4
; DX apunta a la primera cadena
mov dx, TituloL

```

```

BucleMO:
; mostramos cadena
cali Imprimir
; pasamos a la siguiente
add dx, 16
; y repetimos
loop BucleMO

ret ; volver

; Esta rutina recibe en DX la
; dirección de uno de los elementos
; y lo muestra en la posición
; adecuada

Imprimir:
pusha ; guardar registros

; movemos la dirección a BX
mov bx, dx
; y guardamos DX
push dx

; leemos la columna
mov di, [bx]
; incrementamos la dirección
inc bx
; y tomamos la linea
mov dh, [bx]
xor bh, bh
; colocamos el cursor
mov ah, 2
int 10h

; recuperamos la dirección
; de la cadena
pop dx
; incrementamos para
; apuntar al texto
inc dx
i no dx
; y lo imprimimos
mov ah, 9
int 21h

popa ; recuperar registros

ret ; volver

; Esta rutina limpia la pantalla

LimpiaPantalla:
xor dx, dx ; ponemos el cursor
xor bh, bh ; en la esquina superior

```

```

mov ah, 2 ; izquierda
int 10h

; para introducir 2000
; espacios
mov bl, 7
mov al, ' '
mov ex, 2000
mov ah, 9
int 10h

ret ; volver

```

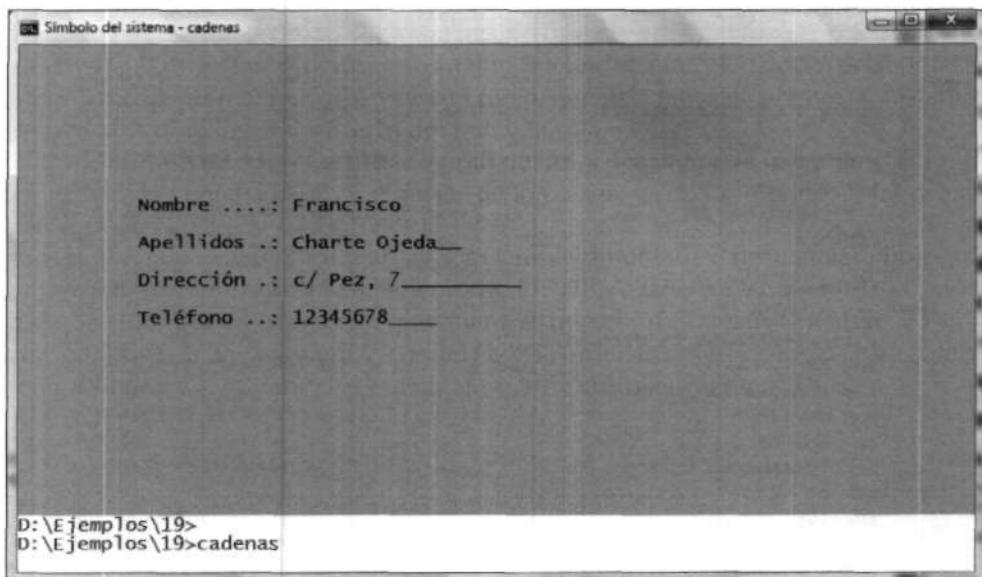


Figura 19.4. Aspecto de la pantalla tras introducir algunos datos.

Resumen

Como ha podido ver en este capítulo, basta conocer tres servicios de la interrupción 10h, y el servicio 0Ah de la interrupción 21h descrito en el último punto, para obtener toda la información que se necesite desde el teclado. Esa información puede ser tan breve como un bit, identificando el estado de una cierta tecla muerta, o tan extensa como una cadena de hasta 255 bytes.

Sabiendo ya cómo interactuar con pantalla y teclado, el paso siguiente, en el capítulo que tiene a continuación, será comunicarnos con la impresora, enviando información que deseamos obtener en papel. Los servicios de la BIOS, como tendrá ocasión de ver, también son muy sencillos en este caso.

20

Acceso a la impresora

La mayoría de las aplicaciones que están escritas en ensamblador tienen como objetivo obtener el mejor rendimiento posible, de ahí que se opte por un lenguaje de tan bajo nivel en detrimento de otros en los cuales sería más fácil conseguir la misma funcionalidad.

Las tareas de impresión, especialmente preparación de informes y similares, no se distinguen por precisar esa velocidad y, además, las actuales herramientas de desarrollo incorporan diseñadores visuales que permiten crear informes a partir de datos sin necesidad de escribir ni una sola línea de programa.

Desde ensamblador, no obstante, también podemos acceder a la impresora para enviarle información. La interrupción BIOS encargada de facilitar los servicios para ello es la 17h.

Los servicios disponibles son sólo tres, facilitando la iniciación de la impresora, el envío de información y recuperación del estado del dispositivo. Su uso, en general, es muy simple.

Actualmente todas las impresoras cuentan con lenguajes, más o menos complejos, mediante los cuales se describe el formato, distribución y contenido de las páginas a imprimir.

Algunos ejemplos son los lenguajes PCL y Postscript. En este capítulo nos centraremos en el estudio de los servicios de la BIOS, sin entrar en la comunicación con la impresora usando un lenguaje en particular. Nos limitaremos, por tanto, al envío de caracteres que se espera que la impresora pase a papel, como hacen la mayoría de los dispositivos de impresión.

Iniciación y estado de la impresora

Normalmente, el primer paso que daremos antes de acceder a la impresora será iniciar el puerto por el que nos vamos a comunicar con ella y comprobar su estado.

Los servicios implicados son el 01h y **02h** de la interrupción **17h**. En ambos casos deberemos facilitar, en el registro DX, el número de puerto.

Normalmente éste será 0, ya que la mayor parte de los sistemas actuales sólo disponen de un conector de impresora, conocido normalmente como LPT1: y que se corresponde con el número 0.

El primer servicio, el 01h, establece el puerto en su estado inicial y, además, devuelve en AH un conjunto de bits indicando el estado de la impresora. El servicio 02h obtiene ese mismo conjunto de bits, en el mismo registro, pero sin efectuar el proceso de iniciación. Es el servicio a utilizar, por tanto, siempre que deseemos comprobar si la impresora está preparada sin necesidad de iniciarla, proceso que, en ocasiones, puede provocar la pérdida de información en caso de que hubiese datos pendientes de imprimir.

Los bits de estado devueltos en el registro AH tienen el significado siguiente:

- **Bit 0:** Estará a 1 si se superó el tiempo de espera para el envío de un dato.
- **Bit 1:** Sin uso.
- **Bit 2:** Sin uso.
- **Bit 3:** Estará a 0 si existe algún error de entrada/salida.
- **Bit 4:** Estará a 1 en caso de que la impresora esté preparada esperando.
- **Bit 5:** Estará a 1 si la impresora no tiene papel.
- **Bit 6:** Reconocimiento.
- **Bit 7:** Está a 1 siempre que la impresora no esté ocupada.

Como es habitual, puede emplear la instrucción and, o test, para comprobar el estado de cada bit individual.

Envío de información a la impresora

Tan sólo tenemos un servicio para el envío de datos a la impresora: el servicio 00h. Además, los datos sólo pueden ser caracteres individuales, no existiendo, en la BIOS, otras posibilidades, aunque puede recurrirse, como en el caso del teclado, a los servicios que ofrece el DOS o el sistema operativo sobre el que vayamos a trabajar.

Al invocar al servicio 00h debemos facilitar dos datos: el puerto de impresora, en DX, y el carácter a imprimir, en AL. Éste, normalmente, será el código ASCII que ya conocemos, aunque dependiendo de la impresora ciertos caracteres podrían tener un significado diferente.

Otra posibilidad, a la hora de enviar caracteres a la impresora, consiste en utilizar el servicio 05h de la interrupción 21h. Ésta espera el carácter a imprimir en el registro DL, enviándolo a la primera impresora conectada al sistema, por lo que no es necesario indicar el número de puerto.

Utilizando tan sólo el servicio 00h de la interrupción 17h podemos, como se muestra en el ejemplo siguiente, imprimir una tabla de códigos ASCII, un recurso que nunca viene mal tener a mano cuando se programa en ensamblador.

Tenga en cuenta que el conjunto de caracteres puede no ser el ASCII dependiendo de la impresora que use.

Las antiguas matriciales tienen el mismo conjunto de caracteres existente en los PC, pero no así las modernas impresoras láser y de inyección de tinta.

```
; EsLd macro imprime el carácter
; entregado como parámetro sin
; modificar registros
%macro ImprimeCaracter 1
    push ax ; guardamos AX
    push dx ; y DX

    ; puerto 0
    xor dx, dx
    ; tomamos el carácter
    mov al, %1
    ; servicio 0
    xor ah, ah
    int 17h ; imprimimos

    pop dx ; recuperar
    pop ax ; registros
%endmacro

segment Pila stack
    resw 512
FinPila:

segment Datos
; Titulo a imprimir al principio
Mensaje db 'CÓDIGOS ASCII', 0
; Esta variable contará las columnas
Columna db 0

; Segmento de código
segment Código
..start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila
```

```
; DS apunta al segmento
; que contiene los datos
mov ax, Datos
mov ds, ax

; imprime el título
cali ImprimeTitulo

; imprimimos un margen
cali Espacios

; comenzamos en la
; columna 10
mov byte [Columnna], 10

; comenzar con el
; código 32 (espacio)
mov al, 32

; número de caracteres
; a imprimir
mov ex, 256-32

Bucle2:
    ; imprimirmos el carácter
    ; que corresponda
    imprimeCaracter al

    ; imprimimos el espacio
    ImprimeCaracter ' '

    ; incrementar AL
    inc al

    ; incrementamos la columna
    inc byte [Columnna]
    ; ¿la columna es la 35?
    emp byte [Columnna], 35

    ; de no ser así seguimos
    jne NoAvance

    ; avanzamos de línea
    ImprimeCaracter 10
    ImprimeCaracter 13

    ; imprimimos un margen
    cali Espacios

    ; comenzamos de nuevo
    ; en la columna 10
    mov byte [Columnna], 10

NoAvance:
    ; y volvemos al primer bucle
    loop Bucle2
```

```

Fin:
    ; terminamos la última linea
ImprimeCaracter 10
ImprimeCaracter 13

    ; y provocamos el avance de página
ImprimeCaracter 12

    ; y salimos
mov ah, 4ch
int 21h

; Esta rutina imprime un titulo
; al principio

ImprimeTitulo:
    ; Imprimimos un margen
    ; cali Espacios

        ;* 31 apunta al mensaje
    mov si/ Mensaje
    cid

    ; DX tiene el puettb
    xor dx, dx
BucleTO:
    lodsb ; leemos un carácter
    or al, al ; Si llcqamoe al fin
    jz FinTitulo ; saltamos

    ; en caso contrario
    ; imprimimos el carácter
    xor ah, ah
    int 17h

    ; seguimos
    jmp BucleTO

FinTitulo:
    ; Saltamos a la linea siguiente
    ImprimeCaracter 10
    ImprimeCaracter 13

    ret ; volver

; Esta rutina imprime un bloque de
; espacios que sirve como margen
;
Espacios:
    push ax ; guardamos
    push ex ; registros

        ;- imprimir 10 espacios
    mov al, ' '

```

```

xor dx, dx
mov ex, 10

BucleEO:
xor ah, ah ; vamos imprimiendo
int 17h ; hasta CX=0
loop BucleEO

pop ex ; recuperar
pop ax ; registros

ret ; y volver

```

De manera análoga, siguiendo pasos similares a los que muestra este programa, podría imprimir cualquier otra información.

Puertos mejorados de impresora

Aunque en sus inicios los puertos de impresora eran básicamente de salida, permitiendo poco control sobre el dispositivo conectado que, generalmente, era una impresora, posteriormente aparecieron mejoras como EPP (*Enhaced Parallel Port*), que aprovechaban líneas del conector para ofrecer funciones adicionales. Como probablemente sabrá, estos puertos permiten, por ejemplo, conectar dispositivos que no son impresoras, como unidades de discos magneto-ópticos e, incluso, facilitan la comunicación entre ordenadores a través de los puertos de impresora.

Los ordenadores que cuentan con puertos de impresora EPP normalmente instalan una BIOS adicional, extendida, que permite la comunicación con este tipo de puertos. En ella existen servicios no tan sólo para el envío de caracteres individuales, sino también para el envío de bloques de caracteres, lectura de bytes individuales y de bloques de bytes. Es decir, tenemos servicios de comunicación en ambas direcciones, como si de un puerto serie se tratase.

Para acceder a los servicios de la BIOS EPP no puede utilizarse la interrupción 17h, ni ninguna otra interrupción de la BIOS estándar. Esa BIOS extendida se instala en una cierta dirección de memoria que es necesario obtener, utilizándola a partir de ese momento para efectuar llamadas mediante la instrucción `cali`.

Con el fin de comprobar si la BIOS EPP está instalada, y obtener su dirección en caso afirmativo, nos serviremos del servicio 02h de la interrupción 17h. Son necesarios varios parámetros que facilitaremos en los registros siguientes:

- AL: Debe estar a 0.
- DX: Indicará el número de puerto.
- CH: Debe contener el carácter 'E'.
- BL: Debe contener el carácter 'P'.
- BH: Debe contener el carácter 'P'.

Tras invocar a la interrupción, para saber si la BIOS EPP está o no disponible tendremos que asegurarnos de que:

- AH contiene el valor 0.
- AL contiene el carácter 'E'.
- CL contiene el carácter 'P'.
- CH contiene el carácter 'P'

Si estas condiciones se cumplen, encontraremos en la pareja de registros DX: BX la dirección del punto de entrada a la BIOS EPP, lo se conoce normalmente como el *vector EPP*, que, a partir de ese momento, usaríamos para invocar a los servicios. El siguiente programa comprueba si está o no instalada la BIOS EPP y lo indica mostrando un mensaje en la consola, aunque no llega a efectuar llamada alguna a sus servicios. Si desea información sobre éstos, para poder aprovechar las posibilidades de los puertos EPP, en la dirección <http://www.atase.com/main/tools/io/eppbios3.pdf> encontrará un documento PDF con la especificación de este tipo de puertos, incluyendo una descripción de todos los servicios y parámetros que necesitan.

```
segment Pila stack
    resw 512
FinPila:

segment Datos
MsgHayEPP db 'Está instalada la BIOS EPP$'
MsgNoHayEPP db 'No está instalada la BIOS EPP5'

; Segmento de código
segment Código
.start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; DS apunta al segmento
    ; que contiene los datos
    mov ax, Datos
    mov ds, ax

    ; Servicio 2 con
    ; AL a 0
    mov ax, 200h
    ; BH y BL tienen P
    mov bx, 'PP'
    ; CH tiene E
    mov ch, 'E'
    ; comprobamos el
    ; puerto 0
    xor dx, dx
    ; llamamos
    int 17h
```

```

; comprobamos que
; AH sea cero
or ah, ah
jnz NoHayEPP

; que AL contenga 'E'
cmp al, 'E'
jnz NoHayEPP

; y que CX tenga 'PP'
cmp ex, 'PP'
jnz NoHayEPP

; Si se pasan tocas las
; condiciones es que
; existe la BIOS EPP
mov dx, MsgHayEPP
jmp Salir

NoHayEPP:
; Si no se cumple alguna
; de las condiciones es
; que no existe la DIOS EPP
mov dx, MsgNoHayEPP

Salir:
; mostramos el mensaje
mov ah, 9
int 21h

; y salimos
mov ah, 4ch
int 21h

```

Resumen

Este tercer capítulo dedicado al estudio de los servicios que nos permiten comunicarnos con los distintos periféricos del PC se ha centrado en la comunicación con la impresora, un dispositivo que, en la mayoría de los casos, tan sólo recibe información si bien es capaz de facilitar un retorno sobre el estado, comunicando incidencias como la falta de papel. Como ha visto, es suficiente con un servicio de la BIOS para imprimir lo que se desee. En este caso la velocidad es algo totalmente secundario, puesto que el puerto de impresora, y el propio dispositivo conectado a ella, no pueden ir a la velocidad a la que el procesador ejecuta las instrucciones.

También se ha mencionado la existencia de los puertos EPP, mediante los cuales es posible conectar a los puertos de impresora otro tipo de dispositivos. Se ha mostrado cómo detectar la presencia de la BIOS EPP, cuyos servicios son igualmente fáciles de usar y hacen posible tanto el envío como la recepción de información.

21

Joystick y ratón

Si bien el teclado es el dispositivo principal para la introducción de datos en la mayoría de las aplicaciones, hay programas en los que puede resultar más adecuado el uso de medios alternativos para que el usuario tome el control. Dos de los dispositivos más usuales son el ratón y las palancas de juegos o *joysticks*. El primero facilita el desplazamiento rápido por la pantalla, permitiendo seleccionar opciones, dibujar, desplazar un contenido, etc. Los joystick, en sus diversas formas, aportan mayor interactividad a la hora de ejecutar juegos.

Originalmente los ratones se conectaban a los PC mediante la entrada de comunicación serie (RS-232), posteriormente aparecieron los ratones con conector PS/2, más adelante los que usan el bus USB, etc. Como es lógico pensar, la comunicación con un ratón diferirá según el tipo de conexión que utilice.

Ésta es la razón de que desde los tiempos de DOS se utilicen controladores, inicialmente un programa que quedaba residente en memoria denominado MOUSE.COM o MOUSE.SYS, que ofrecen a las aplicaciones una interfaz homogénea para utilizar el ratón, indistintamente de las peculiaridades de éste.

En cuanto a los joystick, desde los primeros PC siempre se han conectado a través de un puerto específico de 15 pines que permitía usar uno o dos dispositivos simultáneamente. En la actualidad también se conectan a través del bus USB o bien a través de tecnología inalámbrica.

El objetivo de este capítulo es mostrarle cómo puede utilizar desde programas escritos en ensamblador estos dos dispositivos, sirviéndose para ello de interrupciones que forman parte de la BIOS del PC o bien son agregados por software externo, como es el caso del controlador de ratón antes mencionado.

Uso del ratón

Actualmente todos los ordenadores cuentan con un ratón, o dispositivo equivalente, dada la proliferación de las interfaces gráficas que han ido superponiéndose a todos los sistemas operativos.

Antes de que esto ocurriera, en los tiempos de las interfaces exclusivamente de texto, los ratones no resultaban tan habituales. Ésta es la razón de que en la BIOS los PC no cuenten con servicios específicos para su uso.

La arquitectura extensible de estos sistemas, sin embargo, simplificó la adición de dichos servicios usando para ello un mecanismo idéntico al que hemos conocido en capítulos previos: a través de interrupciones.

Al ejecutar el programa MOUSE.COM, o bien cargar el controlador MOUSE.SYS desde el archivo de configuración CONFIG.SYS, se utiliza uno de los vectores de interrupción que la BTOS deja libre para instalar los servicios del ratón.

Dicho vector es el 33h, por lo que para invocar a cualquier función de estos servicios se asignará su código al registro AX y, a continuación, se usa la instrucción int 33h para ejecutarla.

La tabla 21.1 enumera parte de las funciones que ofrece el controlador de ratón, indicando el código, que se introduciría en el registro AX, y su finalidad. En los siguientes apartados pondremos en práctica algunas de ellas.

Tabla 21.1. Funciones de la interrupción 33h.

Código	Finalidad
00h	Inicializar el ratón.
01h	Mostrar el puntero del ratón.
02h	Ocultar el puntero del ratón.
03h	Obtener información de los botones y posición del puntero.
04h	Establecer la posición del puntero.
05h	Recuperar contador de pulsaciones de botón.
06h	Recuperar contador de liberaciones de botón.
07h	Limitar horizontalmente el desplazamiento del puntero.
08h	Limitar verticalmente el desplazamiento del puntero.
09h	Establecer apariencia del puntero en modo gráfico.
0Ah	Establecer apariencia del puntero en modo texto.
0Bh	Lectura de los contadores de desplazamiento.
0Ch	Instalar gestor de ratón.

Detección e inicialización

Lo primero que debe hacer una aplicación que pretenda utilizar el ratón es inicializarlo, operación que, al tiempo, también le permitirá detectar su existencia y saber con cuántos botones cuenta.

El proceso es realmente sencillo y constaría de los pasos siguientes:

- Se introduce el valor 00h en el registro AX, código de la función que nos interesa ejecutar.
- Invocamos a la interrupción 33h.
- Comprobar que el registro AX contiene el valor FFFFh. De ser así la aplicación sabrá que hay un ratón instalado, de lo contrario no debería intentar seguir usando los servicios.
- En el registro BX se obtiene el número de botones con que cuenta el ratón.

Un programa simple, tal como el que se muestra a continuación, completaría dicho proceso.

Ejecutándolo desde DEBUG (véase la figura 21.1) podemos ver que en el sistema hay instalado un ratón y que cuenta con dos botones.

```
segment Pila stack
    res» 512
FinPila:

        ; Segmento de código
        segment Código
.start:
        ; Configuramos la pila
        mov ax, Pila
        móv ss, ax
        mov sp, FinPila

        ; Servicio 00h
        xor ax, ax
        int 33h

        ; Salimos al sistema
        mov ah, 4ch
        int 21h
```

La inicialización del ratón, a través de este servicio, implica también la colocación del puntero en el centro de la pantalla y su ocultación.

```

Símbolo del sistema - debug hayraton.exe
matched ComDefs
D:\Ejemplos\21>debug hayraton.exe
-r
AX=0000  BX=0000  CX=0260  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1831  IP=0000  NV UP EI PL NZ NA PO NC
1831:0000 B8F117    MOV     AX,17F1
-p

AX=17F1  BX=0000  CX=0260  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1831  IP=0003  NV UP EI PL NZ NA PO NC
1831:0003 8ED0    MOV     SS,AX
-p

AX=17F1  BX=0000  CX=0260  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1831  IP=0008  NV UP EI PL NZ NA PO NC
1831:0008 31C0    XOR     AX,AX
-p

AX=0000  BX=0000  CX=0260  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1831  IP=000A  NV UP EI PL ZR NA PE NC
1831:000A CD33    INT     33
-p

AX=FFFF  BX=0002  CX=0260  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000
DS=17E1  ES=17E1  SS=17F1  CS=1831  IP=000C  NV UP EI PL ZR NA PE NC
1831:000C B44C    MOV     AH,4C

```

Figura 21.1. El programa detecta la existencia de un ratón en el sistema.

Control del puntero del ratón

Una vez que la aplicación sabe que hay un ratón disponible, lo primero que tendrá que hacer será mostrar el puntero del ratón, colocarlo donde necesite la aplicación en cada momento y, en ocasiones, ajustar la apariencia de ese cursor o volver a ocultarlo.

El software que controla el ratón cuenta con un contador interno que determina la visibilidad del puntero.

Siempre que este contador sea mayor o igual a cero el puntero podrá verse, mientras que si es inferior a cero permanecerá oculto. Cuando se inicializa el ratón, a través del servicio OOh, se asigna el valor -1 a dicho contador.

Mediante los servicios Olh y 02h de la interrupción 33h M incrementa y reduce, respectivamente, el valor actual de ese contador. Esto implica que, inmediatamente tras la inicialización, una llamada al servicio Olh mostrará el puntero y una posterior llamada al servicio 02h lo ocultará.

Es posible, sin embargo, llamar varias veces consecutivas a uno u otro. Si el puntero está visible, con el contador a 0, y se usa de nuevo el servicio 0] h dicho valor se incrementará y pasará a ser 1. Esto implica que el puntero siga siendo visible y, además, para ocultarlo habría que recurrir dos veces al servicio 02h.

Conociendo el funcionamiento de estos servicios, podemos preparar dos pequeños programas, a los cuales llamaríamos incraton.asm y redraton.asm, que facilitasen el incremento y reducción del mencionado contador. El código del primero sería el mostrado a continuación, mientras que el segundo solamente diferiría en el código de función a invocar:

```

segment Pila stack
    resw 512
FinPila:

        ; Segmento de código
        segment Código
..start:
        ; Configuramos la pila
        mov ax, Pila
        mov ss, ax
        mov sp, FinPila

        ; Incrementar el contador
        mov ax, 0lh
        int 33h

        ; Salimos al sistema
        mov ah, 4ch
        int 21h

```

En caso de que ejecutemos el programa red ratón tras haber inicializado el ratón, como se ha hecho en la figura 21.2, para conseguir mostrar el puntero será preciso invocar dos veces al programa incraton. El puntero, como puede apreciarse en dicha figura, se encuentra situado inicialmente en el centro de la pantalla y aparece como un cursor de texto en forma de bloque.

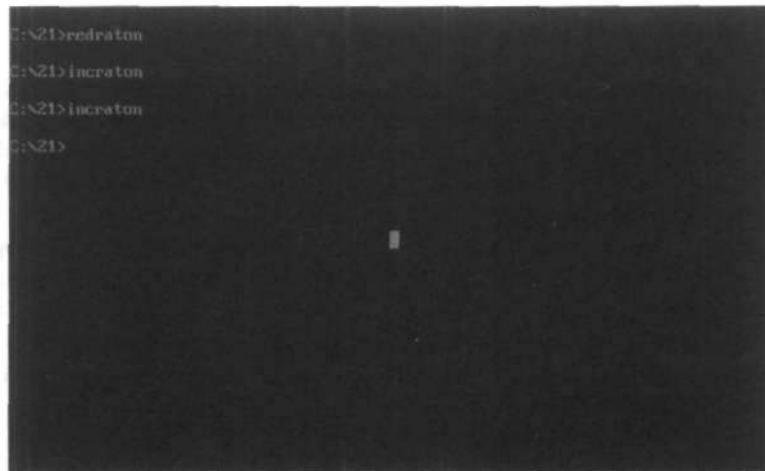


Figura 21.2. El puntero del ratón aparece en el centro de la pantalla.

Aunque tiene una apariencia similar al cursor de texto, el puntero del ratón es un elemento totalmente independientemente. En cuanto se haga visible en pantalla el usuario podrá desplazarlo, moviendo el ratón lógicamente, y su posición no afectará al punto en que se encuentre el cursor de texto en la línea de comandos o aplicación que se esté ejecutando.

La posición del puntero en pantalla también puede ser establecida por el propio programa que está utilizando los servicios del ratón. Al mostrar un menú, por ejemplo, la aplicación podría colocar el puntero del ratón en la primera de sus opciones, facilitando así el proceso de selección por parte del usuario. Para colocar el puntero en cualquier punto de la pantalla habrá que recurrir a la función 04h, facilitando en los registros CX y DX la coordenada horizontal y vertical, respectivamente. Debe tenerse en cuenta, no obstante, que dichas coordenadas se entregan siempre en píxeles, incluso cuando se está trabajando en modo texto.

Trabajando en el modo de texto habitual, el de 80 columnas por 25 líneas, cada carácter se dibuja en una celdilla de 8x8 píxeles. Esto implica que las coordenadas del puntero del ratón, en consecuencia, deberán ser multiplicadas/divididas por un factor de 8, tanto en sentido horizontal como vertical.

Suponiendo que tras mostrar el puntero del ratón deseásemos colocar éste en la décima columna de la octava fila, bastaría con agregar las sentencias siguientes antes de devolver el control al sistema:

```
,- Colocar el cursor en 10,8
mov ex, 10*8
mov dx, 5*8
mov ax, 04h
int 33h
```

Obviamente, el usuario podrá modificar de inmediato la posición fijada con esta función de la interrupción 33h. La única finalidad de establecer una posición concreta, según se indicaba antes, es ayudar al usuario en la selección de un nuevo elemento que aparece en pantalla o situaciones similares.

En este caso, puesto que se conocía en el momento de escribir el programa la posición donde iba a colocarse el puntero, ha sido posible introducir directamente en CX y DX la coordenada correcta dejando que el propio ensamblador efectúe la multiplicación por 8. En la práctica, sin embargo, las coordenadas habrá que calcularlas durante la ejecución, efectuando ese producto mediante desplazamiento de bits (instrucción shl) o bien con la instrucción muí.

Posición del puntero y estado de los botones

Por regla general a las aplicaciones les interesa más conocer la posición actual del puntero del ratón, fijada por el usuario al desplazar el dispositivo físico, que establecerla con el servicio indicado antes.

Asimismo resulta fundamental conocer el estado de los botones, al ser éstos los elementos que el usuario utilizará para indicar lo que quiere hacer.

La función 0 3h de la interrupción 3 3h devuelve en los registros BX, CX y DX información sobre el estado de los botones y la posición actual del puntero del ratón. El contenido concreto de estos registros será:

- BX: Sus tres bits de menor peso, los bits 0,1 y 2, indicarán el estado de los botones izquierdo, derecho y central, respectivamente. Un bit a 1 comunica que el botón se encuentra pulsado, mientras que el valor 0 corresponde a un botón no pulsado. El resto de los bits del registro no tienen en principio ninguna utilidad para el programador.
- CX: Este registro contendrá la coordenada horizontal de la posición actual del puntero del ratón.
- DX: Este registro contendrá la coordenada vertical de la posición actual del puntero del ratón.

La unidad de medida para la posición del puntero es el píxel, según se indicó anteriormente, por lo que si operamos en modo texto será preciso dividir el contenido de CX y DX entre 8 a fin de saber cuál es la columna y fila en que se encuentra.

Veamos en la práctica cómo utilizaríamos estos servicios en un programa, a través de un ejercicio cuya finalidad sería llenar la pantalla de un cierto carácter y, a continuación, dejar que el usuario, utilizando el ratón, altere sus atributos. Para ello se utilizará el botón izquierdo a fin de cambiar aleatoriamente el atributo, mientras que el botón derecho establecería el patrón habitual de blanco sobre negro.

Como se aprecia en el código siguiente, el programa consta de un bloque de inicialización, encargado de llenar la pantalla de asteriscos y comprobar la disponibilidad del ratón, y de un bucle principal en el que se consulta constantemente el estado del ratón. Ésta técnica se denomina *polling* y consiste en que sea la propia aplicación la que rastree periódicamente el dispositivo externo, en este caso el ratón. No es la mejor alternativa, pero funciona correctamente.

```
segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
..start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; ES apuntando a segmento de pantalla
    mov ax, 0b800h
    mov es, ax

    ; que llenamos de asteriscos
    mov al, '*'
```

```

mov ah, 70h
mov ex, 2000
xor di, di

rep stosw

; Servicio 00h
xor ax, ax
int 3 3h

; Seguir únicamente si hay ratón
emp ax, Offffh
jnz salir

,- Mostrar el puntero
mov ax, Qlh
int 33h

bucle: ; Recuperar posición y estado
mov ax, 03h
int 33h

; Si no hay bolones pulsados
or bx, bx
jz bucle ; no hacer nada más

; Si están pulsados los dos botones
emp bx, 03h
jz salir

; Dividir coordenadas entre 8
mov ax, ex
mov el, 3
shr ax, el
shr dx, el
mov ex, ax

; Calcular posición en memoria de pantalla
mov ax, dx ; Fila * 80 * 2
mov di, 160
muí di
add ax, ex ; + Columna * 2
add ax, ex

rmov di, ax
inc di ; Apuntar al atributo

; Incrementar el código de carácter
mov ai, [es:di]
inc al

; Si está pulsado el botón izquierdo
emp bx, 1
jz botonizq

mnv al, 07h ; Si es el derecho

```

botonizq:

```
stosb ; Cambiar el atributo  
jmp bucle
```

salir:

```
; Salimos al sistema  
mov ah, 4ch  
int 21h
```

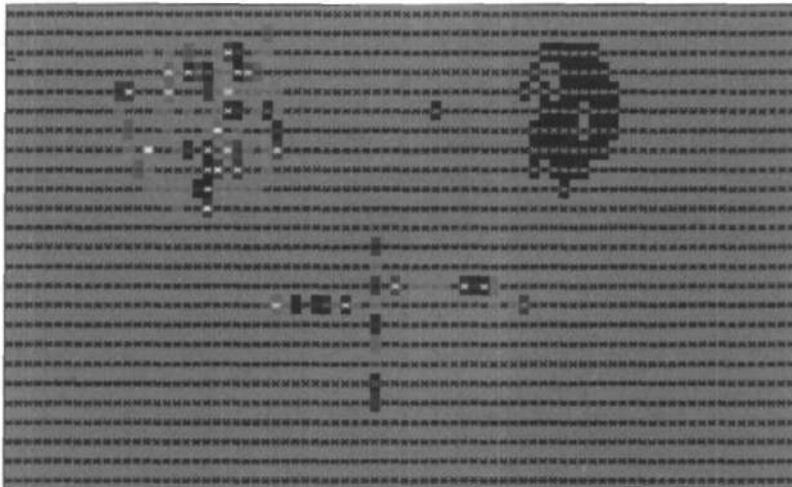


Figura 21.3. El programa en funcionamiento tras alterar el atributo de parte de los caracteres.

En realidad el botón izquierdo no asigna un valor aleatorio a los atributos, sino que incrementa el atributo actual en una unidad. En el tiempo en que se tiene pulsado el botón, sin embargo, el programa ejecutará el bucle un número indeterminado de veces, de ahí que el resultado parezca aleatorio.

Uno de los inconvenientes de la función 03h de los servicios del ratón, a la hora de determinar si el usuario ha pulsado o no un botón, es que si la exploración no se lleva a cabo con mucha frecuencia cabe la posibilidad de que se haya pulsado y liberado un botón entre llamada y llamada a dicha función.

Ésta indicaría que el botón está liberado, pero el usuario ha hecho clic sobre algún elemento existente en pantalla.

Internamente el controlador de ratón registra todas las pulsaciones y liberaciones de botón, datos que pueden ser recogidos a través de las funciones 05h y 06h, respectivamente. Los pasos a seguir para emplear estos dos servicios son los siguientes:

- Se introduce en el registro BX el valor 1, 2 ó 4, según se quiera consultar el botón izquierdo, derecho o central, respectivamente.
- Se invoca a la función 05h para obtener el número de pulsaciones o bien a la función 06h para leer el número de liberaciones. Normalmente estos valores coinciden, pero puede existir una diferencia de una unidad.
- A la vuelta de la llamada el registro BX contendrá el número de pulsaciones o liberaciones, mientras que los registros CX y DX indicarán la posición del puntero del ratón en el momento en que se produjo la última pulsación o liberación.

El contador de pulsaciones/liberaciones del botón consultado se pone a cero tras la lectura. De esta forma es posible efectuar menos llamadas a los servicios de ratón sin por ello perder ninguna actuación del usuario. Pruebe a modificar el ejercicio anterior para emplear las funciones que acaban de describirse en sustitución de la función 03h.

Aspecto del puntero del ratón

Cuando se hace visible el puntero del ratón, éste muestra un aspecto por defecto que es el establecido en el software controlador. En modo de texto ese aspecto es el de un bloque que ocupa toda una celdilla de carácter, mientras que en modo gráfico suele ser una flecha inclinada. En ambos casos, no obstante, es posible modificar la apariencia y establecer otras más adecuada para la finalidad de la aplicación.

Operando en modo de texto el servicio que permite modificar el aspecto del puntero del ratón es el OAh, existiendo dos posibilidades: usar un puntero por hardware del que solamente puede modificarse su tamaño o bien definirse un cursor software obtenido a partir de operaciones lógicas.

Antes de invocar a esta función de la interrupción 33h es necesario asignar valores válidos a los siguientes registros:

- BX: Establece el tipo de cursor que se utilizará. Si se asigna el valor 1 activará el cursor hardware, mientras que el valor 0 optaría por el cursor software.
- CX: Indicará la línea de inicio del cursor hardware o bien la máscara de pantalla si se utiliza un cursor software.
- DX: Indicará la línea de fin del cursor hardware o bien la máscara de cursor si se utiliza un cursor software.

El cursor hardware funciona de manera similar al cursor de texto, siendo las líneas de inicio y fin análogas a las que definen el cursor de texto y que se mencionaron en el capítulo dedicado a los servicios de vídeo. Más interesante resulta el uso del cursor software, ya que éste permite emplear como puntero cualquier carácter, fijo u obtenido a partir de una operación lógica, y con cualquier combinación de color. En este caso los valores introducidos en los registros CX y DX, las denominadas máscaras de pantalla y cursor, han de interpretarse de la siguiente manera:

- Bits 0-7: El byte de menor peso representa la máscara que se aplicará sobre el código ASCII del carácter.
- Bits 8-11: Estos cuatro bits enmascararán el color de primer plano.
- Bits 12-15: Los cuatro bits de mayor peso son la máscara a aplicar al color de fondo y bit de parpadeo.

La máscara de pantalla, contenida en el registro CX, se aplica sobre el contenido actual de la pantalla en la posición donde se encuentre el puntero, utilizando para ello la operación lógica AND. Esto permite conservar o eliminar las partes de ese contenido que interesen. Usando como máscara el valor 0 OFFh, por ejemplo, se conservaría el carácter que haya en la posición, pero no el color. Mediante la máscara FFOOh, por el contrario, se descartaría el código del carácter y conservaría la información de color.

Tras obtener el resultado intermedio que produce la aplicación de la máscara de pantalla, se toma la máscara de cursor, almacenada en DX, y se efectúa una operación lógica XOR, obteniendo el carácter y atributo de color con que aparecerá definitivamente.

El programa mostrado a continuación utiliza la función OAh para definir un cursor software, conservando el carácter de la celdilla donde esté colocado el puntero y cambiando únicamente el color. El resultado es que el cursor aparece como un bloque de fondo azul y con el carácter en color blanco:

```

segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código
.start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; Inicializar y mostrar puntero
    xor ax, ax
    int 33h

    mov ax, 0lh
    int 33h

    ; Configurar el cursor hardware
    mov ax, OAh
    mov f>x, 1
    ; Mantener el carácter
    mov ex, OOFFh
    ; y modificar el atributo
    mov dx, 03700h
    int 33h

    ; Salimos al sistema
    mov ah, 4ch
    int 21h

```

Modificando exclusivamente las líneas en las que se asigna valor a los registros CX y DX es posible configurar cualquier otro tipo de cursor. Sustituyéndolas por las dos sentencias siguientes, por ejemplo, el puntero del ratón sería una flecha que aparecería en la combinación de colores que haya en la pantalla, es decir, se utiliza un carácter concreto y no se alteran los atributos:

```
mov ex, QFFOOh
mov dx, 00018h
```

En la figura 21.4 puede verse una composición en la que aparecen los dos cursores software: en la parte superior el que modifica el atributo, mostrando en blanco sobre azul el carácter que haya en la posición donde esté el puntero; mientras que en la inferior se aprecia la flecha apuntando hacia arriba que aparece como puntero en el segundo caso.

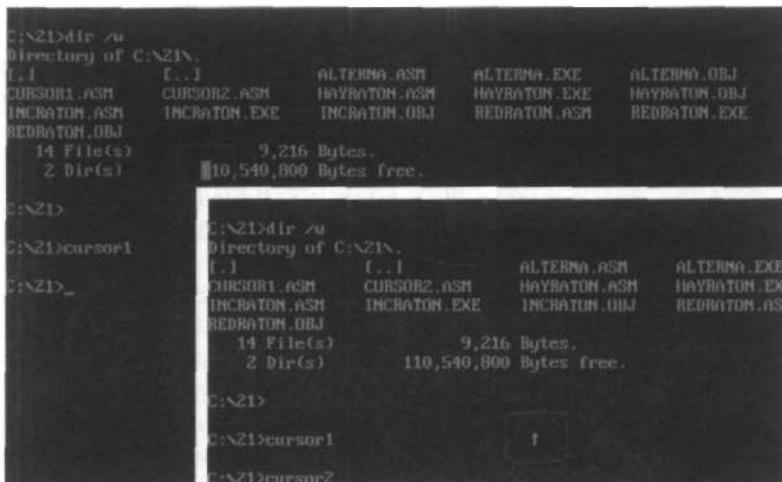


Figura 21.4. Cursores software en modo de texto.

Las posibilidades en modo gráfico, en cuanto al aspecto del puntero del ratón se refiere, resultan más interesantes. En este caso el puntero se define con una máscara de pantalla y una máscara de cursor, al igual que el puntero software en modo texto, aplicando además las mismas operaciones lógicas. La diferencia es la estructura de esas máscaras, ya que en modo gráfico se define como un mapa de bits.

La función encargada de establecer el puntero del ratón en modo gráfico es la 0 9h, siendo preciso facilitar información en los siguientes registros:

- ES y DX: Dirección completa de un bloque de memoria que debe contener la máscara de pantalla y, a continuación, la máscara de cursor. Cada una de ellas estará compuesta de 16 filas de 16 bits, es decir, será un mapa de bits de 16x16.
- BX y CX: Deben contener la coordenada horizontal y vertical, respectivamente, del *hot spot* del puntero. Éste es el punto del cursor que el usuario identificaría

visualmente como el que debe colocar sobre la posición en la que quiere actuar, por ejemplo la punta de una flecha o la intersección de una cruz. Estas coordenadas estarán comprendidas entre 0,0 (esquina superior izquierda) y 15,15 (esquina inferior derecha).

En la máscara de pantalla cada bit a 0 implicará desactivar el píxel correspondiente, al efectuar la operación lógica AND con el contenido actual de la pantalla. Es habitual, por ejemplo, que se desactiven los bits que van a quedar justo alrededor del puntero, favoreciendo así la identificación de éste sobre un fondo que tenga el mismo color.

El comportamiento de la máscara de cursor es, lógicamente, el inverso, de forma que cada bit a 1 corresponderá a un píxel que, al efectuar la operación XOR con el resultado intermedio anterior, quedarán normalmente iluminado, dibujando el puntero.

Todo el proceso quedará mucho más claro con un sencillo ejemplo, consistente en un programa que activará el modo gráfico 5 (véase el capítulo dedicado a los servicios de vídeo), muestra el puntero del ratón y da a éste la forma de una flecha un tanto peculiar. El programa quedará en un bucle hasta que se pulsen simultáneamente los botones izquierdo y derecho del ratón, momento en el que devolverá el control al sistema.

```
segment Pil a stack
    resw 512
FinPila:

segment Datos

; Máscaras de definición del puntero
Puntero:
    dw    001111111111111b
    dw    000111111111111b
    dw    000011111111111b
    dw    000001111111111b
    dw    000000111111111b
    dw    000000011111111b
    dw    000000001111111b
    dw    000000000111111b
    dw    000000000011111b
    dw    000000011111111b
    dw    000100001111111b
    dw    101110001111111b
    dw    ín m i i o o i i n i b
    dw    111111111100011b
    dw    111111111111100b
    dw    111111111111111b

    dw    000000000000000b
    dw    010000000000000b
    dw    011000000000000b
    dw    011100000000000b
    dw    011110000000000b
    dw    011111000000000b
    dw    011111100000000b
    dw    011111110000000b
```

```
dw    0111111111000000b
dw    0111110000000000b
dw    0100011100000000b
dw    0000000011100000b
dw    0000000000011100b
dw    000000000000011b
dw    0000000000000000b

; Segmento de código
segment Código

..start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; ES apuntando al segmento de datos
    mov ax, Datos
    mov es, ax

    ; Modo 640x700
    mov al, 6
    xor ah, ah
    int 10h

    ; Inicializar y mostrar puntero
    xor ax, ax
    int 33h

    mov ax, Olh
    int 33h

    ; Configurar el cursor gráfico
    mov ax, 09h
    mov bx, 0
    mov ex, 0
    mov dx, Puntero
    int 33h

bucle: ; Recuperar posición y estado
    mov ax, 03h
    int 33h

    ; Si están pulsados los dos botones
    cmp bx, 03h
    jnz bucle

    ; volvemos al modo
    ; de video de texto
    mov al, 3
    xor ah, ah
    int 10h

    ; Salimos al sistema
    mov ah, 4ch
    int 21h
```

Las máscaras para el puntero del ratón podrían definirse como dobles palabras en decimal o hexadecimal, pero el uso de la base binaria, tal y como se ha hecho en este programa, hace más fácil apreciar el aspecto que tendrá el cursor (véase la figura 21.5) finalmente.

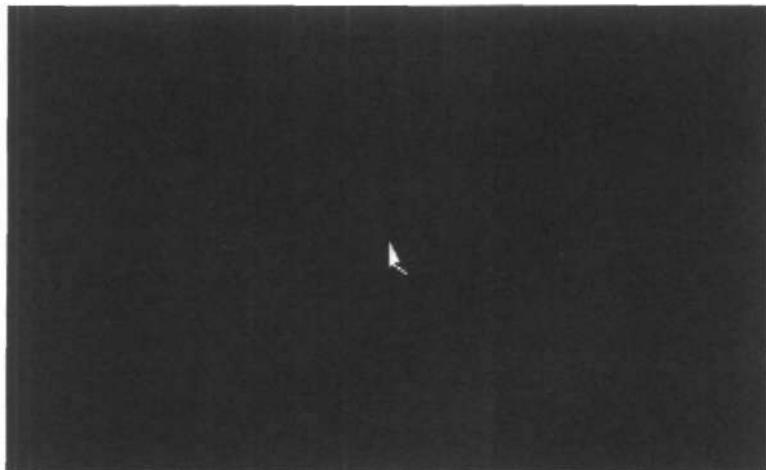


Figura 21.5. Aspecto del cursor en modo gráfico.

En cualquier momento puede restablecerse el aspecto del puntero del ratón, tanto en modo texto como gráfico, sencillamente invocando a la función 0 Oh (inicialización) de los servicios del ratón.

Instalación de una rutina de retorno

Empleando las funciones descritas hasta el momento, para utilizar el ratón como dispositivo de entrada las aplicaciones tendrían obligatoriamente que llamar de manera periódica a una rutina, concretamente a alguno de los servicios que devuelve el estado de los botones, o sus contadores, y la posición del puntero en la pantalla. Tras esta llamada comprobarían si se han pulsado o no los botones que desencadenan acciones, tomando un camino u otro. Es una técnica que interfiere en cierta forma con la actividad principal de la aplicación, ya que ésta debe ocuparse de supervisar la actividad del ratón.

Un método alternativo, más cómodo para el programador, consiste en instruir al controlador de ratón para que sea él mismo, ante la detección de un cierto evento, el que llame a una subrutina de la aplicación. Esta procesaría el evento y devolvería el control, de forma que el bloque principal del programa no tendría que preocuparse en ningún momento por el ratón.

Para instalar una rutina de retorno que procese los eventos del ratón hay que utilizar la función **OCh** de la interrupción 33h. Ésta espera que se facilite en la pareja de registros ES : DX la dirección de la rutina, la llamada será de tipo FAR, y en el registro CX la máscara de sucesos.

De esta máscara tienen utilidad los primeros siete bits, de forma que un bit a 0 indica al controlador que el evento correspondiente no interesa, mientras que un bit a 1 comunicaría lo contrario, de forma que el controlador llamaría a la rutina. La correspondencia entre estos bits y el evento que activan/desactivan es la siguiente:

- **Bit 0:** Cambio en la posición del puntero en pantalla.
- **Bit 1:** Se ha pulsado el botón izquierdo.
- **Bit 2:** Se ha liberado el botón izquierdo.
- **Bit 3:** Se ha pulsado el botón derecho.
- **Bit 4:** Se ha liberado el botón derecho.
- **Bit 5:** Se ha pulsado el botón central.
- **Bit 6:** Se ha liberado el botón central.

Una vez instalada la rutina, cada vez que el controlador la invoque han de tenerse en cuenta los siguientes aspectos:

- No debe modificarse ningún registro, por lo que usualmente las primeras instrucciones se encargan de guardar en la pila aquéllos que van a utilizarse y las últimas se encargan de restaurarlos.
- El registro DS no estará apuntando al segmento de datos del programa, sino al del controlador de ratón. Si se necesita acceder a datos propios será preciso guardar DS, inicializarlo adecuadamente y resLaurarlo al final.
- Desde el código de la rutina puede utilizarse cualquiera de las interrupciones de la BIOS, como las de gestión de vídeo y teclado que conoció en capítulos previos, pero no deben hacerse llamadas a interrupciones DOS, como las que conocerá en capítulos posteriores.

Para que la rutina pueda realizar su trabajo, el controlador del ratón le facilitará, entre otros, los siguientes datos:

- CX y DX: Contendrán la posición actual del puntero del ratón.
- BX: Estado actual de los botones. Los tres bits de menor peso indicarán el estado del botón izquierdo, derecho y central.
- AX: Máscara de evento, de estructura idéntica a la máscara usada durante la instalación de la rutina de retorno y cuya finalidad será indicar qué evento ha generado la llamada.

Nota

Un programa que instale una rutina de retorno para el control del ratón no debe devolver el control al sistema operativo sin desactivarla previamente, ya que de lo contrario podría provocar un fallo del sistema en el momento en que el controlador intente llamar a esa rutina que ya no está en memoria. Basta con inicializar el controlador para desactivar la rutina de retorno.

Para concluir con este tema, los servicios de gestión del ratón, se propone un ejercicio algo más elaborado.

La finalidad del programa sería permitir dibujar a mano alzada en modo gráfico utilizando el ratón, para lo cual se instalará una rutina que, al ejecutarse con el desplazamiento del puntero, tome la posición y active el píxel adecuado. El programa se compondrá de tres partes.

La inicialización consistirá en activar el modo gráfico deseado, inicializar el ratón e instalar la rutina de atención al evento de desplazamiento del puntero.

El código sería el siguiente;

```
segment Pila stack
    resw 512
FinPila:

; Segmento de código
segment Código

..start:
; Configuramos la pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; Modo 610x200
mov al, 6
xor ah, ah
int 10h

; Inicializar y mostrar puntero
xor ax, ax
int 33h

mov ax, 0lh
int 33h

; Instalar rutina
mov ax, 0ch
mov bx, 1
push es
pop es
mov dx, Rutina
int 33h
```

Ejecutada esta porción del código, si el usuario mueve el ratón se llamará de inmediato a la rutina Rutina. Observe cómo se recupera en el registro ES el segmento donde se encuentra dicha rutina, que es el segmento de código, utilizando la pila para llevar el valor de CS a ES.

La segunda porción de código del programa, en orden secuencial tras la anterior, se limitará a esperar la pulsación de una tecla, tras lo cual restaurará el modo de vídeo e inicializará el ratón para desactivar la rutina. Esta sección la formarían las siguientes sentencias:

```
; esperamos la pulsación
; de una tecla
xor ah, ah
int 16h

; volvemos al modo
; de vídeo de texto
mov al, 3
xor ah, ah
int 10h

; y reiniciamos el ratón
xor ax, ax
int 33h

; Salimos al sistema
mov ah, 4ch
int 21h
```

El bloque principal del programa, por tanto, terminará en cuanto se pulse una tecla. Nada especial en este sentido, lo hemos hecho en ejercicios de capítulos previos.

Lo interesante es que mientras se espera dicha pulsación de tecla, encontrándose el programa aparentemente detenido, ante cualquier movimiento del ratón se ejecutará esta rutina:

```
; Rutina de atención al ratón
Rutina:
    pusha ; Guardar los registros

    ; Si no está pulsado
    ; el botón izquierdo
    test bx,01h
    ; terminar
    jz FinRutina

    ; Activar el píxel CX-1,DX-1
    mov ah, 0Ch
    xor bh, bh
    mov al, 1
    dec ex
    dec dx
    int 10h
```

```
FinRutina:
```

```
    popa  
    retf
```

Tras guardar en la pila todos los registros, comprobamos si está pulsado el botón izquierdo y, en caso afirmativo, usamos el servicio OCh de la interrupción 10h para activar el píxel correspondiente.

Fíjese en que la última instrucción es retf, en lugar de ret, indicando así que el retorno es de tipo FAR, es decir, que se recuperará de la pila el contenido de CS e IP, no solamente de IP.

En la figura 21.6 puede ver el programa en funcionamiento.

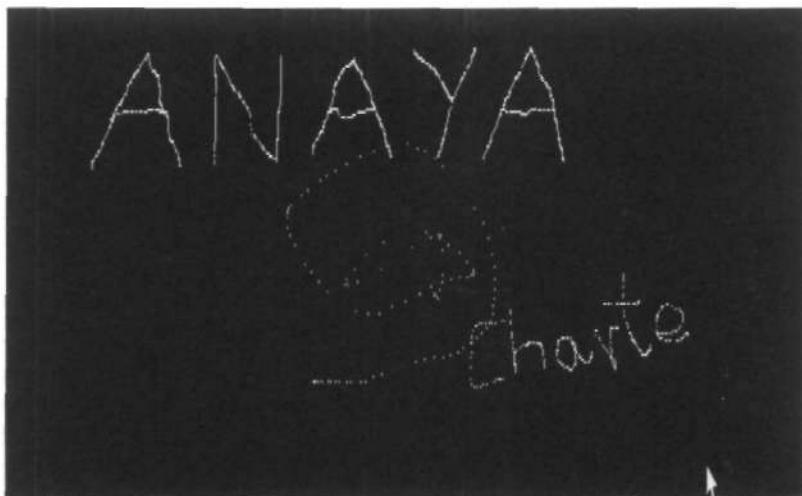


Figura 21.6. El programa de dibujo en funcionamiento.

Uso del joystick

Si bien el joystick y el ratón son dispositivos similares, ya que ambos permiten desplazarse en un plano bidimensional y cuentan con una serie de botones que pueden ser pulsados, la gestión desde las aplicaciones de un joystick resulta mucho más simple ya que no existe un puntero asociado.

La información obtenida del joystick tendrá una utilidad u otra dependiendo del programa que lo utilice. Lo más habitual es que se trate de un juego y sirva para mover algún personaje, nave o similar.

Al igual que ocurre con el ratón, existe una interrupción específica para comunicarse con el joystick. Ésta, sin embargo, sí forma parte de la BIOS de los PC, no siendo necesario alojar en memoria ningún tipo de controlador.

Nota

Los servicios mencionados en realidad se introdujeron en el IBM AT, un sucesor del PC basado en el microprocesador 80286. Los PC actuales heredaron esa característica, como muchas otras de los primeros diseños basados en torno a la familia x86.

Los servicios correspondientes al uso de palancas de juegos aparecen como subfunciones de la función 84h de la interrupción 15h. El código de la subfunción a ejecutar, hay dos sólo, se almacenará en el registro DX.

Existen, por tanto, dos operaciones posibles:

- DX=0: Obtener el estado de los botones del joystick. Éste se devuelve en los cuatro bits de mayor peso del registro AL, correspondiendo el valor 0 a un botón pulsado y el valor 1 a un botón no pulsado.
- DX=1: Lectura de la posición de la palanca de juegos. Los registros AX y BX contendrán la X e Y correspondientes al primer joystick y los registros CX y DX la misma información correspondiente al segundo joystick.

En lugar de tener conectados al sistema dos joystick de 2 ejes, cada uno de ellos con dos botones, también existe la posibilidad de usar una palanca de juegos más sofisticadas, con cuatro botones y tres ejes, por ejemplo.

Las posiciones devueltas por la subfunción 1 estarán comprendidas normalmente en el rango de 0 a 511, si bien esto también podría variar dependiendo del tipo de dispositivo que se tenga conectado.

No existe una función específica para detectar la existencia de un joystick conectado al sistema, pero tras invocar a estos servicios puede comprobarse el bit CF del registro de estado para saber si se ha producido algún fallo.

Si el bit de acarreo se activa indicará que hay un problema, posiblemente porque no haya disponible un joystick.

El siguiente programa muestra cómo se leería el estado de los botones del joystick, verificando que no se produce ningún error.

En realidad el programa no usa el estado de los botones para nada en concreto, aunque puede ejecutarlo desde DEBUG para comprobar el valor devuelto en el registro AL, limitándose a indicar mediante un mensaje la disponibilidad o no de la palanca de juegos (véase la figura 21.7).

```

segment Pila stack
    resw 512
FinPila:

segment Datos
Msg1 db 'Hay un dispositivo conectado$'
Msg2 db 'No se detectan dispositivos$'

; Segmento de código
segment Código

```

```

..start:
    ; Configuramos la pila
    mov ax, Pila
    rao vss, ax
    mov sp, FinPila

    ; y el segmento de datos
    mov ax, Datos
    mov ds, ax

    ; Obtener el estado
    ; de los botones
    mov ah, 84h
    xor dx, dx
    int 15h

    ; Si se activa el carry,
    ; no hay joystick
    je NoHay

    ; Si hay un joystick conectado
    mov dx, Msg1
    jmp Fin

NoHay: mov dx, Msg2

Fin:   mov ah, 9
       int 21h

       ; Salimos al sistema
       mov ah, 4ch
       int 21h

```

The screenshot shows a Windows command-line interface window titled "Símbolo del sistema". The command entered was "D:\Ejemplos\21>nasm -f obj hayjoys.asm". The output shows the assembly code being assembled. When the program runs, it checks for a joystick connection. Since one is detected, it prints "Hay un dispositivo conectado" (A device is connected) to the console.

```

Símbolo del sistema

D:\Ejemplos\21>nasm -f obj hayjoys.asm
D:\Ejemplos\21>alink hayjoys
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file hayjoys.obj
matched Externs
matched ComDefs

D:\Ejemplos\21>hayjoys
Hay un dispositivo conectado
D:\Ejemplos\21>

```

Figura 21.7. El programa detecta la existencia de un joystick en el sistema.

Resumen

En este capítulo ha conocido los servicios necesarios para utilizar desde programas propios dos dispositivos de entrada, el ratón y el joystick, adicionales al teclado, cuyo tratamiento ya se abordó en un capítulo previo.

De los dos el más interesante para la mayoría de las aplicaciones será el ratón, puesto que todos los sistemas actuales cuentan con uno y la interrupción 33h representa un mecanismo eficiente y estándar para acceder al mismo desde programas escritos en ensamblador.

Las palancas de juegos o joysticks se usan casi exclusivamente en juegos y, además, los servicios que ofrece la BIOS para poder utilizarlas resultan anticuados. Es usual que los dispositivos más avanzados incluyan sus propios servicios de bajo nivel, con más funciones que las que se han descrito aquí.

22

Configuración del equipo

En un capítulo previo, dedicado al estudio de la BIOS, conocimos algunos servicios que nos facilitaban información sobre la configuración del equipo, si bien dicha información era parcialmente incompleta y no siempre exacta. La información sobre memoria disponible, por ejemplo, siempre es la misma en equipos actuales, ya que todos ellos cuentan con más de 640 kilobytes de memoria y el servicio que conocemos no reconoce una cantidad superior. Lo mismo cabría decir de otros elementos. La BIOS cuenta con servicios extendidos que pueden facilitarnos más información sobre la configuración del ordenador. Como ya sabe, parte de esa información está almacenada en una memoria CMOS por lo que otra opción, igualmente interesante, consiste en recuperarla leyendo esa memoria. Para ello usaríamos las instrucciones in y out que ya conoce.

El objetivo de este capítulo es mostrarle cómo utilizar esos recursos a fin de poder conocer con mayor exactitud la configuración de su equipo, adaptando así lo mejor posible el comportamiento de las aplicaciones que pueda desarrollar.

Lectura de la memoria CMOS

Comenzaremos tratando el contenido de la memoria CMOS, aprendiendo a leerla e interpretar su contenido. En realidad es algo que ya hemos hecho, al codificar una de las rutinas de capítulos previos para leer los minutos y segundos del reloj en tiempo real. En realidad, es el mismo integrado que mantiene ese reloj el que se aprovecha para recuperar los datos que vamos a conocer a continuación, al tratarse de un elemento que cuenta con una cantidad de memoria más que suficiente para almacenar la hora. Inicialmente esa cantidad era de 64 bytes, utilizándose sólo 10 de ellos para la fecha y hora.

Para comunicarnos con ese integrado usamos dos puertos: el 7 Oh y el 71h. El primero de ellos establece la dirección de la que va a leerse o en la que se piensa escribir, mientras que el segundo es el puerto de datos, en el que se escribirá o del que se leerá. En realidad el puerto 7 Oh tiene otras funciones. Su bit de mayor peso, el bit 7, controla el registro de máscara NMI, no debiéndose modificar su estado para no alterar el normal funcionamiento del ordenador. Antes de enviar una dirección al puerto 7 Oh, por tanto, antes deberíamos leerlo para saber el estado de dicho bit 7, manteniéndolo tal cual a la hora de escribir. Esto no es ningún problema porque los bits empleados para seleccionar la porción de memoria a escribir o leer son sólo los cinco primeros, del bit 0 al 4.

Teniendo esto en cuenta, el proceso general para recuperar un dato de la memoria CMOS sería el siguiente:

```
in al, 70h ; recuperamos el contenido actual
and al, 128 ; 10000000 ponemos menos el bit 7
or al, Dir ; introducimos la dirección a leer
out 70h, al ; y la enviamos al integrado
in al, 71h ; leemos el contenido de ese byte
```

El valor 128 equivale al 10000000 en binario, de tal forma que, al efectuar una operación lógica and, ponemos a cero los siete bits de menor peso sin alterar el octavo, que quedará con su valor. Para introducir la dirección del byte a leer no podemos usar una instrucción mov, ya que ésta eliminaría el contenido de al. Por eso lo que hacemos, en su lugar, es sumar esa dirección. La enviamos al puerto 7 Oh y a continuación leemos el dato que nos devuelve.

Datos contenidos en la CMOS

Sabemos cómo leer datos de la CMOS, pero no la dirección y el significado de cada uno de los bytes que mantiene dicha memoria. En la tabla 22.1 se resumen las direcciones y tañíános de los datos de mayor interés.

Tabla 22.1. Contenido de la memoria CMOS.

Dirección	Longitud	Contenido
0	1	Segundos.
1	1	Segundas de la alarma.
2	1	Minutos.
3	1	Minutos de la alarma.
4	1	Horas.
5	1	Horas de la alarma.
6	1	Día de la semana.
7	1	Día del mes.

Dirección	Longitud	Contenido
8	1	Mes.
9	1	Año.
10	4	Registros de estado A, B, C y D.
14	1	Estado de diagnóstico.
15	1	Estado de apagado.
16	1	Tipo de las disqueteras.
18	1	Tipo de los discos duros.
21	2	Cantidad de memoria base.
23	2	Cantidad de memoria extendida según SETUP.
25	1	Byte de extensión del primer disco duro.
26	1	Byte de extensión del segundo disco duro.
32	8	Parámetros del disco duro.
46	2	Suma de comprobación de la memoria CMOS.
48	2	Cantidad de memoria extendida según comprobación de arranque.
53	8	Parámetros del disco duro.

Los datos apuntados en esta tabla podríamos decir que son los estándares, ya que cada BIOS específica almacena datos adicionales en bloques de memoria más allá de los primeros 64 bytes. Para saber qué datos son y cómo interpretarlos habría que recurrir a la documentación técnica de cada BIOS en concreto.

Los datos relativos a fechas y horas se almacenan en BCD empaquetado, mientras que el resto de cantidades, por ejemplo de memoria, vienen codificadas en binario.

Para poder utilizar muchos de los datos apuntados en la tabla 22.1 resulta imprescindible saber cómo interpretarlos. La fecha y hora, o las cantidades de memoria, son fáciles de comprender, al ser números BCD o binarios, respectivamente. Los tipos de las disqueteras y los discos duros, por el contrario, no pueden interpretarse de manera directa.

El byte que está en la dirección 16 nos indica el tipo de las dos disqueteras que es posible tener en el sistema. El *nibble* de mayor peso, de los bits 4 al 7, corresponden a la primera disquetera (A: en DOS/Windows o fd0 en Unix/Linux), mientras que el de menor peso indica el tipo de la segunda. Los valores posibles para estos *nibbles* son los enumerados en la tabla 22.2.

Tabla 22.2. Significado del *nibble* que indica el tipo de disquetera.

Valor	Significado
0000	Disquetera no instalada.
0001	Disquetera de 5,25 pulgadas y discos de 360 kilobytes.
0010	Disquetera de 5,25 pulgadas y discos de 1,2 megabytes.
0011	Disquetera de 3,5 pulgadas y discos de 720 kilobytes.
0100	Disquetera de 3,5 pulgadas y discos de 1,44 megabytes.

El byte que está en la posición 18 se divide como el anterior, en dos *nibbles* que indican el tipo de los dos primeros discos duros del sistema. Hay que tener en cuenta que en los primeros años de existencia del XT/AT, primeros ordenadores PC que contaban con disco duro, sólo había unos cuantos tipos y no era posible instalar más de dos en un mismo equipo. El *nibble* de mayor peso corresponde al primer disco duro y el de menor peso al segundo. Un valor 0 indicará que el disco no está instalado, los valores 1 a 14 (0001 a 1110) comunican que el disco es de ese tipo, mientras que el valor 15 nos remite al byte de extensión para conocer el tipo de disco. En caso de que el disco no sea de los tipos predefinidos 1 a 14, lo más normal si el equipo es relativamente actual, será necesario recurrir a los bytes de extensión para conocer su tipo. Estos bytes se encuentran en las direcciones 25 y 26, para el primer y segundo disco, pudiendo tener cualquier valor comprendido entre 0 y 255. Cada uno de esos valores correspondería a un tipo predefinido, aunque hay algunos que indican que el disco tiene una geometría definida por el usuario y otros que especifican que la detección de esa geometría se hará de forma automática. Esto es lo que ocurre en las BIOS modernas.

Si el byte de extensión de un disco tiene el valor 48 ó 49, significará que su geometría ha sido definida por el usuario mediante la utilidad de configuración de la BIOS. Podemos recuperar esa geometría del área de parámetros del disco duro. Tenemos una en la dirección 32, para el primer disco, y otra en la dirección 53, para el segundo. En ambos casos se componen de una secuencia de 8 bytes con el significado descrito en la tabla 22.3.

Tabla 22.3. Estructura del área de parámetros de disco duro.

Posición	Longitud	Contenido
0	2	Número de pistas del disco.
2	1	Número de cabezas.
3	2	Pista de escritura precompensada.
5	2	Zona de aterrizaje.
7	1	Número de sectores por pista.

Nota

Observe que la estructura del área de parámetros, con sólo 16 bytes para el número de pistas y ocho para el de sectores y cabezas, resulta insuficiente para la mayoría de discos actuales, de ahí que todas las BIOS modernas cuenten con otro método de direccionamiento del disco y un área de parámetros extendida.

Visualización de parámetros de la CMOS

Para ver en la práctica cómo podemos usar los datos alojados en la memoria CMOS, al menos parte de ellos, vamos a construir un programa que, al ser ejecutado, nos indique la cantidad de memoria, base y extendida, que hay instalada en el sistema, así como el tipo de las disqueteras y los discos duros y los parámetros del primer disco. El resultado sería similar al mostrado en la figura 22.1, correspondiente a un equipo con 16 megabytes de memoria RAM, una disquetera de 3.5 pulgadas y dos discos duros.

```

A:>>cmos
   640 Kb de memoria base - 15360 Kb de memoria extendida
La disquetera A: es de tipo <3.5" y 1.44 Mb>
La disquetera B: es de tipo <No está instalada>
Tipo del disco duro C:    81
Tipo del disco duro D: -  82
Parámetros del primer disco duro
Pistas: 1020
Cabezas: 84
Sectores: 128
A:>>_

```

Figura 22.1. Resultados que facilita el programa por la consola.

El código de este programa es relativamente extenso, habiéndose estructurado en múltiples rutinas que facilitan la comprensión y mantenimiento. La parte principal del programa, con la definición de datos y las sentencias que se ejecutan hasta devolver el control al sistema, es la siguiente:

```

segment Pila stack
    resw 512
FinPila:
segment Datos

```

```

; Cadenas de texto para mostrar
; la información de memoria
MsgMemoriaBase      db '          Ks de memoria base - $'
MsgMemoriaExtendida
                     db '          Ks de memoria extendida'
                     db 13, 10, 10, '$'

; Cadena común para los datos
; de las disqueteras
MsgDisquetera        db 'La disquetera '
LetraDisquetera     db 'A: es de tipo $'

; Tenemos cinco cadenas distintas para
; indicar el tipo de la disquetera
Disquetera0 db '<No está instalada>$'
Disquetera1 db '<5.25" y 360 Ks>$'
Disquetera2 db '<5.25" y 1.2 Mb>3'
Disquetera3 db '<3.5" y 720 Ks>$'
Disquetera4 db '<3.5" y 1.44 Mb>$'

; esta tabla apunta a cada una
; de las cinco cadenas anteriores
TblPunteros dw Disquetera0, Disquetera1,
              dw Disquetera2, Disquetera3,
              dw Disquetera4

; Cadena común para el disco duro
DiseoDuro db 13, 10, 'Tipo del disco duro '
LetraDisco db 'C: - $'

; Cadena para indicar el tipo
; o su ausencia
Noinstalado db 'No está instalado', 13, 10, '$'

AvanceLinea db 13, 10, '$'

; Cadena para mostrar la tabla de parámetros
; del disco duro
MsgParametros db 13, 10
               db 'Parámetros del primer disco duro'
               db 13, 10
               db '_____'
               db 13, 10
               db 'Pistas:   '
Pistas    resb 5
               db 13, 10
               db 'Cabezas:  '
Cabezas   resb 5
               db 13, 10
               db 'Sectores: '
Sectores  resb 5
               db 13, 10, '$'

; Segmento de código
segment Código

```

```

..start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; DS y ES apuntan al segmento
    ; que contiene los datos
    mov ax, Datos
    mov ds, ax
    mov es, ax

    ; Mostramos información sobre
    ; la memoria base
    cali MemoriaBase

    ; y la memoria extendida
    cali MemoriaExtndida

    ; información sobre el tipo
    ; de las disqueteras
    cali Disqueteras

    ; y los discos duros
    cali DiscosDuros

    ; y por último los parámetros
    ; del primer disco
    cali ParametrosDiscoDuro

    ; y salímos
    mov ah, 4ch
    int 21h

```

Como puede ver, tras preparar los registros de segmento el programa se limita a invocar a cinco rutinas que serán las encargadas de ir recuperando la información, introduciéndola en los campos declarados para tal propósito y mostrándolos en la consola.

La primera rutina, MemoriaBase, lee la cantidad de memoria base instalada en el sistema en AX, convirtiéndola en una cadena que mostrará por la consola. Primero lee el byte de mayor peso de la palabra, depositándolo en AH. A continuación lee el byte de menor peso, en AL, e invoca a una renovada rutina Enterocadena que veremos más adelante. Ésta se ha modificado para convertir en una secuencia de dígitos el contenido de AX en lugar de AL, pudiendo tratar números de 16 bits.

Nota

La memoria base del sistema es aquella que está en el primer megabyte de direccionamiento, siempre descontando la cantidad reservada a ROM y áreas de uso de ciertos dispositivos como la pantalla. Típicamente esa memoria siempre es de 640 kilobytes en los equipos actuales.

```
; Esta rutina muestra información
; sobre la memoria base instalada
```

```
MemoriaBase:
    ; leemos cantidad de
    ; memoria base
    mov bl, 22
    cali LeeCMOS

    ; guardamos el byte de
    ; mayor peso en AH
    mov ah, al
    ; y leemos el de
    ; menor peso
    iip.c. bl
    cali LeeCMOS

    ; Tenemos en AX la
    ; cantidad de memoria base

    ; la convertimos a cadena
    mov di, MsgMemoriaBase+4
    cali EnteroCadena
    ; y mostramos
    mov dx, MsgMemoriaBase
    mov ah, 9
    int 21h

    ret ; volver
```

La rutina MemoriaExtendida es muy similar a la anterior, usando la misma técnica para convertir la cantidad en una cadena. La diferencia es la dirección de la CMOS de la que se obtienen los bytes, nada más.

```
/* Esta rutina muestra información
; sobre la memoria extendida instalada
```

```
MemoriaExtendida:
    ; leemos cantidad de
    ; memoria extendida
    mov bl, 49
    cali LeeCMOS

    ; guardamos el byte de
    ; mayor peso en AH
    mov ah, al
    ; y leemos el de
    ; menor peso
    dec bl
    cali LeeCMOS

    ; Tenemos en AX la
    ; cantidad de memoria extendida
```

```

; la convertimos a cadena
mov di, MsgMemoriaExtendida+4
cali EnteroCadena
; y mostramos
mov dx, MsgMemoriaExtendida
mov ah, 9
int 21h

ret ; volver

```

A continuación tenemos la rutina que debe determinar el tipo de las disqueteras, mostrando la información que corresponda. En realidad, como puede verse a continuación, Disqueteras se limita a leer el byte que contiene el tipo de las disqueteras, separar los *nibbles* e invocar con ellos en AL a la rutina ResuelveDisquetera.

```

; Esta rutina muestra el tipo de
; las disqueteras

Disqueteras:
    ; leemos el byte correspondiente
    ; al tipo de las disqueteras
    mov bl, 1C
    cali LeeCMOS

    push ax ; guardamos el dato

    ; nos quedamos con el
    ; nibble de la primera
    ; unidad
    shr al, 4

    ; y resolvemos su tipo
    cali ResuelveDisquetera

    ; cambiamos la letra de A a B
    mov byte [LetraDisquetera], 'B'

    ; recuperamos el valor original
    pop ax

    ; y nos quedamos con el nibble
    ; de menor peso
    and al, 0fh

    ; resolvemos el tipo
    cali ResuelveDisquetera

    ret ; y volvemos

```

Dependiendo del valor que contenga AL, la rutina ResuelveDisquetera debe mostrar un mensaje u otro del conjunto DisqueteraO, Disqueteral, etc. En lugar de efectuar múltiples comparaciones, para determinar cuál de los cinco mensajes hay que imprimir, se ha optado por una solución muy distinta.

Se ha definido en el segmento de datos una tabla, llamada TblPunteros, que contiene las direcciones de cada una de las cadenas de tipo de disquetera. Cada una de esas direcciones ocupa una palabra, es decir, dos bytes. Por eso tomamos el contenido de AL y lo multiplicamos por 2, de tal manera que en AX tendremos 0, 2,4, 6 u 8, dependiendo del tipo de la disquetera. Ese valor se utiliza como desplazamiento sobre la dirección de TblPunteros, recuperando así la dirección de la cadena que debe imprimirse. Es un método que nos ahorra múltiples comparaciones y saltos.

```
; Esta rutina resuelve el tipo
; de una disquetera cuyo código
; ae facilita en AL

ResuelveDisquetera:
    ; guardamos AL
    puñh ax

    ; imprimimos la primera
    ; parte del mensaje
    mov dx, MsgDisquetera
    mov ah, 9
    int 21h

    pop ax ; lo recuperamos

    ; multiplicamos AT,
    ; por 2
    mov bl, 2
    ; para tener en AX
    ; un desplazamiento
    muí bl

    ; BX apunta a la tabla
    ; de punteros
    mov bx, TblPunteros
    ; le sumamos AX
    ; como desplazamiento
    add bx, ax

    ; tomamos la dirección
    ; de la cadena apropiada
    ; en DX
    mov dx, [bx]

    ; y la imprimimos
    mov ah, 9
    int 21h

    ; Avanzamos una linea
    mov dx, AvanceLinea
    mov ah, 9
    int 21h

    ret ; volver
```

La siguiente rutina es DiscosDuros, encargada de leer el byte que contiene el tipo de los discos duros y mostrar la información adecuada. Como la rutina Disqueteras, ésta básicamente lee el byte con los tipos, separa los dos *nibbles* y los facilita a otra rutina, ResuelveDisco, que será la encargada de decidir qué datos mostrar. Antes de llamarla también se almacena en el registro BL la dirección del byte extendido del disco que corresponda, por si fuese necesario recuperarlo.

```
; Esta rutina muestra el tipo de
; los discos duros instalados

DiscosDuros:
    ; recuperamos el byte
    ; con los tipos de disco
    mov bl, 18
    cali LeeCMOS

    ; guardamos el dato
    push ax
    ; y nos quedamos con
    ; el nibble de mayor peso
    shr al, 4

    ; facilitamos la dirección
    ; del byte extendido
    mov bl, 25
    ; y resolvemos el tipo
    cali ResuelveDisco

    ; cambiamos la letra
    ; de C a D
    mov byte [LetraDiscoJ, 'D'

    ; recuperamos el dato
    pop ax
    ; y nos quedamos con el
    ; nibble de menor peso
    and al, 0fh

    ; indicamos la dirección
    ; del byte extendido
    mov bl, 26
    ; y resolvemos el tipo
    cali ResuelveDisco

    ret ; volver
```

Como puede ver en el código siguiente, la rutina ResuelveDisco es algo más compleja ya que debe comprobar tres situaciones distintas: que el contenido de AL sea cero, caso en el que no hay disco instalado; que sea 1111 en binario, 0Fh en hexadecimal, caso en el que se indica que hay que leer el byte extendido, o, por último, que AL esté entre 1 y 14, pudiendo imprimirse directamente. Puesto que la rutina recibe la dirección del byte extendido, en BL, si hubiese que leerlo basta con llamar a la rutina LeeCMOS.

```

; Esta rutina recibe en AL un
; tipo de disco y en BL la
; dirección del byte extendido
; por si fuese necesario

ResuelveDisco:
    ; guardamos el tipo
    push ax

    ; mostramos la cadena común
    mov dx, DiscoDuro
    mov ah, 9
    int 21h

    ; recuperamos el dato
    pop ax

    ; comprobamos si e! tipo es 0
    or al, al
    ; de ser asi no hay disco
    jz DiscoNoInstalado

    ; comprobamos si es 15
    cmp al, 0fh
    ; de no ser asi ya tenemos
    ; en AL el tipo y saltamos
    ; para mostrarlo
    jne DiscoNoExtendido

    ; en caso contrario obtenemos
    ; el tipo extendido, ya que
    ; en BL tenemos su dirección
    cali LeeCMOS

DiscoNoExtendido:
    ; convertimos el tipo en
    ; cadena
    mov di, TipoDisco+4
    xor ah, ah
    cali EnteroCadena
    ; y mostramos el resto del mensaje
    mov dx, TipoDisco
    mov ah, 9
    int 21h

    ret ; volver

DiscoNoInstalado:
    ; indicamos que no hay disco
    ; instalado
    mov dx, Noinstalado
    mov ah, 9
    int 21h

    ret ; volver

```

La última rutina invocada desde el programa principal es ParametrosDiscoDuro que, como se ve seguidamente, se limita a recuperar el número de pistas, cabezas y sectores y mostrarlos en un mensaje compuesto en el segmento de datos de tal forma que puede imprimirse de una vez.

```
; Esta rutina muestra los
; parámetros del primer
; disco duro

ParametrosDiscoDuro:
    ; obtenemos el
    ; número de pistas
    mov bl, 33
    cali LeeCMOS
    mov ah, al
    dec bl
    cali LeeCMOS

    ; y lo convertirnos
    mov di, Pistas+4
    cali EnteroCadena

    ; recuperamos el numpro
    ; de cabezas
    mov bl, 34
    cali LeeCMOS
    xor ah, ah

    ; y lo convertimos
    mov di, Cabezas+4
    cali EnteroCadena

    ; Leemos el número de sectores
    mov bl, 39
    cali LeeCMOS
    xor ah, ah

    ; y lo convertimos
    mov di, Scctores+4
    cali EnteroCadena

    ; mostramos la información
    mov dx, MsgParametros
    mov ah, 9
    int 21h

ret
```

Desde todas las rutinas previas se invoca a LeeCMOS para recuperar distintos datos de la memoria CMOS. Se recibe en el registro BL la dirección de la que leer, devolviéndose el dato en el registro AL. La rutina da los pasos antes descritos para conservar el estado del bit de mayor peso.

```
; Esta rutina recupera
; un byte de información
; de la memoria CMOS.
; Debe indicarse la
; dirección en BL y se
; devuelve el dato en AL
```

LeeCMOS:

```
; leemos el valor
; del puerto 70h
in al, 70h
; ponemos a 0 los 5
; bits de menor peso
and al, 12 8
; y sumamos la
; dirección indicada
or al, bl

; seleccionamos
; dirección
out 70h, al
; y leemos el dato
in al, Vlh

; volvemos
ret
```

Por último tenemos la rutina EnteroCadena, un procedimiento que codificamos por primera vez unos capítulos atrás y que, en este caso, hemos actualizado para que en lugar de números de 8 bits sea capaz de procesar números de 16 bits. Esto facilita el trabajo de mostrar datos como la cantidad de memoria o el número de pistas, que exceden la capacidad de un byte.

```
; Este procedimiento convierte
; el valor de AX en una cadena
; de hasta cinco caracteres
;

; Entrada: AX = número a convertir
;           ES:DI = destino cadena
```

EnteroCadena:

```
; DX debe estar a cero
push dx ; lo guardamos
xor dx, dx

; establecemos valor inicial
mov byte [di], '0'

; comprobamos si AX es cero
or ax, ax
; de ser así, no hay más
; que hacer
jz FinConversion
```

```

push bx ; guardamos bx
; y establecemos el divisor
mov bx, 10

Bucle0:
    ; vamos dividiendo por 10
    div bx

    ; quedándonos con el resto
    ; que convertimos a ASCII
    add di, '0'
    ; y guardamos
    mov [dil], di
    ; retrocediendo al digito anterior
    dec di

    ; eliminamos el contenido
    ; de DX para quedarnos con
    ; el cociente de AX
    xor dx, dx

    ; si el cociente es mayor que 9
    cmp ax, 9
    ; seguimos dividiendo
    ja Bucle0

    ; en caso contrario guardamos
    add al, '0'
    mov [dil], al

    pop bx ; recuperamos BX

FinConversion:
    pop dx ; recuperamos DX
    ret

```

Servicios extendidos de la BIOS

A medida que la arquitectura del PC fue evolucionando, en los XT, AT y posteriores, aparecieron elementos antes inexistentes, como la memoria extendida. En los AT ésta podía llegar como máximo hasta los 16 Mb, barrera que se rompió con la aparición de los 386. Esto justifica la incorporación de distintas extensiones a los servicios estándar de la BIOS, extensiones que en muchos casos son accesibles a través de una única interrupción: la 15h. Con los servicios de dicha interrupción, muy numerosos, puede accederse a la obsoleta posibilidad de usar un cásete para el almacenamiento y recuperación, controlar una palanca de juegos según se indicó en el capítulo previo, acceder a los servicios de gestión de energía, activar el modo protegido, etc. De todos ellos, nos interesan particularmente, en este momento, aquellos que nos facilitan información adicional sobre la configuración de nuestro equipo, por ejemplo sobre la cantidad de memoria. Todos estos servicios activan el indicador de acarreo en caso de que no estén disponibles.

Mediante el servicio 88h obtenemos en AX el número de kilobytes de memoria extendida, es decir, memoria que está por encima de la dirección OFFFFFh. Ésta es la misma información que obteníamos antes de la memoria CMOS.

La memoria extendida se divide, desde un punto de vista lógico, en aquella que está por debajo de los 16 Mb y la que queda por encima. Al segundo bloque tan sólo puede accederse con direccionamiento de 32 bits. Con el servicio 0E801h obtenemos en AX la cantidad de memoria por debajo de los 16 Mb y en BX la que queda por encima de dicha dirección.

Existen servicios adicionales, como el 0E820h, capaces de facilitar un mapa detallado de toda la memoria que hay en el sistema bloque por bloque. No obstante, no son de uso habitual desde DOS al requerir el uso de registros de 32 bits.

Observe el siguiente programa de ejemplo. Usa dos de los servicios indicados para generar el resultado de la figura 22.2. En este caso el equipo cuenta con 48 Mb, apareciendo 47 de ellos como memoria extendida, de los cuales 15 están por debajo de la dirección de los 16 Mb y 32 Mb por encima.

```

segment Pila stack
    resw 512
FinPila:

segment Datos
; Cadena con el mensaje que indicará
; la cantidad total de memoria extendida
Mensaje db 'Hay '
Memoria resb b
    db ' Ks de memoria extendida', 13, 10, '$'

; Cadena para la cantidad de memoria
; existente por debajo de los 16 Mb
MsgLnfl6 db 'Hay '
MemLnfl6 resb 5
    db ' Ks de memoria por debajo de 16 Mb',13,10,'$'

; Cadena para la cantidad de memoria
; existente por encima de los 16 Mb
MsgSupl6 db 'Hay '
MemSupl6 resb 5
    db ' Ks de memoria por encima de 16 Mb?'

; Segmento de código
segment Código
..start:
    ; Configuramos la pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; DS apunta al segmento
    ; que contiene los datos
    mov ax, Datos
    mov ds, ax
    mov es, ax

```

```
; recuperamos información
; sobre la cantidad de
; memoria extendida
mov ah, 88h
int 15h

; convertimos cadena
mov di, Memoria+4
cali EnteroCadena

; y mostramos
mov dx, Mensaje
mov ah, 9
int 21h

; Recuperamos información sobre
; memoria extendida por encima
; y por debajo de los 16 Mb
mov ax, 0E801h
int 15h

; guardamos la que hay
; por encima
push bx

; convertimos la que hay
; por debajo, en AX
mov di, MemInf16+4
cali EnteroCadena

; y mostramos
mov dx, MsgInfl6
mov ah, 9
int 21h

; recuperamos el antiguo
; contenido de BX
pop ax

; viene expresado en bloques
; de 64 Ks
mov bx, 64
muí bx

; convertir a cadena
mov di, MemSuplfc+4
cali EnLeroCadena

; y mostrarlo
mov dx, MsgSupl6
mov ah, 9
int 21h

; y salimos
mov ah, 4ch
int 21h
```

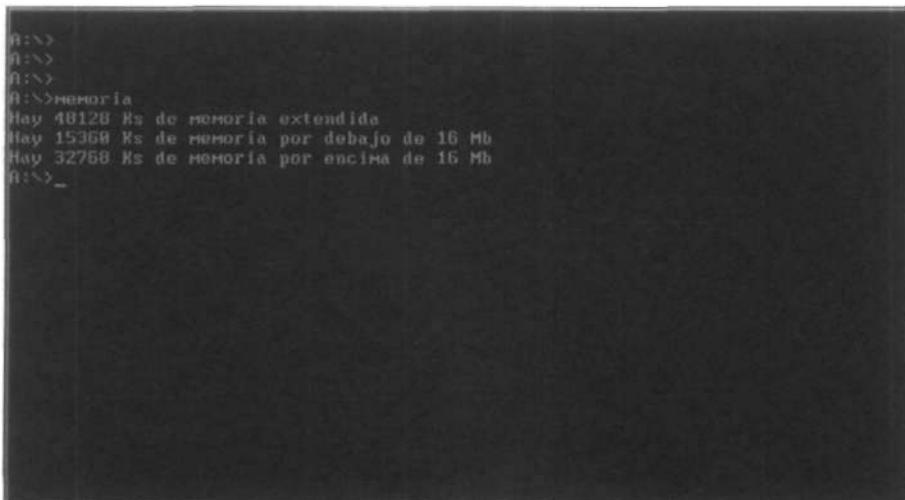
A screenshot of a DOS terminal window. The screen is mostly black with white text. At the top, there are three blank lines starting with 'A:>'. Below them, the text 'B:>' appears three times. Then, the command 'memoria' is entered, followed by its output: 'Hay 48128 Ks de memoria extendida', 'Hay 15360 Ks de memoria por debajo de 16 Mb', and 'Hay 32768 Ks de memoria por encima de 16 Mb'. Finally, 'B:>-' is shown at the bottom.

Figura 22.2. Información sobre la memoria extendida mostrada por el programa de ejemplo.

Resumen

Gran parte de la información de configuración de los PC se aloja en una memoria no volátil conocida como CMOS, memoria que podemos leer y escribir a través de unos puertos de entrada/salida. De esta forma podemos recuperar la fecha y hora actuales, la cantidad de memoria con que cuenta el equipo, el tipo de las disqueteras y discos duros, el estado actual del ordenador después del último reinicio, etc.

Este capítulo es el último que se ha centrado básicamente en el uso de los servicios de la BIOS. A partir del siguiente nos introduciremos en el estudio de los servicios que nos ofrece el DOS, sistema operativo sobre el que, hasta ahora, estamos ejecutando todos nuestros programas de ejemplo. Mediante estos servicios, de más alto nivel que los ofrecidos por la BIOS, se facilitan operaciones de otra forma complejas, como el tratamiento de archivos en disco.

23

**Interrupciones
DOS**

El objetivo de los primeros sistemas operativos era, básicamente, hacer más fácil la entrada/salida de información y facilitar el acceso a los medios de almacenamiento con que contase el ordenador, ofreciendo, además, la posibilidad de efectuar ciertas tareas básicas mediante la introducción de órdenes muy simples. CP/M, DOS (*Disk Operating System*) y Unix comenzaron de esta forma su existencia, en la década de los setenta y ochenta del pasado siglo. Cada sistema operativo ofrece a las aplicaciones sus servicios de distinta manera, dependiendo también en parte del procesador sobre el que funcione el sistema. En el caso de DOS, un sistema creado para los PC, se aprovechó la arquitectura del procesador 8086/8088 para ofrecer los servicios de forma similar a la BIOS, es decir, mediante interrupciones. Para acceder a cualquiera de los servicios que nos ofrece DOS, por tanto, tendremos que facilitar parámetros en los registros de procesador y, a continuación, emplear la instrucción `int` para invocar a la interrupción que corresponda.

Nuestro objetivo, en este capítulo, es familiarizarnos con algunas de esas interrupciones, aunque sin entrar en muchos detalles sobre sus servicios. Conoceremos algunos de ellos, especialmente los de la interrupción 21 h, en los capítulos que siguen a éste.

Interrupciones y versiones de POS

Los servicios ofrecidos por el DOS han ido cambiando a medida que se fueron sucediendo las distintas versiones de este sistema, apareciendo con cada una de ellas nuevos servicios que complementaban o sustituían a otros más antiguos. Esto explica que sigan

existiendo interrupciones como la 20h y la 27h que, a pesar de poder seguir utilizándose, están en desuso al haber aparecido otras opciones mejores.

Desde la primera versión de DOS, la 1.0, hasta la última considerada oficial por Microsoft, la 7.0 que formaba parte de Windows 95, son cientos los servicios que han ido añadiéndose, muchos de ellos específicos para la versión en particular con la que aparecieron. Prácticamente todos esos servicios comparten una misma interrupción, la citada 21h, quedando las demás con las funciones que tenían encomendadas originalmente.

DOS no era un sistema operativo totalmente nuevo, creado desde cero, sino que basa sus raíces en otro sistema previo, llamado CP/M-80, común en microprocesadores de 8 bits como el 8080 y el Z-80. Esto hace que, aún hoy, existan servicios, por ejemplo en la int 21h, que se originaron en CP/M-80 y pueden encontrarse en otros ordenadores y sistemas derivados de éste. No se extrañe, por ejemplo, que parte de los servicios del MSX-DOS, un sistema operativo similar al DOS que formó parte de millones de ordenadores MSX, coincidan en código, parámetros y funcionalidad con los del propio DOS.

El rango principal de interrupciones utilizadas por el DOS es el comprendido entre la interrupción 2 Oh y la 2Fh. La primera de ellas, procedente de la primera versión del DOS, tiene como objetivo finalizar la ejecución de un programa devolviendo el control al sistema, algo para lo que hemos usado repetidamente en todos los ejemplos de capítulos previos la función 4ch de la interrupción 21h. Actualmente, en los sistemas que utilizan Windows, la función es bien distinta, ya que Windows usa dicha interrupción para dar acceso a distintos servicios.

Otra de las interrupciones que ha perdido su función principal, con el surgir de nuevas versiones, es la int 27h, procedente también de la primera versión del DOS. Las aplicaciones DOS no pueden ejecutarse de manera simultánea por una limitación propia del sistema, que es monotarea. Esto significa que, una vez se inicia un programa, no podemos poner en marcha otro hasta que el primero finaliza y devuelve el control al sistema. Esto supone una seria limitación que, en parte, puede ser superada mediante los programas conocidos como TSR (*Terminate and Stay Resident*) o programas residentes. El objetivo de la int 27h es devolver al control al sistema reservando un cierto número de bytes de memoria que, presumiblemente, sería utilizado por un programa que quedaría ahí residente y se activaría de algún modo. En un capítulo posterior verá que existe un nuevo sistema para la gestión de los programas residentes y conocerá todos los detalles necesarios para crear este tipo de aplicaciones con unas ciertas garantías.

Las interrupciones 25h y 26h permiten la lectura y escritura de sectores de disco, respectivamente. Aparecidas en la versión 1.0 del DOS, fueron extendidas posteriormente para superar el límite que les impedía tratar con unidades de más de 32 megabytes y poder llegar hasta los 2 gigabytes en el DOS 3.31. Esa cantidad, que parecía enorme por aquellos tiempos, hoy está más que superada en las unidades que utilizan sistemas de archivos como FAT32, con decenas o incluso cientos de gigabytes de capacidad. En esos casos es necesario usar un servicio de la int 21h para poder leer y escribir.

Por último, mencionar las interrupciones 28h, invocada periódicamente por el DOS mientras está esperando la introducción de un dato por teclado, y la 2Fh, conocida como *multiplex* al ser empleada por multitud de aplicaciones. Las conocerá en mayor detalle posteriormente, al tratar la creación de programas residentes.

Funciones de la interrupción 21 h

La interrupción 21h cuenta con cientos de funciones diferentes, ofrecidas por distintas versiones de DOS, Windows, OS/2, extensores del DOS, sistemas de red como Novell Netware e, incluso, algunos virus que la utilizan para saber si han infectado ya el sistema.

Centrándonos en las funciones pertenecientes al DOS, que son las que nos interesan en este momento, podemos encontrar servicios de entrada y salida por la consola, impresora y puertos de comunicaciones, servicios para controlar la ejecución de los programas, asignar y liberar memoria, operar sobre archivos en disco, obtener y modificar la fecha y hora, etc.

Algunos de esos servicios, en particular los relativos al tratamiento de archivos y otros relacionados con la gestión de memoria expandida/extendida y la creación de aplicaciones residentes, los conocerá en los capítulos que siguen a éste. Por ello aquí nos centraremos en algunas otras de las funciones, aquéllas sobre las que no volveremos posteriormente.

Entrada y salida por la consola

Las funciones de entrada y salida por consola son de las más antiguas del DOS, aparecidas en la primera versión pero en, en realidad, procedentes, como se ha dicho antes, de CP/M-80. Son funciones muy básicas que, en ocasiones, se limitan a invocar al servicio de la BIOS que proceda para dar salida a un carácter o recoger un carácter.

Para seleccionar el servicio se utiliza, como es habitual, el registro AH, introduciendo en él el código de la función a invocar. Los parámetros de salida, habitualmente un carácter, se entregan en el registro DL, mientras que los de entrada, el carácter recuperado, se obtienen en el registro AL. Sabiendo esto, algunas de esas funciones de entrada y salida por consola son las siguientes:

- **Función 1:** Recupera un carácter de la consola, devolviéndolo en AL y mostrándolo en pantalla. Es lo que se conoce como entrada por teclado con eco.
- **Función 2:** Envía el carácter facilitado en DL a la consola.
- **Función 6:** Entrada y salida directa por consola. Si DL contiene un valor distinto de 255 envía ese carácter a la consola, mientras que si es 255 lee un carácter y lo devuelve en AL. En caso de que no haya caracteres esperando ser leídos, en el buffer de teclado, se activa el bit ZF del registro de indicadores.
- **Función 7:** Lee un carácter de la consola y lo devuelve en AL, sin mostrarlo en pantalla.
- **Función 8:** Idéntica a la anterior salvo por el hecho de que ésta comprueba si se ha pulsado la combinación de teclas **Control-ínter**, igual que la función 1, mientras que las funciones 6 y 7 no efectúan esa comprobación.

- **Función 9:** La hemos utilizado repetidamente en muchos ejemplos de capítulos previos para mostrar secuencias de caracteres. La dirección de la cadena se entrega en DS : DX y se espera que un carácter \$ indique el final.
- **Función 10 (0Ah):** También la utilizó en un programa de ejemplo. Su objetivo es facilitar la introducción de secuencias de caracteres, para lo cual hay que facilitar en DS: DX la dirección de un área de memoria cuyo primer byte indicará el número de caracteres a leer como máximo, mientras que el segundo comunicará los leídos efectivamente tras la llamada.
- **Función 11 (0Bh):** Comprueba si hay o no caracteres disponibles para ser leídos, devolviendo en el registro AL el valor 0 si no los hay o 255 en caso contrario.

Las funciones de lectura siempre recuperan un carácter del buffer de teclado, por lo que, en ocasiones, pueden no tener el efecto esperado en caso de que éste no se encuentre vacío. Suponga que quiere usar el servicio 7 simplemente para esperar la pulsación de una tecla, permitiendo ver un cierto resultado en pantalla. En caso de que el buffer de teclado no estuviese vacío, conteniendo pulsaciones previas ya almacenadas, esa pausa nunca llegaría a producirse.

Mediante la función 12 (0Ch) puede limpiar el buffer de teclado, dejándolo vacío, y, a continuación, ejecutar cualquiera de las funciones de lectura de carácter. Para ello facilitaremos, además del código de servicio en AH, un número de función de lectura en el registro AL. Por ejemplo, puede asignar a AL el valor 7 y, a continuación, invocar a esa función para esperar realmente la pulsación de una tecla descartando cualquier contenido previo del buffer de teclado.

Si quiere dejar vacío el buffer de teclado sin recoger ninguna pulsación, invoque a este servicio habiendo asignado el valor 0 al registro AL.

Comunicación serie y paralelo

La consola no es el único dispositivo con el que podemos comunicarnos mediante el envío y recepción de bytes individuales, los puertos serie y la impresora son también dispositivos basados en caracteres. Las funciones disponibles para operar con ellos son muy parecidas a las descritas en el punto anterior, facilitándose los parámetros en los mismos registros.

Para enviar un carácter a la impresora por defecto utilizaremos el servicio 5 de la interrupción 21h. El único parámetro a entregar es el carácter, que asignaremos al registro DL. Tenga en cuenta que este servicio esperará, si es necesario, hasta que la impresora esté preparada para recibir el carácter. También comprueba la pulsación de las teclas **Control-ínter** para interrumpir el proceso.

En cuanto al puerto serie, usado habitualmente para comunicarse con distintos dispositivos externos como los módem, ratones y similares, las funciones disponibles son dos: la 3 y la 4. La primera lee un carácter y lo devuelve en AL, mientras que la segunda envía el carácter contenido en DL. Se usa generalmente el primer puerto de comunicaciones, conocido como COMÍ.

Fecha y hora

Como ya sabe, los PC cuentan con un reloj de tiempo real alimentado por baterías que mantiene la fecha y hora actuales. En su momento vio cómo obtener esos datos leyéndolos directamente del integrado que los mantiene y, de igual forma, podría haberlos modificado puesto que son posiciones de memoria que permiten la escritura. Estas operaciones, sin embargo, resultan mucho más fáciles utilizando las siguientes funciones de la interrupción 2 lh:

- **Función 2Ah:** Devuelve la fecha actual en los registros DL (día del mes), DH (número del mes) y CX (año). También facilita, en el registro AL, el día de la semana, asumiendo que el 0 corresponde al domingo.
- **Función 2Bh:** Establece la fecha actual, que debemos facilitar en los mismos registros que usa el servicio anterior para devolverla. El registro AL nos indicará si se ha modificado sin problemas, en caso de que tenga el valor 0, o si la fecha era errónea y no se ha cambiado, caso éste en el que contendrá el valor 255.
- **Función 2Ch:** Obtiene la hora actual, facilitando las horas, minutos, segundos y centésimas en los registros CH, CL, DH y DL, respectivamente.
- **Función 2Dh:** Establece la hora actual. Debemos entregar las horas, minutos, segundos y centésimas en los registros CH, CL, DH y DL, respectivamente. Tras la llamada, AL indicará si se ha cambiado o no, según tenga el valor 0 ó 255.

PNoa

fIHHHHHHHHH

i

Recuerde que también puede servirse de la interrupción 1Ah para obtener y modificar la fecha y hora, aunque sólo la hemos utilizado en algún ejemplo previo para recuperar datos.

Gestión de vectores

Los vectores de interrupción, que hasta ahora estamos utilizando para invocar a servicios de la BIOS y el DOS, se alojan en memoria RAM, concretamente al inicio del espacio de direccionamiento, lo cual significa que pueden modificarse en caso de ser necesario. Una aplicación puede modificar un vector de interrupción para modificar un servicio ya

existente, actualizándolo o extendiéndolo, así como para interceptar ciertas situaciones, por ejemplo la pulsación de **Control-C** o el desencadenamiento de un error crítico.

Como ya sabe, cada vector se compone de cuatro bytes, dos palabras, que contienen una dirección completa de tipo segmento : desplazamiento. Modificar un vector reviste cierto peligro, ya que si se produce una interrupción entre el cambio de una palabra y otra, mediante dos instrucciones mov, se procedería a ejecutar un código incorrecto. Por ello el DOS ofrece servicios que permiten gestionar estos vectores, evitando que tengamos que hacerlo manualmente.

Por regla general, antes de modificar el valor de un vector de interrupción siempre tendremos que leer el que tiene actualmente, conservándolo por si fuese necesaria su restauración. La función **35h** de la interrupción 21h tiene esa finalidad. Facilitaremos en AL el número del vector de interrupción a leer, que estará comprendido entre 0 y 255, obteniéndose su contenido en los registros ES : BX.

Una vez tenemos el actual contenido del vector, el paso siguiente será su modificación para que apunte a nuestro propio código. En este caso el servicio a emplear es el 25h, facilitándose en AL el numero del vector de interrupción y en DS : DX la dirección que debe asignársele.

A partir de ese momento, cada vez que se genere la interrupción el control se transferirá a nuestro código.

Recuerde que para devolver el control desde una rutina de servicio a una interrupción debe emplearse la instrucción iret, no la instrucción ret, ya que int pone en la pila no sólo la dirección de retorno sino también el registro de indicadores.

Finalización y ejecución de programas

DOS es un sistema operativo que, aparte de ofrecer servicios, debe facilitar la ejecución de aplicaciones. Para ello cuenta con la línea de comandos que utilizamos habitualmente, desde la cual se introduce el nombre del archivo que contiene el programa a ejecutar. Lo que hace el sistema, en ese momento, es preparar un bloque de parámetros por defecto y utilizar un servicio de la interrupción 2 lh, concretamente la función 4Bh, para proceder a su ejecución.

Mientras se ejecuta el programa, el propio sistema queda a la espera de su finalización, que se produce cuando el programa invoca a la función 4Ch, como ya sabe. En ese momento el DOS recupera, mediante la función 4 Dh, el código de salida del programa y lo asigna a una variable de entorno que es conocida como ERRORLEVEL. Pero vayamos por partes.

La función 4Bh, la misma que usa el DOS para ejecutar una aplicación, puede ser utilizada desde el interior de un programa para ejecutar otro. Los parámetros necesarios para ello son los descritos a continuación:

- AL: Contendrá un código indicativo de la operación a efectuar. Para ejecutar un programa asignaremos el valor 0. Existen códigos adicionales para operar con módulos de extensión e, incluso, ejecutar aplicaciones en segundo plano en versiones específicas del DOS.
- DS : DX: Deben contener la dirección de una cadena de caracteres con el camino, nombre y extensión del archivo que contiene el programa a ejecutar. El final de la cadena se indicará mediante un byte 0.
- ES : BX: Contendrán la dirección de un bloque de parámetros en el que se indicará el entorno de ejecución a usar, la dirección de la línea de parámetros y la de dos bloques de control de archivo. El primer dato, una palabra, siempre será 0 para indicar que se utiliza el mismo entorno que facilita el DOS. El segundo será una doble palabra con la dirección de la línea de parámetros, mientras que el tercero y cuarto pueden ser nulos al apuntar a estructuras que actualmente ya no usa ninguna aplicación.

Nota

Los FBC son bloques de parámetros que se usaban en las primeras versiones del DOS para tratar con archivos en disco, pero actualmente están en desuso al aparecer servicios más evolucionados para dichas tareas.

Suponiendo que quisiésemos ejecutar desde una cierta aplicación el programa escrito en un capítulo previo puntocga.exe, ES : BX apuntaría a esta estructura de datos:

```
BloqueParametros
dw 0 ; usar entorno del DOS
dd LineaComandos ; cadena de parámetros
dd FCB1, FCB2 ; FCBS

; La cadena de parámetros está vacía
LineaComandos db 0,13
; como los FCB
FCB1 resb 20
FCB2 resb 20
```

Por su parte, DS : DX apuntarían a la cadena siguiente:

Progl db '\puntocga.exe', 0

No obstante, si intenta utilizar este servicio para ejecutar cualquier programa, por pequeño que sea, siempre obtendrá un error. Este viene indicado por la activación del bit de acarreo, así como por la asignación a AX del valor 8, con el que se indica que no hay memoria suficiente para ejecutar el programa solicitado.

Cuando el DOS pone en marcha un programa no examina su tamaño, a fin de asignarle un bloque de memoria adecuado, sino que, asumiendo que no puede ejecutarse

más de un programa de manera simultánea, le concede toda la memoria que hay disponible en el sistema. Por ello al intentar ejecutar cualquier otro programa no hay memoria disponible. Algo más adelante, en este mismo capítulo, sabrá cuál es la solución a este problema y la comprobará con un ejemplo que ejecuta múltiples programas externos.

En cuanto a la finalización de los programas, básicamente tenemos dos servicios disponibles: el 4Ch, que ya conocemos, y el 31h. El primero devuelve el control al sistema, o al programa que hubiese invocado al que está ejecutándose actualmente, facilitándole un código de salida en el registro AL. Hasta ahora, en los ejemplos de capítulos previos, nos hemos limitado a invocar al servicio 4Ch sin preocuparnos de AL, cuyo valor vuelve al programa de origen y puede ser usado, por ejemplo, para comprobar si el programa terminó satisfactoriamente o, por el contrario, se produjo algún error.

La otra opción, la del servicio 31h, se usa cuando, a pesar de querer finalizar el programa, no se quiere liberar la memoria que tiene asignada. Esto ocurre cuando se crean programas residentes, un tema sobre el que profundizaremos en un próximo capítulo. Los parámetros necesarios son dos: el código de resultado, en AL, y el número de párrafos de memoria que deseemos dejar residentes, en DX.

Si necesitamos, tras haber ejecutado otro programa desde el nuestro, comprobar cuál ha sido el método de finalización y obtener el código de resultado, podemos utilizar el servicio 4Dh. No se necesita parámetro alguno, obteniéndose en AH el citado código de resultado y en AH un valor de los enumerados en la tabla 23.1 indicando cómo se finalizó el programa.

Tabla 23.1. Tipo de finalización devuelto por el servicio 4Dh en el registro AH.

0	De forma normal, con una invocación al servicio 4Ch de la interrupción 21h.
1	Por la pulsación de Control-C .
2	A causa de un error crítico.
3	Quedando residente, mediante el servicio 31h.

Gestión de memoria

Como acaba de decirse, cuando se ejecuta un programa desde la línea de comandos del DOS éste asigna automáticamente toda la memoria disponible para él, de tal manera que el programa puede disponer libremente de la memoria. En el segmento de datos podemos definir campos para reservar memoria, y es posible tener múltiples segmentos de este tipo si fuese necesario.

Hay ciertos casos, sin embargo, en los que el programa puede haber liberado parte de la memoria que el DOS le asignó, por ejemplo al quedarse residente en memoria, y necesitar en un cierto momento un área de memoria para efectuar una operación. Lógicamente, no puede asignarse una dirección aleatoria a los segmentos y esperar que en

ella exista memoria RAM sin utilizar por el sistema, la BIOS u otras aplicaciones. En casos así es cuando tiene sentido la utilización de las funciones de gestión de memoria del DOS, que son tres:

- 4 8h: Este servicio busca un bloque de memoria que tenga el número de párrafos indicados en el registro BX, devolviendo la dirección de segmento en AX. En caso de que no sea posible la asignación se activará el indicador de acarreo. Recuerde que un párrafo son 16 bytes.
- 4 9h: Complementario al anterior, este servicio tiene la finalidad de liberar un bloque de memoria previamente asignado. Debe facilitarse la dirección de segmento, valor que la función 4 8h devolvía en AX, en el registro ES. Como en el caso anterior, consultando el indicador de acarreo podremos saber si se ha producido un error o la función se ejecutó sin problemas.
- 4Ah: Esta función permite cambiar el tamaño de un bloque de memoria previamente asignado, bien sea aumentándolo o reduciéndolo. El registro ES debe contener, como en el caso del servicio 4 9h, la dirección del segmento, mientras que BX contendrá el nuevo tamaño, siempre expresado en párrafos.

Cuando los servicios 4 8h y 4Ah notifican un fallo, activando el indicador de acarreo, existe la posibilidad de que la causa sea la falta de memoria. Es fácil saberlo, ya que en dicho caso el registro AX contendrá el valor 8. Lo más interesante, sin embargo, es que el sistema nos indicará, en el mismo registro BX, el número máximo de párrafos de memoria que es posible asignar.

Nota

Las funciones de asignación de memoria del DOS siempre operan sobre los 640 kilobytes de memoria del sistema y, en ocasiones, sobre un área de memoria que se conoce como *memoria superior* pero, en cualquier caso, siempre en el espacio de direccionamiento máximo de un megabyte. Esto significa que, por muchos megabytes de RAM que tenga nuestro sistema, en principio difícilmente podremos usar desde un programa DOS más de 640 kilobytes. La solución a esta limitación viene de mano de las especificaciones de memoria EMS y XMS, a las cuales está dedicado uno de los capítulos posteriores.

Un programa que ejecuta otros

Para finalizar este capítulo, vamos a ver en la práctica cómo podríamos desde un programa propio ejecutar otros cuya localización y nombre conocemos de antemano. Ya sabemos la función que debemos emplear, así como la problemática que plantea el hecho de que el sistema asigne toda la memoria disponible al programa que pone en marcha. Veamos los pasos que deberíamos dar para solucionar el problema.

Cuando el DOS pone en marcha un programa asigna un bloque de memoria, en realidad toda la memoria que hay disponible, y asigna la dirección a todos los registros de segmento a excepción de CS, que apuntará al segmento de código. Esto significa que DS y ES, por ejemplo, apuntan al área de memoria asignada por el DOS, pudiendo utilizarse esa dirección con el servicio 49h para liberar parte de la memoria, modificando el tamaño del bloque de memoria para reducirlo.

Aquí encontramos el segundo problema: ¿qué tamaño deberíamos dar al bloque de memoria en el que reside nuestro programa? Lógicamente podríamos hacerlo aleatoriamente, a cálculo, pero corremos dos riesgos: el menos grave es que usemos más memoria de la que necesitamos, el más grave que liberemos parte de la memoria donde reside nuestro código, datos o pila, caso en el que el programa, ineludiblemente, fallará y puede hacer fallar al sistema.

Necesitamos, por tanto, algún método para calcular de una forma lo más exacta posible la cantidad de memoria que realmente necesita nuestro programa. Para ello debemos saber que:

- El DOS crea al principio del bloque de memoria asignado para ejecutar un programa una estructura, conocida como PSP (*Program Segment Prefix*), que ocupa 256 bytes.
- Todos los segmentos definidos en el programa deben comenzar siempre en una dirección múltiplo de 16, lo que se conoce como *alineados a párrafo*. De no ser así, no podría accederse a ellos asignando la dirección de inicio a un registro de segmento que, como se trató en su momento, direccionan la memoria en párrafos en lugar de en bytes.
- La suma del PSP, más lo que ocupa cada uno de los segmentos para los ajustes por su alineamiento dan como resultado el número total de bytes que ocupa realmente el programa.

El PSP actúa como cabecera de todo programa ejecutado en DOS, manteniendo la línea de parámetros, punteros a las variables de entorno que afectan al programa, etc. Conocerá algo más de él en un capítulo posterior.

Conociendo estos hechos, calcular cuánta memoria necesitamos no debería resultarnos muy difícil. Basta con saber cuánta memoria ocupa cada uno de los segmentos, es fácil mediante el uso de etiquetas; sumar lo que ocupa el PSP y tener en cuenta que cada segmento comienza en una dirección múltiplo de 16 y, por tanto, pueden existir bytes intermedios que, aunque inutilizados, cuentan a la hora de reservar memoria.

Teniendo en cuenta lo que acaba de decirse, nuestro objetivo será crear un programa que supuestamente actuará como controlador de demostraciones, ejecutando una secuencia de programas externos. En nuestro caso, copiaremos en el directorio de este

programa los archivos puntocga.exe, colorega.exe y colorvga.exe creados en un capítulo previo, ejecutándolos desde éste. Para ello definiremos los datos que pueden verse a continuación.

```
; Segmento de pila
segment Pila stack
    resw 512
FinPila:

; Segmento de datos
segment Datos

; Este campo indicará el número
; de programas a ejecutar
NumProgramas dw 3

; Descripciones y nombres de los programas
Descl db 'Demostración de color en CGA$'
Prog1 db '.\puntocga.exe', 0
Desc2 db 'Demostración de color en ECA$'
Prog2 db '.\colorega.exe', 0
Desc3 db 'Efectos de color en VGA$'
Prog3 db '.\colorvga.exe', 0

; Tabla de punteros a las
; cadenas con anteriores
TblPunteros dw Descl, Prog1
                dw Desc2, Frog2
                dw Desc3, Prog3

; Bloque de parámetros para
; el servicio 4Bh de la int 21h
BloqueParametros
    dw 0 ; usar entorno del DOS
    dd LineaComandos ; cadena de parámetros
    dd FCB1, FCB2 ; FCBí

; La cadena de parámetros está vacía
LineaComandos db 0,13
; como los FCB
FCB1 resb 20
FCB2 resb 20

; Esta etiqueta marca el
; fin del segmento de datos
FinDatos
```

La variable NumProgramas indica el número de programas que van a ejecutarse, facilitándose a continuación una serie de parejas de cadenas con la descripción y nombre de cada uno de esos programas. Justo debajo se define una tabla de punteros con las direcciones de esas cadenas. Esto permite una adaptación muy fácil del programa, ya que basta con modificar NumProgramas, insertar las cadenas y añadir los punteros a TblPunteros para ejecutar otros programas distintos.

También en el segmento de datos se ha definido el bloque de parámetros, Bloque-Parámetros, necesario para invocar a la función 4Bh. Observe que la línea de comandos está vacía, pero podrían entregarse parámetros si fuese necesario. Para ello debería insertar la cadena detrás del byte 0, sustituyendo éste por el número de caracteres que tenga dicha cadena.

A continuación tenemos el inicio del bloque principal del segmento de código:

```
; Segmento de código
segment Código
..start:

; Configuramos los
; registros de pila
mov ax, Pila
mov ss, ax
mov sp, FinPila

; y los registros de
; segmento de datos
mov ax, Datos
mov ds, ax

; Ajustamos la memoria
; asignada a justo la
; que necesitamos
cali AjustaMemoria
```

Preparamos los registros de segmento de pila y datos, tras lo cual invitamos a la rutina AjustaMemoria. Observe que el valor de ES aún no se ha modificado, por lo que sigue apuntando al inicio del bloque de memoria asignado para el programa. El código siguiente es el de la rutina.

```
; F.sfa rutina calcula la memoria
; que necesita el programa y
; reajusta el bloque asignado
; por el sistema.
; Es importante no haber modificado
; el contenido del registro ES
; antes de llamar a esta rutina

AjustaMemoria:
    ; Tomamos en RX la dirección
    ; del final del código más
    ; 256 bytes del PSP
    mov bx,FinCodigo+256

    ; le añadimos lo que ocupa
    ; el segmento de pila
    add bx, FinPila
    ; y también lo que ocupa
    ; el segmento de datos
    add bx, FinDatos
```

```

; dividimos entre 16 para
; convertir a párrafos
shr bx,4

; sumamos un párrafo por
; cada segmento que tenemos:
; pila, datos y código, para
; compensar los ajustes
add bx, 3

; redimensionamo3 el bloque
; de memoria asignado
mov ah,4ah
int 21h

; y volvemos
ret

```

La etiqueta FinCodigo está al final del segmento de código. Sabiendo que todos los segmentos parten de una dirección de desplazamiento 0, una etiqueta que se sitúa al final tendrá como dirección el número de bytes que ocupa todo el código. A dicho número de bytes le sumamos los 256 que ocupa el PSP. A continuación sumamos lo que ocupa el segmento de pila y el segmento de datos, usando también la técnica de la etiqueta puesta al final.

El contenido de BX, en este momento, es el número de bytes que ocupan los tres segmentos más el PSP. Como lo que necesitamos son párrafos, no bytes, dividimos el contenido de BX entre 16 desplazando sus bits cuatro posiciones a la derecha. Acto seguido sumamos tres párrafos para compensar los ajustes a párrafo que pudiesen ser necesarios, son sólo 48 bytes de memoria, y utilizamos el servicio 4AH para ajustar el tamaño del bloque de memoria que nos asignó el DOS. En este momento ya hay memoria libre en el sistema.

Seguimos con el bloque principal de código:

```

; Hacer que BS también
; apunte a los datos
push ds
pop es

; obtenemos el contador
; de programas a ejecutar
mov ex, [NumProgramas]

; BX apunta a la tabla
; de punteros a cadenas
mov bx, TblPunteros

```

Bucle:

```

; obtenemos en DX la
; dirección de una cadena
; con la descripción del
; programa a ejecutar
mov dx,'[bx]

```

```

; la mostramos
mov ah, 9
int 21h
; y esperamos una tecla
mov ah, 0Ch
mov al, 7
int 21h

; nos desplazamos al
; siguiente puntero
add bx, 2

; recuperamos la dirección
; del nombre del programa
mov rix, [bx]
; lo ejecutamos
cali Ejecuta

; pasamos al siguiente puntero
add bx, 2
; y repetimos el proceso
loop Bucle

; Devolvemos el control
; al sistema
mov ah, 4ch
int 21h

```

Tenemos un bucle que se repite tantas veces como indique NumProgramas, recorriendo la tabla de punteros TblPunteros. Acada ciclo tomamos la dirección de la descripción y la imprimimos, esperamos la pulsación de una tecla, avanzamos al siguiente puntero y, almacenándolo en DX, invocamos a la rutina Ejecuta, que será la encargada de ejecutar el programa.

La rutina Ejecuta recibe en DX la dirección de la cadena con el nombre del programa a ejecutar, por lo que nos limitamos a asignar el valor 0 a AL y la dirección del bloque de parámetros a BX, usando el servicio 4Bh para transferir el control temporalmente. Cuando el otro programa termine, la rutina recupera los registros y devuelve el control al bloque principal.

```

; Esta rutina recibe en DX la
; dirección de una cadena con el
; nombre del programa a ejecutar

Ejecuta:
; guardamos registros
push ds
push es
puflha

; ponemos AL a 0 para indicar
; que deseamos ejecutar
xor al, al

```

```

; número del servicio
mov ah, 4Bh

; DS:DX ya tiene la dirección
; del nombre del programa
; ES:BX debe apuntar al
; bloque de parámetros

; dirección del bloque
mov bx, BloqueParametros
; ejecutamos
int 21h

; recuperamos los registros
popa
pop es
pop ds

rct ; y volvemos

; Esta etiqueta marca el final
; del segmento de código
FinCodigo:

```

Tras ensamblar este programa, no olvide colocar en el mismo directorio los programas antes citados. Ejecute el programa principal y compruebe el resultado. En este caso no se han comprobado los posibles errores, algo indispensable en aplicaciones reales ya que la modificación del bloque de memoria o la invocación a la función A Bh pueden fallar.

Resumen

En este primer capítulo dedicado a los servicios del DOS ha conocido algunas de las interrupciones mediante las cuales este sistema ofrece sus funciones, haciendo especial hincapié en la interrupción 21h que, por sus características, es sin duda alguna la más importante. También ha conocido ciertos servicios básicos, como los de entrada y salida de caracteres a dispositivos como la consola, impresora o puerto serie.

Especialmente interesantes resultan los servicios de gestión de memoria, manipulación de vectores de interrupción y control de procesos. Ha visto un ejemplo de cómo usar algunos de esos servicios para reducir la memoria asignada a un programa y ejecutar otros desde éste. El ejemplo, además, puede extenderse fácilmente.

Los capítulos siguientes le servirán para conocer otros servicios, también de la interrupción 21h, relativamente más complejos. Con ellos podrá almacenar y recuperar datos en archivos en disco, acceder directamente a sectores de las unidades de disco, aprovechar la memoria que tenga su sistema más allá de los 640 kilobytes básicos o crear aplicaciones que quedan residentes en memoria.

24

Tratamiento de archivos

La finalidad de la mayoría de las aplicaciones estaría muy limitada si cada vez que finalizasen se perdiera toda la información sobre la que han estado operando. La solución, lógicamente, pasa por almacenar dicha información en un medio persistente, como puede ser un disco o disquete.

El acceso a los archivos desde DOS puede efectuarse por medio de dos sistemas diferentes.

El primero, y más antiguo, consiste en utilizar los FCB (*File Control Block*), unas estructuras de datos que contienen todo lo concerniente al archivo que va a manipularse. Ésta es una técnica incómoda y, tal como se ha dicho, bastante anticuada, de los días del DOS LO.

La técnica alternativa es la que forman los servicios basados en manejadores de archivo *o file handles*. Es una técnica mucho más cómoda de uso, ya que no tenemos que preocuparnos de detalles de bajo nivel.

El único inconveniente es que no puede utilizarse en versiones de DOS previas a la 2.0, una limitación que realmente no es tal puesto que difícilmente nadie usa en la actualidad una versión de DOS previa a la 6.2 o 5.0.

En este capítulo se van a tratar aquellas funciones de la interrupción 21h que tienen que ver con el tratamiento de los archivos basándose en el uso de manejadores de archivos.

Para facilitar el nombre de los archivos con los que se desea trabajar se emplearán cadenas ASCIIZ, es decir, secuencias de caracteres conteniendo el nombre, si es preciso camino y extensión, y terminadas con un byte 0.

Apertura y creación de archivos

^^

Al crear un nuevo archivo, o abrir uno ya existente, deberemos facilitar el nombre, en forma de cadena ASCIIZ, en los registros DS : DX. Una vez que el archivo se haya abierto, o creado, el registro AX contendrá el manejador que nos permitirá movernos por él, leer y escribir datos y cerrarlo.

En caso de que tras la llamada se haya activado el indicador de acarreo sabremos que se ha producido un error, en este caso el contenido de AX no será el manejador sino un código de error.

En los puntos siguientes se indican los servicios que tenemos a nuestra disposición para crear archivos, borrándolos o no en caso de que existiesen; crear archivos temporales con un nombre indefinido, abrir archivos existentes y cerrarlos.

Creación de un nuevo archivo

Para crear un archivo nos serviremos del servicio 3Ch, que toma el nombre del archivo y lo crea incluso aunque ya existiese, borrando su contenido. Los parámetros que necesita son los siguientes:

- DS : DX: Dirección de la cadena ASCIIZ con el nombre del archivo, incluyendo el camino si fuese necesario.
- CX: Atributos que tendrá el nuevo archivo.

El valor del registro CX es una máscara en la que cada bit indica la existencia o no de un cierto atributo.

La tabla 24.1 resume el significado de cada uno de esos bits. Si tras invocar al servicio el indicador de acarreo está a 1, AX contendrá uno de los valores enumerados en la tabla 24.2 comunicándonos cuál es el problema.

Si el indicador de acarreo está a 0, el registro AX contendrá el manejador del archivo creado y abierto.

Tabla 24.1. Significado de los bits de atributo.

Bit	Atributo asociado
0	Archivo sólo de lectura.
1	Archivo oculto.
2	Archivo de sistema.
3	Etiqueta de volumen.
4	Directorio.
5	El archivo ha sido salvaguardado.

Tabla 24.2. Códigos de error devueltos en el registro AX.

Código	Significado
3	No se encuentra el camino indicado.
4	No pueden abrirse más archivos.
5	El archivo es sólo de lectura y no puede eliminarse para volver a crearlo.

Con el servicio 3Ch el archivo indicado se crea incluso aunque ya existiese, borrando su contenido de ser así. Esto puede ser algo indeseado y causar la pérdida de una información que, realmente, queríamos mantener. Suponga que quiere crear el archivo pero, si ya existe, optar por abrirlo. En este caso usaríamos el servicio 5Bh que, tomando y devolviendo los mismos parámetros que el anterior, devolverá en AX el código de error 5Oh en caso de que el archivo ya exista, en lugar de eliminarlo. Este servicio, por tanto, creará el archivo siempre que sea nuevo, es decir, que no haya uno con el mismo nombre en el mismo camino.

Las funciones 3Ch y bBh abren un archivo nuevo, creándolo, por lo que se asume que vamos exclusivamente, a escribir en él ya que no tiene un contenido previo.

Creación de archivos temporales

En ocasiones, el archivo que va a crearse es simplemente para alojar temporalmente alguna información y, por ello, nos da igual el nombre que se le asigne siempre que no coincida con el de un archivo ya existente. En este caso podemos servirnos del servicio 5Ah. Facilitaremos en DS: DX una cadena con el camino donde alojar el archivo y los atributos en ex. Si todo va bien, recibiremos en DS : DX la dirección de una cadena con el nombre del archivo creado y en AX el manejador.

Como en los servicios anteriores, la activación del indicador de acarreo comunicará la existencia de algún error, cuyo código encontraremos en el registro AX. Lo más habitual es que el camino indicado en DS : DX sea incorrecto o no pueda accederse a él por alguna circunstancia.

Apertura y cierre de archivos

Las tres funciones explicadas en los puntos previos abren un archivo tras haberlo creado, estableciéndose el camino, nombre y atributos de dicho archivo. En caso de que deseemos abrir un archivo existente, para leer su contenido, añadir información o ambas

cosas, nos serviremos del servicio 3Dh. Además del nombre completo del archivo en forma de cadena ASCIIZ, en los registros DS : DX, deberemos indicar, en el registro AL, el modo de apertura del archivo.

Los modos de apertura posibles son:

- 0: El archivo se abre sólo para leer de él.
- 1: El archivo se abre sólo para escribir en él.
- 2: El archivo se abre tanto para operaciones de lectura como de escritura.

A diferencia de la función de creación 5Ah, que devuelve un error en caso de que el archivo a crear ya exista, la función de apertura fallará en caso de que el archivo a abrir no exista. En este caso el código de error que encontraremos en AX será el 2. También puede encontrarse alguno de los antes indicados en la tabla 24.2, por ejemplo si intentamos abrir para escritura un archivo que es sólo de lectura o si no es posible la apertura porque ya haya muchos archivos abiertos.

Todos los archivos abiertos deben cerrarse cuando no vaya a seguir operándose sobre ellos, liberando así el manejador correspondiente que podría ser necesario con posterioridad para abrir otro archivo.

La función de cierre es la 3 Eh y sólo precisa como parámetro el manejador en el registro BX. El único error posible es que hayamos facilitado un manejador incorrecto, de un archivo que no está abierto.

Lectura y escritura de datos

Una vez que tengamos abierto el archivo, ya sea nuevo o no, necesitaremos conocer las funciones que nos permitan leer y escribir datos en él. El sistema mantiene por cada archivo abierto un puntero que indica la posición a partir de la que se leerá o en la que se escribirá. Inicialmente ese puntero está al inicio del archivo, desplazándose hacia adelante con cada operación de lectura o escritura. También existe un servicio que nos permite fijarlo donde nos interese en cada momento, así como conocer la posición en que nos encontramos.

Para desplazar el puntero usaremos el servicio 42h. Los parámetros necesarios se facilitarán en los registros siguientes:

- BX: Manejador del archivo, obtenido en AX tras la operación de apertura.
- AL: Punto de partida del desplazamiento.
- CX: DX: Número de bytes a desplazarse.

El registro AL contendrá uno de los valores citados en la tabla 24.3, indicando el punto a partir del cual se desplazará el número de bytes que contienen los registros CX: DX. Dicho número siempre será positivo, por lo que no es posible retroceder directamente, usando un desplazamiento negativo, pero sí utilizando posicionamiento absoluto.

Tabla 24.3. Códigos de inicio de desplazamiento.

Código	Significado
0	El desplazamiento es absoluto, contando desde el inicio del archivo.
1	El desplazamiento es relativo, contando desde la posición actual.
2	El desplazamiento es relativo contando desde el final del archivo.

Si todo va bien, tras la llamada al servicio 4 2h los registros DX: AX contendrá la actual posición del puntero en el archivo. Es fácil, por lo tanto, usar el método de desplazamiento 1, relativo a la posición actual, asignando el valor 0 a los registros CX: DX consiguiendo así en DX: AX la actual posición sin cambiarla. Los servicios de lectura y escritura son el 3Fh y 4 Oh, respectivamente. Los parámetros necesarios son exactamente los mismos:

- BX: Manejador del archivo en el que va a escribirse o del que se va a leer.
- CX: Número de bytes a leer o escribir.
- DS : DX: Dirección de un área de memoria en la que se dejarán los datos leídos o en la que están los datos a escribir.

Tras la invocación a estas funciones encontraremos en AX el número de bytes efectivamente leídos o escritos. Si este número es inferior al indicado en CX, en el momento de la llamada, significará que se ha llegado al final del archivo, en el caso de la lectura, o bien que el dispositivo de destino está lleno, y no se han podido escribir todos los datos. También es necesario comprobar el indicador de acarreo, ya que su activación nos comunicará que se ha producido algún fallo que ha impedido la ejecución satisfactoria de la función.

Guardar y restaurar pantallas

Veamos en la práctica el uso de un buen número de los servicios que hemos conocido mediante un ejemplo. Realmente serán dos los programas. La finalidad del primero será escribir en un archivo el contenido actual de la pantalla de texto. Dicho archivo será creado, en caso de no existir, o bien se le añadirá información al final, si ya existiese. El archivo, por tanto, irá creciendo de tamaño a medida que ejecutemos este primer programa, cuyo código tiene a continuación, si bien puede borrarlo en cualquier momento ya que el programa volverá a crearlo cuando sea necesario.

```

segment Pila stack
    resw 512
FinPila:

segment Datos
; Nombre del archivo donde
; va a guardarse la pantalla
Nombre db 'Pantalla.dat',0

```

```

; Para guardar el manejador
Manejador dw ü

segment Código
..start:
    mov ax, Datos
    mov ds, ax

    ; DS:DX apuntan al nombre
    ; del archivo
    mov dx, Nombre

    ; Lo abrimos para lectura
    ; y escritura
    mov ah, 3Dh
    mov al, 2
    int 21h
    ; si no hubo error saltar
    jnc Abierto
    ; si hubo error creamos
    ; el archivo
    mov dx, Nombre
    mov ah, 3Ch
    xor ex, ex ; atributos normales

    int 2ih

Abierto:
    ; Guardamos el manejador
    ; de acefifio al archivo
    mov bx, ax

    ; movemos el puntero
    ; de lectura al final
    ; del archivo
    mov al, 2
    xor ex, ex
    xor dx, dx
    mov ah, 42h
    int 21h

    ; DS:DX apuntan a la pantalla
    mov ax, 0B800h
    mov ds, ax
    xor dx, dx

    ; guardar 4000 bytes
    mov ex, 4000

    ; los escribimos
    mov ah, 40h
    int 21h

    ; cerramos el archivo
    mov ah, 3Eh
    int 21h

```

```
; devolvemos el control  
; al sistema  
mov ah, 4Ch  
int 21h
```

El objetivo del segundo programa será abrir el archivo creado por el primero, mostrando en secuencia las pantallas que contenga. Para ello irá leyendo usando como área de destino la memoria de pantalla, esperando una tecla entre pantalla y pantalla. Cuando el intento de lectura devuelva en AX el valor 0 se habrá llegado al fin del archivo, momento en que se pone final al proceso.

El código, ampliamente comentado, es el siguiente:

```
seyínen Pila ytauk  
resw 512
```

FinPila:

```
segment Datos  
; Nombre del archivo donde  
; se encuentran las pantallas  
Nombre db 'Pantalla.dat',0  
  
; Mensaje de error  
Error db 'El archivo PANTALLA.DAT no existe.$'
```

```
segment Código  
. . start:  
    mov ax, Datos  
    mov ds, ax  
  
    ; DS:DX apuntan al nombre  
    ; del archivo  
    mov dx, Nombre  
  
    ; Lo abrimos para lectura  
    mov ah, 3Dh  
    xor al, al  
    int 21h  
  
    ; si hubo error saltar  
    je NoExiste  
  
    ; si no hubo error guardamos  
    ; el manejador en BX  
    mov bx, ax  
  
    ; DS apuntará al segmento  
    ; de pantalla  
    mov ax, ÜB8Ü0h  
    mov ds, ax
```

```
Bucle:  
    ; a partir del byte 0  
    xor dx, dx  
    ; 4 000 bytes  
    mov ex, 4000
```

```

; los leemos
niov ah, 3Fh
int 21h

; ¿Es AX = 0?
or ax, ax
; de ser asi terminar
jz FinBucle

; en caso contrario
; esperar una tecla
xor ah, ah
int 16h

;- y continuar con la
; pantalla siguiente
jmp Bucle

FinBucle:
; cerramos el archivo
mov ah, 3Eh
int 21h

jmp Fin

NoExiste:
; ai el archivo no existe
; imprimir el error
mov dx, Frror
mov ah, 9
int 21h

Fin:
; salir al DOS
mov ah, 4Ch
int 21h

```

Ejecute el programa varias veces modificando el contenido de la pantalla de texto. Si quiere, pueda examinar el contenido del archivo Pantalla.dat con algún editor. A continuación, ejecute el segundo programa y observe cómo va mostrando las pantallas previamente almacenadas en dicho archivo.

Borrado, renombrado y otras operaciones con archivos

Además de las funciones para abrir, leer, escribir y cerrar archivos, el DOS cuenta con otras que nos permiten manipularlos como un todo, borrándolos o renombrándolos por ejemplo. Su uso es bastante más sencillo ya que, básicamente, tan sólo hay que facilitar el nombre del archivo afectado, en DS : DX, y poco más. Estos servicios son los descritos a continuación:

- 4 1h: Elimina el archivo cuyo nombre se facilita en DS : DX. El indicador de acceso comunicará el éxito o fracaso de la operación.
- 5 6h: Este servicio cambia el nombre de un archivo, renombrándolo. El nombre actual se entregará en DS : DX y el nuevo ES : DI. Tenga en cuenta que no puede cambiar la localización del archivo con esta función, sólo el nombre.
- 4 3h: Al crear un archivo se establecen unos atributos, según se indicaba anteriormente, que podemos leer y modificar con este servicio. Como en los demás casos, DS: DX apuntará al nombre del archivo, mientras que AL indicará si se quieren leer los atributos, con el valor 0, o modificarlos, con el valor 1. En cualquier caso, los atributos siempre residirán en el registro CX.

Aunque no existe un servicio que facilite la copia directa de archivos, es relativamente fácil hacerlo mediante las funciones de apertura, creación, lectura y escritura vistas antes. Más adelante, en este mismo capítulo, verá un ejemplo de ello.

Unidades y directorios

Los archivos son elementos que se almacenan en unidades de disco, organizándose en estructuras jerárquicas conocidas como directorios.

Las unidades se identifican mediante letras cuando se hace referencia a ellas en la línea de comandos del DOS; pero los servicios de la interrupción 21h utilizan números en su lugar. En cuanto a los directorios, básicamente se tratan como archivos que contienen listas de archivos.

En la interrupción 21 h encontramos funciones que nos permiten saber sobre qué unidad estamos operando, seleccionar otra distinta, cambiar de un directorio a otro, crear y eliminar directorios, leer las listas de archivos que contienen, etc. Los puntos siguientes detallan el uso de algunos de estos servicios.

La unidad por defecto

Cuando va a manipularse un cierto archivo, ya sea abriendolo, eliminándolo o renombrándolo, debemos facilitar, como ya sabe, una cadena en DS : DX con el nombre del archivo. Este nombre puede contener un camino completo, incluyendo la letra de unidad en la que se encuentra el archivo. De no indicarse, siempre se usará la unidad por defecto que, inicialmente, es aquella desde la que se ha puesto en marcha el programa.

Puede saber cuál es actualmente la unidad de disco por defecto invocando al servicio 19h. No se precisa ningún parámetro, obteniéndose en AL un número que será 0 para la unidad A:, 1 para la B:, 2 para la C: y así sucesivamente.

Cambiar la unidad por defecto es igualmente fácil. En este caso el servicio a utilizar es el OEh, facilitando en el registro DL el número de unidad siguiendo la misma notación del servicio anterior. Se recibirá, tras la llamada, el número total de unidades que

pueden existir en el registro AL. Este dato no implica que ese número de unidades exista físicamente, sino que puede existir porque hay letras de unidad disponibles.

El siguiente ejemplo, que hace uso de la rutina EnteroCadena creada en un capítulo previo, muestra por pantalla el número de unidades que hay disponibles. En las últimas versiones del DOS lo habitual es obtener el resultado que puede verse en la figura 24.1 a menos que se haya establecido un límite inferior en el archivo config.sys.

```

segment Pila stack
    resw 512
FinPila:

segment Datos

; Mensaje para indicar el número
; de unidades
Unidades db 'Hay'
Número db ' '
    db ' unidades disponibles.$'

segment Código
.start:
; DS y ES apuntan al
; segmento de datos
    mov ax, Datos
    mov ds, ax
    mov es, ax

; obtenemos la unidad por
; defecto actual
    mov ah, 19h
    int 21h

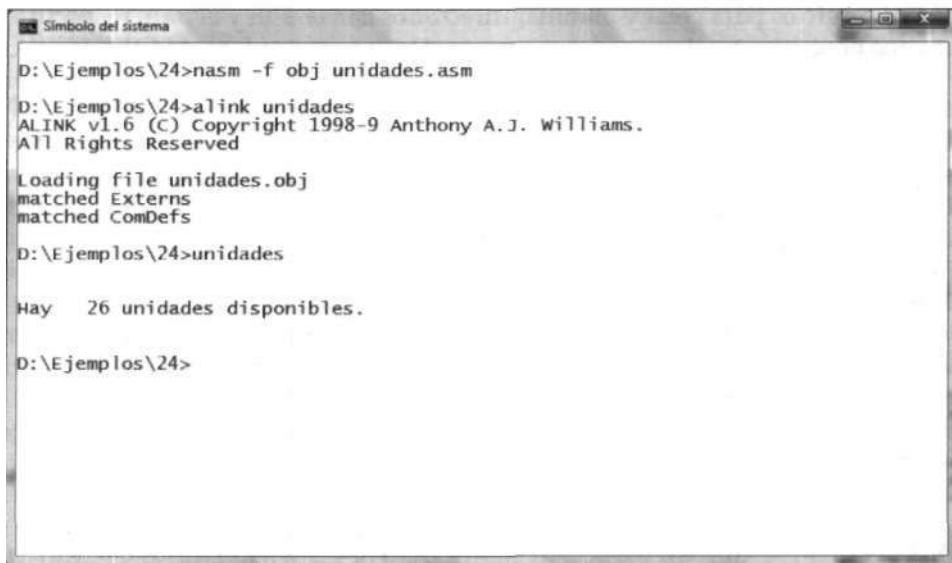
; y la usamos para
; establecerla
    mov di, al
    mov ah, OEH
    int 21h

; obteniendo el número de
; unidades disponibles
    xor ah, ah
    mov di, Número+4
; lo convertimos
    cali EnteroCadena

; y mostramos
    mov dx, Unidades
    mov ah, 9
    int 21h

; devolvemos el control
; al sistema
    mov ah, 4Ch
    int 21b

```



```
D:\Ejemplos\24>nasm -f obj unidades.asm
D:\Ejemplos\24>alink unidades
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file unidades.obj
matched Externs
matched ComDefs

D:\Ejemplos\24>unidades

Hay 26 unidades disponibles.

D:\Ejemplos\24>
```

Figura 24.1. El programa nos indica el número de unidades disponibles.

El directorio actual

Lo mismo que sucede con la unidad, que si no se indica explícitamente se utiliza la establecida por defecto, ocurre también con el directorio. Si queremos abrir o crear un archivo y no indicamos explícitamente un camino, se asume que el camino es el que corresponde al directorio considerado como actual. Podemos tanto obtener como modificar dicho directorio, ya sea en la unidad actual o en cualquier otra.

Para obtener el camino del directorio actual se emplea el servicio 47h. El primer parámetro necesario se facilitará en DL, indicando la unidad cuyo camino quiere obtenerse. En este caso el valor 0 indica la unidad por defecto, la actual, mientras que los valores siguientes harían referencia a las unidades A:, B:, etc. El segundo, facilitado en DS : SI, será la dirección de un área de memoria con al menos 64 bytes libres en los que se introducirá el camino actual. Si vamos a operar reiteradamente sobre archivos que se encuentran en un cierto directorio, lo más fácil es que lo establezcamos como actual. Para ello facilitaremos al servicio 3Bh, en los registros DS : DX, la dirección de una cadena ASCIIZ indicando el nuevo camino. De no existir dicho camino, se activará el indicador de acareo y el registro AX contendrá el valor 3.

Creación y borrado de directorios

Los directorios, al igual que los archivos pero a diferencia de las unidades, pueden crearse y eliminarse. Su contenido, además, puede leerse, como se explica en el punto siguiente.

Los servicios para crear y eliminar directorios son el 39h y el 3Ah, respectivamente. Los dos necesitan exactamente el mismo parámetro: una cadena ASCIIZ con el camino, ya sea absoluto o relativo, del directorio que quiere eliminarse o crearse. Dicha dirección se entregará en los registros DS : DX.

Como ocurre con otras funciones ya descritas, el estado del indicador de acarreo, tras la instrucción int 21h, nos comunicará si la operación ha tenido éxito o no. La creación de un directorio fallará si el camino es incorrecto o el directorio ya existe, mientras que la eliminación puede producir un error por más causas: que el directorio no esté vacío, que el camino corresponda al directorio establecido actualmente como directorio por defecto, etc.

Una vez se ha creado un directorio, en él pueden crearse archivos usando cualquiera de los servicios descritos en los puntos previos. Esto equivaldría a añadir entradas a dicho directorio, estableciendo el nombre, atributos y fecha de creación de cada archivo. Esas entradas pueden recuperarse para saber cuál es el contenido de un directorio, leyendo su contenido.

Archivos existentes en un directorio

Cada vez que utiliza la orden dir del DOS, o alguna otra aplicación facilitándole una referencia en la que se usan los caracteres comodines * y ?, el sistema utiliza una pareja de servicios que le permiten localizar la lista de archivos que se ajustan a dicha referencia.

Esas funciones operan siempre sobre un área de memoria conocida como DTA (*Disk Transfer Área*), que es necesario haber establecido de antemano. Inicialmente, cuando se pone en marcha una aplicación, el sistema establece la DTA a partir del byte 129 del PSP, es decir, la misma zona donde se encuentran los parámetros de entrada al programa.

Podemos modificar la localización de la DTA, estableciendo la que nos interese a nosotros, mediante el servicio 1 Ah. El único parámetro necesario es la nueva dirección, que debe introducirse en la pareja de registros DS : DX. A partir de ese momento, las dos funciones descritas a continuación escribirán la información a partir de dicha dirección.

La recuperación de las entradas existentes en un cierto directorio, o de parte de ellas, se efectúa en dos pasos: primero se invoca al servicio 4Eh para encontrar la primera entrada que coincide con una determinada referencia, usándose después de manera repetida el servicio 4Fh para ir leyendo las entradas siguientes. El servicio 4En establece la referencia de búsqueda, formada por una cadena con el nombre del archivo y un atributo. El primer dato se entregará en DS : DX, pudiendo contener los citados caracteres comodines, mientras que el segundo se facilitará en CX indicando el tipo de entradas que se buscan.

En caso de que exista al menos una entrada que cumpla la referencia, el indicador de acarreo estará a cero y el nombre del archivo podrá encontrarse a partir del byte 30 del área de memoria establecida como DTA. Acto seguido se invocará repetidamente a la función 4Fh sin entregar parámetro alguno, simplemente observando que el indicador de acarreo cambie de estado ya que, cuando se active, indicará que no hay más entradas disponibles.

UDisk

Con el objetivo de ver en funcionamiento muchas de las funciones que se han descrito en este capítulo, vamos a construir un programa de utilidad que nos facilitará la ejecución de operaciones comunes en archivos. El nombre del programa será UDisk. Con él podremos ver el contenido de un directorio, cambiar de unidad o de directorio, borrar, renombrar y copiar archivos, etc.

La extensión y complejidad del código de este programa es bastante superior comparativamente hablando con los propuestos hasta el momento. Por ello se ha dividido en múltiples módulos que van a detallarse a continuación. Se han incluido en el propio código todos los comentarios que son necesarios para que pueda ir viendo el código y leyendo las explicaciones sobre lo que hace, por ello se facilita a continuación todo el programa completo, para facilitarle su lectura.

El programa va a mostrar información en pantalla y solicitarla por teclado de manera intensiva y, por ello, se han preparado unas rutinas, alojadas en archivos independientes, que faciliten estas tareas.

El primero de los archivos, y el más simple, es Colores.inc. En él, como se ve a continuación, simplemente se definen una serie de constantes a fin de no tener que recordar de memoria el código de cada color que necesitemos utilizar.

```
; Constantes para facilitar el uso
; de los colores

AZUL    equ    1
VERDE   equ    2
ROJO    equ    4
CYAN    equ    AZUL+VERDE
MAGENTA equ    ROJO+AZUL
BLANCO   equ    ROJO+VERDE+AZUL
REVERSE  equ    112
BRILLO   equ    00001000b
PARPADEO equ    10000000b
```

En el módulo Borra.inc alojaremos una sencilla rutina cuya única finalidad será borrar la pantalla:

```
; Esta rutina borra la
; pantalla asumiendo que
; el modo siempre es el 3

BORRA:
    xor dx, dx ; ponemos el cursor
    mov ah, 2 ; al principio
    int 10h ; de la pantalla

    mov ex, 2000 ; e imprimimos
    mov al, ' ' ; 2000 espacios
```

```

xor bh, bh
mov bl, 7
mov ah, 9
int 10h

ret ; volver

```

Estos módulos contienen código de uso general, que resulta útil en esta aplicación pero podría, igualmente, aprovecharse en cualquier otra. Lo mismo puede decirse de la rutina EnteroCadena, concretamente la última versión creada en un capítulo previo, que alojaremos en un módulo al que llamaremos Convert.inc.

La siguiente rutina también puede aplicarse a cualquier programa siempre que este defina una estructura con el formato requerido. Se encuentra en el archivo Imprímememe y su finalidad es imprimir un texto en una cierta posición y con un color determinado. Para ello espera recibir en DS : SI la dirección de una lista de parámetros, tal y como se indica en la propia cabecera de la rutina.

```

; Rutina de impresión por pantalla
•
; Al invocar a esta rutina DS:SI
; deben contener la dirección de
; una cadena de bytes con la
; estructura siguiente:
; Linea, Columna, Color, "Texto", 0

IMPRIME:
    ; recogemos la linea en DH
    mov dh, [si]
    inc si ; apuntar al dato siguiente
    ; recoger la columna en DL
    mov di, [sil]

    ; posicionamos el cursor
    mov ah, 2
    xor bh, bh ; página 0
    int 10h

    inc si ; pasar al siguiente dato
    ; recoger el color en BL
    mov bl, [si]
    ; apuntar al primer byte de texto
    inc si
    xor bh, bh ; trabajar en página 0

IMP_BUCLE:
    ; recoger el carácter apuntado por ST
    mov al, [si]
    ; comprobar si es el final
    or al, al
    ; de ser así saltar
    jz IMP_FIN

```

```

; en caso contrario imprimirla
mov ex, 1
mov ah, 9
int 10h

; pasar al siguiente carácter
inc si
; leemos la posición actual
; del cursor
mov ah, 3
int 10      ^,
; incrementamos la columna
inc di
; y recolocamos el cursor
mov ah, 2
int 10h

; repetir
jmp IMP_BUCLF.

IMP_FIN:
; cuando se termine la cadena
ret ; volver

```

Observe que no se usan las funciones que imprimen directamente una cadena de caracteres, implementando todo el proceso mediante código propio. Lo mismo ocurre con la rutina siguiente, contenida en el módulo Peticion.inc, que solicita una cadena de caracteres utilizando la BIOS en lugar de los servicios del DOS, a fin de obtener un mejor control. La rutina también recibe en DS : SI la dirección de una lista de parámetros que le indican en qué posición debe pedir el dato y cuál será su longitud máxima.

```

; La finalidad de esta rutina es
; facilitar la introducción de datos

; Al llamarla DS:SI deben apuntar a
; un descriptor con la siguiente
; estructura:

; Línea, Columna, Color, Longitud, Buffer

PETICIÓN:
; llamar a la rutina LIMPIA
cali LIMPIA

; lleva la dirección de
; descriptor a BX
mov bx, si
; y hacer que SI apunte
; al buffer de almacenamiento
mov si, 4

; guardamos la dirección
; del descriptor del dato
push bx

```

```

; recogemos la línea
mov dh, [bx]
; y la columna
mov di, fbx+1]
xor bh, bh ; página 0
; colocamos el cursor
mov ah, 2
int 10h

; recuperamos la dirección
; del descriptor
pop bx

BUC_PET1:
; esperamos la pulsación
; de una tecla
xor ah, ah
int 16h

; ¿se pulsó Intro?
cmp al, 13
; de ser así saltar
je FTN_PET

; ¿se pulsó la tecla de retroceso?
cmp al, 8
; si no es así saltar
jne SALT_PET1

; Se ha pulsado la tecla
; de retroceso

; Ver si entramos al principio
; del dato
cmp si, 4
; de ser así ignoramos la pulsación
I I e RUC_PF,Tl

; en caso contrario retrocedemos
; reduciendo el puntero
dec si

; guardamos la dirección
; del descriptor
push bx

; leemos la posición actual
; del cursor en pantalla
mov ah, 3
xor bh, bh
int 10

; hacemos retroceder el cursor
dec di
mov ah, 2
int 10h

```

```
; recuperamos la dirección
; del descriptor
pop bx

; continuamos con la petición
jmp BUC_PET1

SALT_PET1: ; No se ha pulsado Intro ni retroceso

; ¿es la flecha hacia arriba?
cmp ax, 4800h
; en caso afirmativo salta
je ARRIBA

; comparar' la Léela con el código 31
cmp al, 31
; si es menor o igual ignorarlo
jle BUC_PET1

; comparar la tecla con el código 123
cmp al, 123
; si es mayor o igual ignorarlo
jge RTJC_PET1

; comparar con el código 96
cmp al, 96
; si es menor o igual saltar
jle SALT_PET

; en caso contrario es una letra
; minúscula y la convertimos
; a mayúscula
and al, 1101111b

SALT_PET:
; Si es un carácter válido introducirlo
; en el buffer.
; BX contiene la dirección de inicio
; del descriptor y SI el número de
; carácter pulsado más 4, que es el
; comienzo del buffer dentro del descriptor
mov [bx+si],al

;• guardamos la dirección
; del descriptor
push bx

;• tomamos el byte de color
mov bl, [bx+2]
xor bh, bh ; página 0
mov ex, 1

; imprimimos el carácter
; que se ha pulsado
mov ah, 9
int 10h
```

```

; incrementamos SI, que apunta
; al siguiente byte del buffer
inc si

; recuperamos la dirección
;- del descriptor
pop bx

; restamos 4 a SI para saber
; la longitud actual del dato
sub si, 4

; leemos el byte de longitud
; del descriptor
mov al, [bx+3]
; convertimos en palabra
; para poder comparar con SI
xur ah, ah

; ver si ya tenemos el número
; de caracteres solicitado
emp ax, si
; de ser así saltar
je FIN_PET

; en caso contrario volver
; a dar su valor a SI
add si, 4

; guardar la dirección
;- del descriptor
push bx

; leemos la posición actual
; del cursor en pantalla
mov ah, 3
xor bh, bh
int 10h

; incrementamos la columna
me di

; y volvemos a colocarlo
mov ah, ?
int 10h

; recuperamos el descriptor
pop bx
; y continuamos pidiendo caracteres
jmp BUC_PET1

```

ARRIBA:

```

; si se pulsó la tecla del cursor
; hacia arriba devolver un 1 en AL
; para notificarlo
mov al, 1 .

```

```

; y terminar
ret

FIN_PET:
; si se ha terminado por la pulsación
; de Intro o al llegar a la longitud
; máxima, damos 0 a AL
xor al, al

ret ; volver

; Esta rutina se encarga de
; preparar el dato que va a
; pedirse

LIMPIA:
; tomar la dirección del dato en DI
mov di, si
; DS y ES apuntan al mismo segmento
push ds
pop es

; incrementar DI
add di, 3
; para leer la longitud del dato
mov el, [di]

; ahora tenemos en CX la longitud
xor oh, oh

; hacer que DI apunte al
; primer carácter
inc di

cid ; autoincremento de DI
mov al, ' ' ; llena con espacios
rep stosb

ret ; volver

```

Estos módulos los añadiremos al archivo principal, udisk.asm, mediante directivas %include, insertando las líneas siguientes al final de dicho archivo.

```
%include "Borra.inc"
%include "Imprime.inc"
%include "Petición.inc"
%include "Convert.inc"
```

Conociendo ya el contenido de los módulos .inc, nos centramos ahora en el programa principal, comenzando por su segmento de datos. Como puede ver, existen múltiples mensajes de aviso y solicitud de datos, así como estructuras para componer el aspecto de la pantalla y solicitar datos. Cada uno de los campos definido está precedido de una explicación sobre cuál es su finalidad.

```

; UDISK 1.0

; Programa de utilidad para manipular
; el sistema de archivos

%include "Colores.inc"
*****
; Segmento de pila

segment Pila stack
    resw 512
FinPila:

; Segmento de datos

segment Datos

,- Mensaje de derechos y
; varios avisos

COPYRIGHT db 'UDISK 1.0',13,10
        db 'Utilidad para manipulación de archivos'
        db 13,10
        db 'Francisco Charte Ojeda',13,10,10,'$'

MEN1      db 'Leyendo el directorio ...$'
MEN2      db 'Directorio vacio.$'
MEN3      db 22,2G,AZUL,'Ordenando ...',0
MEN4      db 21,25,REVERSE + AZUL
        db 'Introduzca la nueva referencia ', 0
MEN5      db 21,25,REVERSE + AZUL
        db 'Introduzca el camino del nuevo directorio ', 0
MEN6      db 22,25,REVERSE + AZUL
        db 'Se produce un error. Pulse una tecla.', 0
MEN7      db 21,25,REVERSE + AZUL
        db 'Introduzca la letra de la nueva unidad ', 0
MENB      db 21,25,REVERSE t AZUL
        db 'Borrar (T)odos o (U)no ', 0
MEN9      db 21,25,REVERSE + AZUL
        db 'Escriba el nuevo nombre del archivo ', 0
MENO     db 21,25,REVERSE + AZUL
        db 'Escriba el nombre del archivo destino ', 0

; Variables para pedir la referencia de archivos
; sobre la que se trabajará, así como el
; directorio al que se desea cambiar

REFER     db 22,25,AZUL,12
        times 12 db ' '
        db 0

```

```
DIREC      db 22,25,AZUL,50
           times 50 db ' '
           db 0

; Los dos campos siguientes servirán para borrar
; la ventana de mensajes

BORR1      db 21,24,BLANCO
           times 54 db ' '
           db 0

BORR2      db 22,24,BLANCO
           times 54 db ' '
           db 0

; Este campo se usará para obtener un
; directorio de los archivos a tratar,
; en principio todos

TODOS      db '*.*', 0

; A medida que vayan leyéndose archivos
; se irán imprimiendo puntos

PUNTO      db '.5'

; Los 128 bytes siguientes se utilizarán
; como área de transferencia de datos

DTA        resb 128

; En este campo se guardará la dirección del
; nombre del último archivo del directorio

FINAL      dw 0

CUENTA    dw 0 ; Número de archivos en el directorio
LINEA     db 0 ; Línea actual en pantalla
PUNT1     dw 0 ; Archivo seleccionado en pantalla
PUNT2     dw 0 ; Primer archivo que aparece en pantalla

; Variable para ir imprimiendo en la ventana
; de archivos los nombres de éstos. Aunque en
; la línea se indica 0 se modificará desde el
; código

LiN_VEN   db 0,6,AZUL
           resb 12
           db 0

; Zona para almacenar los nombres de los
; archivos existentes en el directorio

DIRECTO   resb 12*1024
```

```
; Variable temporal para intercambios
; al ordenar

TEMP      resb 12

; Direcciones de los elementos que
; se están comparando

ELEM1      dw 0
ELEM2      dw 0

; Variables para almacenar los manejadores
; de entrada y salida y buffer de almacenamiento
; temporal para la opción de copia de archivos

HANDLE_IN    dw 0
HANDLEJDUT   dw 0
BUFF_COPIA   resb 1024

; Línea de cabecera de la pantalla.
; En la variable ACTUAL se almacenará el camino
; del directorio actual, y en CADENA el número
; de archivos seleccionados

L1N1      db 0,0,REVERSE + AZUL
          db ' UDISK Versión 1.0
ACTUAL    db '
CADENA    db '
          db ' Archivos      ', 0

; Líneas que componen la pantalla

PAN      db 2, 3,BLANCO,201
          times 16 db 205
          db 187,0

          db 3, 3,BLANCO,186
          times 16 db ' '
          db 186,' F1 - Ordena directorio por nombre',0

          db 4, 3,BLANCO,186
times 16 db ' '
          db 186,0

          db 5, 3,BLANCO,186
          times 16 db ' '
          db 186
          db ' F2 - Ordena directorio por extensión',0

          db 6, 3,BLANCO,186
          times 16 db ' '
          db 186,0

          db 7, 3,BLANCO,186
          times 16 db ' '
          db 186,' F3 - Borra archivo(s)',0
```

```
db  8, 3,BLANCO,186
times 16 db ' '
db 186,0

db  9, 3,BLANCO,186
times 16 db ' '
db 186,' F4 - Renombra archivo',0

db 10, 3,BLANCO,186
times 16 db ' '
db 186,0

db 11, 3,BLANCO,186
times 16 db ' '
db 186,' F5 - Copia archivo',0

db 12, 3,BLANCO,186
times 16 db ' '
db 186,0

db 13, 3,BLANCO,186
times 16 db ' '
db 186,' F6 - Cambia de disco',0

db 14, 3,BLANCO,186
times 16 db ' '
db 186,0

db 15, 3,BLANCO,186
times 16 db ' '
db 186,' F7 - Cambia de directorio',0

db 16, 3,BLANCO,186
times 16 db ' '
db 186,0

db 17, 3,BLANCO,106
times 16 db ' '
db 186
db ' F8 - Establecer referencia de búsqueda',0

db 18, 3,BLANCO,186
times 16 db ' '
db 186,0

db 19, 3,BLANCO,186
times 16 db ' '
db 186,' ESC - Salir al DOS',0

db 20, 3,BLANCO,186
times 16 db ' '
db 186,201
times 54 db 205
db 187,0

db 21, 3,BLANCO,186
times 16 db ' '
```

```

db 186,' ',186
times 54 db ' '
db 186,0

db 22, 3,BLANCO,186
times 16 db ' '
db 186,' ',186
times 54 db ' '
db 186,0

db 23, 3,BLANCO,200
times 16 db 205
db 188,' ',200
times 54 db 205
db 188,0

db 255

; Buffer para obtener el camino
,- del directorio actual

CAMINO resb 128

; Tabla de opciones. Por cada opción
; se almacena un byte con el código
; extendido de la tecla de elección y
; una palabra con la dirección de la
; rutina a ejecutar por dicha tecla

OPCIONES db 72
        dw SUBIR
        db 80
        dw ABAJO
        db 59
        dw OBDF.N1
        db 60
        dw ORDEN2
        db 61
        dw BORR
        db 62
        dw RENOMBRAR
        db 63
        dw COPIAR
        db 64
        dw DISCO
        db 65
        dw DIRECTORIO
        db 66
        dw REFERENCIA

```

El programa comienza configurando los registros habituales y mostrando un mensaje, tras lo cual invoca a una rutina para recuperar todas las entradas del directorio actual y se muestra la máscara de pantalla del programa.

Acto seguido se entra en el bucle principal del programa, que se encarga de detectar las pulsaciones de tecla, invocar a la rutina que corresponda y actualizar la lista de archivos visible en pantalla. El programa finaliza cuando se detecta la pulsación de la tecla Esc.

```

; Segmento de código
segment Código
..start:

INICIO:
    ; Prepararnos los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; y los del segmento de datos
    mov ax, Datos
    mov ds, ax
    mov es, ax

    ; Imprimir el mensaje de copyright
    mov dx,COPYRIGHT
    mov ah,9
    int 21h

    ; Leer los archivos del directorio
    cali LEEDIR

    ; Borrar la pantalla
    cali BORRA

    ; Imprime la máscara del programa
    cali PANTA

    ; En principio la linea 4 será
    ; la resaltada
    mov byte [LINEA], 4

    ; el archivo elegido será el primero
    mov byte [PUNT1], 0

    ; el primer archivo a mostrar
    ; será el mismo
    mov byte [PUNT2], 0

    ; Hacemos desaparecer el cursor
    cali SIN CURSOR

```

BJC CENTRAL

```
; Imprime la lista de archivos  
; en la ventana  
cali LISTA ARCH
```

ESPERA_TECLA:

```
; espera la pulsación de una tecla
xor ah, ah
int 16h

; ¿Es ESC?
cmp al, 27
; de ser así terminar
je FINUDISK

; apuntar a la tabla de opciones
mov si,OPCIONES
; hay 10 opciones disponibles
mov ex,10
```

BUC COMPARA:

```
; comparar la tecla pulsada con
; la de la tabla
cmp [si],ah
; si son iguales saltar
je SALTA_RUTTNA

; no coincide, saltar a la
; opción siguiente
add si,3
; y seguir comparando
loop BUC_COMPARA

; si no coincide ninguna esperar
; otra pulsación
jmp ESPERA_TECLA
```

SALTA_RUTINA:

```
; la tecla coincide
; incrementar SI para que apunte
; a la dirección
inc si
; y llamar a la rutina correspondiente
cali [si]

; tras ejecutar la rutina,
; volver al bucle principal
jmp BOC_CENTRAL
```

; Aquí se llega cuando se pulsa
; la tecla ESC

FIN_UDISK:

```
cali BORRA ; borramos la pantalla
cali CON_CURSOR ; mostramos el cursor
mov ah, 4Ch ; y salimos al DOS
int 21h
```

A partir de aquí, el resto del código está compuesto de rutinas que son invocadas desde el bloque principal o bien desde otras rutinas. Cada una de estas rutinas está precedida de un bloque de comentarios explicando su finalidad y, tal como se indicaba anteriormente, también entre las sentencias se va explicando el proceso paso a paso. Éste es el resto del código:

```
; Aquí se llega cuando se pulsa
; el cursor hacia arriba

SUBIR:
    ; Loma el número de archivo
    ; actual en pantalla
    mov ax, [PUNT1]
    ; ¿es cero?
    or ax, ax
    ; de ser así saltar
    jz FIN_SUBIR

    ; si no 0 pasar al anterior
    dec ax
    ; y guardar el índice
    mov [PUNT1], ax

    ; ¿estamos en la línea 4?
    cmp byte [LINEA], 4
    ; de ser así salta a DESP_ABAJO
    je DESP_ABAJO

    ; en caso contrario
    ; reducimos la línea actual
    dec byte [LINEA]
    ; y volver
    ret

DESPEABAJO:
    ; SJ llegamos aquí es que hay que
    ; subir una línea pero estamos en
    ; la primera.
    ; Se reduce el puntero del primer
    ; archivo a mostrar, con lo que al
    ; volver a imprimir la ventana se
    ; obtendrá un efecto de desplazamiento
    ; sobre la lista de archivos

    ; reducimos el puntero del primer
    ; archivo a mostrar
    dec word [PUNT2]

    ; y borramos el contenido de
    ; la ventana
    cali LIMPIA_VEN

FIN_SUBIR:
    ret ; volver
```

```

; Aquí se llega cuando se pulsa
; el cursor hacia abajo

ABAJO:
; Tomar el número del archivo
; actual en pantalla
mov ax, [PUNT1]
; ¿es el último de la lista?
cmp ax, [CUENTA]
; de ser así salta
je FTN_ABAJO

; incrementamos para pasar
; al archivo siguiente
inc ax
; y lo guardamos
mov [PUNT1], ax

; ¿estamos en la última línea de la ventana?
cmp byte [LINEA], 21

; en caso afirmativo saltar
je DESP-SUBIR

; en caso contrario incrementar
; la línea
inc byte [LINEA]

ret ; y volver

DESPSUBIR:
; si se ha de pasar al archivo siguiente
; pero estamos en la última línea de pantalla,
; se incrementa el puntero del primer archivo
; visible, con lo que la lista dará la
; impresión de desplazarse hacia arriba una
; línea, aunque la linea actual no se mueva

; borramos la ventana de archivos
cali LIMPIA_VF,N

; e incrementamos el puntero
inc word [PUNT2]

FIN_ABAJO:
ret ; volver

; Aquí se llega cuando se pulsa
; la tecla F1, cuya finalidad es
; ordenar la lista de nombres de
; archivos alfabéticamente por
; nombre

ORDEN!:

```

```

; imprimir el mensaje de espera
mov si, MEN3
cali IMPRIME

; mover a CX el número de
; elementos
mov ex, [CUENTA]

BUC_ORDEN0:
push ex ; guardamos CX
; leer en CX el número
; de elementos a ordenar
mov ex, [CUENTA]

; SI apunta al primer archivo
mov si, DIRECTO
; y Di al segundo
mov di, DIRECTO+12

BUC_ORDEN1:
cid ; autoincremento de DI y 51

push ex ; guardar CX
; guardar la dirección
; de los elementos a comparar
mov [ELEM1], si
mov [ELEM2], di

; comparar 8 caracteres
mov ex, 8

; repite la comparación
,- mientras coincida
repe empsb

; si son iguales saltar
jbe BIEN1

; en caso contrario intercambia
; los elementos
cali INTERCAM

BIEN1:
; recuperar la dirección de
; los dos elementos
mov si, [ELEM1]
mov di, [ELEM2]

; y sumar 12 para apuntar
; a los dos siguientes
add si, 12
add di, 12

; recuperamos el contador del
; bucle interno
pop ex

```

```
; y repetimos la comparación
loop BUC_ORDEN1

; recuperar el contador del
; bucle externo
pop ex

; y repetir el ciclo de
; comparaciones
loop BUC_ORDENO

; borrar la ventana de archivos
cali LIMPIA_VEN
; y la de mensajes
cali LIMPIA_MEN

; volver al bucle principal
; del programa
ret

; Aquí se llega cuando se pulsa
; la tecla F2, cuya finalidad es
; ordenar la lista de nombres de
; archivos alfabéticamente por
; extensión del archivo

ORDEN2:
    ; imprimir mensaje de espera
    mov si, MEN3
    cali IMPRIME

    ; CX contiene el número de
    ; ciclos de ordenación
    mov ex, [CUENTA]

BUC1ORDENO:
    ; guardamos CX
    push ex
    ; para asignarle el número
    ; de archivos a ordenar
    mov ex, [CUENTA]

    ; SI apunta al primer archivo
    mov si, DIRECTO
    ; y DI al segundo
    mov di, DIRECTO+12

RUC1 ORDEN1:
    cid ; autoincremento de SI y DI
    push ex ; guardar contador

    ; guardamos las direcciones
    ; de los elementos
    mov [ELEM1], si
    mov [ELEM2], di
```

```

; buscar el '.' de separación en
; el archivo apuntado por ES:DI
raoval, '.'
; en 9 caracteres máximo
raovex, 9

; repite mientras no lo encuentres
repne scasb

; guardar DI, que tiene la dirección
; del punto en el nombre de archivo
push di

; mover SI a DI
mov di, si
; buscar en 9 caracteres
mov ex, 9
; repite mientras no lo encuentres
repne scasb

; pasar la dirección a SI
mov si, di
; y recuperar la de DI
pop di

; Llegados a este punto, SI y DI apuntan a
; la extensión de los nombres de archivo

; comparar tres caracteres
mov ex, 3
; repite mientras sean iguales
repe empsb

; si son iguales salta
jbe BIEN2

; en caso contrario intercambiamos
cali INTERCAM

BIEN2:
; recuperar las direcciones de
; los elementos
mov si, [ELEM1]
mov di, [ELEM2]

; y actualizarlos para apuntar
; a los siguientes elementos
add si, 12
add di, 12

; recuperar el contador de
; archivos a comparar
pop ex

; y seguir comparando
loop BUC1_Orden1

```

```

; recuperar el contador de
; ciclos de ordenación
pop ex

; y repetir
loop BUC1_ORDENO

; limpiar la ventana de archivos
cali LIMPIAJ/EN

; y la de mensajes
cali LIMPIA_MEN

ret ; volver

; Esta rutina es usada por las dos
; anteriores para intercambiar los
; elementos apuntados por ELEM1 y ELEM2

```

INTERCAM:

```

; Llevar el contenido del segundo
; elemento al espacio temporal
mov si, [ELEM2]
mov di, TEMP
mov ex, 12
rep movsb

; copiar el contenido del primer
; - elemento al segundo
mov di, [ELEM2]
mov si, [ELEM1]
mov ex, 12
rep movsb

; por último llevar el área temporal
; al primer elemento
mov di, [ELEM1]
mov si, TEMP
mov ex, 12
rep movsb

ret ; volver

```

```

; A esta rutina se llega cuando se pulsa F3
; Su finalidad es borrar el archivo elegido
; en ese momento o todos los seleccionados
; en la ventana

```

BORR:

```

; imprimir el mensaje que pregunta
; si se desea borrar el archivo
; actual o todos
mov si, MEN8
cali IMPRIME

```

```

; mostrar el cursor
cal] C0N_CURSOR

; colocarnos en la ventana
; de mensajes
mov ah, 2
mov dh, 22
mov di, 25
int 10h

B0RR_1:
;- esperamos la pulsación
; de una tecla
xor ah, ah
int 16h

; convertimos a mayúscula
and al, 11011111b

; ¿es la 'T'?
cmp al, 'T'

; si es así saltar
je B0RJTODOS

; ¿es la 'U'?
cmp al, 'U'

; si no, repetir la solicitud
jne B0RR_1

UNO:
; borrar el archivo elegido
; en este momento
cali B0RFIC

FIN_B0R:
; ocultar el cursor
cali SIN_CURSOR

; limpiar la ventana de mensajes
cali LIMPIAJYIEN

; poner a 0 el número
; de archivos
mov word [CUENTA], 0

; borrar la pantalla
cali B0RRA

; y volver a leer el directorio
jmp INICIO

; Esta rutina tiene el objetivo de borrar
; el archivo indicado por PUNT1
B0R FIC:

```

```

; limpiar el espacio donde va
; a tomarse el nombre
cali LIMPIA_BUF

mov ax, [PUNT1] ; archivo número PUNT1
mov bx, 12 ; 12 caracteres cada nombre
muí bx ; multiplicar

; SI apunta al inicio de la tabla
mov si, DIRECTO
; sumar para acceder al elemento deseado
add si, ax

; copiar el nombre de la tabla al
; espacio de trabajo
cali PASA_BUF

; que está en LIN_VEN+3
mov dx, LIN_VEN+3
mov ah, 41h ; función de borrado
int 21h

rét ; volver

; Esta rutina borra todos los archivos
; seleccionados en la ventana
BORJTODOS:
; tomamos el número de archivos
mov ex, [CUENTA]
; lo incrementamos porque contamos
; desde 0
inc ex
; PUNT1 apuntará al primer archivo
mov word [PUNT1], 0

BUCLE_BOR:
; guardamos CX
push ex

; llamar a la rutina que borra
; el archivo actual
cali BOR_FIC

; incrementar el puntero
inc word [PUNT1]

; recuperar el contador
pop ex

; y seguir
loop BUCLE_BOR

; Establecer la referencia para
; seleccionar de nuevo todos
; los archivos
mov byte [TODOS], •••

```

```
mov byte [TODOS+1], '.'
mov byte [TODOS+2], '*'
mov byte [TODOS+3], 0

; saltar a FIN_BOR
jmp FIN_BOR

; A esta rutina se llega cuando se pulsa F4
; Su finalidad es renombrar el archivo que
; esté elegido en ese momento

RENOMBRAR:
; imprimir el mensaje de petición
; del nuevo nombre para el archivo
mov si, MEN9
cali IMPRIME

; mostrar el cursor
cali CON_CURSOR

; borrar el buffer donde va a
; pedirse el nuevo nombre
mov di, REI'ER+4
mov ex, 12
mov al, ' '
cid
rep stosb

; dato a pedir
mov si, REFER
cali PETICIÓN ; rutina de petición

; Limpiar el buffer a donde pasar
; el nombre del archivo a renombrar
cali LIMPIA_BUF

; PUNT1 contiene el número de
; archivo elegido
mov ax, [PUNT1]
; 12 caracteres por archivos
mov bx, i2
; multiplicar para obtener el
; desplazamiento
muí bx

; SI apunta al elemento que tiene
; el nombre del archivo a renombrar
mov si, DIRECTO
add si, ax

; pasar al buffer, que es LIN_VEN
cali PASA_BUF

; DX apunta al nombre actual
mov dx, LIN_VEN+3
```

```

; y DI al nuevo
mov di, REFER+4

; función de renombrado
mov ah, 56h
int 21h

; borramos la pantalla
cali BORRA

; ponemos a 0 el número de
; archivos leídos
mov word [CUENTA], 0

; ocultar el cursor
cali SIN_CURSOR

; volver a leer el directorio
jmp INICIO

; A esta rutina se llega cuando se pulsa F5
; Su finalidad es copiar el archivo elegido
; en ese momento

COPIAR:
; mostrar el mensaje de petición
; del camino de destino
mov si, MENO
cali IMPRIME

; mostrar el cursor
cali CON_CURSOR

; limpiar el buffer donde va a
; pedirse el camino
mov di, REFER+4
mov al, ' '
mov ex, 12
cid
rep stosb

; pedir el camino de destino
mov si, REFER
cali PETICIÓN

; limpiar el buffer donde va a
; almacenarse el nombre del
; archivo a copiar
cali LIMPIA_BUF

; calcular la dirección del nombre
; del archivo dentro de la tabla
mov ax, [PUNT1]
mov bx, 12

```

```
    muí bx
    mov si, DIRECTO
    add si, ax

    ; copiar el nombre al buffer
    cali PASA_RUF

    ; abrir el archivo para lectura
    mov dx, LIN_VEN+3
    mov ah, 3Dh
    xor al, al
    int 21h

    ; guardamos el manejador
    mov [HANDLE_IN], ax

    ; abrir el archivo de destino
    ; creándolo
    mov dx, REFER+4
    mov ah, 3Ch
    xor ex, ex
    int 21h

    ; guardar el manejador devuelto
    mov [HANDLE_OUT], ax

BUCLE_COPIA:
    ; leemos 1 kilobyte del
    ; archivo de origen
    mov bx, [HANDLE_IN]
    mov ex, 1024

    ; deiándolo en BUFF_COPIA
    mov dx, BUFF^COPIA
    mov ah, 3Fh
    int 21h

    ; saltar si hay error
    je FIN_COPIA

    ; saltar si se llegó
    ; al final del archivo
    or ax, ax
    jz FIN_COPIA

    ; escribimos la información leída
    ; en el archivo de destino
    mov bx, [HANDLEJDUT]
    mov ex, ax
    mov dx, BUFF_COPIA
    mov ah, 40h
    int 21h

    ; repetir el proceso
    jmp BUCLE_COPIA
```

```

FINJ30PIA:
    ; ocultar el cursor
    cali SIN_CURSOR

    ; limpiamos la ventana de mensajes
    cali LIMPIA_MEN

    ; ponemos a 0 el contador de
    ;- archivos
    rmov word [CUENTA], 0

    ; borramos la pantalla
    cali BORRA

    ; y volvemos a leer el directorio
    jmp INICTO

; A esta rutina se llega cuando se pulsa F6
; Su finalidad es cambiar la unidad de disco
; por defecto

DISCO:
    ; imprimir el mensaje de petición
    ; de nueva letra de unidad
    mov si, MEN7
    cali IMPRIME

    ; mostrar el cursor
    cali CON_CURSOR

    ; y ponerlo en la ventana
    ; de mensajes
    mov dh, 22
    mov di, 25
    mov ah, 2
    int 10h

    ; esperar la pulsación de una tecla
    xor ah, ah
    int 16h

    ; convertir a mayúscula
    and al, 11011111b

    ; le restamos 'A' con lo que se
    ; obtiene 0 si era A, 1 si era B, etc.
    sub al, 'A'

    ; pasamos la unidad a DL
    mov di, al

    ; función de cambio de unidad
    mov ah, OEH
    int 21h

```

```
; ocultamos el cursor
cali SIN_CURSOR

; limpiamos la ventana de mensajes
cali LIMPIA_MEN

; borrar la pantalla
cali BORRA

; poner a 0 el contador de archivos
mov word [CUENTA], 0

; leer el directorio de la
; nueva unidad
jmp TNTCTO

; A esta rutina se llega cuando se pulsa F7
; Su finalidad es permitir el cambio del
; directorio actual

DIRECTORIO:
    ; imprimir el mensaje de petición
    ; del nuevo camino
    mov si, MF.N5
    cali IMPRIME

    ; mostrar el cursor
    cali SIN_CURSOR

    ; limpiar el buffer de petición
    mov di, DIREC+4
    mov al, ' '
    mov ex, 50
    cid
    rep stosb

    ; pedir el nuevo camino
    mov si, DIREC
    cali PETICIÓN

    ; cambiar al camino
    ; indicado
    mov dx, DIREC+4
    mov ah, 3Bh
    int 21h

    ; ponemos a 0 el contador
    mov word [CUENTA], 0

    ; borrar la pantalla
    cali BORRA

    ; y volver a leer el directorio
    jmp INICIO
```

; A esta rutina se llega cuando se pulsa F8
; Su finalidad es permitir el cambio de la
; referencia usada para seleccionar los
; archivos y, por ejemplo, poder borrar
; un grupo de archivos

REFERENCIA:

```

; imprimir el mensaje de petición
; de la nueva referencia
mov si, MEN4
cali IMPRTME

; limpiar el buffer donde va
; a pedirse la referencia
mov di, REFER+4
mov ex, 12
mov al, ' '
Cld
rep stosb

,- mostrar el cursor
cali CON_CURSÜR

; pedir la nueva referencia
mov si, REFER
cali PETICIÓN

,- mover la nueva referencia al espacio
; qué usa la rutina de visualización
; de archivos
mov si, REFER+4
mov di, TODOS
mov ex, 12
rep movsb

; borramos la pantalla
cali BORRA

; ponemos a 0 el contador de archivos
mov word [CUENTA], 0

; volver a leer el directorio
; usando la nueva referencia
jmp INICIO

; Su finalidad es permitir el cambio de la
; referencia usada para seleccionar los
; archivos y, por ejemplo, poder borrar
; un grupo de archivos

```

LISTA_ARCH:

```

; SI apunta al inicio de la tabla
mov si, DIRECTO

```

```
; calcular la dirección del
; primer elemento a mostrar,
; que viene indicado por PUNT2
mov ax, [PUNT2]
mov bx, 12
muí bx
add si, ax

; imprimir 18 archivos en pantalla
mov ex, 18

; establecer el color AZUL
mov byte [LIN_VEN+2], AZUL

; la primera linea de la
; ventana es la 4
mov dh, 4

BUC_LISTA:
; guardamos el contador de archivos
push ex

; mover DH a la linea de la variable
; de impresión
mov [LIN_VEN], dh

; incrementar la linea
inc dh
; y guardarla
I-uyh üx

; Limpiamos LIN_VEN
cali LIMPIA_BUF
; y pasa el fichero de la tabla
; apuntado por SI
cali PASA_BUF

; guardar la dirección del archivo
; siguiente
push si
; e imprimirla
mov si, LIN_VEN
cali IMPRIME

pop si ; recuperar la dirección
pop dx ; la linea del cursor
pop ex ; y el contador

; ¿Hemos llegado al final de la tabla?
emp si, [FINAL]
; si es así saltar
jge FIN_BUFFER

; en caso contrario seguir
loop BUC_LISTA
```

```

FIN^BUFFER:
; Una vez impresos los nombres en la
; ventana, se muestra en video inverso el
; nombre del archivo seleccionado
; actualmente

; establecer el color
mov byte [LIN_VEN+2], REVERSE + AZUL
; tomar la linea actual
mov al, [LINEA]
; y ponerla en LIN_VEN
mov [T,TN_VEN], al

; calcular la dirección del archivo
; elegido actualmente
mov si, DIRECTO
mov ax, [PUNTI1]
mov bx, 12
muí bx
add si, ax

; lo pasamos al buffer para
; imprimí rio
cali LIMPIAJBUF
cali PASA_BUF

; mostrar el nombre
mov si, LIN_VEN
cali IMPRIME

ret ; volver

; Esta rutina limpia la wntana de archivos,
; para que al desplazar su contenido no queden
; en las lineas trozos de los nombres anteriores

LIMPIA_VEN:
xor al, al ; 0 lineas
mov bh, BLANCO ; y color blanco
mov ch, 4 ; marca las esquinas
mov el, 6 ; de la ventana de archivos
mov dh, 21
mov di, lfi
mov ah, 6 ; scroll arriba
int 10h ,*borramos

ret ; volver

; Esta rutina borra el contenido de la ventana
; de mensajes

```

```

LIMPIA_MEN:
; borramos la primera linea
mov si, BORR1
cali IMPRIME

```

```

; y la segunda
mov si, BORR2
cali IMPRIME

ret ; volver

; Para muchas operaciones se usa LIN_VEN. La
; finalidad de esta rutina es limpiar el
; buffer de 12 caracteres que tiene dicha
; variable

LIMPIA_BUF:
    ; apuntar al buffer
    mov di, LIN_VEN+3
    mov ex, 12 ; 12 caracteres
    mov al, ' ' ; rellenar de espacios
    rep stosb

    ret ; volver

; Esta rutina pasa los 12 caracteres apuntados
; por SI al buffer de LIN_VEN

PASA_BUF:
    ; DI apunta al destino
    mov di, LIN_VEN+3
    mov ex, 12 ; mover 12 caracteres
    rep movsb

    ret ; volver

; Esta rutina es invocada al
; principio del programa para
; leer el directorio y poner
; los nombres de los archivos
;- en la matriz apuntada por
; la variable DIRECTO

LEEDIR:
    ; Imprimir el mensaje de que
    ; se está leyendo el directorio
    mov dx,MEN1
    mov ah,9
    int 21h

    ; Establecer la dirección de
    ; transferencia de datos
    mov dx,DTA
    mov ah,1Ah
    int 21h

    ; Apuntar a la referencia que
    ; se busca
    mov dx,TODOS

```

```

mov ah,4Eh ; buscar primera coincidencia
xor ex,ex ; archivos normales
int 21h

; si hay archivos que cumplan la
; referencia saltar a SALTO!
jnc SALT01

; El directorio está vacío

; imprimir el mensaje de error
mov dx,MEN2
mov ah,9
int 21h

; y devolver el control al DOS
mov ah,4Ch
int 21h

SALT01:

; DI apunta a la tabla para
; almacenar los nombres
mov di,DIRECTO

BUCLE_LEE :

; SI apunta al nombre dentro de la DTA
mov si,DTA+30
mov ex,12 ; 12 caracteres
rep movsb ; moverlos a la tabla de nombres

; incrementar el contador de archivos leídos
inc byte [CUENTA]

; buscar siguiente coincidencia
mnv ah,4Fh
int 21h

; si no hay más saltar a FIN
je FIN

; en caso contrario imprimir un punto
mov dx,PUNTO
mov ah,9
int 21h

; y repetir el proceso
jmp BUCLE_LEE

FIN:

; reducir CUENTA en una unidad porque
; el primer archivo en la tabla es el 0
dec byte [CUENTA]

```

```

; mover a FINAL la última dirección
; de la tabla
mov [FINAL],di

ret ; volver

; Esta rutina imprime la pantalla definida
; en el segmento de datos, con la información
; del directorio actual y el número de
; archivos leídos

PANTA:
xor di, di ; unidad actual
; apuntar al buffer donde se
; dejará el camino
mov si, CAMINO
; obtenemos el camino actual
mov ah, 4 7h
int 21h

; limpiar el buffer donde va a
; almacenarse el camino
mov di, ACTUAL
mov ex, 30
mov al, ' '
rep stosb

; DI apunta al inicio del buffer
mov di, ACTUAL
; copiar como máximo 30 caracteres
mov ex, 30

BUC_PANTA:
; ¿Es 0 el byte a copiar?
emp byte [si], 0
; en caso afirmativo hemos terminado
je SALTO_PANTAL
; en caso contrario pasar el
; byte de DS:SI a ES:DI
movsb

loop BUC_PANTA ; repetir

SALTO_PANTAL:
; tomar en AX el número de archivos leídos
mov ax, [CUENTA]
; incrementar para contar el 0
inc ax

; DI apunta al buffer donde debe
; dejarse la conversión
mov di, CADENA+4

cali EnteroCadena ; convertir

```

```

; Apuntar a la línea de cabecera
mov si, LIN1
cali IMPRIME ; e imprimir

,- apunta a las líneas que
; componen la pantalla
mov si, PAN

BUC_PANTAI:
    cali TMPRTME ; imprimir la línea

    inc si ; apuntar al byte siguiente
    , si no es 2 55
    cmp byte [si], 255
    ; continuar imprimiendo
    jnz BUC_PANTAI

    ret ; volver

; Esta rutina oculta el cursor asignándole un
; tamaño que no puede tener

STN_C[JRSOR:
    mov ch, 15 ; línea de comienzo y fin 15
    mov el, 15
    mov ah, 1
    int 10h

    ret ; volver

; Esta rutina es complementaria de la anterior,
; volviendo a mostrar el cursor

CON_CURSOR:
    mov ch, 7 ; línea de comienzo y fin 7
    mov el, 7
    mov ah, 1
    int 10h

    ret ; volver

```

Las rutinas que se encargan de ordenar los nombres de archivo, bien sea por nombre o por extensión, usan el conocido método de la burbuja, basado en la comparación de todos los elementos entre sí. Es un método lento pero uno de los más sencillos de implementar. Si los prueba, comprobará que la ordenación realmente es muy rápida al estar codificada en ensamblador.

En la figura 24.2 puede observar el programa en funcionamiento, con un nombre de archivo seleccionado en la lista de la izquierda y tras haber pulsado la opción de borrado. Puede mejorar el programa de diversas maneras, por ejemplo ofreciendo una lista de directorios, facilitando la ejecución de un programa al pulsar la tecla **Intro** teniéndolo seleccionado, etc.

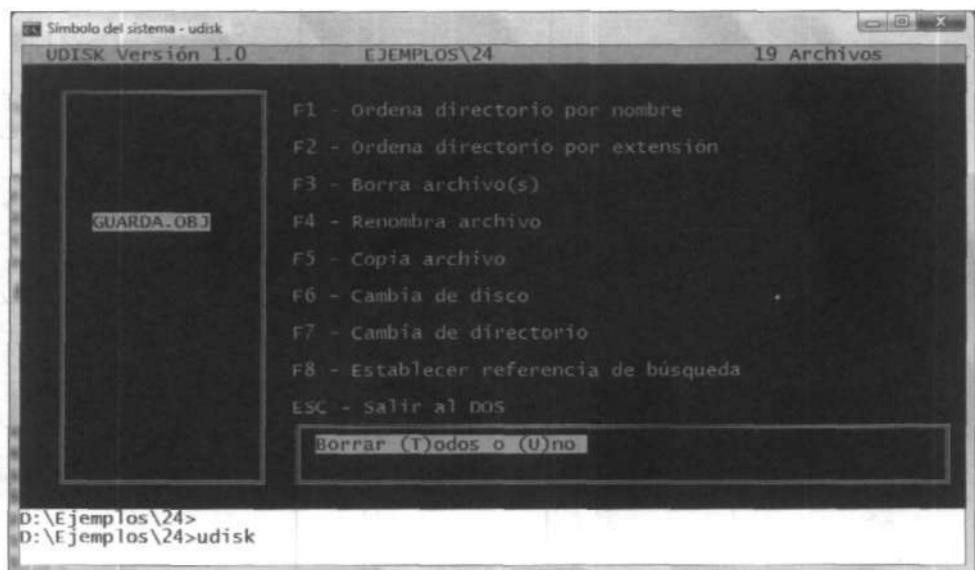


Figura 24.2. El programa UDisk en funcionamiento.

Resumen

Los servicios que nos ofrece el DOS para tratar con archivos en disco facilitan la mayoría de las operaciones, permitiéndonos crear y abrir archivos, leer y escribir información en ellos, eliminarlos y renombrarlos, obtener o modificar sus atributos, etc. También hemos conocido servicios para operar con las unidades de disco y los directorios, recuperando listas de los archivos contenidos en ellos.

En todos los casos se trata de servicios que podríamos considerar de alto nivel, ya que el sistema operativo es el que se encarga de decidir en qué sectores del disco escribir o de qué sectores leer, gestionando asimismo las entradas en la tabla de asignación de archivos.

En el capítulo siguiente tendrá oportunidad de conocer otros métodos de acceso a la información contenida en las unidades de disco, leyendo y escribiendo sectores individuales en lugar de operar con archivos.

25

Acceso a sectores de disco

Además de los servicios de tratamiento de archivos, descritos en el capítulo previo, el DOS también nos ofrece otros mediante los cuales podemos leer y escribir directamente sectores del disco. Son servicios similares a los ofrecidos por la BIOS, que también conocerá en este capítulo, si bien la forma de identificación de los sectores es diferente. Los servicios del DOS pueden además, en caso de error, mostrar mensajes por la consola, algo que no ocurre en ningún caso con los servicios de la BIOS.

El acceso directo a los sectores que forman un disco debe efectuarse siempre de manera cuidadosa, ya que cualquier error podría provocar la pérdida de todos los datos contenidos en él. Hay que tener en cuenta que con estos servicios es perfectamente posible alterar, por ejemplo, las tablas de asignación de archivos.

Tanto los servicios de la BIOS como los del DOS se han visto, con el tiempo, superados por las capacidades de las unidades de disco, cada vez de mayor capacidad. Los servicios originales son incapaces de acceder a tantos sectores porque el método original de direccionarlos es insuficiente. La solución, obvia por otra parte, ha sido la implementación de extensiones a esos servicios originales.

Servicios del DOS

Comenzaremos conociendo los servicios originales del DOS para lectura y escritura de sectores, los servicios conocidos como de *lectura absoluta* y *escritura absoluta*. A ellos se accede mediante las interrupciones 25h y 26h, respectivamente.

Los parámetros necesarios en ambos casos son exactamente los mismos:

- AL: Contendrá el número de la unidad, asociando el valor 0 con la unidad A; el 1 con la B; y así sucesivamente.
- DX: Indicará el número del primer sector a leer.
- CX: Indicará el número de sectores a leer.
- DS : BX: Contendrán la dirección del área de memoria donde se dejarán los datos leídos o de la que se tomarán los datos a escribir.

Las dos interrupciones identifican los sectores del disco mediante una numeración lógica, no física. Se numeran todos los sectores que hay en el disco comenzando por el de la cara 0 y pista 0.

Una vez ejecutadas, estas interrupciones dejan en la pila el contenido del registro de indicadores, que ellas mismas guardan al ser invocadas, siendo el programador el responsable de extraerlo de la pila para evitar problemas. Antes, no obstante, habría que comprobar el indicador de acarreo para saber si se ha producido algún error, en cuyo caso AL contendría uno de los valores explicados en la tabla 25.1.

Tabla 25.1. Errores notificados por las interrupciones 25h y 26h.

Valor de AL	Significado
0	Error de protección.
2	Unidad no preparada.
7	Formato de disco no reconocido.
8	Sector no encontrado.
10	Error de escritura.
11	Error de lectura.

La cantidad de sectores leídos o escritos no puede superar, en total, los 64 kilobytes de información. Teniendo en cuenta que cada sector está compuesto de 512 bytes, una simple división nos indica que el número máximo de sectores que podemos leer o escribir es de 128.

Esto no parecía una limitación seria cuando se creó el DOS 1.0, versión a la que pertenece este servicio, ya que por entonces los discos duros de mayor tamaño tenían entre 10 y 20 megabytes. De hecho, estos servicios están limitados a unidades de disco de menos de 32 megabytes lo que hace que, actualmente, en la práctica sólo se puedan utilizar sobre disquetes.

Esta limitación resulta lógica al utilizarse un registro de 16 bits, DX, para indicar el primer sector a leer o escribir. Puesto que ese registro no puede contener ningún valor superior a 65535, obviamente no podemos leer ni escribir sectores que estén más allá, y 65535 sectores por 512 bytes dan como resultado los 32 megabytes.

Unidades de más de 32 Mb

La solución lógica a las limitaciones de las interrupciones 2 5h y 2 6h pasa por incrementar el número de bits usados para indicar el sector de inicio de la operación. A partir de la versión 3.31 del DOS encontramos una solución en las propias interrupciones 2 5h y 2 6h, ya que estos servicios del DOS se extendieron para poder operar sobre unidades de hasta 2 gigabytes. En la práctica esto significa que es posible trabajar con cualquier unidad de disco o partición en la que esté instalado exclusivamente el DOS, ya que con él no es posible tratar particiones de más de 2 gigabytes.

Nota

Si en un sistema está instalada cualquier versión de Windows actual, las particiones se formatean con una estructura distinta, las más usuales son NTFS y FAT32, que hacen posible trabajar con unidades de más de 2 gigabytes.

Para usar estos servicios extendidos asignaremos a CX el valor 0 FFFFh, a AL el número de unidad y a DS : BX la dirección de una estructura compuesta de estos elementos:

- Una doble palabra indicando el número del primer sector a leer o escribir.
- Una palabra que establece el número de sectores a leer o escribir.
- Una doble palabra con la dirección del área de memoria que se usará para transferir la información.

Como puede ver, el número de sector de inicio ahora se almacena en una doble palabra, lo cual permite acceder a discos de tamaños mucho mayores, concretamente, como se ha dicho, de hasta 2 gigabytes.

Unidades de más de 2 Cb

Si trabajamos en DOS pero tenemos una unidad de más de 2 gigabytes es porque, además, contamos con Windows 95 o una versión posterior de dicho sistema, capaces de formatear las unidades con la estructura FAT32 en lugar de la clásica FAT (*File Allocation Table*). Para leer y escribir sectores en unidades de este tipo, consecuentemente, será preciso usar los servicios que ofrece el propio Windows a través de la interrupción 2 1h.

Las versiones de Windows previas a la 95, como Windows 3.11 y Windows 3.0, seguían utilizando la misma estructura de disco que DOS ya que, en realidad, eran interfaces gráficas que operaban sobre dicho sistema operativo.

El servicio que nos interesa concretamente es el 7305h, que asignaremos al registro AX. Puesto que AL no puede contener el número de unidad, éste pasará al registro DL. El registro CX contendrá el valor OFFFFh, como en el caso de las interrupciones 25h y 26h extendidas, ocurriendo lo mismo con DS : BX, que apuntarán a la estructura antes descrita. Tenga en cuenta que el sector de inicio, al almacenarse como valor de 32 bits, permite trabajar con unidades de hasta 2 terabytes que, hoy por hoy, aún no son corrientes.

Puesto que este servicio se utiliza tanto para leer como para escribir sectores, debe indicarse de algún modo la operación a ejecutar. Con este fin se asignará al registro SI el valor 0, para leer, o 1, para escribir. En este último caso, además, los bits 13 y 14 del mismo registro especificarán el tipo de escritura. Las posibles combinaciones de estos bits son las descritas en la tabla 25.2.

Tabla 25.2. Valores de los bits 13 y 14 de SI al escribir con el servicio 7305h.

Valor	Indica
00	Que se desconoce la naturaleza de los datos a escribir.
01	Que los datos forman parte de la tabla de asignación de archivos (FAT).
10	Que los datos forman parte del directorio.
11	Que los datos pertenecen a un archivo.

Como en el caso de las interrupciones 25h y 26h, el indicador de acarreo nos comunicará si la operación se ha ejecutado o no satisfactoriamente. En caso de fallo, el registro AX contendrá un código de error.

Servicios de la BIOS

La BTOS dispone de una interrupción, la 13h, que da acceso a todos los servicios de bajo nivel relacionados con unidades de disquetes y discos duros. Mediante estos servicios podemos obtener información de las unidades, leer y escribir pistas y darles formato, entre otras operaciones. Vamos a centrarnos especialmente en las de lectura y escritura de sectores, que son las que nos interesan en este capítulo.

El servicio de lectura de sectores es el 02h, precisando los parámetros siguientes:

- **DL:** Contendrá el número de unidad, teniendo en cuenta que las de disquete se numeran como 0 y 1 y los discos duros como 80h, 81h, etc.
- **DH:** Número del cabezal de lectura. En el caso de los disquetes será siempre 0 ó 1.
- **CX:** Este registro se parte en dos secciones. El registro CH, conjuntamente con los bits 6 y 7 de CL, tomados como bits de mayor peso, indican el número de pista a leer, mientras que los seis bits de menor peso de CL especifican el número de sector.

- AL: Número de sectores a leer.
- ES : BX: Dirección del área de memoria donde se dejarán los datos leídos.

Puesto que se dedican 10 bits, 8 de CH más 2 de CL, para determinar el número de pista, tan sólo puede accederse a las primeras 1024 pistas, de la 0 a la 1023, de cualquier disco. Además, no pueden leerse más de 255 sectores, al indicarse el número de sectores en un registro de 8 bits.

En caso de que se produzca un error de lectura se activará el indicador de acarreo. Al operar sobre unidades de disquete, esto puede deberse a que el motor tarda un tiempo en ponerse en marcha y puede no haberlo hecho con la suficiente rapidez para ejecutar la lectura. Por eso la llamada a este servicio, o al de escritura, debería repetirse como mínimo tres veces antes de dar la operación por fallida.

El servicio de escritura de sectores es el 03h, precisando exactamente los mismos parámetros que el de lectura.

La diferencia es que en este caso, como es lógico, el área de memoria apuntada por ES : BX deberá contener los datos a escribir en los sectores.

El hecho de que se empleen los dos bits de mayor peso del registro CL como complemento a CH, para formar el número de pista, sólo se aplica al operar sobre discos duros, ya que los disquetes siempre tienen un número de pistas inferior a 255 y, por tanto, puede indicarse sólo con el registro CH.

Si encuentra repetidamente un error al intentar escribir o leer un cierto sector, puede servirse del servicio Olh de la misma interrupción 13h a fin de obtener información adicional.

No es necesario ningún parámetro de entrada, facilitándose en AH uno de los valores enumerados en la tabla 25.3. Así podremos saber qué causa el problema e intentar solucionarlo.

Tabla 25.3. Códigos de error devueltos en AH por el servicio 01h de la interrupción 13h.

Código	Significado
0	No hubo ningún problema.
3	El disco está protegido contra escritura.
4	No se encuentra el sector solicitado.
8	Error de DMA (<i>Direct Memory Access</i>).
16	Se han leído datos con errores de CRC.
32	Error del controlador de discos.

Código	Significado
64	No se encuentra la pista solicitada.
128	El disco no responde.

Copia de discos

Para finalizar este capítulo terminaremos escribiendo un ejemplo. Su finalidad será facilitar la copia de disquetes, para lo cual se utilizarán los servicios de la BIOS, es decir, la interrupción 13h. Este programa hará uso de las rutinas que están alojadas en los archivos *Borra.inc*, *Imprime.inc* y *Convert.inc* creados en el capítulo previo.

Asumiendo que la mayor parte de los equipos, desde hace bastante tiempo, tan sólo cuentan con una unidad lectora de disquetes y que, además, éstos siempre son de un formato fijo, el de 1,44 Mb, el programa solicitará el disco de origen y almacenará todas sus pistas temporalmente en un archivo. Éste se creará en el directorio raíz del primer disco duro.

A continuación solicitará el disco de destino y efectuará el proceso inverso, escribiendo en sus pistas todo el contenido del archivo previamente creado.

Puesto que el contenido del disco original permanece en un archivo, nada nos impide efectuar múltiples copias del mismo disco. Por eso el programa preguntará si se desean más copias y, en caso afirmativo, pedirá un nuevo disco de destino para repetir el proceso. Copiado un disquete, el último paso será preguntar si se desean copiar más discos, esperándose una respuesta que llevará al programa de nuevo al inicio, solicitando otra vez un disco original, o su fin, devolviendo el control al sistema.

A continuación tiene el código completo del programa, como siempre ampliamente comentado para que le sea más fácil comprender los pasos que van dándose.

```

segment Pila stack
    resw 512
FinPila:

segment Datos
; Nombre del archivo temporal
; para la copia
Archivo db "c:\temp.dat", 0

; Manejador del archivo
Manejador dw 0

; Reservamos espacio para
; leer una pista completa
Buffer resb 18*512

; Cara y pista que están leyéndose
Cara db 0
Pista db 0

```

```
; Mensajes que usará el programa

MEN1 db 1,1,7
      db 'Inserte el disquete original y '
      db 'pulse una tecla ...', 0

MEN2 db 3,1,7
      db 'Inserte el disquete destino y '
      db 'pulse una tecla ...', 0

MEN3 db 10,1,7
      db '¿Desea otra copia de este disco? ', 0

MEN4 db 10,1,7
      db '¿Desea copiar otro disco? ', 0

MEN5 db 10,1,7,'Leyendo ...', 0
MEN6 db 10,1,7,'Escribiendo ...', 0

; Cadena para borrar una determinada
; línea de pantalla

Vacia db 0,1,7
      times 50 db ' '
      db 0

; Cadena para ir imprimiendo por pantalla
; la pista y cara que se procesan

Info db 5,1,7
      db 'Cara :          Pista :          ', 0

*****  
; Segmento de código

segment Código
..start:
Inicio:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax
    mov es, ax

    ; Tomamos el nombre de archivo
    mov dx, Archivo
```

```

; sin atributos especiales
xor ex, ex

; creamos el archivo
mov ah, 3Ch
int 21h

; guardamos el manejador
mov [Manejador], ax

; borramos la pantalla
cali BORRA

; imprimir el mensaje de
; inserción del disco de origen
mov si, MEN1
cali IMPRIME

; esperamos la pulsación
; de una tecla
xor ah, ah
int 16h

; borrar la linea
mov byte [Vacia], ]
mov si, Vacia
cali IMPRIME

; imprimir el mensaje de
; que está leyéndose
mov si, MENS
cali IMPRIME

; En este bucle se irán leyendo las
; pistas que forman el disco, y se
; irán escribiendo en un archivo del
; disco duro

```

Bucle:

```

; Convertimos el número
; de cara en cadena
mov di, Info+9
mov al, [Cara]
inc al
xor ah, ah
cali EnteroCadena

; Hacemos lo mismo con
; el número de pista
mov di, Info+23
mov al, [Pista]
inc al
: xor ah, ah
cali EnteroCadena

```

```

; Imprimimos la linea que
; informa de la cara y
; pista que se leen
mov si, Info
cali IMPRIME

Try:
; Leer de la unidad 0 (A:)
xor di, di
; DH contiene la cara
mov dh, [Cara]
;- y CH la pista
mov ch, [Pista]
; leeremos desde el sector 1
mov el, 1
; los 18 sectores de la pista
mov al, 18
; indicamos el área donde
; dejar la información
mov bx, Buffer
; función de lectura
mov ah, 2

; intentamos leer
int 13h
; si se produce un error
; volver a intentarlo
je Try

; Ponemos en BX el manejador
; del archivo
mov bx, [Manejador]
; Escribir los bytes de
; 18 sectores
mov ex,18*512
; dirección donde están los datos
mov dx, Buffer
; función de escritura en archivo
mov ah, 40h
int 21h ; guardamos la pista

; pasamos a la siguiente pista
inc byte [Pista]
; si no es la 80
emp byte [Pista], 80
; continuar con la siguiente
jne Bucle

; Volvemos a la primera pista
mov byte [Pista], 0

; pasamos a la siguiente cara
inc byte [Cara]
; ¿se han leido las dos caras?
emp byte [Cara], 2

```

```
; de no ser así seguir leyendo
jne Bucle
```

```
; hemos terminado de leer
; el disquete
```

```
; Cerramos el archivo
mov bx, [Manejador]
mov ah, 3Eh
int 21h
```

Destino:

```
; mostramos el mensaje que solicita
; la inserción del disco de destino
mov si, MEN2
call IMPRIME
```

```
; esperamos la pulsación de una tecla
xor ah, ah
int 16h
```

```
; eliminamos lo último escrito en pantalla
mov byte [Vacia], 3
mov si, Vacia
call IMPRIME
```

```
; mostrar el mensaje de que
; está escribiéndose
mov si, MEN6
call IMPRIME
```

```
; abrimos el archivo donde se
; guardó la información
xor al, al
mov ah, 3Dh
mov dx, Archivo
int 21h
```

```
; comenzar a escribir desde la
; cara 0 y pista 0
mov byte [Cara], 0
mov byte [Pista], 0
```

Bucle1:

```
; Introducir en la cadena Info los
; valores de Cara y Pista debidamente
; convertidos a cadena
mov di, Info+9
mov al, [Cara]
inc al
xor ah, ah
call EnteroCadena
```

```
mov di, Info+23
mov al, [Pista]
```

```
inc al
xor ah, ah
cali EnteroCadena

; mostramos la línea informativa
mov si, Info
cali IMPRIME

; Leemos una pista desde el archivo
; en el que se guardó
mov bx, [Manejador]
mov ex, 18*512
; dejando la información en Buffer
mov dx, Buffer
mov ah, 3Fh
int 21h

Try1:
; Unidad 0 (A)
xor di, di
; En DH la cara
mov dh, [Cara]
; y en CH la pista
mov ch, [Pista]
; escribir a partir del sector 1
mov el, 1
; 18 sectores
mov al, 18
; tomando la información que
; acaba de leerse del archivo
mov bx, Buffer
mov ah, 3

; escribimos
int 13h
; reintentando si hay error
je Try1

; incrementamos la pista
inc byte [Pista]
; ¿hemos llegado a la última?
emp byte [Pista], 80
; de no ser así continuar
jne Bucle1

; volver a la primera pista
mov byte [Pista], 0

; pasar a la cara siguiente
inc byte TCaral
; ¿se han escrito las dos caras?
emp byte [Cara], 2
; de no ser así continuar
jne Bucle1

; Hemos terminado
```

```

; Cerramos el archivo
mov bx, [Manejador]
mov ah, 3Eh
int 21h

Respuesta:
; Preguntar si se desea otra
;- copia del mismo disco
mov si, MEN3
cali IMPRIME

; colocamos el cursor
mov ah, 2
xor bh, bh
mov di, 35
mov dh, 10
int 10h

; esperamos la respuesta
xor ah, ah
int 16h
; y la guardamos en la pila
push ax

; borramos el mensaje de pregunta
mov byte [Vacia], 10
mov si, Vacia
cali IMPRIME

; recuperamos la respuesta
pop ax
; y la convertimos a mayúscula
and al, 11011111b

; si no desea otra copia
cmp al, 'N'
; saltar a la etiqueta otra
je Otra

; si desea otra copia
cmp al, 'S'
; saltar a Destino
je Destino

; repetir la pregunta
jmp Respuesta

```

Otra:

```

; Preguntar si se desea
; copiar otro disco
mov si, MEN4
cali IMPRIME

; colocamos el cursor
mov ah, 2
xor bh, bh

```

```

mov di, 35
mov dh, 10
int 10h

; esperar la respuesta
xor ah, ah
int 16h

; la guardamos en la pila
push ax

; eliminamos la pregunta
; de la pantalla
mov si, Vacia
cali IMPRIME

; recuperamos
pop ax
; y convertimos a mayúscula
and al, 11011111b

; si no se desea copiar más
cmp al, 'N'
; terminar el programa
je Salir

; si quiere otra copia
cmp al, 's'
; saltar al principio
je Inicio

; repetir la pregunta
jmp Otra

Salir:
; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

%include "Borra.inc"
%include "Imprime.inc"
%include "Convert.inc"

```

Observe que el programa, tras cada llamada a la interrupción 13h, comprueba si está activo el indicador de acarreo y, de ser así, repite el proceso de lectura o escritura sin usar ningún contador. Esto, en casos extremos en los que no sea posible acceder a la unidad, puede bloquear totalmente el sistema ya que, a diferencia de las interrupciones 25h y 26h, los servicios de la BIOS no verifican automáticamente los errores ni comprueban las combinaciones de teclas que solicitan la interrupción del programa en curso.

Pruebe a sustituir el método de lectura y escritura empleando los servicios del DOS en lugar de los de la BIOS. Para ello debe recordar que dichos servicios no entienden de caras ni pistas, sólo de sectores. Un disquete cuenta con 2880 sectores, que puede ir leyendo en bloques del tamaño que le interese.

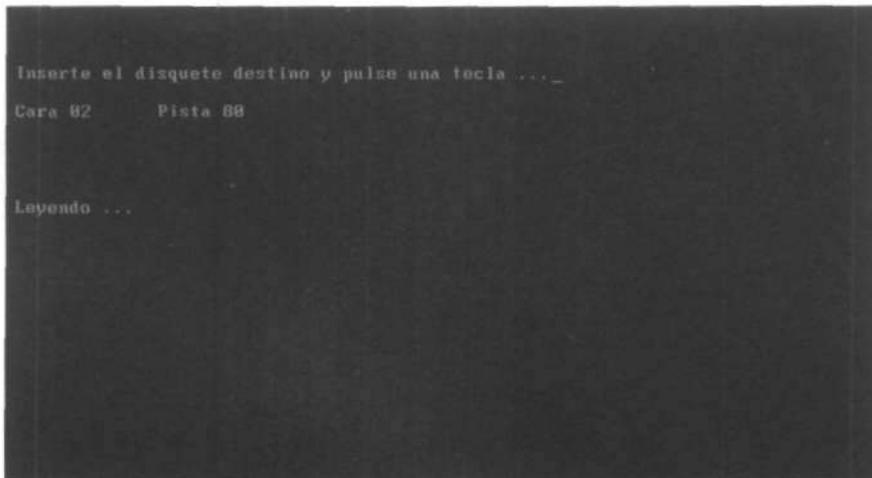


Figura 25.1. El programa de copia de discos en funcionamiento.

Resumen

A diferencia de lo que ocurre con los servicios de tratamiento de archivos y directorios, que son ofrecidos exclusivamente por el DOS y no se ven afectados por aspectos como el tamaño de las unidades de almacenamiento, para leer y escribir sectores las funciones disponibles son varias: la interrupción 13h de la BIOS, las interrupciones 25h y 2 6h del DOS, los servicios extendidos que hay en la interrupción 2 lh son los descritos en este capítulo.

No se han descrito servicios de más bajo nivel con los que podría, por ejemplo, dar formato a los discos, ya sea utilizando formatos estándar predefinidos u otros que podemos haber creado nosotros. Esto, salvo excepciones, no es necesario ya que resulta mucho más fácil efectuar esas tareas desde la línea de comandos del propio sistema operativo.

26

**Memoria
expandida y
extendida en DOS**

Todos los programas de ejemplo desarrollados hasta ahora, en los capítulos previos a éste, han hecho uso exclusivamente del área de memoria conocida como *de sistema*, es decir, los primeros 640 kilobytes de memoria RAM instalados en el sistema. En realidad, y como ya sabe, el DOS asigna automáticamente toda la memoria disponible a un programa cuando éste se pone en marcha aunque, como se vio en el capítulo inicial sobre los servicios del DOS, es posible liberar parte de esa memoria y usar funciones de asignación y liberación de bloques de memoria.

Si bien en principio 640 kilobytes parecían más que suficientes para cualquier tipo de aplicación, de hecho los primeros PC diseñados en torno al microprocesador 8086 contaban únicamente con 16 kilobytes de RAM, el tiempo se encargó de demostrar que la memoria siempre será un bien escaso, independientemente de la cantidad que se tenga. Dadas las limitaciones propias del procesador 8086/8088, presentes en sus sucesores siempre que se opera en modo real y, por tanto, no se tiene acceso al direccionamiento de 32 ó 64 bits, incentivaron la inventiva y dieron como resultado frutos como las especificaciones EMS y XMS que va a conocer, en parte, en este capítulo.

Empezaremos haciendo un poco de memoria sobre las capacidades de direccionamiento de los 8086 y sus limitaciones, que básicamente ya conoce, para, a continuación, conocer otros tipos de memoria que pueden existir en su ordenador y cómo acceder a ella.

Bits, direccionamiento y modos de operación

El microprocesador 8086 causó con su aparición una verdadera revolución, no tanto por su tecnología, aunque estaba por delante de la mayoría de microprocesadores existentes a finales de los setenta, sino por el hecho de ser utilizado por IBM para crear su

primer sistema personal: el PC. El 8086 disponía de un bus de direcciones de 20 bits, lo que le posibilitaba el acceso a 1 megabyte de memoria. Los registros de dicho procesador eran de 16 bits, facilitando la composición de direcciones que estuviesen en el rango de 2^{16} bytes o, lo que es lo mismo, 64 kilobytes.

Sin embargo, y gracias a un acceso segmentado que ha venido utilizándose en todos los ejemplos propuestos, se combinan dos registros, uno llamado de segmento y otro de desplazamiento, para obtener una dirección lineal de 20 bits, que es la que se envía por el bus de direcciones.

Avanzando hasta el 80286, uno de los sucesores del 8086, podemos encontrar un bus de direcciones de 24 bits que, utilizando la misma regla anterior, permite el acceso a 16 megabytes de memoria.

A continuación aparecen los 80386 y 80486, dotados de un bus de 32 bits, el mismo con el que cuentan actualmente los Pentium, que da acceso a la impresionante cantidad de 4 gigabytes de memoria.

Pese a estas posibilidades aún hoy, con procesadores como los Core 2 y Core 17 que tienen un bus de direcciones de 64 bits, los usuarios que trabajan sobre DOS siguen usando el modo real del procesador, por lo que sólo pueden disponer de los conocidísimos 640 kilobytes, al menos directamente, porque las posibilidades de direccionamiento de los 286, 386, 486, Pentium y Core quedan enmascaradas cuando funcionan en ese modo real, emulando el funcionamiento de un 8086/8088.

Esto implica que toda la memoria que queda por encima de la dirección 1.048.575, equivalente a OFFFFh: OOOFh en notación segmento:desplazamiento, no es accesible por medio de un direccionamiento inmediato, debido a que cualquier valor mayor en el registro de desplazamiento, que nos permitiría acceder a casi 64 kilobytes más de memoria, desbordaría el bus de direcciones.

Supon gamos que almacena en el registro DS el valor 0FFFh y en BX el valor 0010h, lo que aparentemente nos permitiría acceder a una dirección de memoria para almacenar o leer un dato. El resultado no sería el esperado, ya que al componer la dirección física nos encontraríamos con un valor de 21 bits, lo que causaría el desbordamiento del bus y la obtención en éste de la dirección 0000000, evidentemente distintita de la que esperábamos y correspondiente al primer vector de interrupción.

El direccionamiento en modo protegido de un 80286 se basa en el uso de selectores y descriptores, mientras que en el 386 y siguientes el bus de 32 bits permite trabajar cómodamente en un modelo de memoria plano. Estas posibilidades desaparecen cuando el procesador emula al ya citado 8086/8088. Sin embargo, sabemos que hay programas que acceden normalmente, trabajando sobre DOS, a más de un megabyte. Es posible gracias a las memorias expandida y extendida.

Memoria expandida

Este tipo de memoria se diseñó para los microprocesadores inferiores de la familia x86, es decir, los 8086/8088, cuyo diseño tiene limitado de forma absoluta el direccionamiento de más de 20 bits. La especificación inicial fue llamada LIM (*Lotus/Intel/Microsoft*),

mediante la cual se permitía a cualquier PC el acceso hasta a un máximo de 4 megabytes que se vieron ampliados hasta 8 en la versión L1M 4.0.

¿Cómo puede un 8086/8088 direccionar esto? Muy simple: no puede. Por ello hay que utilizar un truco bastante antiguo en el diseño de ordenadores como es la paginación de la memoria, método que se utilizaba ya para permitir que un pequeño Z80 pudiese acceder a 4 megabytes de memoria, por ejemplo en los microordenadores MSX. El método no es demasiado complejo. Se dispone de varios bancos de memoria que se dividen en páginas de 16 kilobytes. Un circuito integrado especializado, como puede ser el PPI 8255, se encarga de intercambiar estas páginas, permitiendo ver al microprocesador sólo algunas de ellas en un determinado momento.

Por lo tanto, el microprocesador sólo puede acceder a algunas de estas páginas, que se vuelcan en una determinada porción de memoria en el rango accesible por el microprocesador. La idea queda reflejada en la figura 26.1. El espacio de direccionamiento de 20 bits, accesible por el microprocesador, dispone de una zona o rango de direcciones por medio de las cuales es posible acceder simultáneamente a 4 páginas de 16 kilobytes de la memoria expandida. A este rango de direcciones es a lo que se denomina *marco de página* y, aunque siempre está situado en una misma dirección (que es posible configurar), por medio de él podemos acceder a toda la memoria expandida.

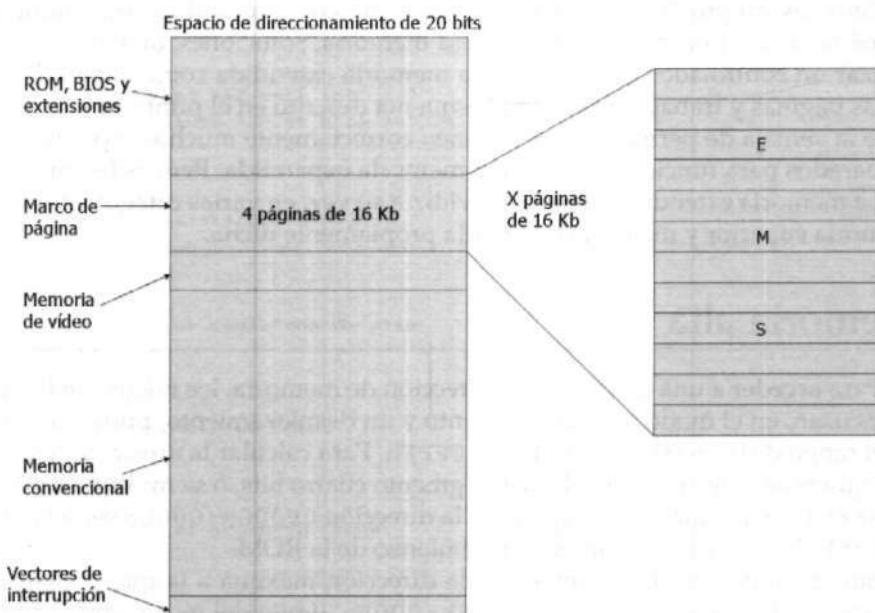


Figura 26.1. Modelo de memoria expandida.

En los sistemas DOS, este tipo de memoria se emplea mediante un controlador, llamado normalmente EMM.SYS, que añade un nuevo servicio a una de las interrupciones del DOS. Gracias a ella se puede acceder a esta memoria y establecer las páginas que deben estar activas en cada momento a través del marco de página.

La mayor ventaja de este tipo de memoria es que permite que un microprocesador limitado, como es el 8086, pierda parte de esas limitaciones. La mayor desventaja es la velocidad y facilidad de acceso, puesto que hay que ir cambiando las páginas a las que se desea acceder. Por tanto, y como conclusión, cabría decir que la memoria expandida no tiene direcciones reales en el rango de direccionamiento del microprocesador, por lo que no se pueden acceder inmediatamente a ella sino que hay que hacerlo a través de un marco de página que nos permite ver algunos de los bloques en que se divide. La memoria expandida puede existir en cualquier ordenador, independientemente del microprocesador.

Memoria extendida

En el momento en que aparece el microprocesador 80286, con su capacidad para direccionar hasta 16 megabytes de memoria gracias al bus de 24 bits, nace la que se denomina memoria extendida. En un sistema 286 o superior que tenga más de un megabyte de memoria, ésta puede utilizarse directamente siempre que trabajemos en el modo nativo del microprocesador, no el de emulación 8086 o modo real.

Pero hay un problema: el DOS no está preparado para utilizar este modo protegido y facilitar a las aplicaciones el uso de esa memoria. Soluciones: una de ellas consiste en utilizar un controlador que trate dicha memoria extendida como expandida, simulando las páginas y trabajando según el esquema descrito en el punto previo. Este método tiene la ventaja de permitir que funcionen correctamente muchos programas que están preparados para funcionar utilizando memoria expandida. Pero es la única ventaja.

La memoria extendida se puede dividir, a su vez, en varias categorías: memoria alta, memoria superior y memoria extendida propiamente dicha.

Memoria alta

Para acceder a una determinada dirección de memoria, los micros de Intel y compatibles usan, en el modo real, un segmento y un desplazamiento, pudiendo estar ambos en el rango de 0 a 65535 o de 0000h a FFFFh. Para calcular la dirección lineal física a la que quiere accederse, se desplaza el segmento cuatro bits, o se multiplica por 16, y se le suma el desplazamiento. Por ejemplo, la dirección 0FOOOh:0000h sería la dirección física 983.040, que es normalmente el comienzo de la ROM.

Sin embargo, ya vimos antes que la dirección máxima a la que podíamos acceder por medio de este método es la 0FFFFh: 000Fh, límite del megabyte, por lo que cualquier dirección superior causaría un desbordamiento, apuntando a donde no se espera. 0FFFFh: 0010h, por ejemplo, sería 0000h:0000h, podríamos decir que le hemos dado "la vuelta al marcador".

Paradójicamente, esto también ocurre en los sistemas utilizados actualmente, con microprocesadores Pentium y similares, que, sin embargo, tienen un bus de direcciones más ancho que les permitiría acoger la dirección de 21 bits 0FFFFh:0010h. Esto ocurre

porque está emulándose un 8086, en todos sus aspectos. Aún así podemos encontrar una solución rápida y fácil para disponer de casi 64 kilobytes más de memoria, consistente en impedir que se produzca ese desbordamiento.

El bus de direcciones de un 80286, por tomar este microprocesador como base, está compuesto por 24 líneas, denominadas A0, A1,... A23. Las últimas cuatro normalmente están desactivadas, lo que causa que se trabaje sólo con las 20 primeras. Si la línea A20 está inactiva, el microprocesador desprecia cualquier valor que supere los 20 bits, impidiendo el acceso a más de 1 megabyte. Si activásemos esa línea impediríamos el desbordamiento y, por tanto, podríamos acceder a esa memoria alta de la misma manera que lo hacemos con el resto de la memoria.

Podemos tanto conocer como modificar el estado de la línea A20 del bus de direcciones mediante el segundo bit de un byte que podemos leer del puerto 92h. Si dicho byte está a 0 es que la línea está inactiva. El programa siguiente, tal como se aprecia en la figura 26.2, informa sobre el estado de la línea permitiéndole, además, modificarla. Con este fin se explora la línea de parámetros que podemos encontrar a partir del byte 128 del PSP, cuya dirección obtenemos mediante el servicio 62h de la interrupción 2 lh.

```
segment Pila stack
    resw 512
FinPila:

segment Datos
; Mensajes informativos
MsgActiva db 'La linea A20 está activa.$'
MsgNoActiva db 'La linea A20 no está activa.$'

; Segmento de código

segment Código
.start:
Inicio:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila
    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax

    ; Obtenemos el segmento donde
    ; está el PSP
    mov ah, 62h
    ilit 21h
    ; lo ponemos en ES
    mov es, bx
    ; DI apunta a la longitud
    ; de la linea de parámetros
    mov di, 80h

    ; Si el contador está a 0
    cmp byte [es:di], 0
```

```

; no hay parámetros
jz NoHayParametros

; En caso contrario leer el
; valor del puerto 92h
in al, 92h

; Si el parámetro facilitado es 1
cmp byte [es:di+ ?], '1'
; activamos la linea A20
je Activar

; en caso contrario la desactivamos
and al, OFDh
jmp CambiarEstado

Activar:
; activamos
or al, 2

CambiarEstado:
; cambiamos el estado de la linea
out 92h, al

NoHayParametros:
; En cualquier caso
mov dx, MsgActiva

; obtenemos el estado de la linea
in al, 92h
; comprobamos su estado
test al, 2
; y mostramos el mensaje que corresponda
jnz Activa

mov dx, MsgNoActiva

Activa:
mov ah, 9
int 21h

Salir:
; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

```

Por lo tanto, la memoria alta, parte de la memoria extendida y conocida como HMA (*High Memory Área*), sólo está disponible en procesadores 286 y superiores, puesto que son los únicos que tienen un bus de direcciones suficientemente ancho como para direccionarla. La HMA está fuera del espacio de direccionamiento de 1 megabyte, pero no fuera del espacio que puede direccionarse con segmento ;desplazamiento, por lo que se puede manipular directamente sin ningún problema siempre que la línea A20 haya sido activada previamente. De hecho, las últimas versiones del DOS usan esa memoria para situar en ella parte del sistema o controladores de dispositivos.

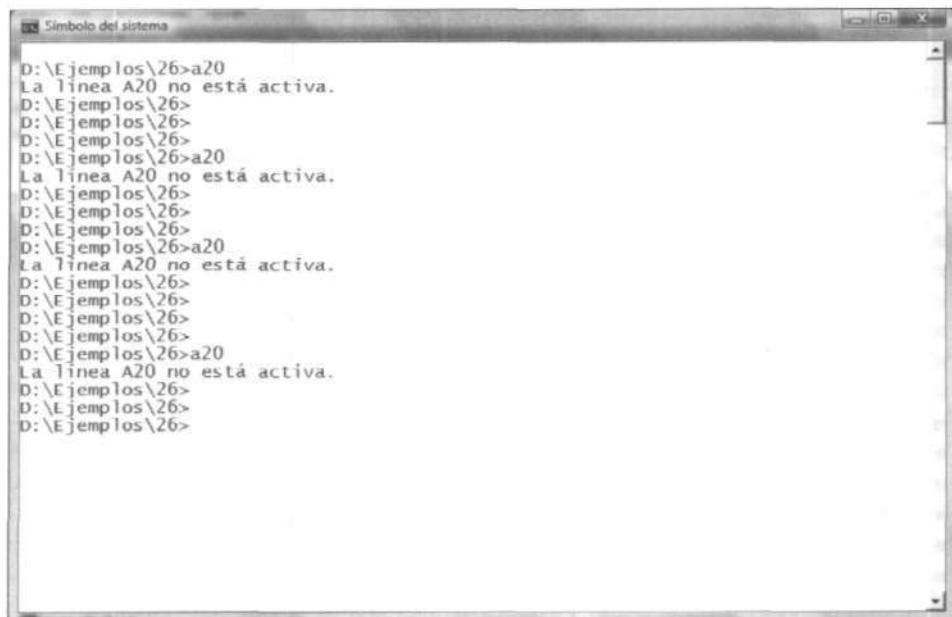


Figura 26.2. El programa muestra y modifica el estado de la línea A20.

Memoria superior

A pesar de que los microprocesadores 8086 / 8088 pueden direccionar hasta un megabyte, todos estamos acostumbrados al límite de 640 kilobytes, hablando del DOS, por supuesto. Esto es así porque parte del espacio de direccionamiento está reservado para memoria de vídeo, la ROM, ampliaciones de la ROM como las de los discos rígidos, el marco de página de la memoria EMS, etc. Sin embargo, siempre quedan rangos de direcciones libres que, por tanto, pueden ser utilizados para albergar memoria RAM.

La memoria superior, que es como se denomina a la memoria que está por encima de los 640 kilobytes pero por debajo del megabyte, se puede manejar de manera similar a la memoria convencional, puesto que está dentro del espacio de direccionamiento del microprocesador, asignando bloques y liberando bloques. Pero el gestor estándar de memoria del DOS, servicio 4Ah, no está preparado para gestionar esta memoria, por lo que es necesario disponer de otro gestor especializado.

Por lo tanto, la memoria superior, como parte de la memoria extendida, solamente puede existir en microprocesadores 286 y superiores, porque son los únicos que disponen de memoria extendida. Se llaman bloques UMB (*Upper Memory Block*) a la memoria que está por encima de los 640 kilobytes pero por debajo del megabyte, siendo posible su direccionamiento con sólo 20 bits. Existen otros métodos por los que se puede dotar a ordenadores 8086/8088 de memoria superior sin tener extendida, asignando RAM a las direcciones por encima de 640 kilobytes. En realidad, el marco de página por el que se accede a la memoria EMS puede identificarse como memoria superior.

Memoria extendida

Se llama así a aquella porción de memoria extendida que realmente está por encima de las posibilidades de direccionamiento sin salir del modo real. El acceso a esta memoria conllevaría la utilización de más de 20 líneas en el bus de direcciones, lo que sólo es posible en modo nativo, es decir, nos sería bastante complicado controlar por nosotros mismos esa memoria desde DOS. La solución, como en el caso de la memoria expandida, viene dada por un controlador que pondrá una serie de funciones a nuestra disposición para asignar bloques, copiar datos en ellos, liberarlos, etc. La diferencia es que este controlador accede directamente a esa memoria sin necesidad de intercambiar páginas, aunque teniendo que entrar y salir del modo protegido o nativo.

Actualmente los sistemas inferiores al 386 han desaparecido, y sólo se comercializan equipos con procesadores Pentium, Core o equivalentes, todos ellos dotados de una cantidad importante de memoria extendida. También es cierto que tanto la memoria alta como la superior suele ser utilizada por los sistemas operativos para *colocar* sus consoladores y código de sistema. Por lo tanto, la mayor parte de la memoria en los PC actuales es extendida, por lo que el resto del capítulo lo dedicaremos a su aprovechamiento.

La especificación XMS

Desde que apareció por primera vez la memoria extendida, hasta hoy, se han utilizado varios métodos distintos para utilizarla. A pesar de ello, el método recomendado se basa en el uso de un controlador que cumpla con las especificaciones XMS lo que nos permitirá no encontrarnos con problemas al trabajar en distintos sistemas. Este mismo controlador también facilitará el uso de la memoria alta y superior. Las últimas versiones de todos los sistemas DOS incorporan sus propios controladores XMS, tanto MS-DOS y PC-DOS como DR-DOS. Tan sólo hemos de instalarlo y comprobar, por ejemplo con la orden MEM, la cantidad de memoria XMS que hay disponible.

C:\>MEM			
Tipo de memoria	Total	Usada	Libre
Convenional	630K	62K	576K
Superior	0K	0K	0K
Reservada	0K	0K	0K
Extendida (XMS)	48.128K	48.128K	0K
Memoria total	48.768K	48.190K	576K
Total menor 1 MB	630K	62K	576K
Programa ejecutable más extenso		576K (589.728 bytes)	
Bloque memoria superior más extenso		0K (8 bytes)	

Figura 26.3. El comando MEM del DOS nos indica la cantidad de memoria existente.

La versión 3.0 de dicha especificación incorpora las funciones que se enumeran en la tabla 26.1. Por medio de ellas podemos utilizar la memoria alta, superior y extendida, además de controlar el estado de la línea A20 sin necesidad de recurrir a las instrucciones de entrada/salida como se ha hecho en el ejemplo previo.

Tabla 26.1. Funciones de la especificación XMS 3.0.

Código	Finalidad
00h	Obtener la versión del controlador.
01h	Asignar memoria alta.
02h	Liberar memoria alta.
03h	Activar globalmente la línea A20.
04h	Desactivar globalmente la línea A20.
05h	Activar localmente la linea A20.
06h	Desactivar localmente la línea A20.
07h	Comprobar el estado de la línea A20.
08h	Obtener la cantidad de memoria extendida libre.
09h	Asignar un bloque de memoria extendida.
0Ah	Liberar un bloque de memoria extendida.
0Bh	Mover un bloque de memoria extendida.
0Ch	Bloquear un bloque de memoria extendida.
0Dh	Desbloquear un bloque de memoria extendida.
0Eh	Obtener información de un identificador.
0Fh	Reasignar un bloque de memoria extendida.
10h	Asignar un bloque de memoria superior.
11h	Liberar un bloque de memoria superior.
12h	Reasignar un bloque de memoria superior.
88h	Obtener información sobre cualquier memoria extendida.
89h	Asignar cualquier bloque de memoria extendida.
8Eh	Obtener un identificador extendido de bloque.
8Fh	Reasignar cualquier bloque de memoria extendida.

Además de poder utilizar cualquiera de estas funciones, tendremos que comprobar si hay instalado o no un controlador XMS. Para ello usaremos la interrupción 2Fh,

conocida como la interrupción múltiple. Antes de ello almacenaremos en el registro AX el valor 430Oh y, tras la llamada, comprobaremos si en AL se ha devuelto el valor 80h. De ser así es que existe un controlador XMS instalado.

El código siguiente comprueba si está instalado el controlador XMS y lo comunica mediante un mensaje por consola.

```

segment Pila stack
    resw 512
FinPila:

segment Datos
; Mensajes informativos
MsgHay db 'Hay instalado un controlador XMS.$'
MsgNoHay db 'No hay instalado un controlador XMS.$'

; Segmento de código

segment Código
..start:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax

    ; asumimos que hay XMS
    mov dx, MsgHay

    ; comprobamos si existe
    mov ax, 4300h
    int 2Fh

    cmp al, 80h

    ; en caso afirmativo saltar
    je Salir

    ; en caso contrario tomar
    ; la dirección del otro mensaje
    mov dx, MsgNoHay

Salir:
    ; mostramos el mensaje
    mov ah, 9
    int 21h

    ; devolvemos el control
    ; al sistema
    mov ah, 4Ch
    int 21h

```

El gestor XMS

Habiendo verificado la existencia de un controlador XMS en el sistema, deberemos obtener la dirección donde se encuentra el gestor XMS, el punto de entrada a todos los servicios que nos ofrece el controlador.

Para ello utilizaremos la subfunción 431 Oh de la misma interrupción múltiple, que nos facilitará en los registros ES : BX dicha dirección.

Teniendo el punto de entrada al gestor, para realizar las llamadas usaremos una instrucción call usando como parámetro una doble palabra en la que, previamente, hayamos alojado los valores obtenidos en ES : BX. Antes de llamar al gestor pondremos en el registro AH el código del servicio a invocar, según la lista de la tabla 26.1. Por ejemplo, podemos obtener la versión del controlador que tenemos instalado usando la función 00h. Al hacerlo obtendremos en el registro AX la versión de XMS y en DX un indicador de existencia de memoria alta (HMA).

El código siguiente, basándonos en el ejemplo anterior que comprobaba la existencia del controlador XMS, obtiene la dirección del gestor y su versión.

Si ejecuta el programa desde una ventana de consola de Windows debe tener en cuenta que este sistema operativo intercepta el funcionamiento del controlador XMS, por lo que los resultados no serán los mismos que si realmente estuviese utilizando DOS como sistema operativo.

En la figura 26.4 se puede observar el resultado de ejecutar el programa en una consola de Windows Vista, mientras que en la figura 26.5 el entorno de ejecución es el de MS-DOS 6.22 sin Windows.

```
segment Pila stack
    resw 512
FinPila:

segment Datos

; Para guardar el punto de entrada
GestorXMS dd 0

; Mensajes informativos
MsgNoHay db 'No hay instalado un controlador XMS.$'

MsgVersion db 'La versión XMS es la '
    Byte1 db 0, '.'
    Byte2 db 0
        db '.', 13, 10, 10, '$'

MsgHayHMA db 'Hay memoria HMA.S'

MsgNoHayHMA db 'No hay memoria HMA.$'

; Segmento de código

segment Código
```

```
. ,start:
Inicio:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax

    ; comprobamos si existe
    mov ax, 4300h
    int 2Fh

    cmp al, 80h

    ; Si no hay controlador no
    ; podemos continuar
    jne NoHayXMS

    ; pedimos el punto de entrada
    ; al gestor XMS
    mov ax, 4310h
    int 2fh

    ; y lo guardamos
    mov [GestorXMS], bx
    mov lGestorXMb+2], es

    ; Solicitamos información de
    ; versión XMS
    xor ah, ah

    ; llamamos al gestor
    cali far [GeslorXMS]

    push dx ; guardamos indicador

    .- Convertimos a ASCII desde BCD
    add ah, '0'
    add al, '0'

    ; componemos el mensaje
    mov [Byte1], ah
    mov [Byte2J], al

    ; y lo mostramos
    mov dx, MsgVersion
    mov ah, 9
    int 21h

    pop dx ; recuperamos indicador
    or dx, dx ; ¿es 0?
    jz NoHayHMA
```

```

; Hay memoria HMA
intov dx, MsgHayHMA
jmp Salir

NoHayHMA:
; No hay memoria HMA
mov dx, MsgNoHayHMA
jmp Salir

NoHayXMS:
; indicamos que no hay controlador
mov dx, MsgNoHay

Salir:
; mostramos el mensaje
mov ah, 9
int 21h

; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

```

Observe cómo se efectúa la llamada al gestor XMS, usando la instrucción call seguida de la palabra far. De esta manera indicamos al ensamblador que se trata de una llamada no a una rutina propia, cuyo código se encuentra en el mismo segmento que la invocación, sino en un segmento diferente.

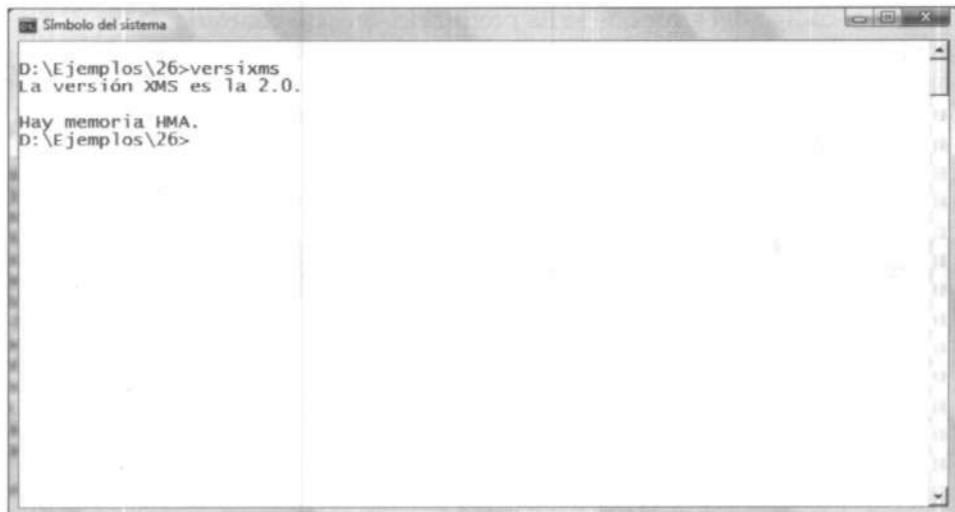


Figura 26.4. Resultado al ejecutar el programa en una ventana de consola de Windows Vista.

```
C:\>ASM>
C:\>ASM>
C:\>ASM>versiónes
La versión XMS es la 3.0.

Hay memoria HMA.
C:\>ASM>
C:\>ASM>
C:\>ASM>_
```

Figura 26.5. Resultado obtenido al ejecutarlo sobre MS-DOS 6.22.

Asignación de EMB

El tamaño de un EMB (*Extended Memory Block*) viene expresado en kilobytes. Por medio de la función 09h podemos asignar un bloque, indicando en el registro DX el número de kilobytes deseados que, lógicamente, no podrá ser mayor que el bloque más grande disponible. Tras llamar al gestor obtendremos en ese mismo registro el identificador del bloque asignado, identificador que necesitaremos posteriormente para ejecutar cada una de las operaciones sobre él.

Cada vez que usemos una de las funciones XMS hemos de tener en cuenta que se pueden producir errores. Si, tras la llamada, el registro AX contiene el valor 0, el registro BL indicará el código del error que se ha producido. En caso contrario, la función se habrá ejecutado sin problemas. Es de vital importancia comprobar el buen funcionamiento del programa mediante esos códigos de error, ya que, de lo contrario, es muy fácil terminar bloqueando totalmente el sistema. La tabla 26.2 enumera los códigos de error más habituales con una pequeña descripción.

Tabla 26.2. Errores notificados por el gestor de memoria XMS.

Código	Descripción
80h	La función solicitada no existe.
82h	Error en la línea A20.
8Eh	Error general del controlador.
8Fh	Error irrecuperable del controlador.
90h	No hay HMA disponible.
91h	La HMA ya está en uso.

Código	Descripción
A0h	Ya está asignada toda la memoria extendida existente.
A1h	No quedan disponibles identificadores XMS.
A2h	El identificador XMS facilitado no es válido.
A3h	El identificador origen del movimiento es inválido.
A4h	El desplazamiento origen del movimiento es inválido.
A5h	El identificador destino del movimiento es inválido.
A6h	El desplazamiento destino del movimiento es inválido.
A7h	El número de bytes a mover es inválido.
B0h	Hay disponible un bloque de memoria superior más pequeño.
B1h	No hay bloques de memoria superior disponibles.

Antes de solicitar un bloque de memoria extendida podemos obtener información sobre el tamaño máximo que hay disponible, utilizando para ello la función 08h. Al hacerlo obtendremos en el registro DX el tamaño total de memoria extendida libre, en kilobytes, y en AX el tamaño del bloque disponible más grande.

El siguiente programa muestra en la consola la memoria extendida total y el tamaño del bloque disponible más grande. Al igual que ocurría al obtener la versión del controlador, el resultado obtenido al ejecutar este programa dependerá de si estamos realmente en un entorno DOS o en una ventana de consola de Windows dado que, en este caso, los parámetros de configuración de la ventana determinarán la cantidad de memoria disponible.

```

segment Pila stack
    resw 512
FinPila:

segment Dalos
; Para guardar el punto de entrada
GestorXMS dd 0

; Mensajes informativos
MsgNoHay db 'No hay instalado un controlador XMS.$'

MsgMemoria db 'La cantidad total de memoria XMS es'
MemTotal db '      kilobfrtes.',13,10,10
        db 'El bloque libre más grande tiene '
MemLibre db '      kilobytes.$'

; Segmento de código
segment Código

```

```
..start:
    ; Preparar los registros de pila
    mov ax, Pila
    raoVss, ax
    mov sp, FinPila

    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax

    ; comprobamos si existe
    mov ax, 4300h
    int 2Fh

    cmp al, 80h

    ; Si no hay controlador no
    ; podemos continuar
    jne NoHayXMS

    ; pedimos el punto de entrada
    ; al gestor XMS
    mov ax, 4310h
    int 2fh

    ; y lo guardamos
    mov [GestorXMS], bx
    mov [CestorXMS+2], es

    ; Solicitamos información de
    ; la memoria total y libre
    mov ah, 8
    ; llamamos al gestor
    cali far [GestorXMS]

    push dx ; guardamos memoria total

    ; ES debe apuntar al segmento de datos
    push ds
    pop es

    ,- Convertimos la memoria libre
    mov di, MemLibre+4
    cali EnteroCadena

    ; recuperamos la memoria total
    pop ax
    ; y convertimos también
    mov di, MemTotal+4
    cali EnteroCadena

    ; apuntamos al mensaje
    mov dx, MsgMemoria
    ; y terminamos
    jmp 'Salir
```

```

NoHayXMS:
    ; indicamos que no hay controlador
    mov dx, MsgNoHay

Salir:
    ; mostramos el mensaje
    mov ah, 9
    int 21h

    ; devolvemos el control
    ; al sistema
    mov ah, 4Ch
    int 21h

%include "Convert.inc"

```

```

C:\>ASIM>
C:\>ASIM>
C:\>ASIM>xmslibre
La cantidad total de memoria XMS es de 48864 kilobytes.
El bloque libre más grande tiene 40964 kilobytes.
C:\>ASIM>_

```

Figura 26.6. El programa indica la cantidad de memoria libre y el bloque de mayor tamaño.

Transferencia de datos

Teniendo asignado ya el bloque de memoria extendida, podemos almacenar información en el usando la función OBh. Esta misma función nos servirá posteriormente para recuperar la información cuando nos sea necesario.

Para llamarla facilitaremos en la pareja de registros DS : SI la dirección de una estructura como la siguiente:

Longitud	dd 0
IdOrigen	dw 0
DireccionOrigen	dd 0
IdDestino	dw 0
DireccionDestino	dd 0

Los campos que componen esta estructura, y su finalidad, son:

- **Longitud:** Contendrá el número de bytes que van a transferirse, teniendo en cuenta que debe ser, obligatoriamente, un número par. Esto es debido a que la función utiliza para la transferencia las instrucciones del grupo lodsw y s Losw, que son más rápidas que las del tipo lodsb y stosb, pero a cambio transfieren sólo palabras.
- **IdOrigen:** Identificador del bloque de memoria que contiene la información a transferir. En caso de que el origen sea una zona de memoria convencional, el identificador será 0.
- **DireccionOrigen:** Dirección de memoria donde se encuentra la información a transferir. En el caso de que el origen sea memoria convencional, esta dirección será el segmento: desplazamiento, mientras que si se trata de un EMB la dirección será realmente un desplazamiento de 32 bits dentro de la longitud total del bloque.
- **IdDestino:** Identificador del bloque de memoria destino que, como en el caso de idOrigen, contendrá o 0 para indicar memoria convencional o el identificador de un EMB que va a recibir la información.
- **DireccionDestino:** Dirección de memoria donde se almacenará la información transferida, con las mismas consideraciones que DireccionOrigen.

La operación de transferencia de información entre memoria extendida y convencional, aunque esta función también puede usarse para transferir bloques entre memoria extendida y extendida o entre memoria convencional y convencional, es el punto más delicado al utilizar los servicios XMS. Nos podemos encontrar con errores tales como que el identificador del bloque XMS no sea válido, que no lo sea la dirección indicada, que el número de bytes a transferir sea inválido, etc. Estos errores serán indicados adecuadamente por los códigos antes indicados y la transferencia, como es lógico, no llegará a efectuarse.

Existe, sin embargo, la posibilidad de que facilitemos una dirección errónea de memoria convencional, causando que la información transferida sobrescriba zonas críticas y provocando la caída del sistema. Aquí es donde hay que extremar las precauciones.

Una vez nos dejé de ser útil, habremos de liberar todo bloque de memoria que hubiéramos asignado. Para ello utilizaremos la función OAh, a la que facilitaremos, en el registro DX, el identificador del EMB que se quiere liberar. Hecho esto el identificador pasará a ser inválido, quedando imposibilitado el acceso a la información que existiese en ese bloque. Aunque, como se aprecia en la tabla 26.1, existen muchas más funciones XMS, con las que se han descrito hay suficiente como para realizar un buen uso de la memoria extendida con que cuente nuestro sistema. Podemos obtener información sobre memoria libre, asignar bloques, mover información a ellos y de ellos y, por último, liberar bloques. Realmente no necesitamos mucho más.

El siguiente programa usa la mayoría de las funciones explicadas para copiar en memoria extendida el contenido actual de la pantalla, borrarlo y, tras esperar la pulsación

de una tecla, restaurarlo. En este caso se utilizan tan sólo 4 kilobytes de memoria extendida, pero si estuviésemos operando en algún modo gráfico la cantidad podría ser de 16, 64 o más kilobytes.

```
segment Pila stack
    resw 512
FinPila:

segment Datos

; Para guardar el punto de entrada
GestorXMS dd 0

; Mensajes informativos
MsgNoHay r1b 'No hay instalado un controlador XM&.5'
MsgFalloAsignacion
    db 'No puede asignarse el bloque.$'
MsgCorrecto db 'Proceso finalizado.$'

; Estructura para las transferencias
DatosTransferencia
    Longitud        dd 4000
    IdOrigen       dw 0
    DireccionOrigen dd 0
    TdDestino      dw 0
    DireccionDestino dd 0

; Segmento de código

segment Código
.start:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax

    ; comprobamos si existe
    mov ax, 4300h
    int 2Fh

    cmp al, 80h

    ; Si no hay controlador! no
    ; podemos continuar
    jne NoHayXMS

    ; pedimos el punto de entrada
    ; al gestor XMS
    mov ax, 4310h
    int 2f1i
```

```
; y lo guardamos
mov [GestorXMS], bx
mov [GestorXMS+2], es

; Solicitamos 4 K de memoria
mov dx, 4
mov ah, 9

; llamamos al gestor
cali far [GestorXMS]

; comprobamos la existencia
; de un error
or ax, ax
jz FalloAlAsignar

; Tenemos en DX el identificador
; del bloque, lo guardamos
mov [IdDestino], dx

; Establecemos la dirección
; de origen
mov word [DireccionOrigen+21,0B800h

; y copiamos la memoria de pantalla
; en el bloque de memoria extendida
mov si, DatosTransferencia
mov ah, 0Bh
cali far [GestorXMS]

; borramos el contenido de pantalla
cali BORRA

; y esperamos la pulsación de una tecla
xor ah, ah
int 16h

; Invertimos los identificadores
mov ax, [IdDestino]
mov [IdOrigen], ax
mov word [IdDestino], 0

; y las direcciones
mov word [DireccionOrigen+2], 0
mov word [DireccionDestino+2], 0B800h

; y copiamos la memoria extendida en
; la de pantalla
mov si, DatosTransferencia
mov ah, 0Bh
cali far [GestorXMS]

; liberamos el bloque de memoria
mov dx, [IdOrigen]
mov ah, 0Ah
cali far [GestorXMS]
```

```

; Terminamos
mov dx, MsgCorrecto
jmp Salir

FalloAlAsignar:
; No se puede asignar la memoria
mov dx, MsgFalloAsignacion
jmp Salir

NoHayXMS:
; indicamos que no hay controlador
mov dx, MsgNoHay

Salir:
; mostramos el mensaje
mov ah, 9
int 21h

; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

#include "Borra.inc"

```

Resumen

Las aplicaciones que se ejecutan sobre DOS se encuentran en un entorno de 16 bits aún cuando el equipo cuente con el procesador más avanzado, ya que en el modo real se simula el comportamiento del arcaico 8086. Es un problema que no encontraremos al crear aplicaciones para Windows o Linux, ya que ambos sistemas cuentan con versiones de 32 y 64 bits y, por tanto, permiten el acceso a toda la memoria disponible utilizando un modelo de direccionamiento lineal o plano.

Debido a las limitaciones del DOS aparecieron, en su día, especificaciones como las de la memoria expandida, o EMS, y memoria extendida, o XMS, siendo esta última la más empleada ya que, desde hace muchos años, los equipos incorporan procesadores 80386 o posteriores, siendo capaces de direccionar hasta 4 gigabytes de memoria. Para ello, los servicios XMS entran y salen del modo protegido cada vez que se accede a la memoria extendida, algo que ocurre de manera totalmente transparente para las aplicaciones DOS pero que consume un cierto tiempo.

27

Programas residentes en DOS

Actualmente la mayoría de sistemas operativos, como en es el caso de Linux y Windows, nos permiten tener varias tareas en ejecución, y la posibilidad de cambiar de una a otra con una simple combinación de teclas o un clic de ratón. Pero a pesar de todo, y aunque para muchos el DOS haya dejado de existir, todavía son miles los usuarios que no saben, y en ocasiones no quieren saber, de entornos gráficos, ventanas ni ratones, sintiéndose muy cómodos con su línea de comandos y sus aplicaciones basadas en texto.

Como es sabido, DOS es un sistema en el cual no existen privilegios de ejecución, multitarea ni nada parecido. Cuando se pone en marcha un programa, desde la línea de comandos, el sistema asigna un bloque de memoria y le transfiere el control, siendo a partir de ese momento dueño del ordenador hasta finalizar su ejecución, momento en el que se devuelve el control al DOS, que libera el bloque de memoria que se asignó. La única posibilidad de mantener dos programas en memoria consiste en asignar un bloque de memoria para poner en él un programa y no liberarlo al salir, habilitando además algún método por el cual esa porción de código que ha quedado residente pueda ser accesible. Ése será el tema que nos ocupe este extenso capítulo.

Aplicación y problemática

Seguramente todo programador se habrá sentido tentado en alguna ocasión de crear una versión residente de ese pequeño programa creado por él que le es tan útil. Un pequeño bloc de notas, una tabla de códigos ASCII o una calculadora, por poner un ejemplo, pueden ser necesarios en cualquier momento y, sin embargo, no es posible disponer de

ellos sin antes abandonar el programa que estamos ejecutando actualmente. En principio la idea puede parecerse buena, hacer accesible ese programa desde cualquier punto con la simple pulsación de una combinación de teclas, pero una vez puestas manos a la obra surgen tantos problemas, que nuestra feliz idea acaba en la papelera tras horas de seguimiento en el depurador y continua tortura del pulsador de reinicio del ordenador. Y es que desarrollar un programa residente que funcione realmente bien no es tan simple como pueda parecer a primera vista, siendo necesario controlar aspectos que en un programa normal nunca se tienen en cuenta. ¿Qué problemas son los que plantean habitualmente los programas residentes? El primero, sin lugar a dudas, su irregular funcionamiento, ya que unas veces al llamarlo opera correctamente, mientras que en otras bloquea totalmente el sistema, generalmente porque ha hecho una llamada al DOS cuando éste estaba ocupado o ha hecho uso de una interrupción no *reentrant*.

Otro problema típico es evitar la instalación más de una vez del programa, hecho que puede consumir innecesariamente la memoria del sistema. También es habitual desarrollar un programa residente que asume que al llamarlo va a estar en modo texto, y al hacerlo con otro modo de vídeo bloquea el programa que se estaba ejecutando.

En este capítulo el lector obtendrá toda la información necesaria para evitar éstos y otros problemas, desarrollando programas residentes de una calidad aceptable. Aunque es posible crear programas de este tipo utilizando lenguajes de alto nivel, como Pascal o C, si deseamos obtener la mayor eficiencia, en cuanto a tamaño del código, y el menor número de problemas, lo más indicado es el uso del ensamblador. Cualquier compilador de alto nivel incluye en el ejecutable código e información innecesarios, que hacen crecer la necesidad de memoria, y si hay algo deseable es que aquellos programas que van a quedar en memoria ocupen lo menos posible. También nos podemos encontrar con otros problemas debido al código que, por ejemplo los compiladores de C, se crea como prólogo/epílogo de las funciones, etc.

A diferencia de la mayoría de ejemplos previos, ensamblados siempre con NASM y enlazados con AIJNK, en este capítulo va a utilizar el ensamblador MASM junto con el enlazador de Microsoft. Como verá en algunos ejemplos posteriores, MASM facilita la creación de estructuras de datos complejas que facilitan ciertas operaciones, lo cual no es posible en NASM. Esto no significa, no obstante, que no pudieramos escribir el código de otra manera para ensamblarlo con NASM, es simplemente una cuestión de elección de la herramienta que más nos facilita el trabajo en cada momento.

Tipos de código residente

Aunque todo el código está compuesto por instrucciones cuya finalidad es realizar una cierta función, cuando se habla de programas residentes el código se puede clasificar según el método de activación que se utiliza. El código que se ha dejado residente

no puede, por sí mismo, obtener el control de la CPU, que es devuelto al sistema operativo para permitir la puesta en marcha de otros programas, siendo necesario que algún evento externo transfiera el control a dicho código.

Basándose en esto podemos distinguir, básicamente, dos tipos de código residente: uno el que se activa por software y otro el que lo hace por hardware. En el primer tipo se engloban, por ejemplo, todos aquellos controladores cuyo código queda residente y a disposición de cualquier programa, generalmente por medio de una interrupción. El ejemplo más cercano lo podemos encontrar en los controladores de memoria extendida, que son utilizados para hacer accesible toda aquella memoria que queda por encima de las posibilidades de direccionamiento del DOS tal y como se vio en el capítulo previo, o en el controlador de ratón, accesible a través de la interrupción 33h que también fue descrita en un capítulo anterior.

El segundo tipo corresponde a aquellos programas que se activan cuando se produce una interrupción generada por hardware, por ejemplo al pulsar una tecla o recibir un carácter por un conectar serie. También se puede dar el caso de que se usen en un mismo código ambos tipos. Por ejemplo, el controlador de ratón se activa con las interrupciones hardware cada vez que se mueve el ratón o se pulsa uno de sus botones, pero también se accede a él por medio de la interrupción software 33h, con la que podremos obtener información de la posición del puntero y el estado de los botones.

El tipo que más interesa a la mayoría de los programadores es el segundo. Por medio de él el código que queda residente se puede activar con la pulsación de una cierta de tecla, con la interrupción de teclado, o activarse cíclicamente cada cierto tiempo, con la interrupción de reloj, por poner los dos ejemplos más usuales. El primer tipo, activación por interrupción software, tiene más interés cuando más que desarrollar un programa residente que tiene una funcionalidad concreta y única, como una calculadora, se desea facilitar a otros programas unos ciertos servicios.

Por poner un ejemplo, se podría desarrollar un controlador gráfico que quedase residente y accesible por medio de una interrupción, que facilitase funciones de dibujo de primitivas gráficas complejas. Este código residente podría ser utilizado desde múltiples programas con diferentes finalidades, como puede ser la creación de un gráfico estadístico, el dibujo creativo o el diseño asistido.

Limitaciones del código residente

Desde el punto de vista del programador del código residente, el primer tipo suele plantear muchos menos problemas. Al activarse desde otro programa, por medio de una interrupción software, el código residente no suele tener limitaciones en cuanto al tiempo que puede emplear en desarrollar su función o la posibilidad de usar los servicios del DOS.

El panorama, sin embargo, es muy distinto cuando el código se activa por medio de una interrupción hardware, ya que hay que limitar el tiempo empleado al mínimo posible, no permitir la múltiple activación, no interrumpir un servicio del DOS si es que se piensa hacer uso de él, etc. Todas estas consideraciones serán tratadas a continuación, pero, comencemos por el principio, ¿cómo se deja algo residente en memoria?

Métodos para dejar código residente en memoria

El método más utilizado, durante mucho tiempo, para evitar la liberación, por parte del DOS, de una porción de la memoria asignada al iniciar un programa fue la interrupción 27h. Al llamar a ésta el registro CS debe contener el segmento en cuyo inicio está el PSP del programa y en DX la dirección, dentro de este segmento, del último byte que quedará residente.

A partir de estos parámetros resulta muy fácil determinar que la porción residente nunca podrá exceder de 64 kilobytes, además no es posible devolver valor de retorno alguno.

Actualmente la interrupción 27h está en desuso y ha dado paso al servicio 31h de la interrupción 21h, que permite la devolución de un código de retorno, facilitándolo en AL, y la posibilidad de dejar residente más de 64 kilobytes ya que el tamaño del bloque de memoria a preservar se indica, también en DX, en número de párrafos. No es necesario indicar el segmento del PSP del programa.

Según lo dicho anteriormente, podemos obtener dos conclusiones: la parte que quedará residente siempre será desde el principio del programa hasta la longitud que se haya facilitado, por lo que todo aquel código que no vaya a quedar residente, si es que lo hay, debe estar al final del programa, con el fin de que se libere la memoria que ocupa. En segundo lugar, debemos conocer la longitud del código, con el fin de facilitar el tamaño del bloque que ha de quedar residente.

Este dato, que es fácil obtener cuando se trabaja en ensamblador según se vio en un capítulo previo, tiene gran importancia, ya que si es inferior al tamaño real, parte del código que nosotros creemos residente será sobrescrito y, al intentar ejecutarlo, podría provocar un bloqueo del sistema.

Tampoco es conveniente que el tamaño facilitado sea mucho mayor al real, ya que ello irá en detrimento de los recursos libres del sistema.

Longitud del código residente

Es posible determinar el tamaño adecuado con bastante exactitud si tenemos en cuenta la estructura del código generado por los compiladores y ensambladores. El PSP (*Prefijo del Segmento de Programa*) es la dirección de inicio del programa, a partir de la cual se encuentra el código propiamente dicho.

A continuación se encuentra el espacio de memoria dedicado a almacenar las variables que se hayan definido en el programa, tanto las que tienen un valor inicial como aquellas que no han sido inicializadas.

El siguiente bloque de memoria contiene el *heap*, o espacio de memoria libre que podemos utilizar en las operaciones de asignación dinámica. Por último tenemos la pila, en la que, como sabemos, los datos se van almacenando desde la dirección más alta hacia la más baja.

Nota

Al programar en ensamblador el orden de los segmentos no los establece el ensamblador, como sí ocurre con los compiladores, por lo que no podemos asumir que el segmento de pila o de datos están después que el de código.

En ensamblador, como se vio en el ejemplo desde el que se ejecutaban otros programas, este cálculo es relativamente fácil de realizar, ya que basta con disponer una etiqueta al final del código que quedará residente, obtener su dirección y convertirla en párrafos, haciendo lo mismo con los demás segmentos.

Algo que tener en cuenta también, es el hecho de que el programa que quedará residente estará precedido de una estructura, el ya mencionado PSP, que ocupa 256 bytes, o lo que es lo mismo, 16 párrafos, así como que cada segmento se alinea siempre en una dirección múltiplo de 16.

Activación del código

Justo antes de devolver el control al DOS, dejando nuestro código residente, debemos prever algún método por el que este código se pueda activar. Como se dijo anteriormente, tenemos dos posibilidades: activación por software, asignando un vector de interrupción para acceder al código, o por hardware, interceptando la interrupción de teclado, reloj, etc.

Tratemos el primer caso, ya que, como veremos después, nos será de utilidad conocerlo aún cuando el que nos interese sea el segundo.

Ciertos controladores, como el del ratón, cuando se alojan en memoria se auto asignan un cierto vector de interrupción, siempre el mismo, porque asumen que ningún otro programa lo utilizará, y porque se asume que ese controlador siempre estará en ese vector de interrupción.

Sin embargo ésta no es una técnica válida de forma general, ya que no podemos asumir que el vector de interrupción 7 6h, por poner un caso, va a estar libre para nuestro programa siempre y en todos los sistemas.

En realidad, si consultamos la conocida recopilación sobre interrupciones de *Ralf Brown* (<http://www.ctyme.com/rbrown.htm>) nos daremos cuenta de que prácticamente todos los vectores de interrupción existentes son utilizados por un programa u otro, quedando muy pocos libres para el usuario. A pesar de ello, también es cierto que en un mismo momento en un sistema no estarán funcionando simultáneamente todos esos programas, por lo que necesariamente habrá vectores libres para nuestro uso, lo único que tenemos que averiguar es cuáles son.

Por lo tanto, el primer paso que tendrá que dar nuestro programa será buscar un vector de interrupción libre, es decir, nulo, que no tenga dirección alguna. Tanto la BIOS como el DOS, y gran parte de controladores, utilizan los primeros 128 vectores, por lo que lo más lógico es buscar a partir de ahí hasta el final. El código, como puede ver en el

ejemplo mostrado a continuación, es bastante simple. Un bucle que recorre los vectores del 128 al 255, si el valor almacenado en el vector es nulo, entonces es que está libre.

```

. *****
; Segmento de pila
. *****

Pila    segment stack 'stack'
        db 256 dup (?)
        FinPila:
Pila    ends

. *****
; Segmento de datos
. *****

Datos   segment 'data'
; Mensajes informativos
MsgVector db 'El vector '
NumVector db '      $'

MsgLibre db 'está libre.', 13, 10, '$'
MsgNoLibre db 'no está libre.', 13, 10, '$'

Datos   ends

. *****
; Segmento de código
. *****

Código  segment 'code'
assume CS:Código, DS:Datos, SS:Pila

Main:
; Configuramos los registros de pila
        mov ax, seg Pila
        mov ss, ax
        mov sp, FinPila

; y los del segmento de datos
        mov ax, seg Datos
        mov ds, ax
        mov es, ax

; vamos a explorar 128 vectores
        mov ex, 128

Bucle:
; calculamos en AX el número
; de vector para contar de
; 128 a 255, no al revés
        mov ax, 256
        sub ax, ex

; guardamos el número de vector
        push ax

; lo convertimos a cadena
        mov di, offset NumVector+3
        cali EnteroCadena

```

```

; y mostramos la primera
; parte del mensaje
mov dx, offset MsgVector
mov ah, 9
int 21h

; recuperamos el número de vector
pop ax

; guardamos ES porque la próxima
; llamada al DOS lo modificará
push es

; obtenemos la dirección del vector
mov ah, 35h
int 21h

; si BX no es cero
or bx, bx
; es que está ocupado
jnz Ocupado

; si ES no es cero
mov bx, es
or bx, bx
; es que está ocupado
jnz Ocupado

; si llegamos aquí es que el
; vector está libre
mov dx, offset MsgLibre
jmp Notifica

Ocupado:
    mov dx, offset MsqNoLibre

Notifica:
    ; notificamos el estado del vector
    mov ah, 9
    int 21h

    ; recuperarnos el anterior
    ; valor de ES
    pop es

    ; y continuamos el bucle
    loop Bucle

    ; devolvemos el control al sistema
    mov ah, 4Ch
    int 21h

Código ends

end Main

```

Para obtener el contenido de los vectores hemos usado el servicio 35h de la interrupción 2 lh, descrito en su momento.

También podríamos haber accedido directamente al área de memoria que ocupan los vectores, puesto que sólo vamos a leerlos.

El programa utiliza la rutina EnteroCadena codificada en los capítulos previos, para lo cual se han hecho un par de retoques, añadiendo la palabra offset donde era preciso, y se ha incluido directamente en el segmento de código, aunque no aparezca en el listado anterior.

Nota

Recuerde que la notación de MASM para definir los segmentos difiere ligeramente de la de NASM, así como la usada para obtener la dirección de un segmento o un identificador, debiendo emplearse seq y offset. Para ensamblar el programa simplemente ejecute el comando mi_vectores.asm, se producirán también el enlace automáticamente y podrá ejecutar el programa.

```

Símbolo del sistema
E1 vector 217 no está libre.
E1 vector 218 no está libre.
E1 vector 219 no está libre.
E1 vector 220 no está libre.
E1 vector 221 no está libre.
E1 vector 222 no está libre.
E1 vector 223 no está libre.
E1 vector 224 no está libre.
E1 vector 225 no está libre.
E1 vector 226 no está libre.
E1 vector 227 no está libre.
E1 vector 228 no está libre.
E1 vector 229 no está libre.
E1 vector 230 no está libre.
E1 vector 231 no está libre.
E1 vector 232 está libre.
E1 vector 233 no está libre.
E1 vector 234 no está libre.
E1 vector 235 no está libre.
E1 vector 236 no está libre.
E1 vector 237 no está libre.
E1 vector 238 no está libre.
E1 vector 239 está libre.
E1 vector 240 está libre.
E1 vector 241 está libre.
E1 vector 242 está libre.
E1 vector 243 está libre.
E1 vector 244 está libre.
E1 vector 245 está libre.
E1 vector 246 está libre.
E1 vector 247 no está libre.
E1 vector 248 no está libre.
E1 vector 249 no está libre.
E1 vector 250 no está libre.
E1 vector 251 no está libre.
E1 vector 252 no está libre.
E1 vector 253 no está libre.
E1 vector 254 no está libre.
E1 vector 255 no está libre.

D:\Ejemplos\27>

```

Figura 27.1. Lista parcial de los vectores mostrados por el programa.

Asignación de un vector de interrupción

Una vez hayamos encontrado un vector libre deberemos asignarlo para nosotros, almacenando en él, con la función 25h de la interrupción 21h, la dirección de la rutina que se ejecutará cada vez que se produzca la interrupción. Cuando se llama a una interrupción, como ya sabe, los parámetros se entregan en registros, y los valores de retorno de la misma forma. La técnica habitual es facilitar en AH el número de función que se desea, si es que la interrupción da varios servicios, y si son necesarios otros parámetros se usan registros adicionales.

Llegados a este punto nos queda un detalle por tener en cuenta. Nuestro programa al ejecutarse busca un vector de interrupción libre, le asigna la dirección del punto de entrada al código y devuelve el control al DOS, quedando residente, y por lo tanto disponible a cualquier software que quiera utilizar sus servicios. Antes de ello, sin embargo, es necesario que nuestro programa pueda ser identificado de alguna forma, o dicho en otras palabras, el software que quiera utilizar nuestros servicios tiene que poder determinar en qué vector se ha instalado, para lo que es necesario un método de identificación. El método más habitualmente utilizado es que la función 0 del gestor de la interrupción devuelva una cadena o código identificativo que otros programas puedan comprobar previamente al uso. Utilizando esta técnica nuestro primer programa residente podría ser similar al que se muestra a continuación:

```
. ****
; Segmento de pila
. **** k
Pila    segment stack 'stack'
        db 256 dup (?)
FinPila:
Pila    ends

. ****
; Segmento de datos

Datos   3egment 'data'

; Mensajes informativos
MsgError db 'No es posible instalar. '
        db 'No se encuentra un vector libre.?'

FinDatos:
Datos   ends

. ****
; Segmento de código
. ****

Código  segment 'code'
assume CS:Código, DS:Datos, SS:Pila

; saltamos al programa principal
; que buscará un vector e instalará
; el programa residente
jmp Main
```

```

; Este fragmento de código
; quedará residente y se
; ejecutará al invocar una
; determinada interrupción

GestorServicio:
    ; si AH no es 0
    or ah, ah
    ; no hacemos nada
    jnz FinGestor

    ; si es 0 devolvemos
    ; 2bb en AH
    mov ah, 2 55

FinGestor:
    ; salimos del gestor
    iret
;-----Fin del gestor

Main: ; Aquí comenzará el programa
; cuando se ejecute desde la
; línea de comandos

    ; Configuramos los registros de pila
    mov ax, seg Pila
    mov ss, ax
    mov sp/ FinPila

    ; y lo3 del segmento de datos
    mov ax, seg Datos
    mov ds, ax

    ; vamos a explorar 128 vectores
    mov ex, 128

Bucle:
    ; calculamos en AX el número
    ; de vector para contar de
    ; - 128 a 255, no al revés
    mov ax, 2 56
    sub ax, ex

    ; obtenemos la dirección del vector
    mov ah, 3 5h
    int 21h

    ; si BX no es cero
    or bx, bx
    ; es que está ocupado
    jnz Ocupado

    ; si ES no es cero
    mov bx, es
    or bx, bx
    ; es que está ocupado
    jnz Ocupado

```

```

; si llegamos aquí es que el
; vector está libre
mov dx, offset GestorServicio
; DS:DX apunta al nuevo gestor
push os
pop ds
; modificamos el vector
mov ah, 25h
int 21h

; calculamos el tamaño actual
; del programa
mov ax, FinPila
; sumando pila, datos y código
add ax, FinDatos
; más el PSP
add ax, Main+256
; convertimos en párrafos
mov el, 4
shr ax, el
; y tenemos en cuenta los
; ajustes
add ax, 3

; dejamos el código residente
; saliendo con el código 0
niöv dx, ax
xor al, al
mov ah, 31h

int 21h

```

Ocupado:

```

; continuamos buscando
loop Bucle

; si llegamos aquí es que no
; hay ningún vector libre
; mostramos el mensaje de error
mov dx, offset MsgError
mov ah, 9
int 21h

; devolvemos el control al sistema
mov      ah, 4Ch
int      21h

```

Código ends

end Main

Observe que hemos dejado la rutina de servicio a la interrupción al inicio del código, justo detrás de los segmentos de pila y datos, de tal forma que al calcular el número de bytes que quedarán residentes podamos descartar todo el código que se encarga de buscar la interrupción e instalar el programa residente.

Si ejecuta este programa no ocurrirá nada especial, siempre que haya vectores de interrupción libres el programa aparentemente terminará y devolverá el control al sistema. No obstante, una parte de él habrá quedado en memoria esperando a ser ejecutada. Con este fin utilizaríamos un programa similar al siguiente:

```
.
*****  

; Segmento de pila  

*****  

Pila    segment stack 'stack'  

        db 256 dup (?)  

FmPila:  

Pila    ends  

*****  

; Segmento de datos  

*****  

Datos   segment 'data'  

; Mensajes informativos  

MsgVpctor db 'Buscando en el vector '  

        NumVector db '      ,13,10,'$'  

MayEncontrado  

        db 'IntGraf está instalado.', 13, 10, '$'  

MsgNoEncontrado  

        db 'IntGraf no está instalado.', 13, 10, '$'  

Datos   ends  

; Segmento de código  

Código  segment 'code'  

assume CS:Código, DS:Datos, SS:Pila  

Main:  

; Configuramos los registros de pila  

        mov ax, seg Pila  

        mov ss, ax  

        mov sp, FinPila  

; y los del segmento de datos  

        mov ax, seg Datos  

        mov da, ax  

        mov es, ax  

; vamos a explorar 128 vectores  

        mov ex, 128  

Bucle:  

; calculamos en AX el número  

; de vector para contar de  

; 128 a 255, no al revés  

        mov ax, 256  

        sub ax, ex
```

```
; guardamos el número de vector
push ax

; lo convertimos a cadena
mov di, offset NumVector+3
cali EnteroCadena

; y mostramos el número de vector
; que está inspeccionándose
mov dx, offset MsgVector
mov ah, 9
int 21h

; recuperamos el número de vector
pop ax

; guardamos ES porque la próxima
; llamada al DOS lo modificará
push es

; obtenemos la dirección del vector
mov ah, 35h
int 21h

; si BX no es cero
or bx, bx
; es que está ocupado
jnz Ocupado

; si ES no es cero
mov bx, es
or bx, bx
; es que está ocupado
jnz Ocupado

; el vector está libre
; y por tanto no lo podemos
; invocar
pop es

; seguimos buscando
loop Bucle

; Hemos encontrado un vector
; que no está a cero
Ocupado:
; recuperamos ES
pop es
; ponemos AH a 0 para invocar
; al servicio de identificación
xor ah, ah
; Modificamos la instrucción
; siguiente para que invoque a
; la interrupción deseada
mov byte ptr [Intermpcion+1], al
```

```

Interrupción:
    int 0 ; invocamos a la interrupción

    ; si nos devuelve 255 en AH
    cmp ah, 255
    ; hemos encontrado nuestro residente
    je Encontrado

    ; en caso contrario seguimos
    loop Bucle

    ; si llegamos aquí es que hemos
    ; recorrido todos los vectores
    ; sin encontrar el programa
    mov dx, offset MsgNoEncontrado
    jmp Notifica

Encontrado:
    mov dx, offset MsgEncontrado

Notifica:
    ; notificamos si se encontró o no
    mov ah, 9
    int 21h

    ; devolvemos el control al sistema
    mov ah, 4Ch
    int 21h

Código ends

end Main

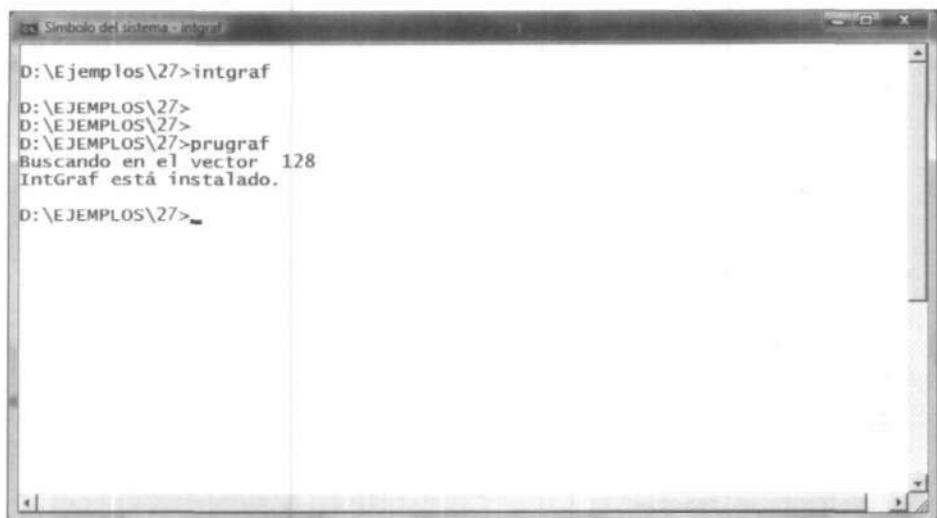
```

En este programa se utiliza una técnica no demasiado habitual, modificando el código del programa desde el propio código. Para localizar la interrupción correspondiente a nuestro programa residente necesitamos ir recorriendo vectores y, si no están vacíos, invocándolos. El problema es que la instrucción `int` tan sólo acepta como parámetro un valor inmediato, no podemos escribir una sentencia como `int al`, por poner un ejemplo.

¿Cómo conseguimos entonces que la instrucción `int` llame a cada ciclo a la interrupción que corresponda? Un método consiste en modificar el código generado por el ensamblador. Para ello debemos saber que una instrucción `int n` se convierte siempre en dos bytes de código, de los cuales el primero es fijo y el segundo indica el número de interrupción. Sabiendo la dirección de memoria donde está ese byte, para lo cual hemos dispuesto una etiqueta, nada nos impide modificarlo asignándole el contenido de AL. Si sigue la ejecución del programa desde DEBUG podrá comprobar cómo al llegar a la instrucción `int` no aparece como `int 0` sino como `int 8 Oh`, que es el primer valor que toma AL.

Tras ejecutar el primer programa, al que he llamado IntGraf, que queda residente, ejecute el segundo, llamado PruGraf para saber si la instalación ha tenido éxito y el número de vector en que está el programa residente.

En la figura 27.2 puede ver cómo se ejecuta el primero y después el segundo notifica que está instalado en el vector de interrupción 128.



```
D:\Ejemplos\27>intgraf
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>prugraf
Buscando en el vector 128
IntGraf está instalado.

D:\EJEMPLOS\27>
```

Figura 27.2. El programa queda residente y memoria y puede accederse a él mediante int 128.

Ocupación en memoria

Tal y como comentaba anteriormente, uno de los fines que debe perseguirse siempre es que el código que vaya a quedar residente ocupe la menor cantidad posible de memoria. Ésta es la razón de que para crear este tipo de programas generalmente se utiliza el lenguaje ensamblador, porque cualquier lenguaje de alto nivel, a pesar de que se utilice C, genera ejecutables que ocupan hasta 80 veces más para un ejemplo como el anterior que, básicamente, no hace nada.

Los propios programas en ensamblador pueden mejorarse para que, al quedar residentes, ocupen el menor espacio posible. Podemos conseguirlo, por ejemplo, no definiendo un segmento de datos al inicio del programa, llevando en su lugar la definición de los datos al final del segmento de código, un área que quedará descartada cuando el programa quede residente. Igualmente, puede ajustarse el tamaño de la pila y el modelo de memoria usado.

El código siguiente es una versión del mismo programa anterior pero con una estructura distinta. Se han empleado ciertas directivas de MASM, como model para establecer el modelo de memoria, y se ha usado una notación diferente para definir los segmentos aunque esto no afecta en nada al código.

El gestor del servicio y la parte de instalación se han definido como procedimientos. En realidad es simplemente una notación más clara a la hora de escribir el código, porque el resultado es el mismo, pero de esta forma los bloques de código quedan delimitados de manera visible.

Lo interesante es que el programa, al quedar residente, ocupa menos. Puede comprobarlo usando el comando mera del DOS para comprobar la memoria libre antes y después de instalar las dos versiones de este programa. El propio ejecutable ocupa unos 200 bytes menos.

```
; INTCRAF1.ASM

;- Este programa funciona de forma idéntica a
; INTGRAF, con la diferencia ocupar menos memoria
; una vez instalado

.MODEL Tiny      ; Modelo de memoria pequeño

.STACK 512       ; 512 bytes de pila

.CODE

Jmp Instalar     ; Saltar a la instalación

; Este procedimiento será el que quede
; residente en memoria
GestorServicios Proc

Or AH, AH        ; Si AH no es cero
Jnz Salir        ; saltar a la etiqueta Salir

; Si AH es cero facilitar la identificación
Mov AH, 255

Salir:
IREt ; Vuelta de la interrupción

GestorServicios EndP

; Este procedimiento se ejecutará tan sólo al
; iniciar el programa en memoria, no quedando
; re3identc

Instalar Proc

; DS apuntando a los mensajes
Mov DX, Seg Mensajel
Mov DS, DX

Mov CX, 80h ; Vamos a inspeccionar 128 vectores
Mov AL, 80h ; a partir del vector 128

; El servicio 35h de la INT 21h nos permite
; leer un vector
Mov AH, 35h

Bucle :
```

```

Int 21h ; Leer el vector indicado en AL

Mov DX, ES ; La dirección se devuelve en ES:BX

; Si tanto ES, que está en DX, como BX son cero
Or BX, DX
Jz VectorLibre ; es que el vector está libre

; En caso contrario saltar al siguiente vector
Inc AL

Loop Bucle ; y repetir el proceso

; Esta etiqueta sirve, sólo indicativa. Se
; llegará a ella cuando habiéndose recorrido
; todos los vectores no se haya encontrado
; uno libre.

NoHayVectorLibre :

; DS:DX apuntando al Mensajel
Mov DX, Offset Mensajel
Mov AH, 9h ; Servicio para imprimir
Int 21h

; Salir sin finalizar la instalación
Jmp SalirInstalacion

; A esta etiqueta se llegará cuando se haya
; encontrado un vector de interrupción libre.

VectorLibre :

; Preservar el número de vector que está en AL
Push AX

Mov BL, AL ; Pasar el número de vector a BL
Mov CL, 4
; Quedarse con los cuatro bits altos o nibble
;- más significativo
Shr BL, CL

Add BL, '0' ; Convertir en número decimal

; Si se trata de un dígito no superior a 9
Cmp BL, '9' + 1
Jl Validol ; la conversión es válida

; En caso contrario convertir en una letra de
; la A a la F
Add BL, 7

Validol:
; Guardar el primer dígito
Mov Byte Ptr [NumVector], BL

; Quedarse con los cuatro bits bajos o nibble
; menos significativo
And AL, OFh

```

```

; Y repetir el anterior proceso de conversión
Add AL, '0'
Cmp AL, '9' + 1
Jl Valido2

Add AL, 7

Valido2:

; para conseguir el segundo dígito
Mov Byte Ptr [NumVector+1], AL

; imprimir el mensaje indicando el
Mov DX, Offset Mensaje2
; vector en que se ha instalado INTGRAF
Mov AH, 9h
Int Jlh

Pop AX ; Recuperar el valor de AL

; Obtener en DS:DX la dirección de
; GestorServicios
Mov DX, Seg GestorServicios
Mov DS, DX
Mov DX, Offset GestorServicios

; y modificar el vector de interrupción
Mov AH, 2 5h
Int 21h ; para que apunte a él

; Dirección de este procedimiento, que no
; quedará residente
Mov DX, Offset Instalar
Mov CL, 4 ; Convertir los bytes
Shr DX, CL ; en párrafos dividiendo por 16

Inc DX ; Un párrafo más por seguridad

; Añadir los 16 párrafos (256 bytes) que ocupa
; el PSP
Add DX, 16

Mov AL, 0,- Código de retorno 0
Mov AH, 31h ; Servicio salir y quedar residente

Int 21h ; Salir quedando residente

; Salir sin terminar la instalación
Salirlnstalacion :

Mov AH, 4Ch

Int 21h

Instalar EndP

; Mensajes

```

```

Mensaje1 Db "Imposible instalar INTGRAF, no hay "
        Db "vectores libres.$"
Mensaje2 Db "INTGRAF instalado en el vector "
NumVector Db      "üüh.$"

End

```

Fiabilidad del método

A pesar de todos los detalles tenidos en cuenta, este método no es totalmente fiable. Si nuestro programa residente está en memoria siempre, antes de que cualquier otro intente localizarlo, no habrá problemas, pero si da la circunstancia de no ser así, el programa que busca nuestro servicio puede bloquear el sistema. En principio porque puede llamar a cualquier otro vector de interrupción, del cual no conoce su funcionamiento ni parámetros, y en segundo lugar porque muchas veces los vectores de interrupción almacenan direcciones que no apuntan a un código de gestión, sino a una cadena, a una tabla de parámetros, etc., por lo que si se realiza una transferencia a dicha interrupción el bloqueo es seguro.

Puede probar esto ejecutando el programa PRUGRAF sin antes haber instalado INTGRAF. El efecto más posible es que el sistema se cuelgue cuando se llegue a llamar a un vector de interrupción que no apunta a código ejecutable.

Para solucionar estos problemas, lo mejor es utilizar un método más sólido y estándar, existente desde la versión 3.0 del DOS. Por medio del vector de interrupción 2Fh es posible *engancharse* a una cadena de gestores de la interrupción, de ahí que se le denomine interrupción múltiple. Cuando un programa necesita hacer uso del código residente de un cierto servicio genera una in L 2Fh, pero puesto que en ella pueden existir varios gestores, antes se deposita en el registro AH un identificador por el cual comunicaremos el gestor al que va dirigido la llamada. Por su parte, nuestro código residente al recibir el control debe comparar el valor de AH con su identificador y, de no coincidir, transferirá la ejecución al siguiente controlador de la cadena. Si nuestro gestor contiene varios servicios, lo habitual es que en el registro AL se indique el deseado, y se usen el resto de registros de uso general para los parámetros que sea necesario pasar.

Supongamos que deseamos utilizar los servicios de un gestor residente cuyo identificador es 123. El primer paso que habremos de dar es comprobar si el gestor está instalado o no, para lo que almacenaremos en AH el identificador, 123, y daremos a AL el valor 0, a lo cual el gestor habitualmente devolverá en AL un valor distinto, confirmando su existencia. El código es tan simple como se muestra a continuación. Según el valor de AL, se da salida a "no está instalado" o "sí está instalado". Una vez se ha realizado esta comprobación podemos utilizar los servicios del gestor.

```

***** + + -*- + -**-*--**-
; Segmento de pila

Pila segment stack 'stack'
    db 256 dup (?)
FinPila:
Pila ends

```

```
*****  
; Segmento de ciatos  
  
Datos segment 'data'  
  
    ; Mensajes  
MsgExiste Db 'El controlador está instalado.$'  
MsgNoExiste Db 'El controlador no está instalado.?'  
  
Datos ends  
  
*****  
; Segmento de código  
*****  
Código segment 'code'  
    assume CS:Código, DS.-Datos, SS:Pila  
Main:  
    ; Configuramos los registros de pila  
    mov ax, seg Pila  
    mov ss, ax  
    mov sp, FinPila  
  
    ; y los del segmento de datos  
    mov ax, seg Datos  
    mov ds, ax  
  
    ; Comprobamos si existe el  
    ; servicio 123  
    Mov AH, 123  
    Xor AL, AL  
    int 2Fh  
  
    ; si no existe  
    Úr AL, AL  
    ; saltar  
    Jz NoExiste  
  
    ; mensaje de que si existe  
    Mov DX, Offset MsgExiste  
    jmp Salir  
  
NoExiste:  
    Mov DX, Offset MsgNoExiste  
  
Salir:  
    ; Mostrar el mensaje  
    Mov AH, 9  
    Int 21h  
  
    ; y salir al sistema  
    Mov AH, 4Ch  
  
    Int 21h  
  
Código ends  
  
end Main
```

Si ejecuta este programa, puesto que no hemos instalado ningún servicio que responda al código 123, siempre obtendrá el mismo resultado: un mensaje indicando que el controlador no está instalado. No obtenemos ningún efecto indeseado a pesar de ello, ya que la interrupción múltiple, y todos los servicios que se incorporan a ella, saben cómo comportarse ante servicios inexistentes.

La interrupción múltiple

Los programas INTGRAF e INTGRAF1, escritos en los puntos previos, consiguen quedar residentes y accesibles a través de una cierta interrupción software, pero con una serie de limitaciones y problemas que ya conocemos. La solución propuesta a dichos problemas pasa por el uso de la interrupción múltiple, número 2Fh, llamada así porque aunque se trate de un sólo vector, en realidad da paso a una cadena de gestores de interrupción que se distinguen entre sí por un código facilitado en el registro AH. De esta forma, en el último ejemplo que hemos escrito, la comprobación de si un cierto programa residente está o no instalado es mucho más simple, sin necesidad de ir recorriendo vectores de interrupción.

Nuestro objetivo inmediato es emplear dicha interrupción, la 2Fh, para que nuestra aplicación residente pueda instalarse y ser detectada con más facilidad, sin necesidad de ir recorriendo todos los vectores de interrupción hasta encontrar uno libre.

Engancharse a la interrupción múltiple

Ya que el vector de interrupción 2Fh es utilizado por multitud de programas, no podemos simplemente sustituirlo haciendo que apunte a nuestro gestor de interrupción, sin más, ya que ello pararía el sistema casi con toda seguridad. Por ello, antes de escribir en el vector la dirección del nuevo gestor utilizando el servicio 25h de la interrupción 2lh, tendremos que obtener la dirección del gestor actual, dirección que conservaremos para poder transferirle el control cuando sea necesario, no interrumpiendo así la cadena de controladores.

Nota

Al interceptar la interrupción múltiple hemos de tener en cuenta que ésta es usada por muchos programas y, por tanto, no debemos romper la cadena de gestores que hay enganchados a ella.

Para leer un vector de interrupción utilizaremos el servicio 35h de la interrupción 21h, como ya hicimos en un ejemplo previo, facilitando en el registro AL el número correspondiente, en este caso 2Fh. Los parámetros serán devueltos en los registros ES, el segmento, y BX, el desplazamiento. Esta operación, preservar la dirección del antiguo

gestor de la interrupción, la realizaremos lógicamente durante la instalación. Hecho esto podremos modificar la dirección del vector, haciendo que éste apunte al gestor que nosotros deseamos instalar.

Según se ha dicho anteriormente, la interrupción múltiple es utilizada por muchos programas. Cuando se genere dicha interrupción, y por lo tanto nuestro gestor reciba el control, hemos de comprobar si realmente la llamada es para nosotros o no. Para ello tendremos que comprobar si el valor que trae el registro AH corresponde con nuestro identificador y, en caso de no ser así, pasaremos el control al siguiente controlador de la cadena, cuya dirección habíamos preservado anteriormente.

En caso de que la llamada a la interrupción 2 Fh sea efectivamente para nuestro gestor, la acción a realizar vendrá generalmente determinada por un código de función que es facilitado en el registro AL. Realizada esta acción, lógicamente, habrá que devolver el control de la llamada. No olvidemos que aunque 2 Fh sea una interrupción software, llamada por otros programas y no por un dispositivo, sigue siendo una interrupción, por lo que al llamarla se guarda el registro de indicadores *oflags* en la pila. Al salir de nuestro controlador este registro debe ser extraído de la pila, lo que habitualmente se hace de forma automática al ejecutar la instrucción *iret*. Sin embargo, y al tratarse de una interrupción software, el registro de indicadores puede ser usado por algún gestor precisamente para devolver alguna indicación, por lo que puede no ser procedente usar la instrucción *iret*. En casos así usaremos la habitual instrucción *ret*, pero disponiendo después un número indicando el número de bytes adicionales a extraer de la pila.

Un primer ejemplo

Veamos en la práctica cómo un programa puede formar parte de la cadena de gestores de la interrupción múltiple, escribiendo el código que se muestra a continuación. Este programa, INT2F. ASM, no tiene más finalidad que la de demostrar lo explicado anteriormente, ya que el programa en sí no hace nada más que notificar que está instalado.

Observe que para volver de la interrupción se ha usado la forma *retf* de la instrucción *rct*, para indicar así al ensamblador que la dirección de retorno es de tipo FAR, ya que al estar trabajando en un modelo de memoria pequeño, el ensamblador por defecto generaría una dirección de retorno NEAR cuando, en realidad, al gestor se le estará llamando desde cualquier otro punto excepto desde nuestro propio programa.

```
; INT2F.ASM
; Este programa queda residente en memoria
; siendo accesible mediante la interrupción
; múltiple. Su código de identificación
; será el 123.

; Modelo de memoria pequeño
.MODEL Sital1
; Pueden usarse instrucciones del 386
.386
```

```
.STACK 512      ; 512 bytes de pila

.CODE ; Inicio del segmento de código
; Generar código de configuración
.STARTUP

        Jmp Instalar    ; Saltar a la instalación

; Para preservar la dirección del
; controlador anterior
Antigua2F Dd  ?

; Este procedimiento será el que quede
; residente en memoria
GestorServicios Proc

        Cmp AH, 123 ; Comprobar si es para nosotros
        Jne NoLoEs

        Mov AX, 54321 ; Devolver otro código
        ; Volver eliminando el registro de indicadores
        ; de la pila
        Retí 2

NoLoEs:
        ; Saltar al siguiente gestor de la lista
        Jmp [CS:Antigua2F]

GestorServicios EndP

; Este procedimiento se ejecutará tan sólo al
; cargar el programa en memoria, no quedando
; residente

Instalar Proc
        ; Obtener la dirección del actual gestor
        Mov AL, 2fh
        Mov AH, 35h
        Int 21h

        ; Preservar la dirección original
        Mov Word Ptr [CS:Antigua2F], BX
        Mov Word Ptr ICS:Antigua2F+2], ES

        ; Instalar en el vector
        Mov DX, Seg GestorServicios
        Mov DS, DX
        ; la dirección apuntando a nuestro gestor
        Mov DX, Offset GestorServicios
        Mov AH, 25h
        Int 21h

        ; Calcular el tamaño a dejar residente
        Mov DX, Offset Instalar
        Mov CL, 4
        Shr DX, CL
```

```

Ino DX
Add DX, 16
Mov AX, 3100h ; Salir y quedar residente
Int 21h
Instalar EndP

```

End

Las macros de MASM, como .MODEL, .STACK, .CODEy . STARTUP, simplifican el código. Con .MODEL definimos el modelo de memoria que usará el programa, lo cual afecta a la disposición de los distintos segmentos. La directiva .386 indica al ensamblador que el procesador de destino es, como mínimo, un 80386 y que, por tanto, pueden utilizarse ciertas instrucciones que no existen en los 8086. La macro . STACK crea el segmento de pila con el tamaño indicado. De no facilitarse un tamaño, se asumirá por defecto una pila de 1 kilobyte. . CODE define el segmento de código y la macro . STARTUP genera todo el código de preparación de los registros de pila y segmento de datos. Tras ensamblar el programa, simplemente escribiendo mi int2f.asm, puede seguirlo con DEBUG y observar el código generado automáticamente por las macros.

El código siguiente corresponde a un programa cuya finalidad es comprobar si INT2F está o no instalado. Ejecútelo antes y después de haber instalado la parte residente anterior, para así comprobar su correcto funcionamiento. Si ejecuta múltiples veces el programa INT2F éste se instalará varias veces, como podrá comprobar mediante la orden MEM/CdeI DOS.

```

.MODEL Small
.386

.STACK S12

.DATA

Msyl Db "No está instalado$"
Msg2 Db "Si está instalado$"

.CODE
.STARTUP

Mov AII, 123 ; Código de nuestro gestor
Tnt 2fh ; comprobar si está instalado

Cmp AX, 54321 ; ¿Ha devuelto el código?
; si no, es que no está instalado
Jne Noinstalado

```

Instalado:

```

; Mostrar el mensaje que corresponda
Mov DX, Offset Msg2
Jmp Salir

```

No instalado:

```
Mov DX, Offset Msg1
```

Salir:

```
Mov AH, 9 ; Sacar el mensaje por pantalla  
Int 21h
```

```
Mov AH, 4Ch ; y salir  
Int 21h
```

```
End
```

```
D:\EJEMPLOS\27>print2f  
No está instalado  
D:\EJEMPLOS\27>  
D:\EJEMPLOS\27>  
D:\EJEMPLOS\27>int2f  
  
D:\EJEMPLOS\27>  
D:\EJEMPLOS\27>  
D:\EJEMPLOS\27>print2f  
Si está instalado  
D:\EJEMPLOS\27>  
D:\EJEMPLOS\27>  
D:\EJEMPLOS\27>
```

Figura 27.3. Comprobamos la instalación del programa residente en la interrupción múltiple.

Cómo evitar la reinstalación

Uno de los fallos más habituales en los programas residentes es el de la reinstalación. Si la parte de instalación del programa no lo evita, el usuario al ejecutarlo varias veces causará que en memoria se instale varias veces la parte residente, con el consiguiente consumo innútil de memoria. Para evitar que nuestro programa residente sea instalado repetidas veces, lo que haremos será incluir en la parte de instalación el código necesario para comprobar si ya está instalado y, en caso afirmativo, terminar el proceso. Esto es lo que se hace en el código siguiente, que es el programa INT2F. ASM, tras haber modificado el ejemplo previo para evitar la reinstalación.

```
; Modelo de memoria pequeño  
.MODEL Small  
; Pueden usarse instrucciones del 386  
.386  
  
.STACK 512 ; 512 bytes de pila
```

```

.CODEF ; Inicio del segmento de código
; Generar código de configuración
.STARTUP

Jmp Instalar ; Saltar a la instalación

; Para preservar la dirección del
; controlador anterior
Antigua2F Dd ?

; Este procedimiento será el que quede
; residente en memoria
GestorServicios Pror,

Cmp AH, 123 ; Comprobar si es para nosotros
Jne NoLoes

Hov AX, 54321 ; Devolver otro código
; Volver eliminando el registro de indicadores
; de la pila
Retf 2

NoLoEs:
; Saltar al siguiente gestor de la lista
Jmp [CS:Antigua2F]

GestorServicios EndP

; Este procedimiento se ejecutará tan sólo al
; cargar el programa en memoria, no quedando
; residente

Instalar Proc

Mov AH, 123 ; Comprobar si ya está instalado
Int 2Fh
Cmp AX, 54321
; Si es así no permitir la reinstalación
Je Yainstalado

; Obtener la dirección del actual gestor
Mov AL, 2fh
Mov AH, 35h
Int 21h

; Preservar la dirección original
Mov Word Ptr [CS:Antigua2F], BX
Mov Word Ptr [CS:Antigua2F+2], ES

; Instalar en el vector
Mov DX, Seg GestorServicios
Mov DS, DX
; la dirección apuntando a nuestro gestor
Mov DX, Offset GestorServicios
Mov AH, 25h
Int 21h

```

```

; Calcular el tamaño a dejar residente
Mov DX, Offset Instalar
Mov CL, 4
Shr DX, CL
Inc DX

Add DX, 16

Mov AX, 3100h ; Salir y quedar residente

Int 21h

Yalnstalado:

,• Mostramos el mensaje
Push CS
Pop DS
Mov DX, Offset Msg
Mov AH, 9
Int 21h
; y devolvemos el control
Mov AH, 4Ch
Int 21h

Instalar EndP

```

Msq Db "El programa ya está instalado\$"

End

En la parte inferior de la figura 27.4 puede ver cómo se comprueba que el residente no está instalado, cómo se instala y, tras verificar que es así, se intenta la reinstalación. El resultado es la impresión de un mensaje de aviso y devolución del control al sistema sin efectuar la instalación.

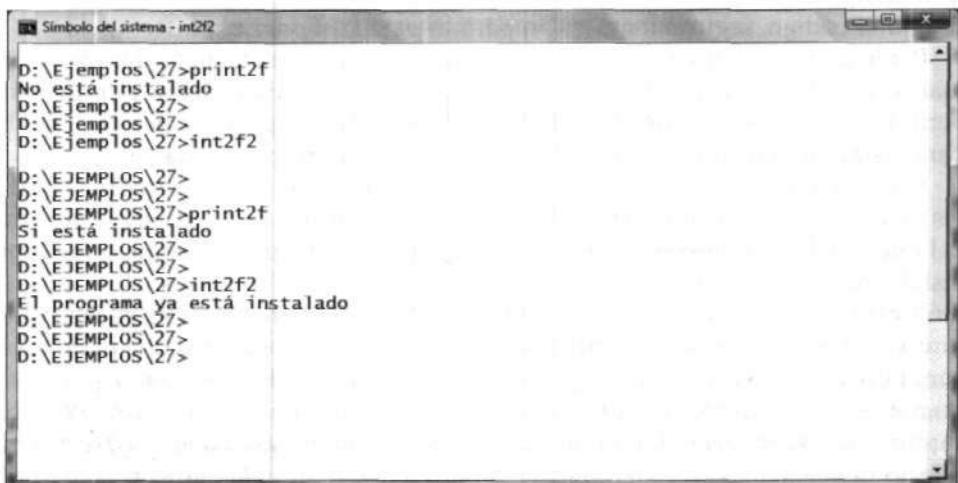


Figura 27.4. El programa ahora no permite la reinstalación.

Nota

Si intenta ejecutar el programa INT2F2 teniendo instalado de antemano INT2F, la anterior versión, verá que automáticamente se notifica que ya está instalado. Esto sucede porque comparten el mismo identificador de gestor y método de notificación de presencia.

Facilitar la desinstalación

Uno de los temas más complejos en el desarrollo de un programa residente es, sin duda alguna, el de la desinstalación. Esta aparente complejidad viene causada más por el desconocimiento del DOS, a nivel interno, que por su dificultad real. Básicamente tenemos que dar dos pasos para dar por completa la desinstalación de un programa residente: restituir las direcciones originales de los vectores de interrupción que hayamos modificado, si es que esto es posible, y liberar la memoria ocupada por el programa.

Restauración de los vectores

Aunque hasta ahora, en los ejemplos previos, sólo estamos modificando un vector de interrupción, el 2Fh, posteriormente veremos que, en la práctica, todos los programas residentes interceptan otras interrupciones, como la de teclado, reloj, disco, etc. Si nuestro programa va a ser desinstalado, y por lo tanto la memoria que ocupa posiblemente sea ocupada por otro programa, está claro que los vectores de interrupción modificados no pueden continuar apuntando a donde lo hacen, ya que en caso de que se produjese la interrupción se ejecutaría un salto a una dirección de memoria en la que puede haber cualquier código, seguramente no preparado para gestionar esa interrupción.

Por lo tanto el hecho de guardar, durante el proceso de instalación, las direcciones actuales de los vectores de interrupción tiene una doble utilidad, ya que por una parte nos permite llamar a los anteriores gestores de nuestro propio gestor y, por otra, nos servirá para restaurar la dirección original al desinstalar nuestro programa.

Sin embargo, hay que tener en cuenta que no podemos, sin más, escribir en los vectores de interrupción la antigua dirección, ya que es posible que otro programa, después del nuestro, haya interceptado ese mismo vector para su uso. Imagine que después de instalar nuestro residente se haya instalado otro que también hace uso de la interrupción múltiple y otras interrupciones. ¿Qué pasaría si nuestro programa escribiese la antigua dirección en el vector de interrupción? Básicamente dos cosas: por una parte estaremos cortando la comunicación al programa que se ha instalado después, que ya no será llamado por esa interrupción. Por otra parte, y lo que es más peligroso, ese otro programa al capturar la interrupción habrá obtenido la dirección antigua, que era precisamente la que apuntaba a nuestro programa, de tal forma que es posible que se realice una llamada a nuestro código cuando éste ya haya sido desinstalado.

Según esto, antes de restaurar un sólo vector de interrupción, si es que hemos modificado varios, debemos asegurarnos que en el momento en que vamos a realizar la desinstalación dichos vectores están todos apuntando a nuestro código. Si esto no es así simplemente debemos rechazar la petición de desinstalación, indicando al usuario que esto no es posible.

También se le puede indicar, como solución, que desinstale primero el programa que haya iniciado después del nuestro y vuelva a intentarlo.

Liberar la memoria

Cuando se inicia un programa en memoria el DOS asigna dos bloques de memoria, almacenándose en uno el código y datos del programa que se va a ejecutar y en otro una serie de variables de entorno. Todo programa, al alojarse en memoria, está siempre precedido por un bloque de 256 bytes conocido como PSP (*Program Segment Prefix/Prefijo del segmento de programa*), conteniendo algunos datos que nos pueden ser de interés. En la tabla 27.1 se indica la dirección dentro del PSP de algunos de los datos que podemos obtener.

Tabla 27.1. Contenido del PSP.

Dirección	Tamaño	Contenido
0h	2	Instrucción int 20h.
2h	2	Segmento de la primera dirección libre detrás del programa.
Ah	4	Dirección de terminación.
Eh	4	Dirección del controlador de Control-Inter .
12h	4	Dirección del controlador de error.
16h	2	Segmento del PSP del proceso padre.
2Ch	2	Segmento del bloque de entorno.
5Ch	16	Primer parámetro pasado.
6Ch	16	Segundo parámetro pasado.
80h	1	Número de bytes en la línea de comandos.
81h	127	Línea de comandos/Área de transferencia.

Al ejecutar un programa, en el momento en que éste se inicia el registro ES contiene el segmento correspondiente a su PSP que, normalmente, conservaremos durante la ejecución del programa para cuando sea necesario acceder a él. Como puede ver en la tabla 27.1, en la dirección 2Ch existe un dato de 2 bytes que es, en realidad, el segmento correspondiente al bloque que contiene los datos del entorno del programa.

Disponiendo de estos dos datos, el segmento del bloque de entorno y el segmento del PSP, lo único que necesitamos es utilizar el servicio 4 9h de la interrupción 2 lh, descrito en un capítulo previo. Este servicio nos permite liberar un bloque de memoria, para lo cual tendremos que entregar el segmento en el registro ES.

Tercera versión de INT2F

Vamos a modificar una vez más el programa INT2F.ASM, en este caso añadiendo el código necesario para hacer posible su desinstalación. Lo primero que haremos, en la parte que se ocupa de la instalación del programa, será comprobar si se ha pasado el parámetro /D. Para ello inspeccionaremos la línea de comandos del DOS, cuya longitud encontraremos en el byte 8 Oh del PSP, encontrándose el contenido a partir del byte 8 lh. En caso de que encontremos la opción mencionada, y tras comprobar que el programa se encuentra instalado, tendremos que comprobar si es posible restaurar los vectores de interrupción, haciéndolo en caso afirmativo.

Llegados a este punto, hay que tener en cuenta que para desinstalar el programa lo que hacemos es volver a ejecutarlo con una opción, en este caso /D. Por lo tanto, en memoria existirán dos copias del programa, una la que ya está instalada, y responde a la interrupción 2 Fh, y otra que está ejecutando la parte de instalación, realizando las comprobaciones anteriores. Esto es importante tenerlo claro, ya que los datos que necesitamos para poder liberar la memoria y restaurar los vectores de interrupción no los tiene la copia que estamos ejecutando en este momento, sino la que ya se encuentra instalada. Por ello el programa, en su parte de instalación, tendrá que comunicarse con la copia que ya hay instalada, indicándole las acciones a tomar.

En el listado mostrado a continuación puede ver que hemos añadido algo de código al gestor de la interrupción, añadiendo un nuevo servicio. En caso de que el registro AL contenga el valor 1, el gestor comprobará si los vectores de interrupción han sido o no modificados desde la instalación, procediendo a restaurar sus valores si procede. En caso de que la desinstalación no sea posible lo indica activando el indicador de acarreo. De esta forma, la parte de instalación del programa que estamos ejecutando puede saber si es posible o no proceder con la liberación de la memoria. En caso afirmativo realiza dos llamadas al servicio 4 9h de la interrupción 2 lh, una con el bloque de entorno y otro con el bloque principal, finalizando así todo el proceso.

```
; INT2F3.ASM
; Esta versión del programa permite
; la desinstalación
;
; Modelo de memoria pequeño
.MODEL Small
; Pueden usarse instrucciones del 386
.386

.STACK 512      ; 512 bytes de pila
```

```
.CODE ; Inicio del segmento de código
; Generar código de configuración
.STARTUP

        Jmp Instalar    ; Saltar a la instalación

; Para preservar la dirección del controlador
; anterior
Antigua2F Dd ?
; Segmento del PSP del programa instalado
SegmentoPSP Dw ?

; Este procedimiento será el que quede
; residente en memoria
GestorServicios Proc

        Cmp AH, 123 ; Comprobar si es para nosotros
        Jne NoLoes

        Cmp AL, 1 ; Orden de desinstalar
; Si no es, procede a la identificación normal
        Jnz Identificación

; Obtener la dirección actual en el vector
        Mov AL, 2Fh
        Mov AH, 35h ; de la interrupción múltiple
        Int. 21h

; Si no coincide con nuestra
        Cmp BX, Offset GestorServicios
        Jne NoSePuede ,• propia dirección es porque
        Mov BX, ES      ; se ha instalado otro programa
        Cmp BX, Seg GestorServicios ; después que este,
; por lo que no es posible llevar a cabo la
; desinstalación
        Jne NoSePuede

; Obtener en DS:DX la antigua dirección
        Mov DX, Word Ptr CS:lAntigua2F1
        Mov DS, Word Ptr CS:[Antigua2F+2]

; Y restituir el vector de interrupción
        Mov AL, 2Fh
        Mov AH, 25h
        Int 21h

; Desactivar el flag de acarreo para indicar
; que todo fue bien
        Clc
        Reti 2 ; volver

NoSePuede:

        .Stc      ; Activar el indicador de acarreo
        Retf 2 ; para indicar que no es posible
```

Identificación:

```
Mov AX, 54321 ; Devolver otro código
; y el segmento del PSP
Mov BX, CS:[SegmcntoPSP]

; Volver eliminando el registro de indicadores
; de la pila
Retf 2
```

NoLoEs:

```
; Saltar al siguiente gestor de la lista
Jmp [CS:Antigua2F]
```

GestorServicios EndP

```
; Este procedimiento se ejecutará tan sólo
; al cargar el programa en memoria, no
; quedando residente
```

Instalar Proc

```
; Preservar el segmento del PSP
Mov CS:[SegmentoPSP], fcs
```

```
Xor CU, CH
; CX contiene la longitud de la linea
; de comandos
Mov CL, ES:[80h]
```

```
; Primer carácter de la linea de comandos
Mov DI, 81h
Mov AL, '/' ; Carácter a buscar
```

```
RepNe Scasb
```

```
; Si no se encontró la barra no hay opciones
Jnz NoHayOpciones
```

```
; Mirar si hay una D detrás de la barra
Cmp Byte Ptr F.S:[DI], 'D'
; en caso contrario ignorar la línea de
; comandos
Jne NoHayOpciones
```

```
; Si se llega a esta etiqueta es porque
; se quiere desinstalar
```

Desinstalar:

```
Mov AH, 123
Int 2Fh ; Comprobar si está instalado
```

```
Cmp AX, 54321
Jne Noinstalado ; No está instalado
```

```
; Preservar el segmento de PSP devuelto
Push BX
```

```
; Indicar a la parte residente que
; restituya el vector
Mov AH, 123
Mov AL, 1
Int 2Fh

; Si no es posible, no continuar
Je NoSePuedeDesinstalar

Pop BX    ; Recuperar el segmento del PSP

Mov DS, BX
; Obtener el segmento de entorno
Mov ES, DS:[2Ch]
Mov AH, 49h ; liberar la memoria que ocupa
Int 21h
Je Fallol ; Indica si hay fallo en liberación

; Liberar el bloque principal del programa
Mov ES, RX
MOV AH, 4 9h
Int 21h
Je Fallo2

; Todo fue bien, el programa se ha desinstalado
Mov DX, Offset Msgü
Jmp Imprimir

Fallol:
; Indicar con un mensaje el error encontrado
Mov DX, Offset Msg5
Jmp Imprimir

Fallo2:

Mov DX, Offset Msg6
Jmp Imprimir

NoSePuedeDesinstalar:

; Desealar el valor que hablamos
; almacenado en la pila
Pop DX

Mov DX, Offset Msg4 ; No se puede desinstalar
Jmp Imprimir

Noinstalado:

; El programa no está instalado actualmente
Mov DX, Offset Msg2
Jmp Imprimir ; terminar

; A partir de aqui tenemos el proceso normal
; en caso de que no se pasen parámetros.
```

NoHayOpciones:

```
; Comprobar si ya está instalado
Mov AH, 123
Int 2Fh
Crap AX, 54321
,- Si es así no permitir la reinstalación
Je Yalnstalado

; Si no está instalado vamos a proceder
; a la instalación

; Obtener la dirección del actual gestor
Mov AL, 2fh
Mov AH, 35h
Int 21h

; Preservar la dirección original
Mov Word Ptr [CS:AnLrgua2F], BX
Mov Word Ptr [CS:Antigua2F+2], ES

; Instalar en el vector
Mov DX, Seg GestorServicios
; la dirección apuntando a nuestro gestor
Mov DS, DX
Mov DX, Offset CestorServicios
Mov AH, 25h
Int 21h

,- Calcular el tamaño a dejar residente
Mov DX, Offset Instalar
Mov CL, 4
Shr DX, CL
Inc DX

Add DX, 16

Mov AX, 3100h ; Salir y quedar residente

Int 21h
```

lalnstalado:

```
Mov DX, Offset Msg
```

Imprimir:

```
Mov AX, Scg Msg ; Obtener la dirección
Mov DS, AX ; del mensaje
Mov AH, 9 ; mostrarlo
Int 21h
Mov AH, 4Ch ; y salir sin instalar
Int 21h
```

Instalar EndP

```
; Mensajes de indicación y error
```

```

Msg Db "El programa ya está instalado$"
Msg2 Db "El programa no está instalado$"
Msg3 Db "El programa ha sido desinstalado$"
Msg4 Db "No es posible desinstalar el programa$" 
Msg5
    Db "Fallo en liberación del bloque de entorno?"
Msg6 Db "Fallo en liberación del bloque principal$"

End

```

En la figura 27.5 puede ver cómo se instala el programa, comprobándose mediante PRINT2F, y cómo se desinstala a continuación. Puede usar el comando mem del DOS para asegurarse de que la memoria ocupada por el residente vuelve a estar libre.

```

Símbolo del sistema - int2f3
D:\Ejemplos\27>print2f
No está instalado
D:\Ejemplos\27>
D:\Ejemplos\27>
D:\Ejemplos\27>int2f3

D:\EJEMPLOS\27>
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>print2f
Si está instalado
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>int2f3 /D
El programa ha sido desinstalado
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>print2f
No está instalado
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>
D:\EJEMPLOS\27>_

```

Figura 27.5. Podemos desinstalar el programa liberando la memoria ocupada.

A vueltas con la pila y el PSP

Cuando un programa se ejecuta desde la línea de comandos, el DOS se ocupa de preparar y establecer el PSP, y el programa usa su propia pila. Sin embargo, cuando la ejecución de un programa está provocada por una interrupción, como es el caso de los programas residentes, no podemos asumir nada de esto. De hecho, si nuestro código no lo tiene en cuenta estaremos usando la pila del programa que se ha interrumpido, y el PSP activo será el de ese programa, y no el nuestro.

Que la pila activa no sea la del programa residente, sino la del programa interrumpido, puede tener consecuencias fatales, ya que si dicha pila no es lo suficientemente grande para mantener los datos depositados en ella lo normal es que el sistema termine

bloqueándose. Imagine que a nuestro código residente lo llama otro programa que dispone de una pila muy pequeña, porque no necesita más, y esta pila se ve desbordada por las necesidades de otro programa para el que no estaba prevista.

El cambio del PSP también tiene su importancia, ya que el hecho de que el programa activo, que se está ejecutando, sea el residente y sin embargo el PSP activo sea el de otro programa, puede confundir a algunas aplicaciones.

Una pila para la parte residente

La pila que definimos al inicio del programa, mediante la directiva `STACK` del ensamblador, se dispone al final del programa y sólo es utilizada por la parte de instalación, ya que al utilizar el servicio 3 lh de la interrupción DOS toda la parte que no va a quedar residente, es decir, el código de instalación, más los datos definidos al final, más la pila, son liberados.

Tendremos, por lo tanto, que definir una pila que quede al final de la parte residente, teniendo en cuenta que la dirección que hemos de tomar como referencia es la del final de la pila, ya que en ésta los datos se van disponiendo desde el final hacia el inicio. Aunque la dirección de esta pila la podemos establecer durante la instalación, será en el momento en que se produzca la interrupción, y se ejecute nuestro gestor, cuando debamos guardar la dirección de la pila que haya en ese momento, que será la del programa interrumpido, y establecer la nuestra. Obviamente, antes de devolver el control deberemos dar el paso contrario, restableciendo la pila anterior.

Para modificar la dirección de la pila tendrá que alterar, como ya sabe, el valor de los registros SS y SP, segmento y desplazamiento, lo que puede hacer manualmente, mediante instrucciones `MOV`, o bien usando la instrucción `LS` en caso de que el procesador sea un 386 o superior, que es lo más habitual.

En cualquier caso, mientras está realizando esta modificación no debe producirse interrupción alguna, ya que se corre el peligro de que la dirección de retorno no se almacene correctamente. Por ello antes de proceder a la modificación deberemos desactivar las interrupciones, activándolas posteriormente.

Cambio del PSP activo

El caso del PSP es similar al de la pila. Deberemos conservar la dirección del PSP del programa interrumpido estableciendo el nuestro, y dar el paso contrario antes de salir. Podemos obtener el segmento del PSP activo mediante el servicio 5 lh de la interrupción 2 lh, que la devolverá en el registro BX. Para modificar el PSP facilitaremos el segmento en el mismo registro, y utilizaremos el servicio 5 Oh de la interrupción 2 lh.

Vamos a servirnos por última vez del programa INT2F para poner en práctica lo que acaba de explicarse, añadiendo a este programa el código necesario para realizar el intercambio de pila y PSP. Para ello hemos añadido al final de la parte residente un espacio de 64 bytes, suficiente por ahora para las necesidades de nuestro programa. En la parte de instalación tomamos el segmento y el desplazamiento del final de la pila, y los

almacenamos en una variable que estará accesible cuando se ejecute la parte residente. Por último hemos añadido al código del gestor el código necesario para preservar la pila del programa interrumpido, fijar la suya, intercambiar el PSP y deshacer todos estos cambios al salir.

Para obtener esta cuarta versión tome como base la anterior, INT2F3, modificándola donde sea preciso. Añada tras la declaración de los campos Antigua2F y SegmentoPSP el código siguiente:

```
; Para almacenar la dirección de la pila del
; otro programa
Pi 1 aAnterior Dd ?
; Dirección de nuestra pila
PilaResidente Dd ?

; Para almacenar el segmento de PSP anterior
SegmentoPSPAnterior Dw ?
```

A continuación se encuentra el código del gestor del servicio que, tras los cambios, quedará como se muestra a continuación. Observe cómo se utilizan los campos anteriores para guardar las direcciones de pila y PSP y establecer los propios, restableciéndose al final. Fíjese también en cómo se ha definido un espacio para pila al final del gestor, ése será el final del área que quede residente.

GestorServicios Proc

```
Cmp AH, 123 ; Comprobar si es para nosotros
Jne NoLoes

; Intercambiar la pila y el PSP

; Desactivar interrupciones mientras
; cambiamos la pila
Cli

; Preservar la dirección de
Mov Word Ptr CS:[PilaAnterior], SP
; la pila del otro programa
Mov Word Ptr CS: [PilaAnterior+2], SS

; Fijar nuestra propia pila
Lss SP, CS:[PilaResidente]

Push AX ; Preservar AL

Mov AH, 51h ; Obtener segmento del PSP activo
Int 21h
Mov CS: ISegmentoPSPAnteriorJ, BX ; Guardarlo

Mov BX, CS:[SegmentoPSP] ; Fijar nuestro PSP
Mov AH, 50h
Int 21h
```

Pop AX

```
Cmp AL, 1 ; Orden de desinstalar
; Si no es, procede a la identificación normal
Jnz Salir
```

```
; Obtener la dirección actual en el vector
Mov AL, 2Fh
Mov AH, 35h ; de la interrupción múltiple
Int 21h
```

```
, " Si no coincide con nuestra
Cmp BX, Offset CestorServicioic
Jne NoSePuede ; propia dirección es porque
Mov BX, ES ; se ha instalado otro programa
Cmp nx, Seg GestorServicios ; después que este,
; por lo que no es posible llevar a cabo la
; desinstalación
Jne NoSePuede
```

```
; Obtener en DS:DX la anLigua dirección
Mov DX, Word Ptr CS:[Antigua2F]
Mov DS, Word Ptr CS:[Antigua2F+2]
; Y restituir el vector de interrupción
Mov AL, 2Fh
Mov AH, 2Sh
Int 21h
```

```
; Desactivar el flag de acarreo para indicar
; que todo fue bien
Clc
```

Jmp Salir ; volver

NoSePuede:

Stc ; Activar el indicador de acarreo

Salir:

```
; Restablecer el PSP
Mov BX, CS:[SegmentoPSPAnterior]
Mov AH, b0h
Int 21h
```

```
Mov AX, 54321 ; Devolver otro código
; y el segmento del PSP
Mov BX, CS:[SegmentoPSP]
```

```
,- Restablecer La pila anterior
Lss SP, CS:[PilaAnterior]
Sti ; Activar las interrupciones
```

```
; Volver eliminando el registro de
; indicadores de la pila
Retf 2
```

```
NoLoEs:  
; Saltar al siguiente gestor de la lista  
Jmp [CS:Antigua2F]
```

GestorServicios EndP

```
; Reservamos 64 bytes para la pila  
EspacioDePila Dh 64 Dup(?)  
; dentro de la parte que va a quedar residente  
EtiquetaPila:
```

El último cambio afecta al código de instalación de la parte residente. Justo detrás de la sentencia Je Yalnslalado, cuando se procede a efectuar el cambio del vector, introduciremos las dos sentencias siguientes:

```
; Preparar la pila del residente  
Mov Word Ptr CS:[PilaRcsidente],Offset EtiquetaPila  
Mov Word Ptr CS:[PilaResidente+2], Seg EtiquetaPila
```

Tal y como puede observar, se guarda en el campo PilaResidente la dirección correspondiente al final de la pila, a fin de que la parte residente pueda utilizarla posteriormente.

Si, tras haberlo ensamblado, prueba el funcionamiento de esta nueva versión del programa, verá que el funcionamiento aparente es idéntico al de INT2F3, ya que los cambios efectuados no podemos verlos a simple vista, a pesar de lo cual hacen que el programa sea más seguro y no afecte negativamente a otros.

Estado del DOS y la BIOS

Un programa residente se activa, por regla general, sin conocimiento del DOS, interrumpiendo al programa que en ese momento tiene el control del sistema. Por ello se deben tener en cuenta muchos aspectos que, habitualmente, son pasados por alto en programas no residentes, ya que de ellos se ocupa el propio sistema operativo. En el punto anterior se han visto algunos de esos aspectos, como el intercambio del PSP del programa interrumpido por el del programa residente, o la necesidad de establecer una pila propia evitando utilizar la del programa interrumpido, que puede no tener la capacidad suficiente para nuestras necesidades.

A pesar de lo que muchos programadores creen, desde el código de un programa residente sí que es posible utilizar los servicios del DOS, así como las interrupciones BIOS que sean necesarias.

Para poder hacer eso, sin embargo, hay que tener en cuenta una serie de indicadores, que unas veces vienen dados por el propio DOS y otras tendremos que crear nosotros mismos.

Partiendo de lo anterior, hay muy pocas acciones que no puedan llevarse a cabo en un programa residente, siendo posible el trabajo con archivos en disco, la entrada de datos por teclado, acceso a la pantalla, etc.

La reentrada y el DOS

El DOS utiliza internamente tres pilas que se usan según el servicio que se esté ejecutando en cada momento. En caso de que se esté ejecutando un servicio de entrada o salida de caracteres, desde el servicio OOh hasta OCh de la interrupción 21h, la pila en uso será la primera. Si se está ejecutando cualquier otro de los servicios DOS, se utilizará la segunda pila, y la tercera se usará sólo en caso de que se esté procesando un error crítico. ¿Qué interés práctico tiene esto para nosotros?, es la clave que nos permitirá utilizar los servicios DOS que nos interesen sin bloquear el sistema.

Mientras el DOS está utilizando una determinada pila, pongamos por caso la primera, el programa residente que se ha activado, interrumpiendo el proceso que estaba en curso, no debe llamar a ningún servicio que utilice esa misma pila, porque ello causaría la sobreescritura y, por lo tanto, pérdida de los datos contenidos en esta pila, principalmente la dirección de retorno desde ese servicio al programa que se ha interrumpido. Consecuentemente, para utilizar un determinado servicio del DOS es de vital importancia conocer antes si la pila correspondiente está en uso y, en caso afirmativo, no se deberá realizar la activación del programa residente, esperando hasta que sea posible esa llamada.

Los indicadores InDOS y ErrorMode

Mediante el servicio 34h de la interrupción DOS, 21h, obtendremos en la pareja de registros ES : BX la dirección del indicador inDos, que ocupa un byte. En caso de que éste indicador esté a uno, significará que el DOS está ejecutando un servicio de la interrupción 21h que usa la segunda pila y, por lo tanto, el programa residente no deberá llamar a ninguno de estos servicios, aunque sí puede utilizar los correspondientes a entrada y salida de caracteres, desde el servicio OOh hasta el OCh.

En el byte anterior al indicador inDos, es decir, la dirección devuelta en ES : BX menos uno, encontraremos el indicador ErrorMode, mediante el cual podremos saber si el DOS se encuentra actualmente procesando un error crítico, caso éste en que se encontrará a uno. Siempre que el indicador ErrorMode se encuentre activado, el programa residente debe abstenerse de realizar llamada alguna al DOS, incluyendo los servicios de tratamiento de caracteres.

La dirección de los dos indicadores anteriores debe ser obtenida durante la instalación del programa residente, y no en el momento en que éste se activa mediante una interrupción ya que en ese momento tendría que utilizar el servicio 34h de la interrupción DOS sin saber si puede o no hacerlo, ya que el DOS puede estar en ese momento ocupado. De acuerdo con esto, la obtención de la dirección será un paso más del proceso de instalación, de tal forma que la parte que quedará residente tan sólo tenga que ocuparse de examinar los indicadores.

Partiendo del conocimiento de los indicadores InDos y ErrorMode podría parecermos que tenemos todo el problema solucionado, y que las llamadas a servicios de la interrupción 21h desde nuestro programa residente no serán ya ningún problema. Sin embargo aún queda algo más, que entenderemos mucho mejor a partir de un ejemplo.

A continuación tiene el código de un programa cuya finalidad es quedar residente y mostrar de forma continua, en la esquina superior derecha de la pantalla, el valor actual de los indicadores InDos y ErrorMode.

El programa en sí no es un modelo perfecto de programa residente, aunque contempla la mayoría de los aspectos que ya conocemos como no permitir la reinstalación, permitir la desinstalación, etc.

```
; INDICADO.ASM

; Muestra el estado de los indicadores
; InDos y ErrorMode

; Modelo de memoria pequeño
.MODEL Small
; Pueden usarse instrucciones del 386
.386

.STACK 512      ; 512 bytes de pila

.CODE ; Inicio del segmento de código
; Generar código de configuración
.STARTUP

Jmp Instalar    ; Saltar a la instalación

; Para preservar la dirección del controlador
; anterior
Antigua2F Dd ?
; Para preservar la dirección
; del controlador anterior
Antigúale Dd ?

; Segmento del PSP del programa instalado
SegmentoPSP Dw ?

; Para almacenar la dirección de la pila del
; otro programa
PilaAnterior Dd ?
; Dirección de nuestra pila
PilaResidente Dd ?

; Para almacenar el segmento de PSP anterior
SegmentoPSPAnterior Dw ?

InDos Dd ?      ; Dirección del indicador InDos

; Este procedimiento será el que quede residente
; en memoria
GestorServicios Proc

Cmp AH, 123 ; Comprobar si es para nosotros
Jne NoLoes
```

```

; Intercambiar la pila y el PSP

Cli ; Desactivar interrupciones mientras
; cambiamos la pila

; Preservar la dirección de la pila
; del otro programa
Mov Word Ptr CS:[PilaAnterior], SP
Mov Word Ptr CS:[PilaAnterior+2], SS

; Fijar nuestra propia pila
Lss SP, CS: [PilaResidente]

Push AX ; Preservar AL

Mov AH, blh ; Obtener segmento del PSP activo
Int 21h
Mov CS:[SegmentoPSPAnterior], BX ; Guardarlo

Mov BX, CS:[SegmentoPSP] ; Fijar nuestro PSP
Mov AH, 50h
Int 21h

Pop AX ; Recuperar AL

Cmp AL, 1 ; Orden de desinstalar
Jnz Salir ; Si no, procede de forma normal

; Obtener la dirección actual en el vector
Mov AL, 2FK
Mov AH, 35h ; de la interrupción múltiple
Int 21h

; Si no coincide con nuestra propia
Cmp BX, Offset GestorServicios
Jne NoSePuede ; dirección es porque se ha
Mov BX, ES ; instalado otro programa
; después que este, por lo que no es posible
; llevar a cabo la desinstalación
Cmp BX, Scq GestorServicios
Jne NoSePuede

; Comprobar ahora el gestor de la ICh
Mov AL, ICh
Mov A1l, 35h
Int 21h

; Si la dirección ha cambiado
Cmp BX, Offset NuevalC
Jne NoSepuede
Mov BX, ES
Cmp BX, Seg NuevalC ; no es posible desinstalar
Jne NoSePuede

; Obtener en DS:DX la antigua dirección
Mov DX, Wórd Ptr CS:[Antigua2F]

```

```

Mov DS, Word Ptr CS:[Antigua2F+2]
; Y restituir el vector de interrupción
Mov AL, 2Fh
Mov AH, 25h
Int 21h

; Restituir el antiguo gestor
Mov DX, Word Ptr CS:[Antigüale]
Mov DS, Word Ptr CS:[AntigualC+2]
Mov AL, ICh
Mov AH, 25h
Int 21h

Cíe ; Desactivar el tlag de acarreo para
Jmp Salir ; indicar que todo fue bien y volver

NoSePuede:

Stc      ; Activar el indicador de acarreo

Salir:
; Restablecer el PSP
Mov BX, CS:[SegmentoPSPAnterior]
Mov AH, 50h
Int 21h

Mov AX, 54321 ; Devolver otro código
Mov BX, CS:[SegmentoPSP] ; y el segmento del PSP

; Restablecer la pila anterior
Lss SP, CS:[PilaAnterior]

Sti ; Activar las interrupciones

Retf 2 ; Volver eliminando el registro de
; indicadores de la pila

NoLoEs:

; Saltar al siguiente gestor de la lista
Jmp [CS:Antigua2F]

GestorServicios EndP

; Gestor para la interrupción ICh

NuevalC Proc

CU ; Desactivar las interrupciones

Push AX ;' Guardar en la pila los registros
Mov AX, ES ; que se van a modificar
Push AX
Push DI

```

```

; Cargar en ES:DI la dirección del
;- indicador InDos
Mov ES, Word Ptr CS: [InDor, + 2]
Mov DI, Word Ptr CS:[InDos]
    ; Obtener en AL dicho indicador
Mov AL, Byte Ptr ES:[DI]
    ; y en AH el indicador ErrorMode
Mov AH, Byte Ptr ES:[DI-1]

Push AX ; Conservar los valores

Mov AX, 0B800h ; Cargar en ES:DI la dirección
Mov ES, AX ; de memoria de video para escribir
Mov DI, 156 ; en ella

Pop AX      ; Recuperar el valor de AX
Push AX

; Convertir a dígito el contenido de AL
Add AL, '0'
Mov AH, 70h ; Negro sobre blanco
St03w        ; escribir el dígito

Pop AX ; Recuperar ahora el valor
Mov AL, AH ; del registro AH
; para escribirlo también en pantalla
Add AL, '0'
Mov AH, 70h
Stosw

Pop DI      ; Recuperar los valores originales
Pop AX      ; de los registros modificados
Mov ES, AX
Pop AX

Sti      ; Activar de nuevo las interrupciones

; Pasar el control al controlador anterior
Jmp [CS:AntigualC]
NuevalC EndP

; Reservamos 64 bytes para la pila
; dentro de la parte que va a quedar residente
EspacioDePila Db 64 Dup(?)
EtiquetaPila:

; Este procedimiento se ejecutará tan sólo
; al cargar el programa en memoria, no
; quedando residente

Instalar Proc

; Preservar el segmento del PSP
Mov CS:[SegmentoPSP], ES

```

```
; CX contiene la longitud de la línea de comandos
Xor CH, CH
Mov CL, ES:[80h]

; Primer carácter de la línea de comandos
Mov DI, 81h
Mov AL, '/' ; Carácter a buscar

RepNe Scasb

Jnz NoHayOpciones ; Si no se encontró la barra
; no hay opciones

; Mirar si hay una D detrás de la barra
Cmp Byte ftr ES:[DI], 'U'

; en caso contrario ignorar la línea de comandos
Jne NoHayOpciones

Desinstalar: ; Si se llega a esta etiqueta es
; porque se quiere desinstalar

Mov AH, 123
Int 2Fh ; Comprobar si está instalado

Cmp AX, 54321
Jne NoInstalado ; No está instalado

Push BX ; Preservar el segmento de PSP devuelto

; Indicar a la parte residente que restituya
; el vector
Mov AH, 123
Mov AL, 1
Int 2Fh

; Si no es posible, no continuar
Je NoSePuedeDesinstalar

Pop BX ; Recuperar el segmento del PSP

Mov DS, BX
Mov ES, DS:[2Ch] ; Obtener el segmento de entorno
Mov AH, 49h ; liberar la memoria que ocupa

Int 21h
Je Fallo1 ; Indica si hay un fallo en liberación

; Liberar el bloque principal del programa
Mov ES, BX
Mov AH, 49h
Int 21h
Je Fallo2

Mov DX, Offset Msg3 ; Todo fue bien, el
Jmp Imprimir ; programa se ha desinstalado
```

Fallol:

```
; Indicar con un mensaje el error encontrado
Mov DX, Offset Msg5
Jmp Imprimir
```

Fallo2:

```
Mov DX, Offset Msg6
Jmp Imprimir
```

NoSePuedeDesinstalar:

```
Pop DX ; Descartar el valor que hablamos
; almacenado en la pila
Mov DX, offset Msg4 ; No se puede desinstalar
Jmp Imprimir
```

NoInstalado:

```
; El programa no está instalado actualmente
Mov DX, Offset Msg2
Jmp Imprimir ; terminar
```

```
; A partir de aquí tenemos el proceso normal
; en caso de que no se pasen parámetros.
```

NoHayOpciones:

```
Mov AH, 123 ; Comprobar si ya está instalado
Int 2Fh
Cmp AX, 54321
; Si es así no permitir la reinstalación
Je YNoInstalado
```

```
; Si no está instalado vamos a proceder
; a la instalación
```

```
; Preparar la pila del residente
Mov Word Ptr CS:[PilaResidente],Offset EtiquetaPila
Mov Word Ptr CS:[PilaResidente+2], Seg EtiquetaPila
```

```
; Obtener la dirección del actual gestor
Mov AL, 2Fh
Mov AH, 35h
Int 21h
```

```
; Preservar la dirección original
Mov Word Ptr CS:[Antigua2F], RX
Mov Word Ptr CS:[Antigua2F+2], ES
```

```
Mov AL, ICh • ; Obtener la dirección
Mov AH, 35h ; del actual gestor de la
Int 21h ; interrupción ICh
```

```

; Guardar la antigua dirección
Mov Word Ptr CS:[Antigúale], BX
Mov Word Ptr CS : [AntigualC-t 2] , ES

Mov AH, 34h      ; Obtener la dirección del InDos
Int 21h

Mov Word Ptr CS:[InDos], BX      ; y preservarla
Mov Word Ptr CS:[InDos+2], ES

; Instalar en el vector la dirección
; apuntando a nuestro gestor
Mov DX, Seg GestorServicios
Mov DS, DX
Mov DX, Offset GestorServicios
Mov AL, 2Fh
Mov AH, 2ih
Int 21h

; Instalar el nuevo gestor para la ICh
Mov DX, Seg NuevalC
Mov DS, DX
Mov DX, Offset NuevalC
Mov AH, 25h
Mov AL, ICh
Int 21h

; Calcular el tamaño a dejar residente
Mov DX, Offset Instalar
Mov CL, 4
Shr DX, CL
Inc DX
Add DX, 16

Mov AX, 3100h ; Salir y quedar residente

Int 21h

```

Yalnstalado:

```
Mov DX, Offset Msg
```

Imprimir:

```

; Obtener la dirección del mensaje
Mov AX, Seg Msg
Mov DS, AX
Mov AH, 9      ; mostrarlo
Int 21h

Mov AH, 4Ch ; y salir sin instalar
Int 21h

```

Instalar EndP

```
; Mensajes de indicación y error
```

```

Msg Db "El programa ya está instalado$"
Msg2 Db "El programa no está instalado$"
Msg3 Db "El programa ha sido desinstalado$"
Msg'1 Db "No es posible dcsinstalar el programa$"
Msg5
Db "Fallo en liberación del bloque de entornoS"
Msg6 Db "Fallo en liberación del bloque principáis"

End

```

Como podrá observar, en la parte de instalación del programa se modifica el contenido del vector de la interrupción ICh, haciendo que apunte a un procedimiento nuestro. Esta interrupción se genera de forma continua, con una frecuencia de 18,2 veces por segundo, por lo que es apropiada para mantener en pantalla una información constantemente actualizada. Además también se utiliza el servicio 34h de la interrupción DOS para obtener la dirección del indicador InDos, que se conserva para ser utilizada posteriormente por la parte residente.

El nuevo gestor para la interrupción 1ch es muy simple y rápido, ya que lo único que hace es obtener el valor actual de la dirección de memoria almacenada en InDos y ErrorMode, mostrándola en la esquina superior derecha de la pantalla en vídeo inverso. Para conseguir esta visualización se ha utilizado el acceso directo a memoria, sin tener en cuenta el tipo de adaptador ni el modo en que éste se encuentra, se ha asumido que el programa se va a ejecutar desde la línea de comandos en un modo de texto normal en una VGA.

Este tipo de suposiciones son las que no debe realizar un buen programa residente, pero en este caso de lo que se trata es de realizar un programa lo más simple posible que nos permita inspeccionar el estado de los indicadores citados y, por ello, se ha optado por este método de trabajo con el fin de no hacer más extenso el código.

En la parte que se ocupa de la desinstalación del programa puede observar que ahora hay un elemento más a tener en cuenta, y es que si el vector de la interrupción 1 Ch ha sido modificado por otro programa no será posible llevar a cabo la desinstalación.

Al ejecutar el programa por primera vez se instalará y quedará residente, apareciendo de forma inmediata en pantalla el valor de los indicadores InDos y ErrorMode. Esta visualización continua en pantalla aún cuando inicie otros programas, como puede ser el editor del DOS o el programa ScanDisk. En la figura 27.6 puede ver cómo en un determinado momento de la ejecución del programa ScanDisk los dos indicadores toman el valor 0, lo que quiere decir que en ese momento podríamos utilizar sin problemas cualquiera de los servicios de la interrupción 21h.

En la figura 27.7 se muestra el caso contrario, los dos indicadores se encuentran a 1 al estar el DOS procesando un error crítico, concretamente se ha intentado acceder a una unidad en la que no había disco insertado. En esta situación el programa no debe realizar ningún tipo de llamada al DOS.

Si ejecuta múltiples aplicaciones teniendo instalado este programa de ejemplo, podrá comprobar que el indicador inDOS a veces toma otros valores, además del 0 y el 1. Es habitual ver aparecer de vez en cuando un 2, lo que indica que desde un servicio DOS se ha llamado a otro servicio DOS, caso éste en que el indicador InDOS es incrementado con cada llamada y reducido a la salida.



Figura 27.6. Momento en el que los dos indicadores están a cero.

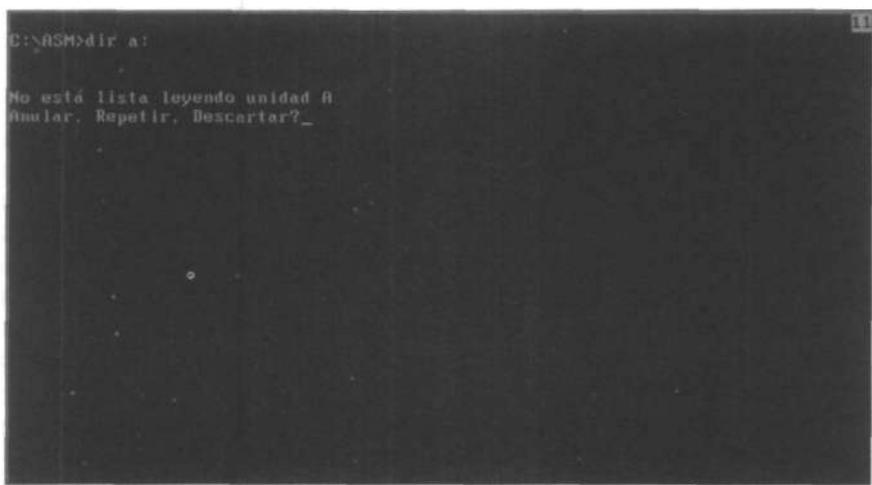


Figura 27.7. Durante el tratamiento de errores críticos ambos indicadores están a uno.

La interrupción 28h

Como habrá podido observar al ejecutar el programa anterior, mientras el DOS está esperando una entrada por teclado, por ejemplo en la línea de comandos, el indicador InDOS siempre está a 1. Este es el problema que queda por resolver que se mencionaba antes, mostrado claramente por este programa. El DOS mientras está esperando una entrada por teclado, momento en el cual un programa residente podría sin problemas activarse y usar los servicios del DOS a excepción de los de entrada y salida de caracteres, y por ello activa el indicador InDOS en previsión de que una orden del usuario haga

necesaria la ejecución de un servicio de la interrupción 21h. Por lo tanto, un programa que quedase residente y observase el indicador inDOS, antes de activarse, podría quedar parado indefinidamente mientras no se ejecute algún programa que provoque que dicho indicador se ponga a cero.

La solución a este problema la encontraremos en la interrupción 28h, una interrupción software que es generada por el DOS, y otros programas, mientras está esperando una acción del usuario. Nuestro programa residente deberá modificar el vector correspondiente a esta interrupción, desviándolo a un gestor propio que debe encadenarse a ella. De esta forma, aunque el indicador InDOS esté a uno el programa sabrá que si se produce una interrupción 28h la activación se puede realizar de forma segura, y es posible utilizar cualquiera de los servicios de la interrupción DOS a excepción de los de entrada y salida de caracteres, que son los que está utilizando el propio DOS para solicitar la entrada del usuario.

Nuestro programa no deberá asumir que si se genera una interrupción 28h el sistema está totalmente libre, y por lo tanto es posible la activación. Siempre será necesario comprobar que el indicador ErrorMode no esté a uno, caso éste en que no será posible la llamada a ningún servicio DOS, aún cuando se esté generando la citada interrupción.

Estructura del programa residente

Como estamos comprobando, la activación de un programa residente que pretende hacer uso de los servicios de la interrupción DOS se va complicando poco a poco, a causa de la cantidad de elementos que hay que controlar. Supongamos que nuestro programa residente se va a activar mediante la pulsación de una cierta combinación de teclas, para lo cual hemos interceptado la interrupción 9h generada por el teclado. En el momento en que delectamos la pulsación de activación no podemos asumir que todo el sistema está a nuestra disposición, por lo que en lugar de ejecutar la parte residente lo que haremos será activar un indicador, mediante el cual sabremos que el usuario ha solicitado dicha ejecución.

Este indicador deberá ser consultado frecuentemente, por ejemplo en el gestor del vector correspondiente a la interrupción ICh, conjuntamente con la comprobación de los indicadores InDOS y ErrorMode, de tal forma que si nuestro indicador está a uno y los otros dos están a cero, la activación se puede realizar. Por último también tendremos que comprobar el indicador de activación en el gestor de la interrupción 28h, verificando además que no se está procesando un error crítico. Según esto, el código asociado a cada uno de los vectores de interrupción sería:

- Int 9h: Si se produce la pulsación de activación, hay que dar el valor 1 al indicador Activar.
- Int ICh: Si Activar es 1 hay que comprobar el valor de InDOS y ErrorMode. En caso de que ambos estén a cero se puede activar la parte residente.
- Int 28h: Si Activar es 1 hay que comprobar el valor de ErrorMode, activando la parte residente si está a cero.

Por lo tanto la parte residente se podrá activar en dos casos: cuando los indicadores InDOS y ErrorMode estén a cero, porque el DOS no esté ocupado ejecutando algún servicio de la interrupción 21h, o bien cuando se genere una interrupción 28h, lo que indicará que el programa que se está ejecutando está libre.

Los servicios de entrada y salida de caracteres

Los servicios OOh a OCh de la interrupción DOS facilitan funciones fácilmente realizables mediante otros servicios DOS o bien mediante interrupciones BIOS, por lo que en la mayoría de las ocasiones no será necesario utilizarlos. Sabemos que estos servicios los podemos utilizar siempre que el DOS no esté ya ejecutando uno de ellos, ya que en ese caso sobrescribiríamos la pila.

Cuando el DOS se encuentra en la línea de comandos, punto desde el cual se puede activar un programa residente, el indicador InDOS está a 1, pero realmente el DOS está utilizando uno de los servicios de entrada/salida de caracteres para solicitar una orden.

Por lo tanto, un programa residente que se active porque el indicador InDOS está a cero no encuentra el sistema en las mismas condiciones que ese mismo programa activado por la interrupción 28h, caso éste en que sabemos a ciencia cierta que se está ejecutando un servicio de entrada/salida de caracteres.

La tercera pila del DOS, la que se utiliza en un error crítico, estará siempre sin utilizar al activar el programa residente, ya que una de las condiciones para que esta activación se produzca es precisamente que el indicador ErrorMode esté a cero. Puesto que los servicios OOh a OCh de la interrupción 21h se basan en este indicador para utilizar una pila u otra, nuestro programa puede activar el indicador ErrorMode antes de llamar a uno de esos servicios, consiguiendo así utilizarlos aún cuando el DOS ya está ejecutando uno de ellos, pero mientras nuestro programa almacena su dirección en la pila de gestión de errores críticos, el DOS almacena la suya en la habitual pila de los servicios de entrada y salida de caracteres. De esta forma podemos eliminar una de las pocas limitaciones que nos quedan, en cuanto a qué servicios del DOS es posible usar desde un programa residente.

Las interrupciones BIOS

De forma similar a lo que ocurre con los servicios de la interrupción DOS, las interrupciones BIOS tampoco son reentrantes, lo que quiere decir que si nuestro programa residente necesita hacer uso de una cierta interrupción BIOS, por ejemplo de pantalla o disco, no deberá activarse hasta que no exista ningún servicio de estas interrupciones en curso.

A diferencia de lo que ocurre con los servicios del DOS, que mantienen un indicador mediante el cual nosotros podemos saber si está en curso la ejecución de un servicio, las interrupciones BIOS no mantienen ningún tipo de indicador, por lo que inicialmente no disponemos de un método ya preparado con el que poder determinar si podemos o no llamar a una cierta interrupción.

La solución a este problema pasa por instalar un gestor para cada una de las interrupciones que pretendamos utilizar en nuestro programa, de tal forma que a la entrada activemos un indicador, a continuación llamaremos a la dirección que inicialmente existía en el vector, y a la vuelta desactivemos el indicador. Podemos utilizar un indicador para cada una de las interrupciones, o bien uno conjunto, que se incremente con cada llamada y se reduzca a cada salida, de forma similar a como el DOS utiliza el indicador InDOS. Al interceptar una interrupción de pantalla, disco, etc., hemos de tener en cuenta que esas llamadas envían una serie de parámetros en determinados registros, que deben ser siempre conservados. Es decir, debemos tener siempre la precaución de que nuestro código no modifique registro alguno, a no ser que así lo deseemos para conseguir una cierta acción.

Tiempo de interrupción de un residente

Uno de los factores determinantes del funcionamiento de un programa residente es el tiempo que utiliza para realizar su función que, por regla general, debe ser lo más corto posible, evitando así interferir con la ejecución de otros programas. Particularmente hay que evitar que el programa pueda ser activado una segunda vez antes de terminar una primera ejecución, lo que ocurre a veces si el programa se activa con la interrupción de reloj, que es llamada con la frecuencia citada anteriormente, y en su proceso de ejecución emplea más tiempo del que tarda en llegar una segunda interrupción.

Según qué interrupciones sean las que den paso a nuestro programa, a veces podemos optar por inhibirlas durante la ejecución del código residente, mediante la orden cli, reactivándolas antes de salir, con sti. Sin embargo, y como regla general, un programa debe desactivar las interrupciones el menor tiempo posible.

Otra opción, mucho más eficiente y lógica, consiste en utilizar un indicador que se active cuando se inicia la ejecución de la parte residente, de tal forma que el gestor o gestores que se encargan de la activación puedan comprobar dicho indicador, evitando así la reactivación. De esta forma, la estructura básica de un residente se completaría con los pasos siguientes:

- Int 9h: Si se produce la pulsación de activación, hay que dar el valor 1 al indicador Activar.
- Int 1Ch: Si Activado es 0 y Activar es 1 hay que comprobar el valor de InDOS y ErrorModc. En caso de que ambos estén a cero se puede activar la parte residente, dando el valor 1 a Activado, y el valor 0 a Activar.
- Int 28h: Si Activado es 0 y Activar es 1 hay que comprobar el valor de FileMode, activando la parte residente si está a cero, dando el valor 1 a Activado y el valor 0 a Activar.

A pesar de estas precauciones, el tiempo de ejecución de un programa residente puede afectar a otros servicios y otros programas. Supongamos que nuestro código no hace uso de ningún servicio de disco, por lo que en principio no parecería necesario controlar la

correspondiente interrupción. Sin embargo, si nuestro programa es activado en el momento en que está en curso una operación de disco, interrumpiéndola, y el tiempo de ejecución es demasiado prolongado, podemos provocar un error en el funcionamiento del programa interrumpido e incluso en el sistema, si era éste el que estaba accediendo a disco para, por ejemplo, alojar un programa en memoria. Con el fin de evitar estos problemas nos serviremos una vez más de un indicador, en este caso asociado a la interrupción 13h.

Por último, nos podemos encontrar el caso de que nuestro programa residente solicite alguna información al usuario, como puede ser un nombre de archivo para guardar una información. Mientras el usuario no introduzca dicha información nuestro programa estará esperando y, por tanto, el programa interrumpido se mantendrá en ese estado mucho más tiempo. Es más, otros programas residentes y procesos de ejecución en segundo plano pueden quedar detenidos, ya que mientras que nuestro programa esté usando un servicio DOS para solicitar un dato el indicador InDOS estará activado. Para evitar esta situación podemos comportarnos como lo hace el DOS, llamando a la interrupción 28h mientras estamos esperando la introducción de datos por parte del usuario, de tal forma que otros programas que se guíen por los mismos indicadores que el nuestro puedan continuar con su ejecución.

Estado de otros elementos del sistema

Cuando un programa residente se activa, tras haber realizado todas las comprobaciones pertinentes para hacerlo de una forma segura, tiene que cerciorarse de que ninguna interrupción, externa o provocada por un error, devolverá accidentalmente el control al programa que se interrumpió o al DOS, para lo cual será necesario modificar algunos vectores de interrupción adicionales.

Anteriormente vimos cómo en el momento en que un programa residente se activaba era necesario establecer nuestra propia pila, para no utilizar la del programa interrumpido, y nuestro propio PSP. Además de estas dos estructuras de datos, existen otros elementos que debemos tener en cuenta, como el área de transferencia a disco, que nuestro programa utilizará en caso de que necesitemos realizar operaciones con archivos, o ciertos vectores de interrupción, como el de gestión de error crítico o tratamiento de pulsación de **Control-ínter**, que pueden estar apuntando a otra aplicación o al propio DOS.

Intercambio de la DTA

Los archivos que vaya a utilizar la parte residente de nuestro programa pueden ser abiertos bien durante el proceso de instalación, guardando los manejadores, o bien en el momento en que dicha parte residente sea activada. El primer caso tiene la ventaja de que, cuando se produce la activación, el archivo ya está abierto y, por lo tanto, se puede trabajar directamente sobre él sin necesidad de emplear más tiempo en el proceso de apertura. Sin embargo, esto no es siempre posible puesto que puede ocurrir que el

nombre y camino del archivo a tratar no se conozcan en el momento de la instalación, siendo facilitados posteriormente por el usuario del programa. En este caso es obvio que la apertura se habrá de realizar en la parte residente del programa.

Todas las operaciones con archivos que se realizan mediante los servicios de la interrupción 2 lh, a excepción de las obsoletas de entrada y salida de caracteres, utilizan la DTA (*Disk Transfer Área*) como espacio de trabajo. Inicialmente la DTA se halla situada dentro del propio PSP del programa, pero éste es libre de modificar su posición y longitud, especialmente si va a trabajar con archivos en los cuales se van a tratar bloques de más de 128 bytes. Por lo tanto, no podemos asumir que intercambiando el PSP del programa interrumpido por el del residente ya hemos cambiado también la DTA.

Aunque es posible utilizar la DTA del programa que hemos interrumpido para llevar a cabo nuestras operaciones con archivos, lo más aconsejable es que realicemos una operación de intercambio y fijemos nuestra propia DTA. Esto, que puede ser opcional si la apertura de los archivos se realiza en la parte residente, es condición indispensable en caso de que dicha apertura se efectúe durante el proceso de instalación.

Cuando se ejecuta desde la línea de comandos el programa residente, y éste realiza todas las comprobaciones y operaciones necesarias para instalarse en memoria, la DTA activa es la de dicho programa, y como parte del PSP quedará también residente. Si en ese proceso de instalación abrimos cualquier archivo que nos vaya a ser necesario, la información correspondiente se almacenará en la DTA activa en ese momento, que es la del propio programa residente. Si posteriormente, en el momento en que la parte que ha quedado residente se activa, queremos realizar cualquier operación sobre el archivo que nosotros suponemos abierto, hemos de tener en cuenta que la DTA activa es la del programa que hemos interrumpido, en la que lógicamente no se encontrará la información adecuada para poder llevar a cabo el trabajo. En el mejor de los casos puede ocurrir que operemos sobre uno de los archivos del programa interrumpido, provocando consiguientemente fallos en dicho programa.

En caso de que el archivo sobre el que la parte residente va a trabajar sea abierto durante la activación, lo normal es que no encontremos problema alguno a no ser que en la DTA del programa interrumpido no haya espacio suficiente para la operación. Por esto lo más lógico, y correcto, es que utilicemos nuestra propia DTA en todos los casos.

Para establecer la dirección de nuestra DTA usaremos el servicio 1Ah de la interrupción 21h, facilitando en la pareja de registros DS : DX la dirección completa, segmento y desplazamiento. Obviamente, antes de realizar esta operación deberemos guardar la actual dirección de DTA, la del programa interrumpido, con el fin de restablecerla antes de devolver el control. Para ello utilizaremos el servicio 2Fh de la interrupción 2 lh (no confunda este servicio del DOS con la interrupción 2Fh), que nos devolverá en la pareja de registros ES : BX la dirección completa de la DTA actual.

En caso de que nuestro programa residente, tras su activación por parte del usuario, pueda modificar el directorio actual de trabajo, e incluso la unidad de disco activa en ese momento, siempre deberemos restituir el directorio y unidad originales antes de devolver el control. Imagine el trastorno que puede causarle al programa interrumpido si éste asumía en ese momento que se encontraba en una determinada unidad y directorio, y tras ser interrumpido, de lo cual el programa no tiene conocimiento, se encuentra con un directorio o unidad distintos.

Gestión de errores críticos

Ciertas operaciones, en particular las que tienen que ver con la entrada y salida de información a disco, pueden encontrarse con errores de difícil resolución, a los que se llama habitualmente errores críticos. Imagine, por ejemplo, que va a abrir un archivo en un disco flexible, que suponemos se encuentra en la unidad A:, y al intentar hacerlo dicha unidad no contiene el disco, momento en el cual aparece el conocido mensaje Anular/Reintentar/Ignorar.

Si estando nuestro programa residente activado se produce un error de este tipo, y aparece el mensaje anterior, el usuario puede optar por anular la operación. Esto causaría que el DOS intentase cancelar la operación del programa residente, devolviendo el control al propio DOS, posiblemente dejando vectores de interrupción asignados a la parte residente. Esto finalmente puede provocar una caída del sistema.

Cuando el DOS detecta una condición de error crítico, como el descrito, genera una interrupción 24h, enviando distintos datos acerca del error en varios registros. Por defecto el vector de interrupción 24h está dirigido al propio DOS, a una rutina que muestra el conocido mensaje citado anteriormente, en el que se da opción al usuario para que finalice la operación, la ignore o reintente. Dependiendo de la respuesta que dé el usuario, el gestor de la interrupción 24h devuelve un código, con el que comunica al DOS la operación que debe llevar a cabo.

Está claro que si deseamos evitar la aparición del mensaje anterior mientras nuestro programa residente está activo, no tendremos más remedio que modificar el contenido del vector de interrupción 24h haciendo que éste apunte a nuestro programa. Para gestionar adecuadamente los errores necesitaremos obtener algo de información acerca de ellos.

Al generarse una interrupción 24h, previamente el DOS deposita en los registros AX y DI información acerca del error que se ha producido. El byte de menor peso de DI puede contener cualquiera de los valores mostrados en la tabla 27.2, especificando el tipo de error que se ha producido.

Tabla 27.2. Códigos de error crítico.

Valor	Descripción
0	Se ha intentado escribir en un disco protegido.
1	Unidad desconocida.
2	Unidad no preparada.
3	Comando desconocido.
4	Error de CRC en los datos.
5	Tamaño incorrecto de la estructura de petición de comando.
6	Error de posicionamiento.
7	Tipo de soporte desconocido.

Valor	Descripción
8	Sector no encontrado.
9	Impresora sin papel.
10	Fallo de escritura.
11	Fallo de lectura.
12	Fallo general.

Mediante el contenido del registro AX podremos determinar dónde se ha producido el error descrito por DI y qué operaciones son posibles. En caso de que el error se haya producido al intentar acceder a una unidad de disco, el registro AL contendrá el número de unidad, que será 0 para la unidad A:, 1 para la B:, etc. El registro AH se estructura como una serie de indicadores. El significado de cada bit de este registro se encuentra en la tabla 27.3.

Tabla 27.3. Indicadores de error en el registro AH.

Bits	Significado
0	Si está a 0 el error se produjo durante una operación de lectura, si está a 1 en una operación de escritura.
1 y 2	Lugar del error en una unidad de disco: 00 - En el área de MS-DOS. 01 - En la FAT. 10 - En el directorio. 11 - En el área de datos.
3	Si está a 1 la operación puede ser cancelada.
4	Si está a 1 la operación puede ser reintentada.
5	Si está a 1 la operación puede ser ignorada.
7	Si está a cero el error ha sido en un dispositivo de bloques, como una unidad de disco. Si está a uno el error ha sido entonces en un dispositivo de caracteres, como la impresora.

Respuesta del controlador de error crítico

El controlador de errores críticos, cuya dirección se almacena en el vector de interrupción 24h, es llamado por el DOS con el fin de obtener una respuesta a partir de la cual el DOS sabrá si tiene que cancelar la ejecución de la operación, cancelar el programa, reintentar la operación o simplemente ignorar el error. En principio el controlador de errores

críticos está fijado por el propio DOS, pero si nosotros lo modificamos para que apunte a nuestro programa, tendremos que responder al DOS como éste espera. Aunque la respuesta que nuestro controlador dé debería depender de los valores de los registros AX y DI, la mayoría de los programas residentes optan simplemente por ignorar los errores. En cualquier caso, antes de devolver el control al DOS, mediante la instrucción iret, nuestro controlador deberá asignar a AL uno de los valores indicados en la tabla 27.4.

Tabla 27.4. Respuesta del controlador de errores críticos en el registro AL.

Valor	Significado
0	Ignorar el error.
1	Reintentar la operación.
2	Terminar el programa.
3	Terminar la función.

Otros aspectos a tener en cuenta

El controlador de errores críticos está limitado en cuanto a los servicios de que puede hacer uso, ya que muchas llamadas podrían provocar un nuevo error. Básicamente podemos utilizar las funciones del DOS de entrada y salida de caracteres, además de la función 5 9h que nos permite obtener una mayor información acerca del error que se ha producido. A diferencia de otros controladores de interrupciones, como el de reloj, interrupción múltiple, etc., en el controlador de error crítico nunca se llamará al anterior gestor de ese vector, por lo cual al instalar nuestro propio sistema de gestión de errores críticos estamos anulando, temporalmente, el de cualquier otro programa que anteriormente se hubiese asignado el vector de interrupción 2 4h.

Lógicamente cuando la parte residente de nuestro programa se dispone a devolver el control al programa interrumpido, de igual forma que se restituye el PSP, la pila y la DTA también se deberá devolver su valor original al vector de interrupción 2 4h, con el fin de que un posterior error no cause la activación indirecta de nuestro programa.

División por cero

Cuando un programa realiza una operación de división en la cual el divisor es cero, el procesador genera lo que se conoce como una excepción. A diferencia de una interrupción, que puede ser generada por el hardware, cualquier dispositivo, o software, mediante la instrucción int, una excepción se genera por un fallo en la propia CPU, como es la imposibilidad de hallar una solución satisfactoria a una división por cero. Ante esto el procesador transfiere el control a la dirección almacenada en el vector de interrupción cero, controlado habitualmente por el DOS y que pone fin a la ejecución del programa que ha causado la excepción.

Tomemos como ejemplo el código que se muestra a continuación, en el cual se efectúa una operación de división y a continuación se muestra un mensaje por pantalla para, finalmente, devolver el control al sistema operativo. Sin embargo, el mensaje no se llegará a visualizar nunca ya que el divisor, almacenado en DL, es cero, por lo que se provocará la excepción anteriormente citada. Ejecute el programa y observe el resultado.

```
; OivCcro.Asm
; Realiza una división por cero causando
; la interrupción del programa.

.Model tasmall
.Stack 512
.Data
Mensaje Db 'Su lia terminado el proceso?'
.Code
.Startup
Xor DX, DX ; DX = 0
Div DL      ; Dividimos AX entre DL, que es 0
; Imprimir el mensaje
Mov DX, $0F0h ; Mensaje
Mov AH, 9
Int 21h
; y terminar
Mov AH, 4Ch
Int 21h
End
```

Imagine que esta operación de división por cero se ha realizado en el código de nuestro programa residente, una vez que estaba activado. Nos encontramos con un caso similar al de un error crítico, ya que el sistema operativo intentará poner fin al programa que ha causado la excepción sin tener en cuenta que dicho programa está residente y ha modificado vectores de interrupción y otras estructuras. Por lo tanto, en caso de que en nuestro programa se vayan a realizar operaciones de división en las que no sabemos de antemano si el divisor será o no cero, necesitaremos modificar el vector de interrupción cero dirigiéndolo a nuestro programa.

Tratamiento de excepciones

Los procesadores de Intel distinguen tres tipos de excepciones, dependiendo del origen e importancia que tengan. Una excepción de cancelación tiene lugar cuando es imposible continuar con la ejecución, generalmente por un fallo importante que puede

encontrarse en el propio procesador. Otro tipo de excepción es el de interrupción o trampa, que es provocada por el procesador a petición de programas tales como depuradores con el fin de facilitar la ejecución paso a paso. Una excepción de este tipo se caracteriza porque el puntero de programa, el registro IP, apunta a la instrucción siguiente a la que ha generado la excepción.

El último tipo de excepción, y el que nos interesa a nosotros, es la excepción de fallo, que se produce cuando una cierta instrucción no puede ser ejecutada por el procesador. Al producirse una excepción de este tipo el puntero de programa contiene la dirección de la instrucción que ha causado la excepción, de tal forma que es posible repetir su ejecución en caso de que se consiga subsanar la condición que ha provocado el error. Ante una excepción de fallo, el controlador de la interrupción cero puede tomar básicamente tres caminos: evitar que se produzca de nuevo el error, haciendo que el divisor sea distinto de cero, cancelar la ejecución del programa o bien modificar la dirección de retorno para que IP apunte a la instrucción siguiente.

Por lo tanto, ante una excepción de división por cero nuestro programa residente puede informar al usuario del fallo, restituir todos los vectores de interrupción que tenga capturados, restablecer estructuras como el PSP, la DTA y la pila y, por último, devolver el control al programa interrumpido, evitando así que sea el DOS el que se haga cargo de poner fin a la ejecución.

También podemos optar por cualquiera de las dos posibilidades citadas antes, modificando el divisor para evitar que se produzca de nuevo la excepción o modificando la dirección de retorno para IP, haciendo que no se ejecute de nuevo la operación de división. Esto último es lo que se hace en el programa que se muestra a continuación, que modifica el vector de interrupción cero haciendo que éste apunte a un procedimiento propio en el cual se accede a la dirección de retorno de IP y se incrementa, con el fin de evitar su nueva ejecución. Observe que en caso de no llevar a cabo esta operación, ejecutando tan sólo la instrucción `i ret`, el procesador entraría en un bucle sin fin, ya que la operación de división por cero llamaría al gestor y éste devolvería el control a la instrucción que está generando la excepción, causando el bloqueo del sistema.

```
; DivCeroI.Asm
; Realiza una división por cero controlándola

.Model Small
.Stack 512
.Data
Mensaje Db 'Se ha terminado el proceso?'
; Para guardar el contenido del vector 0
AnteriorVectorO Dd ?

.Code
```

```

; Éste será el nuevo gestor para
; la interrupción ü
NuevoGestorO      Proc
    Push BP ; Preservar BP
    Mov BP, SP ; Obtener la dirección final
    Inc BP ; Apuntar al desplazamiento
    Inc BP ; de retorno para IRET
    Inc Word Ptr SS:[BP] ; Incrementarlo para
    Inc Word Ptr SS:[BP] ; saltar la operación Div
    Pop BP ; recuperar BP
    Iret ; Volver
NuevoGestorO      Endp

.Startup

Entrada Proc

    Mov AX, 3500h ; Obtenemos la actual
    Int 21h ; dirección en el vector 0

    ; Preservamos la dirección
    Mov Word Ptr [AnteriorVector0], BX
    Mov Word Ptr [AnteriorVector0+2], ES

    Mov AX, DS
    Push AX ; Preservar DS

    Mov DX, Seq NuevoGestorO
    Mov DS, DX
    Mov DX, Offset NuevoGestorO

    Mov AX, 2500h ; Establecemos nuestro
    Int 21h , - propio controlador

    Pop AX
    Mov DS, AX ; Recuperamos DS

    Xor DX, DX ; DX - U
    Div DL , - Dividimos AX entre DL, que es 0

    ; Imprimir el mensaje
    Mov DX, Offset Mensaje
    Mov AH, 9
    Int 21h

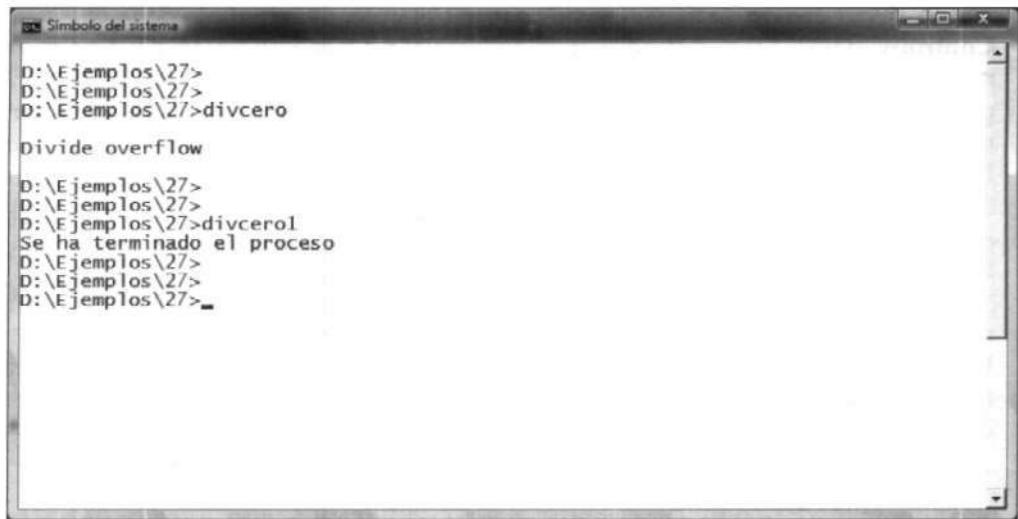
    ; Restituimos el controlador original
    Mov DX, Word Ptr [AnteriorVector0]
    Mov DS, Word Ptr [AnteriorVector0+2]
    Mov AX, 2500h
    Int 21h
    ; y terminar
    Mov AH, 4Ch
    Int 21h

Entrada Endp

End

```

Observe la diferencia existente entre ejecutar el programa DIVCERO y ejecutar DIVCERO1, mostrada en la figura 27.8. En el primer caso la excepción de fallo es controlada por el sistema operativo, que informa sobre ello y detiene la ejecución del programa, que no llega a mostrar el mensaje. En el segundo caso la excepción se gestiona en el interior del propio programa, por lo cual la ejecución termina satisfactoriamente. Sin embargo hay que tener en cuenta que la operación de división no ha llegado a efectuarse, por lo cual cualquier proceso que dependiese del resultado de dicha operación actuaría con un valor erróneo.



```
D:\Ejemplos\27>
D:\Ejemplos\27>
D:\Ejemplos\27>divcero
Divide overflow
D:\Ejemplos\27>
D:\Ejemplos\27>
D:\Ejemplos\27>divcerol
Se ha terminado el proceso
D:\Ejemplos\27>
D:\Ejemplos\27>
D:\Ejemplos\27>
```

Figura 27.8. División por cero sin controlar y controlada mediante la interrupción cero.

Tratamiento de Control-C y Control-Ínter

La pulsación de la tecla **Control** junto con la tecla **ínter**, llamada **Break** en algunos teclados, o la tecla C tienen un tratamiento especial, en el primer caso por parte de la BIOS y en el segundo por parte del propio sistema operativo.

Cuando el controlador asignado a la interrupción hardware 9h, correspondiente a la IRQ1 generada por el teclado, detecta la pulsación de la combinación **Control-ínter**, transfiere el control de inmediato a la dirección almacenada en el vector de interrupción 1Bh, o dicho de otra forma, genera dicha interrupción. El resultado dependerá de que dicho controlador haya sido o no modificado durante el proceso de arranque del DOS, ya que por defecto apunta a una instrucción iret, con lo cual la pulsación simplemente se ignora. Pero si en el archivo CONFIG.SYS existe la línea **BREAK=ON**, se instala un nuevo controlador cuya finalidad es poner fin de forma inmediata al programa que se está ejecutando.

La pulsación de la combinación **Control-C** es recogida de forma normal por la BIOS y el DOS, pero cuando éste la detecta, bien en la lectura de caracteres del buffer de teclado

o de forma más inmediata si está activado el indicador BREAK del sistema operativo, se procede de forma similar a la descrita anteriormente, poniendo fin a la ejecución del programa. De hecho el controlador de la interrupción 1Bh, generada por la pulsación de **Control-ínter**, puede limitarse a almacenar en el buffer de teclado el carácter ^C, dejando así que sea el sistema operativo el que se encargue de interrumpir el programa.

Inhibición del tratamiento de Control-C

El DOS cuenta con un indicador, llamado BREAK, que puede estar activado o desactivado, dependiendo de lo cual se procede al tratamiento inmediato de la pulsación de **Control-C** o no. Dicho indicador se puede consultar y modificar desde la línea de comandos del DOS, con la orden interna BREAK. Nuestro programa residente también puede modificar este indicador al activarse, de tal forma que evite una posible interrupción durante su funcionamiento. Para ello tendremos que utilizar dos subfunciones de la función 33h de la interrupción 21h. La primera de ellas, subfunción 0, nos devuelve en el registro DL el estado actual del indicador, que será 0 si está desactivado o 1 si está activado. Mediante la subfunción 1 podremos modificar el valor de este indicador, que facilitaremos en el registro DL.

Para comprender mejor el funcionamiento del indicador BREAK y las dos subfunciones comentadas, fijémonos en código siguiente, correspondiente al programa INVBREAK. Este programa utiliza la subfunción 0 de la función 33h de la interrupción 21h para obtener el estado actual del indicador BREAK, estado que invierte a continuación, para lo cual utiliza la subfunción 1.

```
; InvBreak.Asm
; Invierte el estado del indicador DREAK
; de 1 DOS

.MODEL SMALL
.STACK 512
.DATA
Mensaje    Db    'Indicador BREAK está $'
Activado   Db    'activo.$'
Desactivado Db    'inactivo.$'

.CODE
.Startup

Mov 1)X, Offset Mensaje
Mov AH, 9 ; Imprimimos la primera parte
Int 21h ,* del mensaje

Mov AX, 3300h ; Obtener el estado actual
Int 21h ; del indicador DOS
```

```
Or ÜL, DL ; Comprobar el estado  
Jz EstaDesactivado
```

EstaActivado:

```
Xor DL, DL ; Desactivar  
Push DX  
Mov DX, Offset Desactivado  
Jmp Salir
```

Ki gDesactivado:

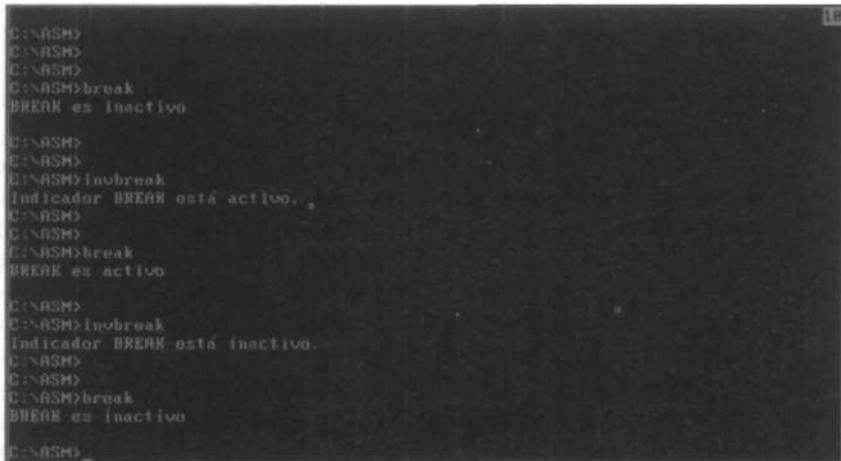
```
Mov DL, 1 ; Activar  
Push DX  
Mov DX, Offset Activado
```

Ral i r:

```
Mov AH, 9 ; Imprimimos la segunda parte  
Int 21h ; de la raddrfl  
  
Pop DX ; Recuperamos DL  
Mov AX, 3301h ; y establecemos el nuevo estado  
Int 21h  
  
Mov AH, 4Ch ; Salimos al DOS  
Int 21h
```

End

En la figura 27.9 puede observar cómo se ha usado la orden interna BREAK del DOS para comprobar el estado del indicador, y el programa INVBREAK para modificarlo. El programa en sí no tiene más finalidad que la de demostrar el tratamiento de ese indicador, ya que la propia orden BREAK del DOS permite su modificación.



The screenshot shows a DOS terminal window with the following session:

```
C:\>  
C:\>  
C:\>  
C:\>break  
BREAK es inactivo.  
  
C:\>  
C:\>  
C:\>invbreak  
Indicador BREAK est&aacute; activo..  
C:\>  
C:\>  
C:\>break  
BREAK es activo.  
  
C:\>  
C:\>invbreak  
Indicador BREAK est&aacute; inactivo.  
C:\>  
C:\>  
C:\>break  
BREAK es inactivo.  
  
C:\>
```

Figura 27.9. Manipulamos el estado del indicador BREAK del sistema.

Inhibición del tratamiento de Control-Inter

En caso de que el vector de interrupción 1Bh esté apuntando a algún gestor que se encargue de detener el programa que se esté ejecutando, el único método del que disponemos para poder evitarlo consiste en modificar dicho vector, haciendo para ello que éste apunte a un procedimiento de nuestro programa, que se puede limitar a una simple instrucción iret, de tal forma que la pulsación de dicha tecla sea ignorada por completo.

Si se desea, es posible incluir otro tratamiento más elaborado, por ejemplo indicando al usuario que ha pulsado **Control-ínter** y preguntándole si desea poner fin al programa, caso éste en que habría que restituir todos los vectores y estructuras que se hayan modificado, devolviendo el control al programa interrumpido.

Otros aspectos a tener en cuenta

En caso de que nuestro programa desactive el indicador BREAK del DOS al ponerse en marcha, y lo restituya al salir sin haber realizado ninguna entrada por teclado, es posible que el programa que hemos interrumpido y al que devolvemos el control encuentre el carácter ^C en el buffer de teclado, causando su interrupción.

Imagine que el usuario pulsara dicha combinación de teclas con el fin de interrumpir nuestro residente y, sin embargo, terminase interrumpiendo el programa que había *debajo*.

Si queremos evitar situaciones como ésta no tendremos más remedio que modificar también el vector de interrupción 23h, sin desactivar el indicador BREAK, con el fin de proceder nosotros mismos al tratamiento de **Control-C**.

Acceso a la pantalla

La mayoría de los programas residentes que necesitan mostrar algún tipo de información, o bien recoger datos de la pantalla, lo hacen accediendo directamente a la memoria de vídeo, aún cuando también es posible utilizar el DOS o la BIOS, siempre que sea seguro hacerlo.

Sin embargo, antes de proceder con cualquier operación de visualización o captura de información, un elemento que se ha de tener en cuenta es si el modo actual de vídeo es el que nosotros esperamos, puesto que no podemos asumir que el programa residente va a ser activado siempre estando en modo texto o bien siempre en un determinado modo gráfico.

El primer paso que habrá de dar nuestro programa será determinar el modo actual de vídeo, para lo cual utilizaremos el byte situado en 0040h: 0049h, que contiene el código correspondiente a dicho modo. En el caso de que el modo de vídeo sea el adecuado, procederemos según sea necesario, en caso contrario, si el programa residente no puede funcionar en el modo actual, se dará un aviso al usuario, por ejemplo mediante un pitido.

Salvaguarda del contenido de la pantalla

En caso de que el programa residente necesite visualizar algo en pantalla, antes de hacerlo deberá salvaguardar distintos parámetros, como el contenido actual de la memoria de vídeo, la posición del cursor y, en caso de que se trate de un modo gráfico, también el registro de la paleta de color y cualquier otro parámetro que vaya a ser modificado.

Para determinar la dirección de memoria correspondiente a la página de vídeo que se está visualizando en un determinado momento en pantalla, leeremos el contenido de la palabra que se encuentra en 0040h:004En. Esta indica el desplazamiento a partir del segmento inicial de vídeo. También podemos conocer el tamaño de dicha página, en bytes, dato que se almacena en la palabra que se encuentra en la dirección 0040h:004Ch.

Salvaguardar el contenido de una página de vídeo en modo de texto es fácil, ya que generalmente sólo son necesarios 4000 bytes, pero si el modo actual es un modo gráfico, el tema se complica, ya que puede ser necesario un bloque de memoria mucho más extenso, que en el momento en que el programa residente es activado puede o no estar disponible. En casos como éste una opción al problema es utilizar un archivo en disco, o bien acceder a la memoria extendida que tenga el sistema con el fin de almacenar temporalmente el contenido de la pantalla y restituirlo posteriormente, usando para ello los servicios descritos en el capítulo previo.

Aunque no es una técnica muy habitual, un programa residente puede cambiar el modo de vídeo en que se encontraba en ese momento el programa interrumpido. En este caso hay que tener en cuenta que dicho programa puede estar usando otras zonas de la memoria de vídeo, por ejemplo para almacenar varias páginas de texto, que seguramente se verán destruidas al cambiar de modo. Para evitar problemas como éste sería necesario salvaguardar toda la memoria de vídeo, así como todos los parámetros mencionados anteriormente de posición y forma del cursor, paleta de color, etc.

Estado del teclado

A pesar de no tener la importancia que tienen los parámetros de vídeo, también el teclado cuenta con una serie de indicadores que pueden ser modificados durante la ejecución del programa residente y que deberían ser restituidos antes de devolver el control al programa interrumpido.

Cuando el programa residente se activa, la tecla de bloqueo de mayúsculas, inserción o bloqueo de números pueden encontrarse en un determinado estado, que puede ser modificado por el usuario mientras interactúa con el programa residente, por ejemplo para facilitar un nombre de archivo.

Aunque no se trate de parámetros de gran interés, el programa residente al activarse puede obtener el estado de las teclas mencionadas, simplemente leyendo el byte situado en la dirección 0040h:0017h, en el que los bits 4 a 7 contienen el estado de las teclas de bloqueo de desplazamiento, bloqueo de números, bloqueo de mayúsculas e inserción, respectivamente. Este byte puede ser restaurado posteriormente, antes de devolver el control, de tal forma que el programa interrumpido se encuentre con el mismo estado que tenía previamente.

Estado del ratón

Al igual que la pantalla cuenta con un cursor, que en un determinado momento se encuentra en un punto en concreto, si estamos usando un ratón existirá otro cursor, del cual la pantalla no tiene conocimiento alguno. Si nuestro programa residente va a hacer uso de algún servicio que modifique la posición o algún otro parámetro del ratón, antes de ello se ha de asegurar de salvaguardar el estado actual en que se encuentre dicho dispositivo, con el fin de restituirlo al salir.

Lo primero que tendremos que hacer, en caso de que vayamos a utilizar el ratón, será determinar si éste está o no instalado, y en caso afirmativo utilizaremos el servicio 15h de la interrupción 33h, que nos indicará el tamaño que ha de tener el bloque de memoria para preservar los parámetros actuales del ratón. A continuación usaremos el servicio 16h, que nos permitirá guardar el estado actual, de tal forma que posteriormente, a la salida, podamos restituirlo simplemente llamando al servicio 17h.

Activación por teclado

Si bien ya conocemos la mayor parte de los elementos a tener en cuenta en el diseño de un programa residente, aún no hemos aprendido a utilizar la interrupción de teclado con el fin de que dicho programa sea activado mediante una combinación de teclas, en lugar de hacerlo mediante una interrupción. Se ha dejado este apartado casi para el final con el objetivo de que, al iniciar el desarrollo de nuestro primer programa residente útil, pudiésemos tener todos los puntos precisos en cuenta, evitando así cualquier problema que pudiese surgir.

El teclado de un PC cuenta en su interior con una matriz de contactos y un pequeño integrado, habitualmente un 8741, 8742 o alguno compatible, que es el encargado, entre otras funciones, de transmitir al ordenador lo que se denomina *scan code* o código de la tecla que se ha pulsado. Este código no tiene nada que ver con el código ASCII de un carácter, de hecho algunas de las teclas no tienen tal correspondencia. Es el controlador de teclado, normalmente el programa KEYB del DOS, el que se encarga de traducir cada código de teclado a su correspondiente código ASCII, en caso de que esto sea necesario.

Cada vez que el controlador de teclado detecta el cierre de un contacto, o lo que es lo mismo, la pulsación de una tecla, genera la IRQ1, cuyo resultado será la ejecución del actual gestor de la interrupción 9h. El gestor habitual de esta interrupción lee el código, a través del puerto 60h, y lo almacena en la cola circular de teclado, poniéndolo así a disposición del sistema operativo y los programas dicha información.

Interceptar la interrupción de teclado

El punto más adecuado para que un programa residente detecte la pulsación de la tecla de activación se encuentra, precisamente, en la interrupción 9h que, como se ha dicho anteriormente, es generada con cada pulsación.

Existen otros métodos, como la exploración a intervalos del *buffer* de teclado, pero siempre imponen más limitaciones.

La única dificultad que encontraremos al escribir nuestro propio gestor para la interrupción de teclado está en la interpretación de los códigos que éste envía, ya que deberemos saber qué código es el que corresponde a la tecla de activación que hemos seleccionado, así como la gestión del teclado a través de los puertos correspondientes, con el fin de eliminar la pulsación de tecla que nos ha activado y que ésta no llegue al programa que se está ejecutando en ese momento.

La técnica explicada en este artículo para la gestión de la interrupción de teclado puede no ser válida en algunos sistemas PS/2, en los cuales sería necesario interceptar la función **4Fh** de la interrupción **15h**.

Control del teclado a bajo nivel

A diferencia de lo que ocurre con los códigos ASCII, que no identifican todas las teclas existentes, los códigos de teclado existen para todas las teclas, incluidas aquellas que aparentemente no generan una interrupción, como las teclas **Mayús**, **Control** o **Alt**.

El código de teclado de una determinada pulsación se genera internamente, en el propio teclado, y se envía al registro A del PPI, siendo leído por el sistema en respuesta a la generación de la interrupción 9h. Una vez que la pulsación ha sido leída, es necesario enviar al PPI un comando para que sepa que ya ha sido leída y, por último, será necesario enviar una señal de fin de interrupción al controlador de interrupciones. Tanto el PPI como el PIC, denominados también 8255 y 8259, son dos circuitos integrados que se describieron en los primeros capítulos.

Por tanto, si nosotros interceptamos el vector correspondiente a la interrupción 9h, lo primero que tendremos que hacer será leer el código de teclado, simplemente utilizando la instrucción in sobre el puerto 60h, lo que nos devolverá el contenido actual del registro A del PPI. A continuación comprobaremos si el código obtenido corresponde al de la tecla de activación, de no ser así transferiremos el control al anterior gestor de teclado, lo que permitirá que todo siga funcionando de forma normal.

Si se pulsa la tecla de activación de nuestro programa residente, tendremos que evitar que ésta llegue posteriormente hasta el programa que se está ejecutando. Para ello lo primero que haremos será indicar al PPI que ya la hemos leído, para lo cual leeremos el valor actual del registro B, mediante el puerto 61h, y pondremos a 1 el bit de más peso, enviándolo por el mismo puerto. A continuación enviaremos a dicho puerto el valor que tenía originalmente, tras lo cual la gestión de lectura de la pulsación queda finalizada. El último paso será enviar la señal de fin de interrupción, escribiendo el valor 2 Oh en el puerto 2 Oh, lo que equivale a activar el bit EOI (*End Of Interrupt*) de la segunda palabra del comando de operación en el controlador de interrupciones.

En resumen, el proceso a llevar a cabo en el gestor de la interrupción 9h por parte de nuestro programa residente, sería similar al mostrado en este fragmento de código:

```
In AL, 60h ; Leemos la pulsación
Cmp AL, Código ; Si es el código de activación
Jmp Activación ; Saltar
```

```
Pushf ; Si no es simular la INT 9
Call Antigualnt9 ; llamando al anterior gestor
```

Activación:

```
In AL, 61h ; Leemos el contenido del registro B
Or AL, 80h ; Activamos el bit 7
Out 61h, AL ; y lo enviamos
And AL, 7Fh ; Desactivamos el bit 7
Out 61h, AL ; y dejamos el contenido original
Mov AL, 20h
Out 20h, AL ; Fin de interrupción
```

Iret

Códigos de teclado

Obviamente, para poder comparar el código de una pulsación con el que debe activar nuestro programa lo primero que necesitaremos es conocer los códigos generados por las distintas teclas. La figura 27.10 muestra gráficamente la disposición habitual en un teclado extendido de 102 teclas, el más usado antes de la aparición de los teclados específicos para Windows, indicándose el código correspondiente a cada una de las teclas.



Figura 27.10. Códigos de teclado.

Los códigos mostrados en la figura 27.10 son los que generan las teclas al ser pulsadas, aunque ciertas teclas generan más de un código. Por ejemplo, las teclas correspondientes a edición extendidas, que se encuentran en la parte central del teclado, generan varios códigos, con el fin de que sea posible distinguirlas de las mismas teclas existentes en el área numérica. Sin embargo, para la función que nosotros buscamos que es permitir la activación del programa mediante una cierta tecla, tendremos suficiente con los códigos genéricos. Esto tendrá como efecto que si seleccionamos una tecla especial, como puede

ser una de las de edición, la activación pueda llevarse a cabo tanto con la tecla que existe en la parte extendida como la que hay en el área numérica.

Además del código generado al pulsar la tecla, nuestro controlador para la interrupción 9h también recibirá un código cada vez que se libera una tecla. Este código será el mismo que se recibe al pulsar, pero con el bit de más peso puesto a 1, o lo que es lo mismo, el código de pulsación más 128.

Con el fin de facilitarnos el trabajo de la obtención de los códigos correspondientes a una cierta tecla, tanto de pulsación como de liberación, vamos a escribir un programa cuya finalidad será asignarse el vector de interrupción 9h y mostrar por pantalla todos los códigos que reciba. Puede ver a continuación el código de este programa, con todos los comentarios necesarios para facilitar su comprensión.

```
; Códigos.ASM

; Intercepta la interrupción de teclado con el
; lin de ir mostrando por pantalla los códigos
; generados por las pulsaciones.

.Model Small
.Stack 512
.Data

; Para componer la cadena decimal
; correspondiente cada código
CifraDecimal    Db      3 Dup(?)
                  Db      ", S"

; Secuencia para avanzar una linea
NuevaLinea      Db  13, 10, "$"

; Para conservar la dirección del anterior
; gestor de la interrupción
AnteriorGestor Dd ?

; Indicador de salida del programa
Indicador  Db 0

.Code
.Startup

Mov AX, 3509h ; Obtener la dirección actual
Int 21h , - del vector de interrupción 9

; y guardarla en AnteriorGestor
Mov Word Ptr [AnteriorGestor], BX
Mov Word Ptr [AnteriorGestor+2], ES

; Fijar el nuevo gestor para la interrupción
Mov DX, Seg NuevoGestor
Mov DS, DX
Mov DX, Offset NuevoGestor
```

```

Mov AX, 2509b
Int 21h

Mov AX, @Data ; Volver a cargar la

Mov DS, AX ; dirección del segmento de datos

Bucle:

; El programa se quedará en este
; bucle sin hacer nada hasta que Indicador
; deje de ser 0
Mov AL, [Indicador]
Or AL, AL
Jz Bucle
; Devolver al vector su antigua dirección
Mov DX, Word Ptr [Anterior/Gestor]
Mov DS, Word Ptr [AnteriorGestor+2]
Mov AX, 2509h
Int 21h

Mov AH, 4Ch ; Y salir al DOS
Int 21h

; Este procedimiento se encarga de
; convertir el dato facilitado en AI,
; en una cadena, imprimiéndola en la
; posición actual en pantalla.

Imprime      Proc

Push DX ; Preservar los registros
Push AX

Xor AH, AH ; Eliminar el contenido de AH

Mov DL, 100 ; Obtener las centenas
Div DL

Add AL, '0' , - Convertir en digito
Mov Byte Ptr [CifraDecimal], AL

Mov AL, AH ; Tomar el resto en AL
Xor AH, AH ; y borrar AH

Mov DL, 10
Div DL ; Dividir el contenido de AX entre 10

Add AL, '0' ; Convertir en carácter
Add AH, '0' ; cada digito

; Pasarlos a la cadena
Mov Word Ptr [CifraDecimal+1], AX

Mov DX, Offset CifraDecimal
Mov AH, 9 ; Imprimir la cadena
Int 21»

```

```
Pop AX ; Recuperar el valor de AX
Push AX

Cmp AL, 1 ; ¿Se ha pulsado ESC?
Jne NoEscape
    ; En caso afirmativo activar el Indicador
    ; provocando el fin del programa
Mov [Indicador], AL

NoEscape:

Cmp AL, 28 ; ¿Se ha pulsado INTRO?
Jne Nolntro
    ; Si es así saltar una linea
Mov DX, Offset NuevaLinea
Mov AH, 9
Int 21h

Nolntro:

Pop AX ; Recuperar los registros
Pop DX

Ret ; y volver

Imprime      Endp

; Éste será el nuevo gestor de la interrupción
; de teclado

NuevoGestor Proc

Push AX ; Preservar los registros a modificar

In AL, 60h ; Leer el código de teclado
Push AX ; y guardarlo en la pila

In AL, 61h ; Indicar que ya se ha leído
Or AL, 80h ; el código
Out 61h, AL
And AL, 7Fh
Out 61h, AL

Mov AL, 20h ; Enviar EOI al controlador
Out 20h, AL ; de interrupciones

Pop AX ; Recuperar el código
Cali Imprime ; e imprimirllo

Pop AX ; Recuperar el contenido original de AX
Iret ; y terminar la interrupción

NuevoGestor Endp

End
```

Al ejecutar el programa verá aparecer inmediatamente un código, correspondiente a la liberación de la tecla **Intro**, ya que el programa se aloja en memoria y se activa seguramente antes de que la hayamos liberado.

A continuación podemos ir pulsando teclas, de tal forma que en pantalla veamos los códigos de pulsación y liberación generados. En cualquier momento podemos pulsar la tecla **Intro** para pasar de una línea a la siguiente, con el fin de permitirnos hacer una separación y facilitar la lectura de los códigos. Para abandonar el programa tan sólo tenemos que pulsar la tecla Esc, lo que provocará que se devuelva al vector de interrupción su antiguo valor y se regrese a la línea de comandos. En la figura 27.11 puede ver un momento del funcionamiento del programa.

Figura 27.11. El programa va facilitando las secuencias de códigos correspondientes a las teclas pulsadas.

Combinaciones de teclas

Normalmente la activación de un programa residente tiene lugar mediante la combinación de unas ciertas teclas, y no ante la pulsación de una sola ya que esto impediría el normal uso de dicha tecla. No podemos, por ejemplo, activar nuestro programa cuando se pulse la tecla P, ya que esto provocaría que el usuario no pudiese utilizar tal carácter en ningún programa. Sin embargo, sí que podríamos usar esa tecla en combinación con otra, como **Control** o **Alt** e incluso una combinación de ellas.

En el teclado del PC existen una serie de teclas que no tienen función alguna por sí solas, y cuya finalidad es modificar en cierta forma el funcionamiento normal de otras teclas. La tecla **Mayúsculas** se combina con cualquier tecla alfabética para obtener una letra mayúscula, o con una tecla numérica para obtener un símbolo.

La tecla **Control** también se combina con ciertas teclas para conseguir lo que se denominan códigos de control. Mediante la tecla **Alt** existen menos asociaciones hechas de antemano, aunque se puede usar junto con la tecla **Control** y una tercera tecla para conseguir ciertos caracteres especiales.

Ejecutando el programa CÓDIGOS podrá ver que las teclas que hemos mencionado generan también sus códigos, al igual que cualquier otra. Pero, además, la BIOS mantiene unos bytes de estado mediante los cuales podremos saber en cualquier momento si una de estas teclas está pulsada o no. Esto nos servirá en el momento en que detectemos la pulsación de la tecla de activación de nuestro programa, tras lo cual comprobaremos si la tecla con la ha de combinar está o no pulsada, activándose en caso afirmativo o llamando al anterior vector de interrupción en caso contrario.

Suponga, por ejemplo, que el programa se ha de activar mediante la combinación **Control-P**. Cuando en nuestro gestor de la interrupción 9h se detecte la pulsación de la tecla P, examinaremos un byte mediante el cual sabremos si está pulsada la tecla **Control**, procediendo a la activación si es así.

Bytes de estado del teclado

Aunque vimos en su momento cómo usar los servicios de la interrupción 16h para obtener cierta información del teclado, sirvan los párrafos siguientes como recordatorio al mecanismo que nos permitirá definir la combinación de teclas para activar nuestro programa.

En el segmento 4 Oh existen múltiples parámetros relacionados con el teclado. En la dirección 4 Oh:17h existe un byte de estado cuyos cuatro bits de mayor peso contienen el estado de las teclas de bloqueo de inserción, mayúsculas, números y desplazamiento. Pues bien, ese mismo byte, en sus cuatro bits de menor peso, contiene el estado actual de las teclas **Alt**, **Control**, **Mayús izquierda** y **Mayús derecha**. En la tabla 27.5 se resume el contenido de este byte de estado.

Tabla 27.5. Bits del byte de estado 40h:17h.

Bit	Significado
0	Mayúsculas derecha pulsada.
1	Mayúsculas izquierda pulsada.
2	Control pulsada.
3	Alt pulsada.
4	Desplazamiento activo.
5	Bloqueo números activo.
6	Bloqueo mayúsculas activo.
7	Insertar activo.

Como puede observar, en el byte de estado que acabamos de describir no se hace ninguna diferenciación entre la tecla **Control** situada en la parte izquierda del teclado y la existente en la parte derecha, y lo mismo ocurre con la tecla **Alt**. Esto es así porque en los teclados más antiguos esas teclas no existían y, para reflejar su estado, se han tenido que habilitar nuevos bytes de estado, concretamente en las direcciones 40h:18h y 40h:96h.

Además de permitirnos diferenciar las teclas citadas, este byte de estado también nos indica si las teclas de bloqueo están pulsadas en ese instante. En las tablas 27.6 y 27.7 se muestran los bits de interés de estos bytes de estado.

Tabla 27.6. Bits del byte de estado 40h:18h.

Bit	Significado
0	Control izquierda pulsada.
1	Alt izquierda pulsada.
2	PetSis pulsada.
3	Pausa activada.
4	Desplazamiento pulsada.
5	Bloqueo números pulsada.
6	Bloqueo mayúsculas pulsada.
7	Insertar pulsada.

Tabla 27.7. Bits del byte de estado 40h:96h.

Bit	Significado
2	Control derecha pulsada.
3	Alt derecha pulsada.

Esquema general de un programa residente

Una vez que conocemos el método por el cual podemos activar nuestro programa residente mediante una cierta combinación de teclas, podemos pasar al diseño completo, ya que tenemos todos los datos necesarios para ello.

Recopilemos toda la información necesaria, facilitada a lo largo de este capítulo, representando esquemáticamente los pasos que tendría que dar nuestro programa. Partiremos de cuatro grandes bloques que agrupan el código que se ejecutará en cada momento en nuestro programa. Estos bloques son:

- **Instalación:** Tiene lugar cuando se ejecuta el programa desde la línea de comandos del DOS.
- **Desinstalación:** Tiene lugar cuando al ejecutar el programa desde la línea de comandos se facilita un cierto parámetro.
- **Supervisión:** Este bloque agrupa todo el código que se estará ejecutando de forma continua, como son los gestores de las distintas interrupciones.
- **Activación:** Tiene lugar cuando se solicita la activación de la parte residente y el proceso de supervisión informa que es seguro hacerlo.

Instalación

Los pasos que daremos al ejecutar el programa desde la línea de comandos, sin haber entregado parámetro alguno, serán los siguientes:

- Comprobar que el programa no está ya instalado, si es así indicarlo entonces y no continuar.
- Salvaguardar el contenido de los vectores de interrupción a modificar, sustituyéndolos por los nuestros. Básicamente tendremos que modificar los vectores 9h, 10h, 13h, ICh, 28h y 2Fh.
- Calcular el tamaño de código y quedar residente.

Desinstalación

Al ejecutar el programa facilitando un cierto parámetro deberá proceder a su desinstalación si es que es posible, para lo cual daremos los siguientes pasos:

- Comprobar si los vectores que modificamos durante la instalación siguen apuntando a nuestros gestores. Si no es así salir indicando entonces que no es posible la desinstalación.
- Restablecer los antiguos valores de todos los vectores de interrupción modificados.
- Liberar la memoria ocupada por el programa y el bloque de entorno.

Supervisión

El proceso de supervisión tendrá lugar en los distintos gestores de interrupción, mediante los cuales determinaremos si hay que activar el programa y si es seguro hacerlo en un determinado momento. Durante este proceso usaremos los indicadores Activar, que se pondrá a 1 cuando el usuario solicite la activación, e InBios, que nos servirá para saber cuándo está en curso una llamada a la BIOS.

Gestor de INT 9h

- Leer código de tecla.
- Si no es la tecla de activación llamar al antiguo gestor.
- Mirar el byte de estado de teclado para comprobar si está pulsada la tecla de combinación. Si no es así llamar al antiguo gestor.
- Activar = 1, borrar la pulsación de tecla y salir.

Gestor de INT 1Ch

- Llamar al antiguo gestor.
- Si Activar == 0 terminar.
- Si ErrorMode > 0 terminar.
- Si InDos > 0 terminar.
- Si InBios > 0 terminar.
- Activar la parte residente.

Gestor de INT 28h

- Llamar al antiguo gestor.
- Si Activar = 0 terminar.
- Si ErrorMode > 0 terminar.
- Si InBios > 0 terminar.
- Activar la parle residente.

Gestor de INT 1Qh e INT 13h

- Sumar 1 a InBios .
- Llamar al antiguo gestor.
- Restar 1 a InBios .

Activación

La activación se realizará mediante una llamada desde el gestor de la interrupción ICh o la interrupción _8h, momento en el cual el sistema en general, tanto el DOS como la BIOS, se encuentran en situación de poder ser interrumpidos. En este proceso

utilizaremos un indicador más, Activado, que estará a 1 cuando la parte residente ya esté activa. Daremos los siguientes pasos:

- Activar = 0. Borrar el indicador de activación.
- Si Activado = 1 terminar.
- Si el modo de vídeo actual no es soportado por nuestro programa terminar.
- Activado = 1. Activar el indicador de que la parte residente está en ejecución.
- Salvaguardar las direcciones actuales de pila, PSP y DTA.
- Establecer la nueva pila, PSP y DTA.
- Salvaguardar el contenido actual de los vectores de error crítico, **Control-ínter** y **Control-C**.
- Establecer los nuevos gestores en los vectores anteriores.
- Salvaguardar todos los parámetros que vayan a ser modificados, como la posición del cursor, modo de vídeo, directorio actual, etc.
- Ejecutar la parte residente.
- Restablecer los parámetros guardados anteriormente.
- Restablecer los vectores de interrupción anteriores.
- Restablecer DTA, PSP y Pila.
- Activado = 0. Borrar el indicador de activado.
- Terminar.

Otros factores a tener en cuenta

De todos los parámetros que se indican en los puntos anteriores, el programa sólo debería salvaguardar y restaurar aquellos que vayan a ser modificados. No tiene ningún sentido que se guarde el modo de vídeo actual y la posición del cursor si el programa no accede a pantalla, de esta forma lo único que estaríamos haciendo sería ocupar algo más de memoria y hacer el proceso de activación un poco más lento, todo ello de forma innecesaria.

Además de vectores de interrupción y otros parámetros, hemos de recordar que nuestro programa se activa interrumpiendo la ejecución de otro programa, por lo cual todos los registros que vayan a ser modificados deberán ser almacenados en la pila, con el fin de restituir sus valores originales justo antes de salir, de tal forma que el programa interrumpido no se vea afectado en ningún momento por nuestro funcionamiento.

Aunque la interceptación de la interrupción 1Ch funcionará correctamente en la mayoría de los casos, con el fin de periódicamente ver si es posible activar el programa, ciertos programas manipulan de alguna forma este vector impidiendo su funcionamiento

correcto. Por ejemplo, si intentamos activar un programa residente desde la interrupción ICh estando dentro del programa EDIT del DOS, veremos que no es posible. La solución es muy simple y consiste en interceptar la interrupción 8h en lugar de la ICh.

Una tabla de códigos ASCII residente

Básicamente hemos terminado de conocer todos los elementos necesarios para poder desarrollar un programa residente completo. Se ha confeccionado de forma esquemática una guía con las distintas fases del funcionamiento y los puntos que es necesario llevar a cabo en cada una de dichas fases.

Llegados a este punto podemos pasar a la codificación de nuestro programa completo, cuya finalidad será mostrar una tabla de códigos ASCII con sus correspondientes caracteres, permitiéndonos seleccionar cualquiera de ellos transfiriéndolo al programa que en ese momento se está ejecutando.

Podrá encontrar el código completo del programa en el archivo TBLASCTT. ASM, con amplios comentarios que facilitan su comprensión. A continuación vamos a comentar distintos fragmentos de código de manera individual.

La instalación

Cuando el programa TBLASCTT se ejecuta, llamándolo desde la línea de comandos, el control se transfiere directamente al procedimiento Instalar, que será la única parte del programa que no quede residente en memoria.

El proceso de instalación del programa TBLASCIIf es similar al que usábamos en los ejemplos anteriores. En el procedimiento Instalar primero examinamos la línea de comandos, para ver si se ha facilitado la opción /D, que sería la petición de desinstalación. Dependiendo de ello se intenta la instalación o la desinstalación del programa, para lo cual es determinante la interrupción 2Fh, o mejor dicho el gestor que nosotros instalamos en dicha interrupción, ya que él nos permitirá saber si el programa está o no ya instalado, no permitiendo la reinstalación, y se ocupará de restituir los vectores de interrupción originales y facilitar la información necesaria para completar la desinstalación.

Para facilitar el proceso de obtención y modificación de los vectores de interrupción, al inicio del programa se ha definido una tabla conteniendo por cada vector su número, la dirección de gestor original, y la del gestor propio que vamos a instalar, tal y como puede ver en el fragmento que se muestra a continuación.

```
; Estructura para mantener la información
; de cada uno de los vectores de interrupción
; a modificar.
BlqInt Struc
    Numero        Db ?
    AnteriorGestor Dd ?
    NuevoGestor   Dd ?
BlqInt Ends
```

```

; Datos de cada una de las interrupciones
INT09    Blqlnt    <09h, ?, Gestor09>
INT10    Dlqlnt    <10h, ?, Gestor10>
INT13    Blqlnt    <13h, ?, Gestor13>
INT08    Blqlnt    <08h, ?, Gestor08>
INT28    Blqlnt    <28h, ?, Gestor28>
INT2F    Blqlnt    <2Fh, ?, Gestor2F>
INT1B    Blqlnt    <1Bh, ?, Gestor1B>
INT23    Blqlnt    <23h, ?, Gestor23>
INT24    Blqlnt    <24h, ?, Gestor24>

```

Observe cómo se define una estructura de datos, mediante Struct, para simplificar después la creación de la tabla de interrupciones. Ésta es una posibilidad con la que no cuenta NASM.

De esta forma, en el proceso de instalación no es necesario tratar cada interrupción de forma individual, sino que se crea un bucle que recorre la tabla, como puede ver en el fragmento siguiente, ahorrando así muchas líneas de código.

```

; A continuación obtenemos el contenido
; actual de los vectores de interrupción
; y los modificamos, haciendo que apunten
; a nuestros gestores.

Mov CX, 6 ; Modificar seis vectores
; Dirección del primer bloque
Mov ST, Offset INT09

BucleInt:
    ; Ponemos en AL el número del vector
    Mov AL, [SI].Blqlnt.Numero
    Mov AH, 35h
    Tst 21h ; y obtenemos el contenido actual

    ; Salvaguardamos la dirección
    Mov Word Ptr [ST].Blqlnt.AnteriorGestor[ü], BX
    Mov Word Ptr [SI].Blqlnt.AnteriorGestor[2] , ES

    ; Obtener la dirección del nuevo gestor
    Mov DX, Word Ptr [SI] .Blqlnt.Nuevo-Gestor [0]
    Mov AH, 2 5h
    Tst 21h ; y escribirla en el vector

    Add SI, 9 ; Saltar al siguiente bloque
Loop BucleInt

```

El proceso de instalación del programa, si llega a término de forma satisfactoria, finaliza con la impresión de un mensaje con una pequeña ayuda y queda residente a espera de su activación.

La activación del programa tendrá lugar mediante la tecla **Alt** pulsada conjuntamente con cualquier otra, cuyo código se define como una constante al principio del programa con el fin de facilitar su modificación.

Petición de activación

A lo largo de este capítulo hemos visto que un programa residente puede ser activado mediante una interrupción software, ejecutando la instrucción `int`, o bien mediante una interrupción hardware, que puede venir provocada por el teclado, el reloj, puerto serie, etc.

En este caso el programa se activará mediante la interrupción provocada por el teclado, por lo cual hemos modificado el vector de interrupción 9h haciendo que apunte a nuestro gestor, cuyo código se muestra a continuación:

```

; Gestor de la interrupción de teclado

Gestor09 Proc
    Push AX ; Salvaguardamos AX

    In AL, PPI_PORT_A ; Leemos el código de tecla
    ; ¿Es la tecla de activación?
    Cmp AL, TECLA_ACTIVACION
    Jne NoEsPeticion ; si no saltar

    Push ES      ; Salvaguardar ES
    ; Cargar en AL el contenido del
    Mov AX, SEGMENTO_BIOS
    Mov ES, AX ; indicador de estado almacenado
    Mov AL, liS: [SHIFT_STATUS]

    Pop ES      ; Recuperamos ES

    And AL, 8 ;Comprobar si está pulsada la tecla ALT
    Jz NoEsPeticion ; Si no así no continuar

    In AL, PPI_PORT_B ; Indicar que la tecla ya
    Or AL, BOh ; ha sido leída
    Out PPI_PORT_B, AL
    And AL, 7Fh
    Out PPI_PORT_B, AL

    Mov AL, EOI ; Enviar EOI al controlador
    Out PIC, AL ; de interrupciones

    ; Ver si estamos ahora mismo activos
    Cmp CS:[Activado], 1

    ; Si es así no seguir
    Je NolndicarActivar

    Mov CS:Activar, 1 ; Activar el indicador

NolndicarActivar:

    Pop AX ; Recuperar AX
    Iret    ; y salir

```

NoEsPeticion:

```
Pop AX ; Recuperar AX

Pushf ; Meter el registro de flags
; y llamar al anterior gestor
Cali CS:INT09.AnteriorGestor

iret ; Volver

Gestor09 Endp
```

To primero que hacemos es leer el código de la tecla, que se encuentra en el puerto A del PPI. Comparamos este código con el de la tecla de activación y, en caso de que coincida, miramos si está pulsada la tecla **Alt**, caso éste en el cual será necesario poner a 1 el indicador Activar a no ser que el Activado esté ya a 1, porque en ese caso el programa residente ya está activo, no siendo posible su activación.

Observe que se han eliminado del código la mayor parte de valores inmediatos y se han sustituido por constantes que hacen más fácil la comprensión.

Todas estas constantes se han definido al inicio del módulo de código tal como se ve a continuación:

```
; Definición de constantes

; Código de la tecla de activación del programa
TECLA ACTIVACIÓN EQU 20 ; T

; Identificador del programa para la INT 2Fh
ID_PROGRAMA EQU 123

; Código de confirmación del programa
CONFIRMACIÓN EQU 54321

; Con estos códigos se dibujarán los bordes
; de la tabla de caracteres
ESQ1 EQU 201
ESQ2 EQU 200
ESQ3 EQU 187
ESQ4 EQU 188
HORZ EQU 205
VERT EQU 186

; Segmento de memoria de video
SEG_PANTALLA EQU 0B800h

; Bytes que ocupa una pantalla
; de texto
BYTES_PANTALLA EQU 4000

; Atributo para la tabla.
ATTR_TABLA EQU 1Fh
```

```

; Ancho de la tabla menos laterales
ANCHO_TABLA      EQU      72

; Número de bytes de cada linea
; de pantalla en modo texto
BYTES_LINEA      EQU      160

; Número de columnas y filas en la tabla
COLUMNAS         EQU      12
FILAS            EQU      22

; Segmento de datos de la BIOS
SEGMENTO_BIOS    EQU      40h

; Número de caracteres que ocupa cada
; entrada en la tabla
CARACTFRF,S_ELEMENTO EQU      6

; Códigos de las teclas a controlar en
; el programa
CURSOR_ARRIBA    EQU      72
CURSOR_IZQUIERDA EQU      75
CURSOR_DERECHA   EQU      77
CURSOR_ABAJO     EQU      80
TECLA_ESCAPE     EQU      1
TECLA_INSERTAR   EQU      82

; Distintas direcciones de puertos
Pri_PORT_A        EQU      60h
PPI_PORT_B        FQU      61h
PIC               EQU      20h
EOI               EQU      20h

; Distintas direcciones de datos
MODOJVIDEO        EQU      49h
PAGINA_VIDEO      EQU      62h
QFFSET_PAGINA_ACTUAL FQU      4Eh
PESCRITURA_TECLADO EQU      1Ch
INICIO_BUFFER_TECLADO EQU      1Eh
FIN_BUFFER_TECLADO EQU      3Eh
SHIFT_STATUS       EQU      17h
SEG_BLOQUE_ENTORNO EQU      2Ch

; Tamaño de la pila para la parte residente
TAMANO_PILA       EQU      128

```

La existencia de estas constantes al inicio del programa también hace más sencilla la modificación de ciertos aspectos, como la tecla de activación, el atributo con el que se mostrará la tabla de códigos ASCII o su ancho, el identificador con el que se instalará la parte residente, etc.

Observe que la pulsación de la combinación **Alt-tecla** es ignorada siempre, tanto si la activación es posible como si no lo es. Si la tecla pulsada es cualquier otra nos limitamos a llamar al anterior gestor de teclado, que generalmente se encargará de almacenar la pulsación en el buffer de teclado.

Estado de la BIOS

Aunque nuestro programa no va a hacer uso de los servicios de acceso a disco, es importante que nunca interrumpa una operación en curso ya que ello podría provocar un fallo e incluso una pérdida de datos. De igual forma, tampoco podemos activar el programa en caso de que se esté procesando una llamada a la interrupción 1 Oh, ya que el programa sí usa esa interrupción.

Al inicio del programa se ha definido un indicador, llamado InBios, cuyo valor inicial es cero. Cada vez que se entra en la interrupción 1Oh, vídeo, o 13h, disco, dicho indicador se incrementa, siendo reducido a la salida. Los gestores de estas dos interrupciones son iguales, con la única diferencia de la llamada al anterior gestor. Su código es el siguiente:

```
; Gestor de la interrupción de video

Gestor10 Proc

    ; incrementar el indicador InBios
    Inc CS:[InBios]

    ; Llamar al anterior gestor
    Pushf
    Cali CS:INT10.AnteriorGestor

    ; Decrementar el indicador InBios
    Dec CS:[InBios]

    Iret ; y volver
Gestor10 Endp

; Gestor de la interrupción de disco

Gestor13 Proc

    ; Incrementar el indicador InBios
    Tnc CS: [InBios]

    ; Llamar al anterior gestor
    Pushf
    Cali CS:INT13.AnteriorGestor

    ; Decrementar el indicador InBios
    Dec CS:[InBios]

    Iret ; y volver
Gestor13 Endp
```

En caso de que el programa utilizase alguna otra interrupción de la BIOS, sería necesario modificar su vector de interrupción haciendo que apuntase a un gestor similar a éstos, evitando así que nuestro programa interrumpa cualquier función en curso.

La activación

Como hemos visto en el gestor de la interrupción de teclado, al pulsar la combinación de activación lo que se hace es poner a 1 el indicador Activar, pero no se activa la parte residente, sino que se devuelve el control al sistema. Una vez que Activar tiene el valor 1, la activación de la parte residente vendrá dada desde el gestor de la interrupción de reloj o bien desde el de la interrupción 28h, dependiendo del estado actual del sistema.

Ambos gestores utilizan el procedimiento CompruebaEstado para ver si la activación es segura en ese momento y, en caso afirmativo, llaman al procedimiento ActivarResidente, que es el que se encarga de llevar a cabo todo el cambio de contexto, fijando la pila, PSP, etc.

En el fragmento siguiente se muestra el código del procedimiento CompruebaEstado. Se inicia examinando si hay una petición de activación y si el programa no está ya activo. Si esto es así se pasa a comprobar el estado del sistema, consultando los indicadores ErrorMode, InDos e InBios. La indicación de si la activación es satisfactoria o no se realiza a través del indicador de acarreo, que se activa o desactiva según proceda.

```
; Este procedimiento se encarga de hacer
; todas las comprobaciones necesarias para
; saber si hay que activar la parte residente
; y si es seguro hacerlo en este momento.
```

```
CompruebaEstado Proc
    Push ES      ; Preservar los registros a modificar
    Push SI
    Push AX
    Push BX

    Mov AL, CS:[Activar]
    Or AL, AL   ; Comprobar si está a 1 el indicador
    Jz NoActivar ; en caso contrario no activar

    Mov AL, CS:[Activado]
    Or AL, AL ; Comprobar si el programa está activo
    Jnz NoActivar ; de ser así no continuar

    ; Cargamos en ES:BX la dirección almacenada
    ; - en InDOS
    Mov AX, Word Ptr CS:[InDos+2]
    Mov ES, AX
    Mov BX, Word Ptr CS:[InDosJ

    ; Cargar en AL el indicador ErrorMode
    Mov AL, ES:[BX-1]
    Or AL, AL
    Jnz NoActivar ; Si está activado no continuar

    Mov AL, CS:[TnBios]
    Or AL, AL ; Si hay algún servicio BIOS en curso
    Jnz NoActivar ; no continuar
```

```

; Si la llamada es desde INT28
Mov AL, CS:[Enlnt28]
Or AL, AL
Jnz SiActivar ; Se puede activar

; en caso contrario mirar el InDos
Mov AL, ES:[BX]
Or AL, AL ; Si no está a cero
Jnz NoActivar ; no se puede activar

SiActivar: ; La activación es posible

    Clc ; Poner a cero el carry
    Jmp SalirComprobacion

NoActivar: ; La activación no es posible

    Stc ; Indicar que no se debe activar

SalirComprobacion:

    Pop DX
    Pop AX
    Pop SI ; Recuperar registros
    Pop ES

    Ret ; y volver

CompruebaF.sr.ad0 Endp

```

En caso de que la activación del programa residente sea posible, hay que llevar a cabo una de las operaciones más importantes, como es el cambio de contexto, que comprende el establecimiento del PSP, intercambio de la **pila**, modificación de los vectores de interrupción de error crítico, **Control-ínter** y **Control-C**, y en caso de que sea necesario, en éste no lo es, de la DTA.

Además, puesto que este programa accede a pantalla, es necesario obtener la dirección actual de la página activa de vídeo, salvaguardando su contenido, así como la posición del cursor. Todo este trabajo se lleva a cabo en el procedimiento **ActivarResidente**, en su primera parte, justo antes de la llamada a **TablaAscii**. La segunda parte del procedimiento se encarga de realizar la operación inversa, restituyendo el contenido de la pantalla, posición del cursor, vectores de interrupción, PSP y pila, devolviendo por fin el control al programa interrumpido.

```

; Este procedimiento se encarga de hacer
; todo lo necesario para poner en marcha
; la parte residente, activando su pila,
; su PSP, etc.

```

ActivarResidente Proc

```

Mov CS:[Activar], 0 ; Poner a cero el indicador
Mov CS:[Activado], 1 ; Indicar que está activo

```

```

; Intercambiar la pila y el PSP

Cli      ; Desactivar interrupciones mientras
; cambiamos la pila

; Preservar la dirección de la pila
; del otro programa
Mov Word Ptr CS:[PilaAnterior], SP
Mov Word Ptr CS:[PilaAnterior+2], SS

; Fijar nuestra propia pila
Lss SP, CS:[PilaResidente]

Sti

Push DS ; Preservar los valores de los registros
Push ES ; que se van a ver afectados
Push AX
Push BX
Push CX
Push DX
Push SI
Push DT

Mov AH, 51h      ; Obtener segmento del PSP activo
Int 21h
Mov CS:[PSPAnterior], BX ; Guardarlo

Mov DX, CS:[PSPResidente] ; Fijar nuestro PSP
Mov AH, 50h
Int 21h

; Modificar los vectores de interrupción de
; Ctrl-Break, Ctrl-C y error crítico

Mov CX, 3 ; Son tres vectores

Mov SI, Offset INT1B ; a partir de INT1B

BucleInt1:
Mov AL, CS:[SI].Blqlnt.Numero
Mov AH, 35h ; Obtenemos el contenido
Int 21h      ; actual del vector

; y lo guardamos
Mov Word Ptr CS:[SI].Blqlnt.AnteriorGestor[0], BX
Mov Word Ptr CS:[SI].Blqlnt.AnteriorGestor[2], ES

; Cargamos en DS:DX la dirección de
; nuestro propio gestor
Mov DX, Word Ptr CS:[SI].Blqlnt.NuevoGestor[0]
Mov DS, Word Ptr CS:[SI].Blqlnt.NuevoGestor[2]
Mov AH, 25h ; y la escribimos en el vector
Int 21h

```

```

Add SI, 9 ; Pasar al siguiente bloque
Loop BucleIntl ; y continuar

Mov AX, SEGMENTO_BIOS ; Mirar si estamos
Mov ES, AX ; en el modo de video adecuado
Cmp Byte Ptr ES:[MODO_VIDEO], 3
Je ModoAdecuado

Mov CL, 10
Cali Pitido ; Si no es asi provocar un pitido
Jmp ModoNoAdecuado ; y no continuar

ModoAdecuado:

; Obtener en SI la dirección en la que
; se almacena la posición del cursor
Mov SI, 50h
Mov AL, Byte Ptr ES:[PAGINA_VIDEO]
Mov CS:[PaginaActiva], AL
Xor AH, AH
Add SI, AX

; Obtener la posición del cursor
Mov AX, ES:[SI]
Mov CS:[Cursor], AX ; y guardarla

; Cargamos en CX el número de bytes
Mov CX, BYTF.ñ_PANTAT,T,A

; Cargamos en SI el desplazamiento
; en el segmento de memoria de video
; de la página actual
Mov SI, ES:[OFFSET_PAGINA_ACTUAL]
Mov CS:[OffsetPagina], SI

Mov AX, SEG_PANTALLA ; DS:SI apuntan al contenido
Mov DS, AX ; actual de la pantalla

Mov AX, Seg Pantalla
Mov ES, AX ; ES:DI apuntan a nuestro bloque
Mov DI, Offset Pantalla ; de memoria

Cid ; Copiar hacia adelante

Rep Movsb ; todo el contenido

Cali TablaAscii ; Mostrar la tabla ASCII

; Realizamos la operación inversa,
; restituyendo el contenido de la pantalla

Mov AX, SEGMENTO_BIOS
Mov ES, AX

; Obtenemos de nuevo el tamaño en bytes
Mov CX, BYTES_PANTALLA

```

```

; y el desplazamiento de la página activa
; en el segmento de memoria de video
Mov DI, ES:[OFFSET_PAGINA_ACTUAL]

Mov AX, SEG_PANTALLA
Mov ES, AX ; ES:DI apuntan ahora a la página
            ; actual de video

Mov AX, Seq Pantalla
Mov DS, AX ; DS:SI apuntan a nuestro bloque,
Mov SI, Offset Pantalla ; que tiene los datos

Cid : Copiar hacia adelante

Rep Movsb ; todo el contenido

; Restituir la posición del cursor
Mov DX, CS:[Cursor]
Mov AH, 2
Mov BH, CS:[PaginaActiva]
Int 10h

```

ModoNoAdecuado:

```

; Una vez que hemos terminado debemos dejar
; el sistema tal y como lo encontramos al
; provocar la interrupción.

Mov CX, 3      ; Restituimos los tres vectores
Mov SI, Offset INT1I3

```

BucleInt2:

```

; Cargamos en AL el número de interrupción
Mov AL, CS:[SI].Blqlnt.Numero
; y en DS:DX la dirección del gestor original
Mov DX, Word Ptr CS:[SI].Blqlnt.AnteriorGestor[0]
Mov DS, Word Ptr CS:[SI].Blqlnt.AnteriorGestor/2J

```

```

Mov AH, 25h ; lo escribimos en el vector
Int 21h

```

```

Add SI, 9 ; Pasar al siguiente bloque
Loop BucleInt2 ; y repetir

```

```

; Dejamos el PSP anterior
Mov BX, CS:[PSPAnterior]
Mov AH, 50h
Int 21h

```

```

Pop DI      ; Recuperamos el contenido
Pop SI      ; de los registros
Pop DX
Pop CX
Pop BX
Pop AX

```

```

Pop ES
Pop DS

Cli
Lss SP, CS:[PilaAnterior] ; Recuperamos la pila
Sti

Mov CS:[Activado], 0 ; y terminamos
Ret

ActivarRe.sidente Kndp

```

Mostrar la tabla de códigos ASCII

La mayor parte del código del programa TDLASCII no tiene nada que ver con su funcionalidad última, que es mostrar una tabla de códigos ASCII y los caracteres correspondientes, permitiendo seleccionar cualquiera de ellos. El código encargado de hacer esto se ha aislado en el procedimiento TablaAscii que se puede ver a continuación.

En este procedimiento se accede directamente a la memoria de vídeo de la página actual para dibujar una tabla, conteniendo un carácter, un guión y su código correspondiente. Una vez que la tabla está dibujada se sitúa el cursor en uno de los caracteres y se permite la utilización de las teclas de desplazamiento del cursor para moverse a cualquier otro.

Es posible abandonar la tabla ASCII, devolviendo el control al programa interrumpido, bien pulsando la tecla Esc, que simplemente la cierra, o bien pulsando la tecla **Insert**, caso éste en que se calculará el código del carácter elegido y se insertará en el buffer de teclado, de tal forma que al devolver el control al programa que se estaba ejecutando éste lea de forma inmediata el carácter.

```

; Este procedimiento contiene el código
; con la funcionalidad del programa, en
; este caso mostrando una tabla de códigos
; ASCII.

```

TablaAscii Proc

```

; Cargamos en ES:DI la dirección
; de pantalla
Mov AX, SEC_PANTALLA
Mov ES, AX
Mov DI, CS:[OffsetPagina]

Mov AH, ATTR_TABLA ; Atributo de texto

Mov AL, ESQ1 ; Primera esquina
Stosw

Mov AL, HORZ ; Carácter horizontal
Mov CL, ANCHO_TABLA ; para dibujar la
Rep Stosw ; parte superior

```

```

Mov AL, ESQ3

Stosw ; Otra esquina

Xor DL, DL ; Contador de lineas

BucleLineas:
    ; Tomar la dirección base
    Mov DI, CS:[OffsetPagina]

    ; Y sumar H1 incremento necesario
    ; para llegar a la linea actual
    Mov AL, BYTES_LINEA
    Inc DL
    Muí DL ; BYTESJLINEA * Linea actual
    Dec DL
    Add DI, AX

    ; Cargamos de nuevo el atributo
    Mov AH, ATTR_TABLA

    Mov AL, VERT ; Margen izquierdo
    Stosw

    Mov AL, COLUMNAS ; columnas por fila
    Muí DL ; Obtenemos en AL el código del primer
    ; carácter de esta linea

    Mov AH, ATTR_TAD.TiA , - Preparamos el atributo
    Mov CX, COLUMNAS ; y el contador del bucle

BucleColumnas:
    Stosw ; Imprimimos el carácter

    Push AX ; Guardamos el código

    Mov AL, '-' ; Un guión une a cada carácter
    Stosw ; con su código

    Pop AX ; Recuperamos código

    ; Mostramos el código ASCII correspondiente
    Cali ImprimeCodigo

    Inc AL ; Pasamos al siguiente carácter

    Oí AL, AL ; Ver si ya están todos
    Jz FinTabla ; si es así no continuar

    Loop BucleColumnas ; Hasta terminar la fila

    Mov AL, VERT ; Margen derecho
    Stosw

    Inc DL ; Incrementamos el contador de lineas

```

```

Jmp BucleLineas ; Hasta terminar toda la tabla

FinTabla: ; Ya se han mostrado los 256 caracteres

Mov CX, 18 ; Resto de espacios en la fila
Mov AL, ' ' ; rellenarlo
Rep Stosw

Mov AL, VERT ; Marqen derecho
Stosw

; Tomar la dirección base
Mov DI, CS:[OffsetPagina]

; Y sumar el incremento necesario
; para llegar a la última linea
Mov AL, BYTES_LINEA
Inc DL
Inc DL
Muí DL
Add DI, AX

; Cargamos de nuevo el atributo
Mov AH, ATTR_TABLA

Mov AL, ESQ2 ; Una esquina
Stosw

Mov AL, HORZ ; La linea inferior
MOV CX, ANCHO_TABLA
Rep Stosw

Mov AL, ESQ4 ; y la otra esquina
Stosw

; Hemos terminado de dibujar la tabla, ahora
; entramos en el bucle de selección de carácter

; Dejar ES apuntando al segmento de datos
; del BIOS
Mov AX, SEGMENTO_BIOS
Mov ES, AX

```

BucleTeclado:

```

; Calcular la columna de pantalla
; correspondiente a la columna actual de
; la tabla
Mov AL, CS:[Columna]
Mov DL, CARACTERES_ELEMENTO
Muí DL
Inc AL
Mov DI,, AL ; Columna del carácter elegido

Mov DH, CS:[Fila]
Inc DH ; Fila del carácter elegido

```

```
Mov BH, CS:[PaginaActiva]
Mov AH, 2
Int 10h      ; Posicionamos el cursor
```

EsperaTecla:

```
Xor AH, AH
Int 16h ; Esperamos la pulsación de una tecla

; Según la tecla que se haya
; pulsado pasamos el control
; a una etiqueta u otra.

Cmp AH, CURSOR_ARRIBA
Je Arriba
Cmp AH, CURSOR_IZQUIERDA
Je Izquierda
Cmp AH, CURSOR_DERECHA
Jo Derecha
Cmp AH, CURSOR_ABAJO
Je Abajo
Cmp AH, TECLA_ESCAPE
Je SalirBucleTeclado
Cmp AH, TECLA_INSERTAR
Je Insertar

; Si no es ninguna de las teclas anteriores
Mov CL, 8 ; generar un pitido
Cali Pitido
Jmp EsperaTecla ; y volver a esperar una tecla
```

Arriba: ; Se ha pulsado el cursor arriba

```
Cmp CS:[Fila], 0 ; Si ya estamos en la primera
Jz EsperaTecla ; fila, no continuar

Dec CS:[Fila) ; Decrementar la fila
Jmp BucleTeclado ; y volver
```

Izquierda: ; Se ha pulsado el cursor a la izquierda

```
; Si ya estamos en la primera
Cmp CS:[Columna], 0
Jz EsperaTecla ; columna, no continuar

Dec CS:[Columna] ; Decrementar la columna
Jmp BucleTeclado ; y volver-
```

Derecha: ; Se ha pulsado el cursor a la derecha

```
; Si estamos en la última columna de la fila
Cmp CS:[Columna], COLUMNAS-1
Je EsperaTecla ; ignorar la pulsación

Inc CS:[Columna] ; Incrementar la columna
Jmp BucleTeclado ; y volver
```

```

Abajo: ; Se ha pulsado el cursor abajo

; Si estamos en la última fila de la tabla
Cmp CS:[Fila], FILAS-1
Je EsperaTecla ; ignorar la pulsación

Inc CS:[Filal : Incrementar la fila
Jmp BucleTeclado ; y volver

; Insertar la tecla en el buffer de teclado
Insertar:

Mov AL, CS:[Fila] ; Tomamos la fila actual
Mov DL, COLUMNAS ; la multiplicamos por el
Muí DL ; número de columnas en cada fila
Add AL, CS:[Columna] ; y le sumamos la columna

; Ya tenemos en AL el código ASCII del
; carácter elegido en la tabla

; Obtenemos en BX el puntero de escritura
; en el buffer de teclado
Mov BX, ES:[PESCRITURAJTECLADO]
Mov ES:[BX], AL ; insertamos la tecla

Inc BX ; Incrementamos el puntero
; e insertamos el sean code
Mov Byte Ptr ES:[BX], 0

Inc BX ; Volvemos a incrementar el puntero
,- Ver ej estamos al final del buffer de teclado
Cmp BX, FIN_BUFFER_TECLADO
Jb NoHaySalto ; Si no es así saltar
; En caso necesario apuntar al inicio del buffer
Mov BX, INICIO_BUFFER_TECLADO

NoHaySalto:
; Escribir el nuevo puntero de escritura
Mov ES: [PESCRITURA_TECLAD01 , BX

SalirBucleTeclado:

Rct ; Salir

TablaAscii Endp

```

Simplemente modificando el código del procedimiento TablaAscii podríamos construir un programa residente con cualquier otra finalidad, no siendo necesario retopear el resto del código prácticamente en nada. Las constantes definidas al principio nos permiten elegir la tecla de activación, el identificador con el que se conocerá el programa mediante la interrupción 2Fh y el tamaño de la pila. En caso de que el nuevo código necesitase realizar alguna operación con archivos, habría que añadir al procedimiento ActivarResidente el código necesario para obtener la DTA del programa interrumpido e intercambiarla por la nuestra.

Otros gestores de interrupción

El código del gestor de la interrupción múltiple es similar al de otros programas de puntos previos, facilitando básicamente la identificación del programa y la información necesaria para facilitar su desinstalación.

Aunque el identificador del programa dentro de la interrupción está definido como una constante al principio del código, con el fin de permitir su modificación de forma simple, también sería posible modificar el código del programa para que el identificador no fuese uno fijo, sino que se pudiera seleccionar uno cualquiera dentro de un rango, evitando de esta forma utilizar el mismo identificador que otro programa que ya estuviese instalado.

Los gestores de las interrupciones de **Control-ínter**, **Control-C** y error crítico se limitan simplemente a ignorar la interrupción, devolviendo el control de forma inmediata. Al activar el programa podrá comprobar que la pulsación de las dos teclas anteriores simplemente genera un pitido.

Procedimientos adicionales

Además de los procedimientos descritos, en el código encontrará otros adicionales, como **Pitido**, que genera un pitido durante un segundo, o **ImprimeCodigo**, que convierte el código facilitado en AL en tres dígitos y lo imprime. Estos procedimientos son de utilidad general, pudiendo ser eliminados del código en caso de no utilizarse y ser usados en cualquier otro programa.

Para producir el pitido, se utilizan los registros del PPI para manipular el estado del altavoz del sistema.

Es posible controlar la frecuencia y la duración, de hecho la frecuencia se facilita en el registro CL antes de invocar a **Pitido**.

El código de este procedimiento es el siguiente:

```
; Este procedimiento tiene como única
; finalidad generar un pitido durante
; un segundo, para indicar algún tipo
; de error.
; El tono del pitido dependerá del
; valor que se facilite en CL.
```

```
Pitido Proc
Push AX ; Preservamos registros a modificar

; Ponemos a cero el contador de tiempo
Mov CS:[ContadorTiempo], 0

; Fijamos el tipo de activación
Mov AL, 10110110b
Out 43h, AL
Xor AL, AL ; y la frecuencia del pitido
```

```

Out 42h, AL
Mov AL, CL ; CL tiene la frecuencia
Out 42h, AL

In AL, PPI_PÜRT_B ; Activamos el pitido
Or AL, 3
Out PPI_PORT_B, AL

BuclePitido:
    ; Esperamos aproximadamente un segundo
    Cmp Byte Ptr CS:[ContadorTiempol], 18
    Jb BuclePitido

    In AL, PPI_PORT_B      ; Desactivamos el pitido
    And AL, 252
    Out PPI PORT B, AL

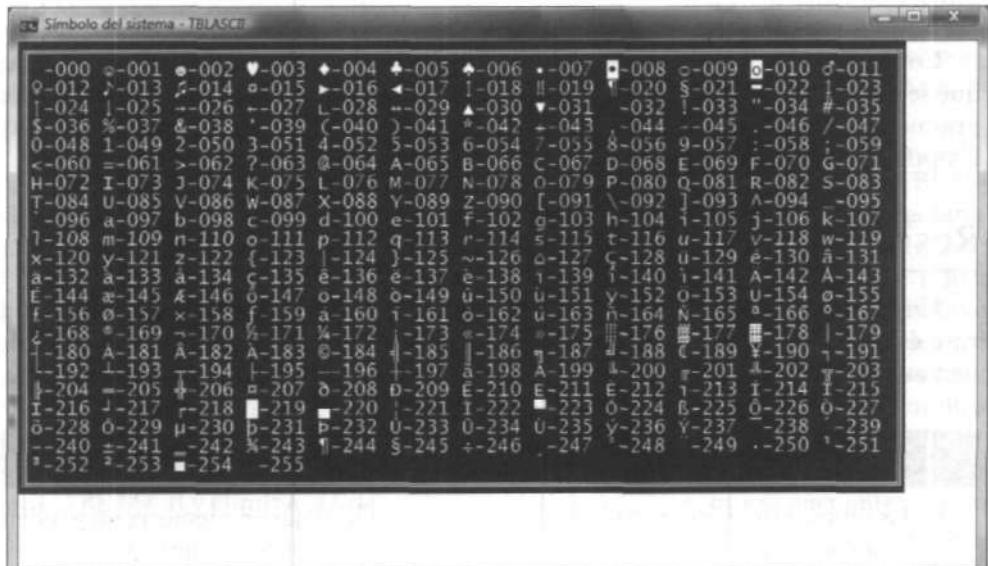
    Pop AX      ; Recuperamos registros

    Ret          ; Volver
Pitido Endp

```

Funcionamiento del programa

Una vez que el programa ha sido cargado en memoria, quedando residente, bastará con pulsar la combinación correspondiente, en este caso **Control-T**, para activarlo. En la figura 27.12 se muestra el aspecto del programa.



La ocupación del programa en memoria es de 5824 bytes, de los cuales 4000 corresponden al espacio reservado para salvaguardar el contenido actual de la pantalla y 304 al bloque de entorno, por lo que la ocupación real del código y datos del programa es de 1520 bytes. (La ocupación del bloque de entorno depende del contenido de variables como PATH, por lo que en su sistema seguramente será distinta.)

Si asumimos que el resto de la memoria de vídeo no está siendo usada, al ser el modo actual de texto, podríamos utilizarla para guardar el contenido de la página actual, evitando así esos 4000 bytes de más.

El bloque de entorno, que no es utilizado por la parte residente del programa, también puede ser liberado antes de salir de la instalación, en lugar de hacerlo durante la desinstalación. Con estas modificaciones la ocupación del programa quedaría en los citados 1520 bytes.

Aplicaciones residentes y Windows

Durante muchos años, el único sistema operativo utilizado por los usuarios de sistemas compatibles PC ha sido el DOS, aunque hoy tenga una presencia más bien testimonial. La aparición de Windows 3.0 puede ser considerada el punto de inflexión para el uso de entornos gráficos sobre PC, ya que versiones anteriores de Windows y otros sistemas no tuvieron demasiado éxito. En la actualidad la mayoría de los usuarios utilizan Windows, bien sea en sustitución de DOS o como complemento a éste, lo que causa que en el diseño de un programa residente deba tenerse en cuenta este aspecto.

En los puntos siguientes vamos a conocer las limitaciones que impone Windows al funcionamiento de nuestro programa residente, y los métodos que tenemos al alcance para evitar tales límites.

Con el fin de realizar diversas pruebas, nos vamos a servir del programa TBLASC11 que se desarrollado en el punto previo. Si desea mantener la versión original de este programa, asegúrese de realizar una copia antes de llevar a cabo las modificaciones que se propondrán en los siguientes puntos.

Residentes globales y locales

Un programa residente puede ser instalado en memoria antes de iniciar Windows, caso éste en el que se habla de un residente global; o bien posteriormente, abriendo una ventana DOS, encontrándonos entonces con un residente local a esa ventana.

Nota

En las últimas versiones de Windows ya no es posible alojar en memoria un residente con antelación, pero en hace años se cargaba primero el DOS, a continuación los residentes que fuesen necesarios y, finalmente, se iniciaba Windows.

La mayoría de los programas residentes han sido diseñados teniendo en cuenta la filosofía de funcionamiento del DOS, que no permite ejecutar simultáneamente más de una aplicación. Por ello al cargar un programa residente en una ventana DOS, el comportamiento en la mayoría de los casos será idéntico al mostrado por ese mismo programa ejecutándose fuera de Windows, ya que éste inicializa una máquina virtual (VM en adelante) para cada ventana DOS, lo que garantiza que el programa crea que se está ejecutando él sólo, y que tiene a su disposición todos los recursos del sistema.

Si después de haber cargado un programa residente de forma local, en una ventana DOS, abrimos una segunda ventana DOS y pulsamos la combinación de teclas de activación, veremos que no ocurre nada. Esto se debe a que la segunda ventana DOS no conoce el estado de la primera, y por tanto no sabe que el programa residente está cargado ni nada acerca de su activación.

Puede realizar una prueba cargando Windows y ejecutando el programa TBLASCII anterior en una ventana DOS, comprobando que funciona correctamente. A continuación abra una segunda ventana DOS y luego pulse **Alt-T**, observará que no hay ninguna reacción.

En contraposición a lo que ocurre con un residente local, si la instalación del programa se realiza antes de haber cargado Windows, convirtiéndose por lo tanto en un residente global, cualquier ventana DOS que abramos tendrá acceso a dicho programa, sin necesidad de instalarlo. Es más, si el programa está correctamente diseñado no será posible la reinstalación.

Problemas de un residente global

Si antes de iniciar Windows (en las versiones en que esto es posible) instala el programa TBLASCII, y posteriormente abre múltiples ventanas DOS, podrá comprobar que la activación es posible desde cualquiera de las ventanas. Sin embargo, no será posible la activación simultánea en más de una ventana, ya que el programa está diseñado para que no sea posible la activación múltiple.

Esto tiene sentido en DOS, donde no es posible la ejecución simultánea de más de un programa, pero en Windows, y en concreto en una ventana DOS, el programa TBLASCII debería permitir la activación siempre y cuando ésta no se solicitase en una ventana en la que ya estuviese activo.

Cuando el programa TBLASCII se activa, el indicador Activado se pone 1, lo que impide que mientras que está en funcionamiento pueda ser activado de nuevo. Lo que ocurre es que cuando intentamos la activación desde otra ventana DOS, el indicador Activado se encuentra a 1, y por lo tanto no es posible la activación. De esto se deduce que el programa residente está utilizando los mismos datos siempre, en las múltiples ventanas.

Para verlo aún más claro hagamos la siguiente prueba. Edite el código fuente del programa TBLASCII y elimine la línea Mov CS : [Activado] , 1, que se encuentra al principio del procedimiento Activar/Residente. Una vez obtenido el ejecutable instálelo, a continuación inicie Windows y abra dos ventanas DOS. En la primera borre la pantalla y a continuación pulse **Alt-T** para hacer aparecer la tabla de códigos ASCII. Seguidamente

pase a la segunda ventana, obtenga un directorio por pantalla y pulse **Alt-T** para activar el programa, que en este caso mostrará la tabla sin problemas, ya que no tiene constancia de que haya otra copia activa.

Hasta aquí todo parece estar correcto, pero si ahora vuelve a la primera ventana DOS y pulsa la tecla Esc, para hacer desaparecer la tabla ASCII, observará que la pantalla no aparece vacía, tal y como la dejó, sino que en ella se muestra el directorio que obtuvo en la segunda ventana DOS. Está claro que aunque el programa se activa dos veces, tan sólo existe una copia de los datos que se utilizan, y por eso al salvaguardar el contenido de la segunda ventana DOS se sobrescribió el de la primera.

Si realiza diversas pruebas activando TBLASCII en dos o más ventanas DOS se puede llegar a encontrar algunos problemas, y es que al compartir los mismos datos no sólo se sobrescribe el contenido de la pantalla, sino que ocurre lo mismo con el resto de los elementos que utiliza el programa residente, incluidos tanto los vectores de interrupción de **Control-ínter**, **Control-C** y error crítico, que se establecen al inicio y restablecen al desactivar, como los datos acerca del PSP y la pila.

En resumidas cuentas, el resultado final de las pruebas puede terminar con un fallo y el cierre de la ventana DOS en el mejor de los casos, y con un bloqueo total del sistema en el peor.

Los problemas pueden ir a más si, por ejemplo, intentamos desinstalar el programa en una ventana mientras está activo en otra. En TBLASCII no se contempla la posibilidad de que esto pueda ocurrir, ya que mientras que el programa está activo no es posible introducir en la línea de comandos del DOS la orden necesaria para la desinstalación. Pero en Windows sí que puede ocurrir, al existir múltiples ventanas DOS.

Iniciación de Windows

Ante la necesidad del usuario de utilizar el entorno Windows, un programa residente puede actuar básicamente de dos formas: inhibiendo su funcionamiento, informando al usuario de que no es posible su uso mientras esté Windows en funcionamiento, o bien adecuándose a la forma de trabajo Windows, lo que implica algo más de trabajo.

En cualquiera de los casos anteriores es básico que el programa residente detecte el arranque de Windows, momento en el cual se tomará el camino oportuno. Por suerte, para poder hacer esto ya tenemos parte del trabajo realizado, ya que un programa residente que deseé obtener una notificación del arranque de Windows lo primero que tiene que hacer es interceptar la interrupción **2Fh**, algo que TBLASCII ya hace.

Cuando Windows es cargado desde el DOS, antes de pasar al proceso de inicialización genera una interrupción **2Fh**, con el código **1605h** en AX. En ese momento el programa residente puede tomar varios caminos. En caso de que el programa residente no deba estar instalado al iniciar Windows, deberemos mostrar un mensaje indicándolo y devolveremos el control almacenando en CX un valor distinto de cero, lo que causará que el inicio de Windows se detenga.

En el CD-ROM podrá encontrar una versión de TBLASCII, con el nombre **TBLASCI**, que hace precisamente esto, impedir el arranque de Windows hasta en tanto no se desinstale el programa.

Funcionamiento bajo Windows

Un programa residente que decida ejecutarse bajo Windows debe escoger un camino para hacerlo apropiadamente, sin plantear los problemas expuestos anteriormente. Uno de estos caminos consiste en no permitir la activación desde múltiples ventanas DOS, algo que el programa TBLASC11 original ya hace. Sin embargo, ésta no es la mejor opción porque el usuario puede tener TBLASCII activado en una ventana mientras está trabajando en otra, momento en el cual necesita también la tabla de códigos ASCII. Si elegimos el método de no permitir la múltiple activación, el usuario se verá forzado a cambiar a la primera ventana, desactivar el programa, cambiar de nuevo a la segunda y proceder como lo hubiese hecho en un principio.

Otra de las posibilidades consiste en identificar la VM en la que se activa el programa, de tal forma que si posteriormente se intenta la activación desde otra ventana, otra VM, sea posible conmutar a aquellas en la que el programa ya está activado. Aunque esta solución puede no ser la apropiada para el programa TBLASCII, sí resultará de utilidad en otros casos.

Por último tenemos la opción de solicitar a Windows, en el momento en que éste se inicia, que cree un bloque de datos independiente para cada ventana DOS, de tal forma que cada vez que el programa residente sea activado, se encuentre con datos independientes para cada VM. Esta tercera posibilidad sería la más interesante para conseguir que TBLASCII funcione adecuadamente.

A vueltas con las VM

Como se dijo anteriormente, cada una de las ventanas DOS se ejecuta en una VM diferente. Cada VM que abre Windows cuenta con un identificador, que puede ser obtenido mediante el servicio 1683h de la interrupción múltiple. El identificador es un número de 16 bits que se devuelve en el registro BX.

Mediante la función 16 8Bh de la misma interrupción es posible forzar a Windows el cambio a una VM específica, concretamente a aquella cuyo identificador se facilita en el registro BX.

Nota

Las dos funciones de la interrupción ? Fh que son citadas en este apartado funcionan en Windows 3.1, Windows 95, Windows 98 y Me, pero el cambio de núcleo que se produjo a partir de Windows 2000, y posteriormente Windows XP y Vista, restringe ciertas operaciones por parte de las aplicaciones, entre ellas el cambio de la ventana activa.

Partiendo de estas dos funciones, no sería demasiado complicado modificar el código del programa TBLASCII de tal forma que, al activarse por primera vez, guardase el identificador de VM, de tal forma que al intentar una segunda activación desde otra

ventana fuese posible commutar a la primera. Esto es lo que hace una segunda versión de TBLASCII que encontrará en el CD-ROM con el nombre TBLASC2.ASM.

Para comprobar el funcionamiento de TBLASC2, instálelo antes de iniciar Windows. Posteriormente abra dos o más ventanas DOS a pantalla completa, active TBLASC2 en una de ellas, y después intente hacerlo desde cualquiera de las otras.

Copias individuales de datos

Como vimos anteriormente, un programa residente que ha sido cargado de forma local, después de entrar en Windows, no se encuentra con ningún problema, ya que cada ventana DOS cuenta con una copia del programa. Un residente global, por el contrario, comparte todos los datos entre las distintas ventanas DOS.

Para evitar los problemas de funcionamiento que se derivan de este hecho, debemos pedir a Windows que cada vez que se abra una ventana DOS prepare un bloque de datos individual para nuestro programa, de forma que, aunque éste sea global, los datos que a nosotros nos interesen sean locales.

Con el fin de conseguir esto tendremos que definir dos estructuras de datos, según el patrón que se muestra a continuación. Cuando el gestor de la interrupción múltiple de nuestro programa reciba el mensaje de iniciación de Windows, servicio 1605h visto anteriormente, lo primero que debe hacer es pasar el control al siguiente gestor, cuya dirección obtuvimos durante la instalación.

Ala vuelta deberemos salvaguardar el contenido de la pareja de registros ES: BX en el campo NextDevPtr de la estructura R lo que inicialización, con el fin de crear una cadena con estructuras que le sirva a Windows para identificar a todos los programas que respondan. A continuación cargaremos en esa misma pareja de registros, ES : BX, la dirección de nuestra propia estructura BloqueInicializacion, y devolveremos el control mediante la instrucción iret.

```
; Estructura para indicar la dirección
; y tamaño de cada bloque de datos.
BloqueDatos Struc
    DireccionBloque      Dd  ?
    TamanoBloque         Dw  ?
BloqueDatos EndS

; Estructura a devolver en respuesta a la
; llamada a la INT 2Fh con el servicio 1605h
BloqueTnicializacion Struc
    VersionWin           üb   3, 10
    NextDevPtr            Dd   ?
    VirtDevFilePtr        Dd   0
    ReferenceData         Dd   0
    InstanceDataPtr       Dd   ?
BloqueTnicializacion EndS
```

Un programa residente puede tener uno o más bloques de datos susceptibles de ser creados automáticamente por Windows en cada VIVÍ. En el caso del programa TBLASCII,

nos encontramos con un primer bloque que contiene todos los datos definidos al principio y un segundo bloque que comprende la pila, espacio que tampoco debe ser compartido simultáneamente por dos copias del programa ya que uno podría intentar utilizar direcciones almacenadas por el otro, con el consiguiente problema.

Por cada bloque de datos que necesitemos crear definiremos una estructura Bloque-Datos, almacenando en el miembro DireccionBloque la dirección de inicio de ese bloque, y en TamanoBloque el número de bytes que ocupa. Dado que cada programa puede tener un número de bloques distintos, para indicar a Windows el final de la cadena crearemos una estructura BloqueDatos con los dos miembros anteriores a cero.

Dado que la dirección que se pasa a Windows en ES : BX es la de la estructura BloqueInicializacion, en ésta debe existir alguna referencia a la lista de estructuras BloqueDatos. Concretamente el miembro instanceDataPtr es el encargado de almacenar la dirección de inicio de la mencionada lista.

En el código siguiente se muestran las modificaciones que habría que efectuar en el gestor de la interrupción múltiple del programa TBLASCII con el fin de que cada VM cuente con su propia copia de los datos y la pila.

En el CD-ROM podrá encontrar el programa completo con esta modificación, con el nombre TBLASC3.ASM.

```
Gestor2F Proc
    Cmp AX, 1605h      ; ¿Se va a iniciar Windows?
    ; Si no es así proceder normal
    Jne ProcesoNormal

    Pushf                ; Llamar al anterior gestor
    Cali CS:INT2F.AnteriorGesLor

    ; Almacenar la dirección del
    ; gestor anterior en la estructura
    Mov Word Ptr CS:Inicializacion.NextDevPtr[0], BX
    Mov Word Ptr CS:Inicializacion.NextDevPtr[2], ES

    ; Obtener la dirección de inicio y tamaño
    ; del bloque de datos y almacenarlo
    Mov BX, Offset INT09
    Mov Word Ptr CS:DatosTSR.DireccionBloque, BX
    Mov Word Ptr CS:DatosTSR.DireccionBloque12J, CS
    Mov CS:DatosTSR.TamanoBloque, BytesDatos

    ; Obtener la dirección de inicio y tamaño
    ; de la pila y almacenarlo
    Mov BX, Offset EspacioDePila
    Mov Word Ptr CS:PilaTSR.DireccionBloque, BX
    Mov Word Ptr CS:PilaTSR.DireccionBloque[2], CS
    Mov CS:PilaTSR.TamanoBloque, TAMANO_PILA

    .- Tomar la dirección de las estructuras anteriores
    ; y almacenarla en el bloque de inicialización
    Mov BX, Offset DatosTSR
    Mov Word Ptr CS:Inicializacion.InstanceDataPtr, BX
    Mov Word Ptr CS:Inicializacion.InstanceDataPtr[2], CS
```

```

Push CS ; ES:BX apuntando a nuestra propia
Pop ES ; estructura de inicialización
Mov BX, offset Inicialización

Iret ; terminar

```

ProcesoNormal:

```

; Comprobar si es para nosotros
Cmp AH, IDPROGRAMA
Jne NoLoes

```

Dados estos pasos no tendremos que preocuparnos más de los problemas anteriores, ya que el programa funcionará como si se hubiese cargado localmente. Esta versión de TBLASC11 podría considerarse prácticamente definitiva, ya que funciona correctamente tanto en DOS como en Windows, y tanto si se carga de forma global como si se hace de forma local.

Secciones críticas

A diferencia de lo que ocurre en DOS, en el entorno Windows es posible ejecutar múltiples aplicaciones de forma simultánea, y es Windows el que se encarga de ir activando cíclicamente cada programa. Esto quiere decir que nuestro programa residente puede ver interrumpida su ejecución en cualquier momento, incluso en algunos puntos que pueden ser críticos, como el momento en que se intercambia la pila del programa interrumpido por la del residente. Windows dispone de dos servicios, a los que se denomina *Begin Critical Section* y *End Critical Section*, que nuestro programa puede utilizar mediante la interrupción múltiple, concretamente a través de los servicios 1681h y 1682h, cuya finalidad es evitar que mientras se ejecuta el código comprendido entre las dos llamadas el control pueda ser transferido a otro proceso.

Por lo tanto, acciones tales como el intercambio de la pila pueden precederse de una llamada al servicio 1681h de la interrupción múltiple, que se complementará mediante una llamada al servicio 1682h en cuanto dicho cambio se haya efectuado.

Al igual que ocurre con la desactivación de las interrupciones, que deben ser activadas tan pronto como sea posible, una llamada a *Begin Critical Section* debe ser correspondida en el menor tiempo con una llamada a *End Critical Section*.

Ejecución de programas Windows

Muchos programas residentes, que son útiles en DOS, no tienen mucho sentido en Windows, ya que este entorno permite la ejecución simultánea de varias tareas. Suponga que ha diseñado un programa residente que muestra un calendario en pantalla, si el usuario inicia Windows y desea usar su programa, no tendrá más remedio que abrir una ventana DOS para poder activarlo. Desde luego sería mucho más fácil iniciar una utilidad similar que funcionase en el propio entorno de Windows, de tal forma que con una pulsación de tecla se pudiese acceder a él, de forma similar a como se haría en DOS.

Además del mensaje de iniciación de Windows, comentado anteriormente, nuestro gestor de la interrupción múltiple también recibirá en el momento de la carga una llamada con el valor 160Bh en AX. En respuesta a esta llamada el programa puede identificarse, facilitando una serie de datos a Windows, y además puede indicar a éste que cargue un programa, librería o controlador.

En el código que hay más adelante se muestra la estructura necesaria para responder al mensaje de identificación. El contenido de cada uno de los campos es el siguiente:

- **SiguienteTSR**: En este campo almacenaremos el contenido de la pareja de registros ES : DI antes de dar cualquier otro paso.
- **SegmentoPSP**: Segmento del PSP del programa residente.
- **VersionEstructura**: Siempre contendrá el valor que se muestra en el código siguiente.
- **Indicadores**: Este campo contendrá un valor que le indicará a Windows la acción que deseamos llevar a cabo, de entre las mostradas en la tabla 27.8.
- **Visualización**: Mediante este campo podremos pasar a Windows indicadores de visualización, como SWSHOWMAXIMIZED, por ejemplo.
- **Comando**: Contendrá la dirección de una cadena contenido la línea de comandos a pasar a Windows, terminada con un carácter nulo.
- **FirmaTSR**: Contendrá la dirección de un bloque identificativo, con una palabra al principio que indique la longitud y una cadena a continuación con el nombre del programa.
- **PunteroDatos**: Este campo puede contener una dirección cualquiera que el programa necesite, Windows no hace uso alguno de este dato.

```
; Bloque de datos para indentificación
; del programa y cargar una aplicación
; Windows.

RLoqueldentificacion Struc
    SiguienteTSR        Dd      ?
    SegmentoPSP         Dw      ?
    VersionEstructura   Dw      100h
    Indicadores         Dw      1
    Visualización       Dw      0
    Comando             Dd      ?
    Reservado           Dd      0
    FirmaTSR            Dd      ?
    PunteroDatos        Dd      ?
Bloqueldentificacion Ends

Identificación Bloqueldentificacion <>

CadenaComando     Db      'TBLWIN.EXE', 0

BloqueFirmaTSR     Dw      15
                    Db      'Tabla ASCII.', 0
```

Tabla 27.8. Comandos de iniciación.

Comando	Significado
0	No realizar ninguna acción.
1	Ejecutar una aplicación Windows.
2	Cargar una biblioteca de enlace dinámico.
4	Cargar un controlador.

En el código mostrado a continuación puede ver cómo se ha modificado el gestor de la interrupción múltiple del programa TBLASCII para responder al mensaje de identificación y, utilizando la definición de datos del fragmento anterior, cargar y ejecutar el programa TBLWIN.EXE. Este programa lo único que hace es mostrar un mensaje en una ventana Windows y su único fin es servir de ejemplo.

En la práctica el programa a cargar sustituiría a TBLASCII con un programa Windows que tuviese la misma finalidad.

El programa TBLWIN.EXE está incluido en el CD-ROM y, para que pueda ser ejecutado en el momento en que se inicia Windows, deberá copiarlo en su sistema en un directorio que se encuentre en el camino de búsqueda. En caso de que al iniciar Windows no se encuentre el archivo TBLWIN.EXE, aparecerá una ventana informativa indicando que no es posible ejecutar el programa especificado por el programa residente. Observe en este caso que Windows ha identificado perfectamente al programa TBLASCII, gracias a que como respuesta al mensaje 160Bh hemos facilitado los datos necesarios.

```
Gestor2F Proc
    Cmp AX, 160Bh ; ¿Petición de identificación?
    Jne CargaWindows ; Si no saltar

        ; Almacenar ES:DI en SiguienteTSR
    Mov Word Ptr CS:Identificación.SiguienteTSR, DI

    Mov Word Ptr CS:Identificación.SiguienteTSR[2], ES

    Push AX

        ; Obtener el segmento del PSP y
        ; almacenarlo en la estructura
    Mov AX, CS:PSPResidente
    Mov CS:Identificación.SegmentoPSP, AX
        ; Pasar la dirección de la cadena de comando
    Mov AX, Offset CadenaComando
    Mov Word Ptr CS:IdenLiricacion.Comando, AX
    Mov Word Ptr CS:Identificación.Comando[2], CS

        ; Pasar la dirección con la firma
    Mov AX, Offset BloqueFi rmaTSR
    Mov Word Ptr CS:Identificación.FirmaTSR, AX
    Mov Word Ptr CS:Identificación.FirmaTSR[2], CS
```

```
Pop AX
Mov DI, Offset Identificación
Push CS      ; ES:DI apuntando a Identificación
Pop ES

; Saltar al siguiente gestor de la lista
Jmp CS:INT2F.AnteriorGestor
```

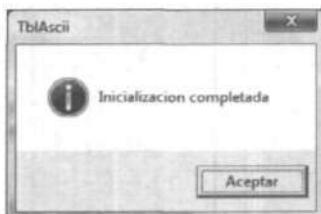


Figura 27.13. Ventana mostrada por el programa TblWin.

Resumen

Como ha podido ver en este extenso capítulo, la creación de aplicaciones que quedan residentes en memoria no es una tarea simple en DOS, a pesar de lo que pudiera parecer inicialmente. Si se pretende que el programa se comporte siempre adecuadamente, sin interferir con el DOS, la BIOS y las demás aplicaciones que se ejecuten, es necesario tomar en consideración multitud de aspectos, como los problemas de reentrada del DOS y la BIOS, la posibilidad de interrumpir operaciones de acceso a disco, la activación mediante la interrupción de teclado, etc.

Para contemplar todos esos aspectos, como se ha hecho en el programa TBLASCII, es necesario contar con un buen conocimiento del funcionamiento del ordenador, en general, y del DOS, en particular.

Esto nos ha servido como excusa para, en este capítulo, describir temas que no habían sido abordados con anterioridad, en los capítulos precedentes.

28

Servicios de Windows

La mayor parte de los capítulos de este libro se centran en el estudio de las distintas instrucciones del lenguaje ensamblador de los procesadores de la familia x86, empleando DOS como sistema operativo para la construcción de los ejemplos por su simplicidad. Tanto los usuarios de DOS como los de Windows y Linux puede generalmente ejecutar cualquier programa DOS, ya sea en una ventana de consola o la emulación DOS de Linux, mientras que las personas que usan exclusivamente DOS no pueden ejecutar programas escritos para otros sistemas.

Esto no significa, como ya pudo verse en el segundo capítulo, que no sea posible escribir aplicaciones para Windows o Linux usando el lenguaje ensamblador. De hecho, y siempre que el proceso requiera velocidad y un código compacto, el lenguaje ensamblador será siempre la mejor opción a la hora de escribir código. No tiene sentido su uso, por el contrario, cuando lo que se persigue es la creación de una interfaz de usuario que quedará a la espera de entradas por teclado o tareas similares.

Nuestro objetivo, en este capítulo, es familiarizarnos con el proceso de generación de aplicaciones para Windows en ensamblador, aprendido a utilizar elementos como los archivos con definiciones o las bibliotecas de importación. También conoceremos la estructura básica de una aplicación Windows, mostrando una ventana estándar y respondiendo a las acciones del usuario.

Para ello deberemos conocer algunos de los servicios del API de Windows, una extensísima biblioteca de miles de funciones que facilitan prácticamente cualquier tarea, desde las más fundamentales, como la asignación y la liberación de memoria, hasta las más avanzadas, como el control de dispositivos multimedia, la comunicación a través de Internet, etc.

Como es lógico, resulta imposible abarcar en un libro como éste ni siquiera una pequeña fracción del API de Windows, nos centraremos en unas pocas decenas de funciones, ya que para ello serían precisas varios miles de páginas. No obstante, Microsoft facilita, en la sede de MSDN (*Microsoft Developer Network*, <http://MSDN.Microsoft.com>) una referencia de todos esos servicios.

Herramientas necesarias

Además del ensamblador y el enlazador, prácticamente las únicas herramientas que hemos empleado hasta ahora si exceptuamos el depurador, para crear aplicaciones para Windows necesitaremos, asimismo, archivos con la definición de los servicios y las constantes y estructuras que usan, así como bibliotecas de importación que permitan al enlazador vincular nuestro programa con las bibliotecas que contienen las funciones a las que se pretende invocar.

A diferencia de DOS, los servicios de Windows no son accesibles mediante interrupciones software, sino que son funciones o rutinas a las que hay que llamar mediante la instrucción *call*.

Los parámetros no se facilitan en registros sino en la pila, concretamente en orden inverso a como aparecen en la documentación de referencia de Microsoft puesto que la convención de llamada que se emplea, conocida como *stdcall*, facilita los parámetros en dicho orden.

Para poder acceder a esas funciones necesitamos, por una parte, conocer sus nombres así como la lista de parámetros que necesitan. En algunas ocasiones dichos parámetros son punteros a estructuras de datos, cuya configuración también es preciso saber de antemano.

Aunque podríamos construir dichas definiciones previas, conocidas como *prototipos*, manualmente, escribiendo código, el objetivo es difícilmente alcanzable si se pretenden utilizar la mayor parte de los servicios de Windows, ya que son miles de definiciones las necesarias.

Afortunadamente, hay disponibles archivos de inclusión que contienen todos esos datos, dispuestos simplemente para ser usados en nuestros programas. Los hay disponibles tanto para MASM/TASM como para NASM.

En la carpeta \Herramientas\Windows\MASM32\Include encontrará los que vamos a utilizar a continuación con MASM.

Además de conocer los nombres de las funciones y su lista de parámetros, a la hora de enlazar los programas será necesario saber en qué bibliotecas de enlace dinámico, o DLL, se encuentran las funciones a las que va a invocarse.

Ciertas funciones se encuentran en kernel32.dll, otras en user32.dll, un tercer grupo en opendlg32.dll, etc.

La vinculación entre nuestros programas y esas DLL se establece en el momento del enlazado, al llamar a link. Por ello esta herramienta necesita lo que se conoce como *bibliotecas de importación*, unos módulos que contienen toda la información sobre el contenido de las DLL.

Nota

En la carpeta \Herramientas\Windows32\MASM32 encontrará tanto los archivos de inclusión como las bibliotecas de importación necesarias para usar la mayoría de servicios de Windows. Estos archivos pueden ser generados mediante herramientas como iMPDEF/iMPLiBoDUMPDEF pero, por comodidad, en las subcarpetas `include` y `Lib` encontrará los archivos resultantes del proceso.

En caso de que vayamos a crear para nuestros programas elementos como cuadros de diálogo, menús o imágenes, otra herramienta indispensable es el compilador de recursos. Éste toma una descripción textual del recurso, por ejemplo posiciones, textos y atributos de los elementos de un cuadro de diálogo, y genera un archivo de recursos que, posteriormente, puede ser incluido en el ejecutable durante el proceso de enlazado. En la subcarpeta Bin de Herramientas\Windows\MASM32, la misma que contiene el ensamblador y el enlazador, encontrará la herramienta `re` (*Resource Compiler*), el compilador de recursos de Microsoft.

Inclusión de definiciones y bibliotecas

Al crear una aplicación para Windows, usando MASM, tendremos que incluir en cada módulo los archivos de definiciones que contengan los prototipos de las funciones que vamos a utilizar, así como las bibliotecas de importación correspondientes. Con este fin usaremos las directivas `include` e `includelib`, por ejemplo:

```
include windowa.inc  
include kernel32.inc  
include user32.inc  
  
includelib kerne.L32.lib  
includelib user32.lib
```

Para poder hacer referencia a los archivos de inclusión y bibliotecas de importación como se hace aquí, sin indicar explícitamente el camino en el que se encuentra, es necesario facilitar ese camino al ensamblador y enlazador. La ventaja es que, de cambiarse dicho camino, no tendremos en ningún caso que tocar el código de los programas, bastará con facilitarlo adecuadamente a mi o link.

Ensamblado y enlace

Tal como ya se indicase en el octavo capítulo, al crear nuestro primer programa para Windows, para ensamblar un programa con MASM obteniendo un módulo objeto para Windows es necesario facilitar una serie de opciones, concretamente /c, /Cp y /coff. Con la primera indicamos a MASM que sólo ensamble el código, sin invocar automáticamente al enlazador. La segunda hace que, al enlazar, no se pierda el orden de minúsculas

y mayúsculas que se hubiese utilizado en el código original, lo cual es importante puesto que Windows distingue entre mayúsculas y minúsculas a la hora, por ejemplo, de localizar sus servicios. Por último, la opción /coff provoca que MASM genere un módulo en formato COFF, el que usa Windows, en lugar de MZ, que es el ejecutable desde DOS.

Además de las anteriores, mediante la opción /I indicaremos el camino en el que se encuentran los archivos de inclusión, a los cuales se hace referencia con la directiva include. Esto, como se indicaba antes, evitará el uso de caminos absolutos en el código, haciéndolo más independiente. En cuanto al ensamblado, las opciones a entregar a LINK son dos: /SUBSYSTEM:WINDOWS, para que el ejecutable generado sea válido para Windows, y /LTBPATH:camino, indicando el camino en el que se encuentran las bibliotecas de importación.

Para evitar tener que introducir todos estos parámetros y los caminos cada vez que necesite ensamblar un ejemplo, lo más sencillo es crear un archivo de proceso por lotes. Incluya las dos líneas siguientes en un archivo llamado asm.bat. A partir de este momento, cada vez que necesite ensamblar un ejemplo para Windows bastará con que introduzca el comando asm archivo, sin facilitar extensión.

```
mi /c /Cp /coff /I\ASM\Herramientas\Windows\MASM32\Include %1.asm
link /LIBPATH:\ASM\HRrr,imicntas\Windows\MASM32\Lib /SUBSYSTEM:WINDOWS %1
```

Lógicamente, tendrá que modificar los caminos indicados en las opciones /I de MASM y /LIBPATH de LINK para que se ajusten a la configuración de su sistema.

Invocación a funciones Windows

La mayor parte del código de un programa Windows, a pesar de estar escrito en ensamblador, estará compuesto de llamadas a funciones del sistema. Estas llamadas se efectúan mediante la instrucción call, como si de una rutina cualquiera se tratase. Hay que tener en cuenta que en Windows nuestros programas son aplicaciones de 32 bits, no de 16, y que tenemos acceso a un modelo de memoria plano en el que se direccionan hasta 4 gigabytes de memoria sin necesidad de registros de segmento ni nada parecido. No hay que preocuparse, por tanto, por el hecho de que esas llamadas sean o no de tipo far, ya que todas las invocaciones son de 32 bits y almacenan en la pila el registro EIP, extensión de IP a 32 bits.

Antes de efectuar la llamada, no hay que olvidar poner en la pila la lista de parámetros que la función a invocar necesite. Recuerde también que el orden de los parámetros en la pila será el inverso al indicado en la definición de la función.

Por ejemplo, si necesitamos llamar a la función EnableWindow debemos saber que precisa dos parámetros: el manejador de la ventana a manipular y una constante indicando si la ventana se activará o desactivará. Para llamar desde ensamblador el proceso sería el siguiente:

```
Puah TRUE
PUSH hMiVentana
CALL EnableWindow
```

TRUE es un valor de 32 bits definido en Windows . inc como constante, pudiendo utilizarse a modo de valor inmediato. Lo que hacemos, por tanto, es incluir en la pila el segundo parámetro, después el primero y, por último, llamamos a la función. El parámetro que devuelve ésta, también un valor de 32 bits, lo encontraremos en el registro eax.

Recuerde que al programar para Windows lo está haciendo para un sistema operativo de 32 bits y, por tanto, puede usar los registros extendidos como eax, ebx, edx, ecx, es i y edi. En ocasiones esto es imprescindible, por ejemplo al recoger el valor devuelto por una llamada a una función.

Si olvidáramos insertar en la pila un parámetro necesario para una función a la que llamamos, lo más seguro es que la aplicación tenga problemas y sea cerrada por el sistema, ya que éste extraerá de la pila datos inapropiados y afectará a otros aspectos. Para evitar esto, la alternativa consiste en llamar a las funciones con la directiva invoke tal y como se hizo en el ejemplo del segundo capítulo. Gracias a esta directiva, el fragmento de código anterior podría resumirse en la siguiente sentencia:

```
invoke EnableWindows, hMiVentana, TRUE
```

Para poder usar invoke es necesario contar de antemano con un proLolipo de la función a la que va a llamarse, prototipo que se crea con la directiva proto. Por ejemplo:

```
EnableWindow Proto :DWORD, ;DWORD
```

Con este prototipo se indica que la función EnableWindow precisa dos parámetros de tipo DWORD, es decir, de 32 bits. Lo que contienen los archivos de inclusión son, precisamente, los prototipos de las funciones Windows creadas con la directiva Proto, de tal forma que podemos utilizar invoke no sólo para simplificar el código, sino también para estar seguros de que la lista de parámetros entregada coincide al menos en el número y tipo de parámetros.

En la Web de MSDN (véase la figura 28.1) puede encontrar una lista de todas las directivas que contempla MASM, tanto relacionadas con la invocación a funciones como de otros tipos.

Estructura básica de una aplicación Windows

Además de saber cómo invocar a las funciones de Windows, para poder crear aplicaciones para este sistema es imprescindible saber qué hace cada función y cuál es la estructura general de cualquier programa Windows. Lo primero, saber qué hace cada función, podemos obtenerlo de la ayuda de referencia ofrecida por Microsoft en su Web MSDN o bien de cualquier producto de desarrollo para Windows.

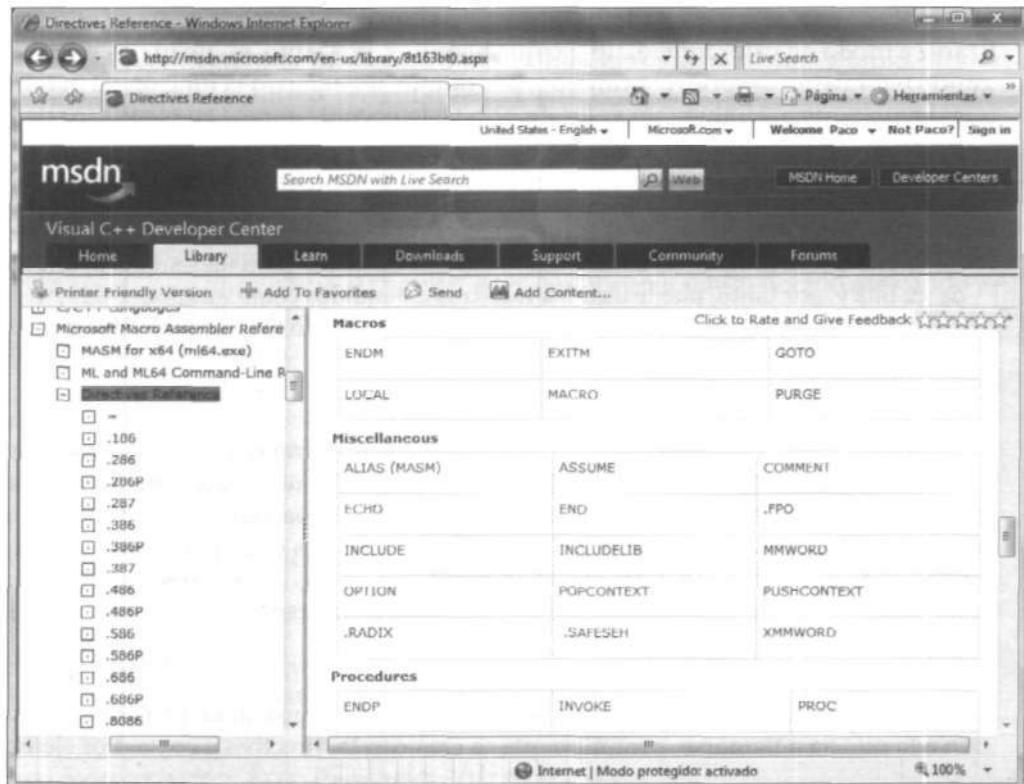


Figura 28.1. Referencia de directivas que entiende MASM.

Al crear un programa para Windows no tendremos que preocuparnos de leer directamente el teclado, acceder a la memoria de vídeo para mostrar datos y tareas similares. A cambio, deberemos actuar como lo hacen la mayoría de aplicaciones: creando y mostrando una ventana que, una vez es visible, queda a la espera de las acciones del usuario.

Lógicamente hablamos de la aplicación estándar de Windows, porque también podríamos escribir en ensamblador bibliotecas de enlace dinámico con código para ser usado desde otras aplicaciones, controladores de dispositivos, etc.

La clase de ventana

Para poder crear una ventana, alojando en ella los parámetros que nos interesen, lo primero que debemos hacer es registrar en el sistema sus características, lo que se denomina la *clase de ventana*. Existen una serie de clases predefinidas, como los botones, listas, botones de radio, etc., pero lo habitual es que cada aplicación defina su propia clase, fin para el cual se utiliza la función `RegisterClassEx`. Ésta se encuentra alojada en la DLL user32.dll, lo cual significa que necesitaremos el archivo de inclusión user32.inc y la biblioteca de importación user32.lib.

Esta función necesita como parámetro la dirección de una estructura WNDCLASSEX, definida en el archivo windows.inc como sigue:

```
WNDCLASSEX STRUCT
    cbSize          DWORD      ?
    style           DWORD      ?
    lpfnWndProc    DWORD      ?
    AClssExtra     DWORD      ?
    cbWndExtra     DWORD      ?
    hlnstance      DWORD      ?
    hlcon          DWORD      ?
    hCursor         DWORD      ?
    hbrBackqround  DWORD      ?
    LpszMenuNamq   DWORD      ?
    IpozClassName  DWORD      ?
    hlconSm        DWORD      ?
WNDCLASSEX ENDS
```

Como puede suponer, antes de llamar a RegisterClassEx, con la dirección de una estructura de este tipo, será preciso dar valor a los distintos campos con que cuenta, para lo cual es imprescindible que sepamos que finalidad tiene cada uno:

- **cbSize**: Debe contener el tamaño, en bytes, de toda la estructura de datos. Es fácil calcularla, ya que se compone de doce campos de tipo DWORD, es decir, de 12 elementos de 32 bits, lo cual equivale a 48 bytes.

Nota

Para evitar calcular el tamaño de una estructura cada vez que necesite conocer su ocupación, puede usar la directiva size seguida del nombre de la estructura.

- **style**: Cada uno de los bits de este campo tiene un significado, existiendo una serie de constantes que permiten su activación simplemente combinándolas. En la tabla 28.1 se enumeran algunas de ellas.
- **lpfnWndProc**: Puntero a una función o rutina que será la encargada de gestionar todas las ventanas de la clase que está definiéndose.
- **cbClssExtra**: Número de bytes que quieren reservarse detrás de la estructura de datos de la clase de ventana para el almacenamiento de datos propios.
- **cbWndExtra**: Número de bytes que quieren reservarse detrás de la estructura de datos asociada a cada ventana de esta clase que se cree.
- **hlnstance**: Las aplicaciones Windows obtienen de parte del sistema, al ponerse en marcha, un identificador global conocido como *hlnstance*. Al crear una clase de ventana es necesario facilitarlo en este campo, pudiendo recuperarlo mediante la función GetModuleHandle.

- hIcon: Identificador del ícono asociado a la ventana. Puede ser 0 para utilizar un ícono por defecto.
- hCursor: Identificador del cursor asociado a la ventana. Puede ser 0 para usar un cursor por defecto.
- hbrBackground: Identificador a una brocha para llenar el fondo o constante de color para rellenarlo. La *brocha* es un recurso de Windows que cuenta con un color y opcionalmente atributos como una trama o patrón de relleno.
- lpszMenuName: Dirección de una cadena de caracteres con el nombre del menú a asociar a esta clase de ventana. Puede ser 0 para no utilizar menú alguno.
- lpszClassName: Dirección de una cadena de caracteres con el nombre que identificará a esta clase de ventana que está definiéndose.
- hIconSm: Identificador del ícono pequeño asociado a esta clase de ventana. Puede ser 0 para utilizar hIcon y, si éste es cero, un ícono por defecto.

Nota

Por regla general, todas las cadenas de caracteres facilitadas a Windows deberán estar terminadas con un carácter nulo.

Tabla 28.1. Constantes que identifican valores a usar en el segundo campo de WNDCLASSEX.

Constante	Significado
CS_NOCLOSE	Desactivar el botón de cierre de las ventanas.
CS_OWNDC	Asignar un contexto de dispositivo único para cada ventana.
CS_HREDRAW	Redibujar el contenido si se cambia el ancho de la ventana.
CS_VREDRAW	Redibujar el contenido si se cambia el alto de la ventana.
CS_SAVEBITS	Guardar el área de pantalla a la que se superponga la ventana para restaurarla de manera inmediata después.

En el siguiente fragmento de código se muestra cómo preparar, sólo con los datos imprescindibles, una estructura WNDCLASS básica utilizándola para registrar una clase de ventana.

```
.data
    ; Estructura non todos los datos
    ; necesarios para registrar la
    ; clase de ventana
    ClaseVentana WNDCLASSEX <size WNDCLASSEX, 0, ProcVentana,0,0,0,0,0,0,0,0,
    NombreClase,0>
```

```

; Identificador de la clase de ventana
NombreClase db 'MiClaseVentana',0

.code
Main:

; Obtenemos el i denl. if icadür de la
;- aplicación en EAX
invoke GetModuleHandle, 0

; y lo colocamos donde debe estar
; en la estructura
mov [ClaseVentana.hlnstance], eax

; Registraremos la clase de ventana
invoke RegisterClassEx, offset ClaseVentana

; Devolvemos el control
invoke Fxi tProcess, 0

; EsLe procedimiento debería procesar
; los mensaje provenientes de las ventanas
ProcVentana:

```

Creación de ventanas

Tras dar el paso anterior, lo único que tenemos es una estructura de datos alojada en memoria y reconocida por Windows como una clase para crear ventanas. No tenemos, por ahora, nada visible. Tomando como base dicha estructura, utilizaremos la función CreateWindowEx para crear las ventanas que necesitemos, mostrándolas u ocultándolas mediante la función ShowWindow.

Podemos crear múltiples ventanas a partir de la misma clase. Esto es lo que permite, por ejemplo, la inclusión de varios botones o listas en una misma ventana. Esos elementos son ventanas que aparecen en el interior de otras.

La función CreateWindowEx necesita un gran número de parámetros y, a diferencia de RegisterClassEx, no existe ninguna estructura para facilitarlos, por lo que deben entregarse en el momento de la llamada.

Los parámetros son los siguientes:

- dwExStyle: Estilo extendido de la ventana. Es un valor de 32 bits en el que cada bit establece una cierta característica. Las constantes enumeradas en la tabla 28.2 nos facilitan la asignación a este campo.
- lpClassName: Dirección de una cadena con el nombre de la clase de ventana a crear. Este elemento es el vínculo entre la ventana que va a crearse y la clase definida previamente.
- lpWindowName: Dirección de una cadena con el texto que aparecerá como título de la ventana.

- dwStyle: Estilo básico de la ventana. Es similar a dwExStyle pero, en este caso, las constantes de estilo son las de la tabla 28.3.
- x: Posición horizontal de la ventana en la pantalla. Puede utilizarse el valor CW_USEDEFAULT para dejar que sea el sistema quien coloque la ventana.
- y: Posición vertical de la ventana en la pantalla. Puede utilizarse el mismo valor CW_USEDEFAULT del campo anterior.
- nWidth: Anchura que tendrá la ventana inicialmente. Se puede utilizar CW_USEDEFAULT para obtener una anchura por defecto.
- nHeight: Altura que tendrá la ventana inicialmente. Se puede utilizar CW_USEDEFAULT para obtener una altura por defecto.
- hWndParent: Si la ventana a crear va a estar incluida en otra, convirtiéndose en una *ventana hija*, este parámetro contendrá el identificador de la ventana que actuará como contenedor o *padre*.
- hMenu: Identificador de un menú en caso de que no desee utilizarse el indicado en la clase de ventana.
- hInstance: El mismo valor facilitado en el campo hinstance de la estructura WNDCLASSEX.
- lpParam: Dirección de una estructura CREATESTRUCT con parámetros adicionales de creación en caso de que sean necesarios.

Tabla 28.2. Constantes de estilo extendido para las ventanas.

Constante	Significado
WS_EX_ACCEPTFILES	Indica que la ventana aceptará que se suelten archivos sobre ella tomados desde el Explorador de Windows o bien una herramienta similar.
WS_EX_CLIENTEDGE	Indica que la ventana tendrá un borde resaltado alrededor.
WS_EX_MDICHILD	La ventana creada es una ventana hija MDI (<i>Multiple Document Interface</i>).
WS_EX_STATICEDGE	La ventana tendrá un borde tridimensional típico de elementos que no interactúan con el usuario.
WS_EX_TOOLWINDOW	Indica que la ventana creada será del tipo flotante, con una barra de título de menor tamaño y que no aparece en la barra de tareas de Windows.
WS_EX_TOPMOST	La ventana permanecerá sobre todas las demás.
WS_EX_TRANSPARENT	La ventana será transparente, dejando ver las que hay debajo.
WS_EX_WINDOWEDGE	Indica que la ventana tendrá un borde resaltado alrededor.

Tabla 28.3. Constante de estilo básico para las ventanas.

Constante	Significado
WS_BORDER	La ventana tendrá un borde alrededor.
WS_CAPTION	La ventana contará con una barra de título.
WS_CHILD	La ventana será hija de otra.
WS_DISABLED	La ventana creada estará inicialmente desactivada.
WS_HSCROLL	La ventana tendrá una barra de desplazamiento horizontal.
WS_MAXIMIZEBOX	La ventana contará con un botón para maximizar la ventana.
WS_MINIMIZEBOX	La ventana contará con un botón para minimizar la ventana.
WS_MAXIMIZE	Inicialmente la ventana estará maximizada.
WS_MINIMIZE	Inicialmente la ventana estará minimizada.
WS_OVERLAPPED	Crea una ventana apilada típica, con título y borde.
WS_POPUP	La ventana será de tipo emergente.
WS_SIZEBOX	La ventana tendrá un borde para la alteración del tamaño inicial.
WS_SYSMENU	La ventana tendrá un menú de sistema.
WS_VISIBLE	La ventana estará inicialmente visible.
WS_VSCROLL	La ventana tendrá una barra de desplazamiento vertical.

Suponiendo que hubiésemos definido en la sección de datos una cadena de caracteres con un título, llamando Nombre Ventana al campo, la siguiente sentencia sería suficiente para crear una ventana de la clase definida previamente:

```
; Creamos la ventana
invoke CreateWindowEx, 0,
    Offset NombreClasc,
    Offset NombreVentana,
    WS_VISIBLE Or WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CWJSEDEFAULT,
    0, 0, ClaseVentana.hlnstance, 0
```

En caso de que, después de ejecutar esta sentencia, el valor de eax sea cero, significará que la ventana no ha podido crearse. En caso contrario el valor de eax es el manejador de la ventana que, normalmente, guardaremos para poder actuar posteriormente sobre ella.

Por ejemplo, tras crear la ventana lo habitual es hacerla visible y actualizar su estado, para lo cual se invoca a las funciones ShowWindow y UpdateWindow, respectivamente, facilitando el citado manejador como primer parámetro.

```

; Si no ha podido crearse la ventana
ur eax, eax
; salir
j2 Salir

; en caso contrario mostrarla
invoke ShowWindow, eax, SW_SHOWNORMAL
; y actúa 1 i zarla
invoke UpdateWindow, eax

```

En las actuales versiones de Windows, si la ventana se ha creado de antemano con el atributo ws_VISIBILE no son necesarias las llamadas a las funciones ShowWindow y UpdateWindow.

Proceso de mensajes

A pesar de que ejecute todas las instrucciones de registro de clase de ventana, creación y visualización de la ventana, un programa Windows que sólo haga lo anterior terminará prácticamente de manera inmediata, sin llegar a ver nada en pantalla. Es lógico, ya que el programa ejecuta todas esa secuencia de órdenes pero, a continuación, devuelve el control al sistema, que termina la aplicación y destruye los recursos que le pertenezcan y que ya no son útiles. Una vez que se ha creado una ventana, Windows comienza a enviarle mensajes que deben ser procesados para que se haga visible, reciba eventos generados por el teclado y el ratón o el propio sistema operativo. Cada aplicación cuenta con una cola de mensajes y un procedimiento encargado de procesarlos, cuya dirección se facilitó en el momento de registrar la clase de ventana. En este momento ni recuperamos esos mensajes, tras haber creado la ventana, ni procesamos nada en el procedimiento que, en nuestro caso, habíamos llamado ProcVentana.

La recuperación de mensajes de la cola se efectúa con la función GetMessage, enviándose al procedimiento de proceso que corresponde mediante la función DispatchMessage. Debe tener en cuenta que una misma aplicación puede contar con ventanas de distintas clases y, por tanto, con varios procedimientos de proceso. La cola de mensajes, sin embargo, es única y se lee y despacha en un único punto, generalmente tras haber creado las ventanas. El primer parámetro que hay que entregar a GetMessage es la dirección de una estructura como la siguiente:

```

MSG STRUCT
    hwnd    DWORD    ?
    message DWORD    ?
    wParam   DWORD    ?
    lParam   DWORD    ?
    time    DWORD    ?
    p.t     P0INT   o
MSG ENDS

```

En ella se devolverá el manejador de la ventana a la que va dirigido el mensaje, el mensaje propiamente dicho, los parámetros que le acompañan, la hora en que se generó y el punto en el que estaba el cursor del ratón en ese instante. Dicha estructura, como muchas otras, está definida en Windows.inc, por lo que podemos usarla sin más. Además son necesarios otros tres parámetros cuyo objetivo es actuar como filtros de los mensajes a obtener, pudiendo ser 0 todos ellos.

Si GetMessage devuelve el valor 0 en eax significará que hay que salir del bucle de proceso de mensajes, devolviendo el control al sistema para que pueda poner fin a la ejecución del programa. En caso contrario, el mensaje obtenido en la estructura se entregará como único parámetro a la función DispatchMessage. Ésta se encargará de enviarla al procedimiento de proceso que corresponda según el manejador de ventaja que hay en MSG.hwnd.

Los bucles de proceso de mensajes son, básicamente, siempre iguales, ya que repiten las llamadas a GetMessage e InvokeMessage hasta que eax toma el valor 0, momento en el que se interrumpe el bucle y devuelve el control. La diferencia la encontraremos en los procedimientos de proceso de mensajes, el procedimiento que recibirá parte de los parámetros de la estructura MSG y deberá responder adecuadamente. Este procedimiento recibe cuatro argumentos que, en nuestro caso, definiríamos de la siguiente manera:

```
ProcVentana proc hWnd:HWND, uMsg:UINT,  
wParam:WPARAM, lParam:LPARAM
```

hWnd es el manejador de la ventana afectada por el mensaje uMsg, mientras que wParam y lParam contendrán parámetros dependientes de dicho mensaje. Dependiendo de su estructura y necesidades, una aplicación necesitará procesar eventos de teclado y ratón, selección de opciones, pulsación sobre los botones que contenga la ventana, etc. No obstante, mientras se ejecute recibirá muchos más mensajes, por ejemplo al cambiar la ventana de posición o tamaño, cambiar algún parámetro global de sistema y situaciones similares. Todos los mensajes que no vaya a procesar la aplicación por sí misma deben ser remitidos a la función DefWindowProc, que ejecutará el código definido por defecto para cada mensaje.

El programa completo

Con lo que hemos aprendido hasta ahora, a pesar de ser sólo los fundamentos más básicos, tenemos suficiente para desarrollar un programa Windows completo que muestre una ventana con un título, redimensionable y que responda a los botones para minimizar, maximizar y cerrar. El código de este programa sería el mostrado a continuación:

```
.586 ; Asumirnos un procesador Pentium  
; Trabajaremos con un modelo  
; de memoria plano  
.model flat,stdcall  
  
; Incluimos definiciones de estructuras  
; y constantes  
include Windows.inc
```

```

; así como los prototipos de las funciones
; de uso más habitual
include kernel32.inc
include user32.inc

; Importamos las bibliotecas para que el
; enlazador pueda vincular adecuadamente
; las llamadas
includelib kernel32.lib
includelib user32.lib

.Data

; Estructura con todos los datos
; necesarios para registrar la
; clase de ventana
ClaseVentana WNDCLASSEX <size WNDCLASSEX, 0, ProcVentana, 0, 0, 0, 0, 0, COI,OR_
BACKGROUND,0, NombreClase,0>

; Identificador de la clase de ventana
NombreClase db 'MiClaseVentana',0

; Título de la ventana
NombreVentana db 'Título de la ventana', 0

; Para ir recuperando mensajes
Mensaje MSG <>

ClaseBoton db 'RtITTON',0
TituloBoton db 'Púlsame',U

```

, Code

Main:

```

; Obtenemos el identificador de la
; aplicación en EAX
invoke GetModuleHandle, 0
; y lo colocamos donde debe estar
; en la estructura
mov ;ClaseVentana.hinstance] , eax

; Registraremos la clase de ventana
invoke RegisterClassEx, offset ClaseVentana

; Creamos la ventana
invoke CreateWindowEx, 0,
    Offset NombreClase,
    Offset NombreVentana,
    WS_VISIBLE Or WSJDVERLAPPEDWINDOW,
    CWJSEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT,
    0, 0, ClaseVentana.hinstance, 0

```

```

; Si no ha podido crearse la ventana
or eax, eax
; salir
jz Salir

Bucle: ; Bucle de proceso de mensajes

; Obtenemos un mensaje de la cola
invoke GetMessage, Offset Mensaje, 0, 0, 0

; Si EAX = 0 hay que salir
or eax, eax
jz Salir

; En caso contrario lo despachamos
invoke DispatchMessage/ Offset Mensaje

; y seguimos esperando mensajes
jmp Bucle

Salir: ; Se ha recibido EAX - 0 hay que terminar
; Facilitamos en EAX el contenido
; del wParam del mensaje
invov eax, Mensaje.wPaxam

; y volvemos
ret

; Este procedimiento debería procesar
; los mensaje provenientes de las ventanas
ProcVentana proc hWnd:HWND, uMesg:UINT,
    wParam:WPARAM, lParam:LPARAM

    ; Comprobamos el mensaje
    emp uMesg, WM_DESTROY
    je Quit ; y saltamos al punto adecuado

    ; Pedimos a Windows que procese el
    ; mensaje él mismo
    invoke DefWindowProc, hWnd,uMesg,wParam,lParam

    ret ; y volvemos

Quit: ; Se ha pedido la salida
; Llamamos a PostQuitMessage
invoke PostQuitMessage, Ü

; Ponemos EAX a 0
xor eax, eax

ret ; y volvemos

ProcVentana Endp

end Main

```

Observe que en el procedimiento ProcVentana se procesa un único mensaje: WM DESTROY. Éste se genera cuando el usuario cierra la ventana y, por tanto, el programa debe llegar a su fin. En este caso llamamos a la función PostQuitMessage, que se encargará de desencadenar el proceso de finalización. Todos los demás mensajes se pasan a DefWindowProc, obteniendo un programa que, básicamente, no hace más que mostrar la ventana y poco más.

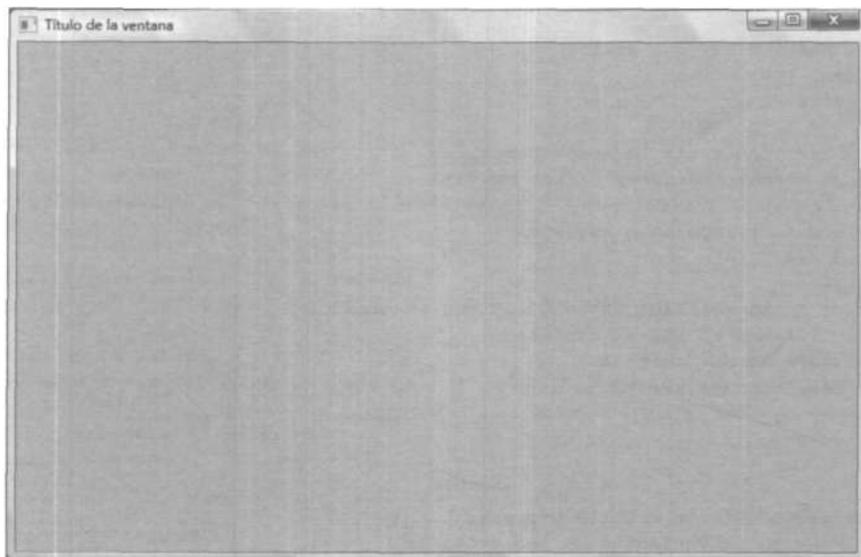


Figura 28.2. Aspecto inicial de la ventana mostrada por el programa.

Uso de controles

Una ventana completamente vacía no es un elemento especialmente útil, aunque por sí sola sea más llamativa, visualmente hablando, que la mayoría de los programas que hemos escrito en los capítulos previos. Lo que tenemos, en este momento, es una superficie sobre la que trabajar, una superficie que está preparada para que aportemos el contenido que necesite nuestra aplicación.

Básicamente existen dos métodos para poner algo dentro de una ventana: utilizar controles o bien responder a los mensajes que solicitan que se dibuje el contenido de la ventana. Salvo que la aplicación sea de tipo gráfico, lo más habitual es incluir en la ventana los controles que se necesiten, enviándole mensajes y respondiendo a los que ellos nos envíen. Los controles son de clases de ventanas predefinidas, de tal forma que podemos crear directamente ventanas de dichas clases e incluirlas en nuestras ventanas. Para no confundir los términos, a las ventanas de clases predefinidas, y que cuentan ya con una cierta funcionalidad, se les llama controles. En la tabla 28.4 se enumeran los controles más corrientes indicándose el nombre de clase que les corresponde.

Tabla 28.4. Controles y sus nombres de clase.

Clase	Definición
BUTTON	Una ventana con aspecto de botón rectangular y un texto en su interior. Puede tomar otras apariencias, como un botón de radio o un botón con una marca en su interior.
STATIC	Texto que puede colocarse en la ventana como elemento estático.
EDIT	Área de rectangular para la introducción o edición de datos.
LISTBOX	Lista de elementos de los cuales puede seleccionarse uno o varios.
SCROLLBAR	Control que facilita el desplazamiento horizontal o vertical.

Cada uno de estos controles cuenta con estilos diferentes y generan distintos mensajes cuando se actúa sobre ellos. Algunos, como los botones, envían un mensaje para notificar que se han pulsado. Otros, como las cajas de texto, mantienen un valor en su interior que puede ser leído o escrito mediante el envío de mensajes. Vamos a tratar algunos de estos detalles.

Añadir un control a una ventana

Los controles son ventanas y, como tales, son creados mediante la misma función CreateWindowEx que hemos empleado anteriormente para crear la ventana de nuestro programa.

La única diferencia es que el nombre de clase será uno de los citados en la tabla 28.4, u otro de los nombres de clase con que cuenta Windows, y que posiblemente usemos estilos específicos para este tipo de control.

Si creamos el control como una ventana independiente, al mostrarlo aparecerá en pantalla y se comportará como una ventana corriente. Para incluirlo en el interior de la ventana de nuestra aplicación, que es lo que nos interesa, deberemos modificar dos de los parámetros de la llamada a CreateWindowEx:

- dwStyle: Hay que añadir al estilo básico la constante WS_CHILD, indicando así que la ventana a crear será hija de otra.
- hWndParent: En lugar de 0, este parámetro deberá ser el manejador de la ventana creada previamente, estableciendo de esta forma la relación entre ventana hija y padre.

Por ejemplo, si quisieramos añadir un botón a la ventana del ejemplo anterior insertaríamos el código siguiente justo antes de la etiqueta Bucle. Lógicamente, habría que definir los cambios ClaseBoton y TituloBoton. El primero de ellos sería la secuencia de caracteres BUTTON, mientras que el segundo contendría el texto a mostrar en el interior del botón.

```

; Insertamos un botón
invoke CreateWindowEx, 0,
    Offset ClaseBoton,
    Offset TituloBoton,
    WS_CHILD Or WS_VISIBLE,
    40, 40, 120, 48,
    eax, 0, ClaseVentana.hlnstance, 0

```

Al ejecutar el programa, la ventana ahora tendrá el aspecto que puede verse en la figura 28.3. A pesar de que el botón puede ser pulsado, y visualmente responde a esta pulsación, no se produce ninguna acción puesto que no estamos respondiendo a los mensajes que envía a la ventana.

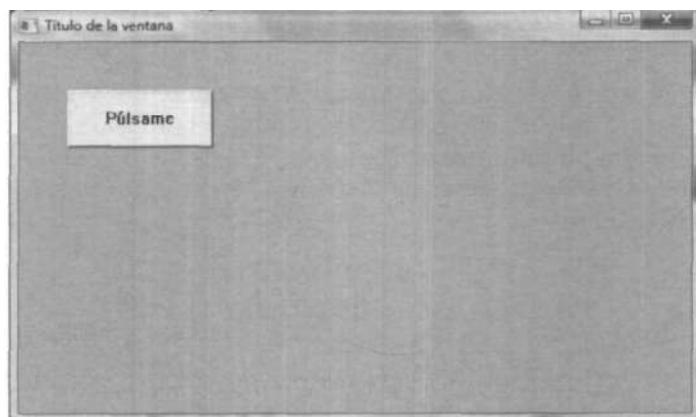


Figura 28.3. Aspecto de la ventana con el botón.

Botones

Uno de los elementos más recurrentes en las ventanas de Windows son los botones, ya sean como el anterior, que muestra un título y al pulsarse generan un mensaje, o de cualquier otro de los tipos disponibles. Además de conocer los distintos tipos, y cómo indicar el que deseamos en cada caso, necesitaremos conocer algunos mensajes para poder aprovechar su funcionalidad.

Cuando se crea una ventana de la clase BUTTON y no se facilita ningún indicador adicional en wsstyle, se asume que el estilo del botón será BS_PUSHBUTTON, correspondiente a un botón estándar que puede ser pulsado. Además de éste, tenemos también los estilos siguientes:

- **BSDEFPUSHBUTTON:** Idéntico en apariencia y funcionalidad a BS_PUSHBUTTON, con la única diferencia de que actúa como botón por defecto y, por tanto, al pulsar **Intro** es el que aparece como pulsado.
- **BS_CHECKBOX/BS_AUTOCHECKBOX:** El botón aparece como un recuadro que puede tener una marca en su interior. La diferencia entre uno y otro estilo es que

el segundo activa y desactiva dicha marca automáticamente, mientras que el primero tan sólo lo notifica a la aplicación y espera que ésta le indique el estado que debe tomar.

- BS_3STATE/BS_AUTO3STATE: Similar al estilo anterior, con la diferencia de que el botón puede estar en tres estados diferentes: marcado, desmarcado o bien indeterminado.
- RS_RADIOBUTTON/BS_AUTORADIOBUTTON: El botón aparece como un círculo que puede o no tener una marca en su interior, en combinación exclusiva con otros botones del mismo tipo. Esto es lo que se conoce habitualmente como *botón de radio*.
- BS_GROUPBOX: A pesar de ser un estilo de la clase de ventana BUTTON, en realidad al aplicarlo no se obtiene un botón sino un área rectangular cuyo objetivo es acoger otros controles. Lo que se conoce normalmente como un *grupo*.

Todos los estilos de botón, a excepción de los que tienen el estilo BS_GROUPBOX, envían a la ventana en la que está el botón un mensaje WM_COMMAND acompañado de un código de notificación representado por la constante BN_CLICKED. Este código de notificación lo encontraremos en el parámetro wParam, mientras que lParam contendrá el manejador de ventana del control que ha generado el mensaje.

Envío de mensajes a ventanas

La comunicación entre nuestra aplicación y los controles insertados en la ventana no es una sola dirección, con la recepción de mensajes de notificación, sino totalmente bidireccional, siendo posible el envío de mensajes. Con ellos podemos tanto obtener información del control como alterar su estado. La función encargada de enviar un mensaje a una cierta ventana, ya sea ésta un control o no, es SendMessage, siendo necesarios los parámetros siguientes:

- hWndControl: Manejador de la ventana a la que va a enviarse el mensaje.
- uMsg: Mensaje a enviar. Normalmente se utilizará alguna de las constantes predefinidas.
- wParam: Primer parámetro asociado al mensaje.
- lParam: Segundo parámetro asociado al mensaje.

Lógicamente, los parámetros facilitados en wParam y lParam dependerán del mensaje que se entregue como segundo parámetro.

Mediante el mensaje WM_GETTEXT, por ejemplo, podemos recuperar el título o nombre del control, debiendo facilitarse en wParam el número máximo de caracteres a recuperar y en lParam la dirección del área de memoria donde se dejarán. Por el contrario, WMSETTEXT, que modifica ese mismo texto, sólo necesita en lParam la dirección del nuevo título, no teniendo ninguna utilidad wParam.

Al trabajar con botones, dos mensajes especialmente interesantes son BM_GETCHECK y BMSETCHECK. El primero nos facilita el estado actual de botones tipo de radio o bien *checkbox*, mientras que el segundo modifica dicho estado. BM_GETCHECK no necesita parámetros y devuelve como resultado uno de los valores enumerados en la tabla 28.5, mientras que el segundo necesita que entreguemos como wParam uno de esos valores.

Tabla 28.5. Constantes que identifican los valores de BM_GETCHECK/BM_SETCHECK.

Constante	Estado del botón
BST_UNCHECKED	El botón no está marcado.
BST_CHECKED	El botón está marcado.
BST_INDETERMINATE	El botón se encuentra en estado indeterminado.

Estos estados no pueden aplicarse a los botones corrientes, que simplemente generan un mensaje al ser pulsados.

Nota

Windows cuenta con muchas funciones que efectúan tareas equivalentes al envío de ciertos mensajes. Para establecer el texto de una ventana, por ejemplo, podemos utilizar el mensaje WM_SKTTEXT pero también llamar a la función SetWindowText facilitándole el manejador de la ventana y la dirección del texto a establecer.

Un ejemplo

Veamos un ejemplo demostrativo de cómo usar la clase BUTTON para añadir algunos botones a la ventana, detectando la pulsación sobre ellos y, en uno de los casos, indicando también el estado actual. El código del programa es el mostrado a continuación:

```
.586 ; Asumimos un procesador Pentium
; Trabajaremos con un modelo
; de memoria plano
.model flat, stdcall

.stack

; Incluimos definiciones de estructuras
; y constantes
include Windows.inc

; así como los prototipos de las funciones
; de uso más habitual
include kernel^2.inc
include user32.inc
```

```

; Importamos las bibliotecas para que el
; enlazador pueda vincular adecuadamente
; las llamadas
includelib kernel32.lib
includelib user32.lib

; Estructura para preparar los datos
; de los controles que incluiremos
DatosControl Struct
    Manejador DWORD ?
    Clase    DWORD ?
    PosX    DWORD ?
    POSY    DWORD ?
    Ancho   DWORD ?
    Alto    DWORD ?
    Estilo  DWORD ?
    Texto   DWORD ?
DatosControl Ends

>Data

; Estructura con todos los datos
; necesarios para registrar la
; clase de ventana
ClaseVentana WNDCLASSEX <size WNDCLASSEX, 0, ProcVentana,0,0,0,0,COLOR_
BTNSHADOW,0, NombreClase,0>

; Tdidentificador de la clase de ventana
NombreClase db 'MiClaseVentana',ü

; Titulo de la ventana
NombreVentana db 'Título de la ventana', 0

; Manejador de la ventana
Manejador DWORD ?

; Para ir recuperando mensajes
Mensaje MSG <>

; Número de controles a insertar
NumControles DWORD 7



---


; NO INCLUIR NADA ENTRE ESTAS ESTRUCTURAS
Btn1 DdosControl <?,ClaseBoton, 24,24,164,48,BS_PUSHBUTTON,MsgPush>
Btn2 DatosControl <?,ClaseBoton, 260,24,164,48,BS_DEFPUSHBUTTON,MsgDefPush>
Btn3 DatosControl <?,ClaseBoton, 24, 120,164,48,BS_AUTOCHECKBOX,MsgCheckBox>
Btn4 DatosControl <?,ClaseBoton,260,120,164,48,BS_AUTOSTATE,Msg3State>
Btn5 DatosControl <?,ClaseBoton,24,216,164,48,BS_AUTORADIOBUTTON,MsgRadio1>
Btn6 DatosControl <?,ClaseBoton,260,216,164,48,BS_AUTORADIOBUTTON,MsgRadio2>

Lbll DatosControl <?,ClaseTexto,24,168,164,24,HH_CENTER,MsgInactivo>


---


; Textos para los controles
MsgPush db 'Botón normal',0
MsgDefPush db 'Botón por defecto',0

```

```

MsgCheckBox db 'Marcar/üesmarcar',0
Msg3State db 'Tres estados',0
MsgRadio1 db 'Opción 1',0
MsgRadio2 db 'Opción 2',0

MsgThaotivo db 'Inactivo',ü
MsgActivo db 'Activo',0

; Nombres de las clases de los controles
ClaseRoton db 'BUTTON', 0
ClaseTexto db 'STATIC', 0

; Espacio temporal para recuperar el
; texto de los controles
Texto db 128 dup(0)

```

.Code

Main:

```

; Obtenemos el identificador de la
; aplicación en EAX
invoke GetModuleHandle, 0
; y lo colocamos donde debe estar
; en la estructura
mov [ClaseVentana.hlnstance], eax

; Registraremos la clase de ventana
invoke RegisterClassEx, offset ClaseVentana

; Creamos la ventana
invoke CreateWindowEx, 0,
    Offset NombreClase,
    Offset NombreVentana,
    WS_VISIBLE Or WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CWJJSSEDEFAULT,
    0, 0, ClaseVentana.hlnstance, 0

; Si no ha podido crearse la ventana
or eax, eax
; salir
iz Salir

; Guardamos el manejador
mov Manejador, eax

; Apuntamos al primer control a crear
mov esi, Offset Bt.nl
; Número de controles a crear
mov ecx, NumControles

```

CreaControles:

```

; Guardamos los registros ECX y ESI
push ecx
push esi

```

```
; Añadimos los estilos apropiados
or [esi] .DatosControl.Estoilo,
    WS_CHILD Or WS_VISIBLE
; Y creamos el control
invoke CreateWindowEx, 0,
[esi].DatosControl.Clase,
[esi].DatosControl.Texto,
[esi].DatosControl.Estoilo,
[esi].DatosControl.PosX,
[esi].DatosControl.PosY,
[esi].DatosControl.Ancho,
[esi].DatosControl.Alto,
Manejador, 0, ClascVentana.hlnstance, 0

; Guardamos su manejador
rmov [esi] .DatosControl.Manejador, eax

; Recuperamos los registros
pop esi
pop ecx

; Hacemos avanzar ESI
add esi, size DatosControl

; Y seguimos hasta terminar el bucle
loop CreaControles
```

```
; Bucle de proceso de mensajes
```

Bucle:

```
; Obtenemos un mensaje de la cola
invoke GetMessage, Offset Mensaje, 0, 0, 0
; Si EAX = 0 hay que salir
or eax, eax
jz Salir

; En caso contrario lo despachamos
invoke DispatchMessage, Offset Mensaje

; y seguimos esperando mensajes
jmp Bucle
```

Salir: ; Se ha recibido EAX = 0 hay que terminar
; Facilitamos en EAX el contenido
; del wParam del mensaje
mov eax, Mensaje.wParam
; y volvemos
ret

```
; Este procedimiento deberia procesar
; los mensaje provenientes de las ventanas
```

```

ProcVentana proc hWnd:HWND, uMesg:UINT,
               wParam:WPARAM, lParam:LPARAM

        ; Comprobamos el mensaje
        cmp uMesg, WM_DESTROY
        je Quit ; y saltamos al punto adecuado

        ; Ver si es la pulsación de un botón
        cmp uMesg, WM_COMMAND
        je ClicBoton

        ; Pedimos a Windows que procese el
        ; mensaje él mismo
        invoke DefWindowProc, hWnd,uMesg,wParam,lParam

        ret ; y volvemos

ClicBoton: ; Se ha pulsado uno de los botones

        ; Obtenemos el texto que contiene el botón
        ; pulsado
        invoke SendMessage, lParam, WM_GETTEXT,
               128, offset Texto
        ; y lo mostramos como título de ventana
        invoke SendMessage, Manejador, WM_SETTEXT,
               0, offset Texto

        ; Obtenemos el estado del botón AUTOCHECKBOX
        invoke SendMessage,Btn3.Manejador,BM_GETCHECK,0,0
        ; Si no está pulsado
        cmp eax, BST_CHECKED
        jne Inactivo ; saltamos

        ; Indicar el estado del botón en la
        ; etiqueta de texto
        invoke SetWindowText,Lbl1.Manejador,
               offset MsgActivo

        ret ; volver

Inactivo:
        invoke SetWindowText,Lbl1.Manejador,
               offset MsgInactivo

        ret

Quit: ; Se ha pedido la salida
      ; Llamamos a PostQuitMessage
      invoke PostQuitMessage, 0
      ; Ponemos EAX a 0
      xor eax, eax
      ret ; y volvemos

ProcVentana Endp

end Main

```

Para facilitar el proceso de creación de los controles, y no tener que invocar casi una decena de veces a la función CreateWindowEx con toda su lista de parámetros, se ha definido al inicio del módulo una estructura capaz de contener el manejador de cada control, su clase, posición, tamaño, estilo y título. Es importante que las variables definidas con esta estructura como base estén contiguas en el código, ya que de lo contrario el bucle codificado más adelante no funcionaría adecuadamente.

Al recibir el mensaje de pulsación de un botón, independientemente del que sea, se obtiene el texto, mediante el mensaje WMGETTEXT, y se establece como título de la ventana. El resultado es que al ir pulsando los botones en el título de la ventana aparecerá el nombre del botón pulsado. Además, se utiliza el mensaje BM_GETCHECK para comprobar el estado del tercer botón, indicándolo mediante un mensaje en una etiqueta de texto que se ha dispuesto debajo. En la figura 28.4 puede ver el aspecto de la ventana tras haber pulsado algunos de los botones que aparecen en ella.

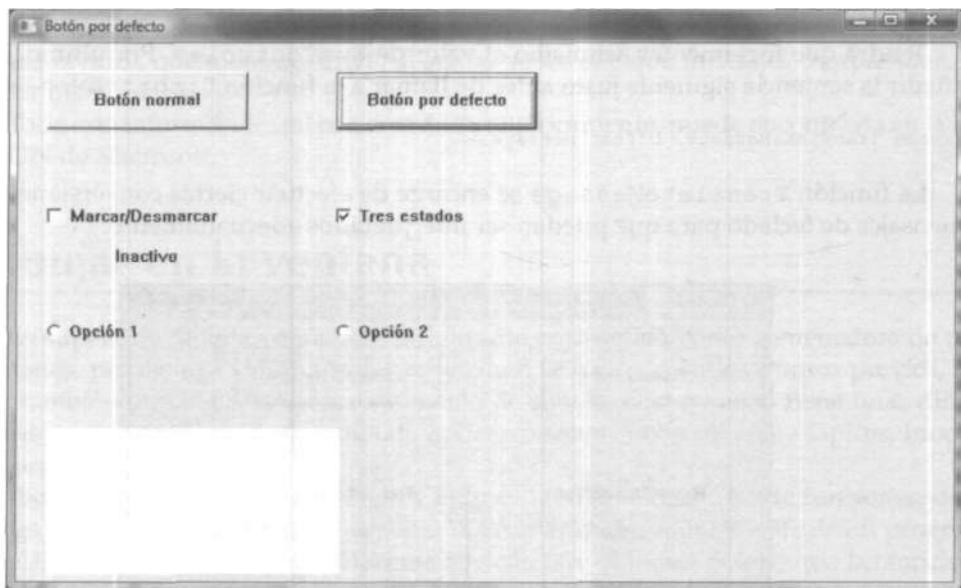


Figura 28.4. La ventana con los botones.

Textos

Si necesitamos mostrar un texto en el interior de una ventana, el método más fácil consiste en incluir una ventana de clase STATIC, lo que se conoce normalmente como una etiqueta de texto. Lo hemos hecho en el ejemplo previo, justo debajo del tercero de los botones. El texto que contiene una etiqueta puede modificarse en cualquier momento mediante el mensaje WM_SETTEXT. Existen múltiples estilos que, por ejemplo, establecen la alineación del texto en el interior del control y si debe mostrarse una imagen en lugar de un texto.

Cuando el texto debe ser introducido o editado por el usuario, en lugar del control STATIC se utilizará el control EDIT. Su apariencia es la de un área rectangular en la que aparece el texto y un cursor, de tal forma que el usuario puede desplazarse y efectuar tareas básicas de edición como el borrado, la inserción, copiar y pegar a través del portapapeles, etc.

Los estilos de EDIT permiten configurar el área de entrada de datos consiguiendo, por ejemplo, campos limitados a un cierto número de caracteres y que sólo permiten la introducción de números, campos que convierten automáticamente a mayúsculas o minúsculas, que facilitan la edición de múltiples líneas de texto, etc.

Basándonos en el programa de ejemplo del punto anterior, añadiendo el elemento siguiente a la lista de estructuras con datos de controles conseguiremos añadir un área de edición de varias líneas:

```
Editl DatosControl <?, ClaseEdit, 24, 240, 240, 24*5, WS_BORDER Or ES_LEFT Or ES_MULTILINE  
Or ES^AUTOSCROLL Or ES_WANTRETURN, M.sgEdit>
```

Tendrá que incrementar asimismo el valor de NumControles. Por último, deberá añadir la sentencia siguiente justo antes de llamar a la función DispatchMessage:

```
invoke TranslateMessage, Offset Mensaje
```

La función TranslateMessage se encarga de efectuar ciertas conversiones en los mensajes de teclado para que puedan ser interpretados adecuadamente.

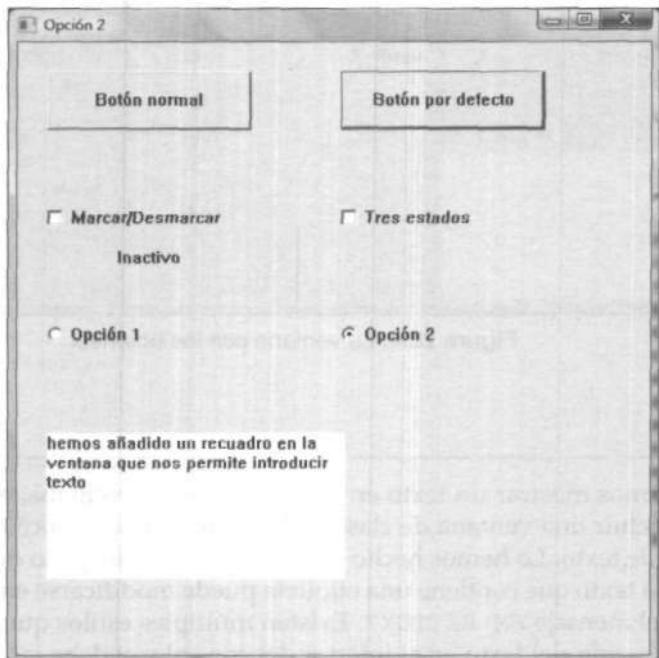


Figura 28.5. La ventana con el recuadro de texto.

Nota

Puede detectar las actuaciones sobre el área de edición de texto interceptando el mensaje WM_COMMAND e identificando el control mediante el parámetro lParam. Para manipular su contenido utilice los mismos mensajes que ya conoce: WM_GETTEXT y WM_SETTEXT.

Otros controles

Aunque tan sólo se ha demostrado el uso de tres de las clases de controles con que cuenta Windows: los botones, el texto estático y los campos de edición, los fundamentos para usar el resto de tipos de controles son exactamente los mismos. Lo único que necesitará será información de referencia para saber qué estilos específicos se aplican a cada control, qué mensajes envía al actuar sobre él y qué mensajes comprender para poder manipularlo.

Toda esta información, tal como se decía anteriormente, puede encontrarla en la sede MSDN de Microsoft.

Dibujar en la ventana

La superficie de una ventana es útil no sólo para actuar como contenedora de otras ventanas, por ejemplo los controles tal y como se ha visto en los puntos previos, sino que también puede usarse como un lienzo de dibujo. Dicho lienzo tiene unas dimensiones y unas coordenadas, pudiendo aplicarse sobre él objetos como lápices, brochas, regiones o tipos de letra.

Todos estos elementos se definen y aplican mediante multitud de funciones, de las cuales vamos a conocer tan sólo media docena a fin de poder construir un programa más simbólico de lo que puede hacerse que con otro objetivo, puesto que las funciones CDI (*Graphics Device Interface*), que son las usadas para dibujar, son cientos y algunas de cierta complejidad, como casi todo en Windows.

Cuando el sistema operativo necesita que una ventana muestre su contenido, porque se hace visible por primera vez, cambia de tamaño o deja de estar tapada por otra, genera automáticamente un mensaje WM_PAINT que va dirigido a la ventana interesada.

Ante este mensaje, que hasta el momento hemos ignorado, deberemos dar los pasos siguientes:

- Obtener los parámetros que necesitamos para poder dibujar, llamando para ello a la función BeginPaint.
- Dibujar utilizando las funciones que se precisen del CDI.
- Notificar al sistema que hemos terminado, invocando a la función EndPaint.

Nota

Las funciones `BeginPaint` y `EndPaint`, así como todas las relativas a GDI, se encuentran en la biblioteca de enlace dinámico `gdi32.dll`, por lo que será necesario importar el archivo `gdi32.inc` y la biblioteca `gdi32.lib`.

Las funciones para dibujar son multitud, necesitando todas ellas lo que se conoce como un *contexto de dispositivo* o DC (*Device Context*). A la hora de aplicar un cierto lápiz, brocha o tipo de texto, mover el puntero de dibujo, trazar una línea recta o curva, dibujar polígonos, etc., siempre facilitaremos ese elemento como primer parámetro.

Lógicamente, cada ventana tendrá un DC exclusivo para su uso. En el caso de los mensajes `WM_PAINT`, ese DC se obtiene con la llamada a `BeginPaint`, a la cual facilitaremos como primer parámetro el manejador de la ventana y como segundo la dirección de una estructura `PAINTSTRUCT`, definida en `Windows . inc`. El primer campo de esa estructura, `hdc`, contiene el DC.

Disponiendo ya del DC, podemos establecer el puntero de dibujo en la posición que nos interese con la función `MoveToEx`, dibujar líneas con `LineTo`, imprimir textos con `TextOut`, etc. Tan sólo tenemos que consultar la lista de funciones disponibles.

Partiendo del programa inicial que mostraba una ventana vacía, al principio del capítulo, vamos a añadir una serie de modificaciones para conseguir dibujar en ella una línea y un texto. Lo primero será insertar al principio del código estas dos directivas:

```
; Añadimos los datos de GDI
include gdi32.inc
includelib gdi32.lib
```

Sin ellas no podríamos usar ninguna de las funciones GDI. A continuación incluiremos en el segmento de datos la definición de estos dos campos:

```
; Datos necesarios para dibujar
Datos PAINTSTRUCT <>
Texto db 'Dibujando en ensamblador',0
```

`Datos` es la variable que nos permitirá obtener toda la información que necesitamos para dibujar, mientras que `Texto` es simplemente una cadena de texto que mostraremos en el interior de la ventana.

Desplazándonos ya hasta el bucle de proceso de mensajes, después de comprobar si el mensaje es `WM_DESTROY` añadiremos las dos sentencias mostradas a continuación, cuyo objetivo es desviar la ejecución a la etiqueta `Dibujar` en caso de que se reciba el mensaje `WM_PAINT`.

```
; Si hay que dibujar la ventana
emp uMsg, WM_PAINT
je Dibujar ; Saltar
```

Por último, tenemos el bloque de instrucciones que se encargarán de dibujar la línea y el texto en la ventana:

```

Dibujar: ; Hay que dibujar en la ventana

; Obtenemos los datos que necesitamos
invoke BeginPaint,hWnd,offset Datos

; Nos movemos a una cierta posición
invoke MoveToEx,Datos.hdc, 100,100,0

; Dibujamos una linea
invoke LineTo,Datos.hdc, 320, 240

; Establecemos colores de fondo y tinta
; para el texto
invoke SctBkColor,Datos.hdc,0FFFFFFh
invoke SetTextColor,Datos.hdc,0FFh
; y lo imprimimos
invoke TextOut.,Datos.hdc, 320, 240,
    offset Texto,sizeof Texto

; Hemos terminado de dibujar
invokfi EndPaint,int,hWnd,offset Datos

ret ; volvemos

```

Observe que se utiliza la orden `sizeof` para obtener la longitud de la cadena de caracteres a mostrar, último parámetro que precisa la función `TextOut`. Si utilizásemos `size` obtendríamos el valor 1, ya que `Texto` está definido como una secuencia de bytes, mientras que `sizeof` se encarga de contar el número de caracteres y devolver el valor adecuado. Al ejecutar el programa obtendrá el resultado que puede observarse en la figura 28.6. No es una obra de arte, desde luego, pero sí un punto de partida para probar las demás funciones de dibujo con que cuenta Windows.

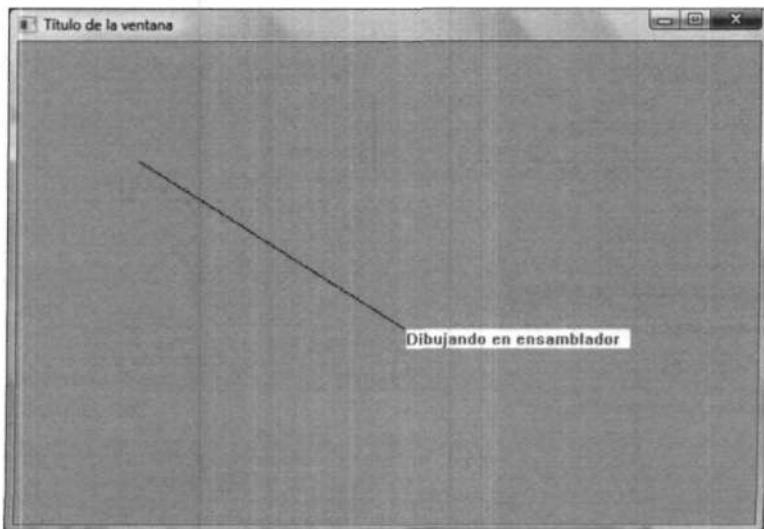


Figura 28.6. Aspecto de la ventana con la línea y el texto.

Resumen

Este capítulo nos ha servido para aprender a crear nuestras propias aplicaciones para Windows usando el lenguaje ensamblador, obteniendo así ejecutables considerablemente más pequeños de los que generaría cualquier herramienta actual, al tiempo que se obtiene un mejor rendimiento. No cabe duda, sin embargo, de que crear aplicaciones para Windows con este lenguaje resulta complejo, ya que es preciso conocer multitud de funciones, sus parámetros, códigos de retorno, etc. Toda esta información, como se ha dicho varias veces a lo largo del capítulo, puede encontrarse en MSDN. Si es programador en algún lenguaje de alto nivel para este sistema, como Visual Basic o Delphi, posiblemente ni conozca las funciones del API de Windows, ya que dichas herramientas las ocultan tras los componentes y clases de alto nivel.

Nuestro objetivo en el próximo capítulo será similar al que hemos conseguido en éste, pero ocupándonos de los servicios de Linux en vez de los de Windows. Algo en común es que, una vez se hayan adquirido las bases sobre cómo crear aplicaciones para ese sistema, todo se basa en disponer de información de referencia sobre funciones y saber cómo llamarlas, y poco más.

29

Servicios de Linux

En el capítulo previo ha aprendido a crear aplicaciones para un sistema operativo, Windows en ese caso, para lo cual tan sólo ha tenido que conocer los servicios propios de dicho sistema, ya que el lenguaje ensamblador es exactamente el mismo y, por tanto, se usan las mismas instrucciones que conocimos en los primeros capítulos del libro y se utilizaron en los ejemplos desarrollados para DOS en los siguientes.

Como en Windows, al trabajar en Linux nos encontramos con un sistema operativo de 32 ó 64 bits, no de 16 como DOS, y por tanto el entorno de ejecución es el modo protegido. Esto significa que, al igual que en Windows, no tenemos acceso directo a la BIOS ni a posiciones absolutas de memoria, como el área que ocupa la pantalla, a no ser que desarrollemos un controlador de dispositivo o, en general, una aplicación que se ejecute en modo *kernel*, con los mismos privilegios que el núcleo de sistema.

Sin embargo, la mayor parte de los programas que se crean para Linux, como para los demás sistemas, son aplicaciones de usuario final, para lo cual basta con conocer qué servicios nos ofrece el sistema y cómo podemos acceder a ellos.

El objetivo de este capítulo, como el del anterior, es mostrarle los pasos básicos para crear una aplicación para Linux, sin entrar en la descripción de todos los servicios y sus parámetros ya que ésta es una información de referencia que, por regla general, puede encontrar en cualquier distribución de Linux actual.

Herramientas necesarias

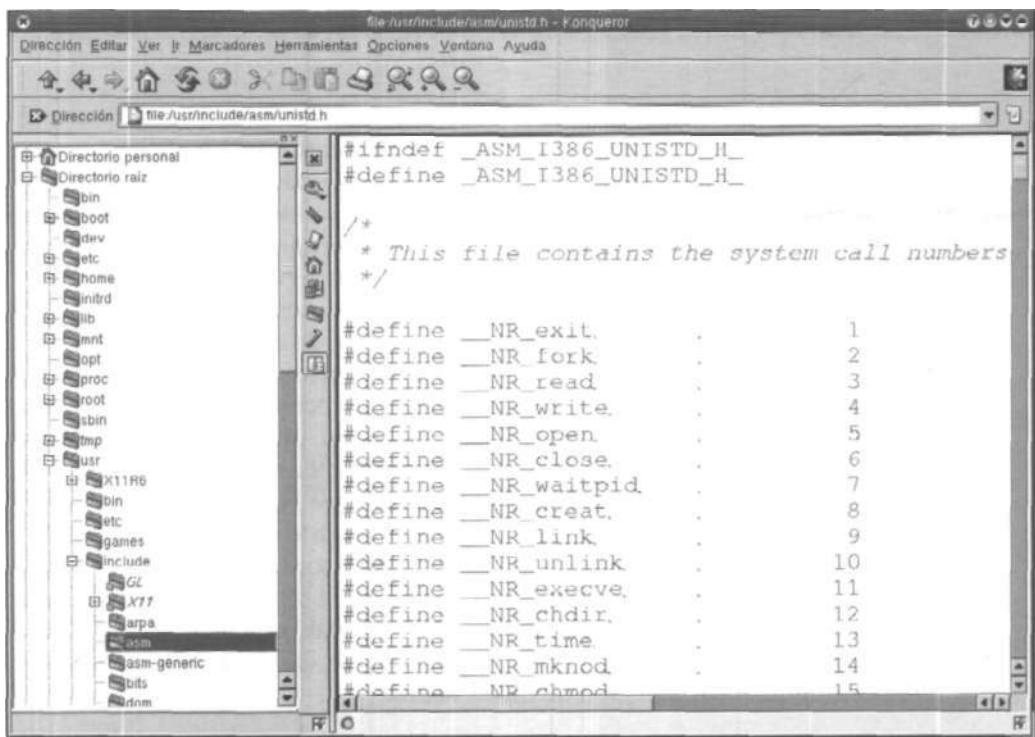
En un capítulo previo vimos que podíamos crear una aplicación Linux simplemente con un ensamblador, como puede ser NASM, y un enlazador, finalidad para la cual sirve el propio compilador gcc presente en toda instalación Linux. Obviamente puede optar

por otras herramientas, como el ensamblador as o gas y el enlazador ld, debiendo en ese caso adaptarse a la sintaxis propia de esas herramientas. En este capítulo, puesto que ya lo conocemos, será NASM el ensamblador que utilicemos.

Dependiendo de los recursos que vayamos a utilizar para implementar nuestra funcionalidad, accediendo a los servicios del núcleo o bien a través de la biblioteca estándar de funciones de C del sistema, necesitaremos unos elementos u otros, principalmente archivos de cabecera con los que conocer el código de cada uno de los servicios y, eventualmente, su lista de parámetros.

También puede necesitar las bibliotecas correspondientes a servicios no básicos a los que desee acceder.

En el directorio /usr/include/asm encontrará distintos archivos de cabecera, como el archivo unistd.h que aparece en la figura 29.1. Éste concretamente contiene una enumeración de los distintos servicios del núcleo, lo que se conoce como *kernel*, asignándole un identificador simbólico al código de cada uno de ellos.



```
#ifndef __ASM_I386_UNISTD_H__
#define __ASM_I386_UNISTD_H__

/*
 * This file contains the system call numbers
 */

#define __NR_exit          .      1
#define __NR_fork          .      2
#define __NR_read           .      3
#define __NR_write          .      4
#define __NR_open            .      5
#define __NR_close           .      6
#define __NR_waitpid        .      7
#define __NR_creat           .      8
#define __NR_link             .      9
#define __NR_unlink          .     10
#define __NR_execve          .     11
#define __NR_chdir           .     12
#define __NR_time            .     13
#define __NR_mknod           .     14
#define __NR_chmod           .     15
```

Figura 29.1. Localizamos los archivos de cabecera.

Puede ocurrir que el servicio que quiere usar no esté disponible en el sistema y, por tanto, deba instalarlo de antemano. Para ello usará la herramienta de instalación de paquetes de la distribución que tenga instalada. Consulte la documentación en línea si no sabe cómo hacerlo.

Servicios del núcleo de Linux

^

Como sistema operativo de 32 bits que opera en modo protegido, Linux impide la invocación directa de los servicios de la BIOS y, lógicamente, los servicios DOS no existen, igual que no están presentes los de Windows a menos que utilicemos alguno de los emuladores que hay disponibles para esos dos sistemas.

A cambio contamos con los servicios del núcleo de Linux, accesibles a través de la interrupción 8 Oh. Podríamos comparar ésta con la interrupción 2 lh del DOS, ya que da acceso a prácticamente todos los servicios del sistema, aquellos que son fundamentales para todas las aplicaciones.

Antes de invocar a la interrupción 8 Oh hay que depositar en el registro eax el número del servicio al que quiere llamarse. En el archivo de cabecera indicado anteriormente, unistd.h, se facilita una serie de constantes que hacen innecesario conocer de memoria los códigos numéricos, lo cual simplifica el trabajo y, además, hace que el código sea algo más legible.

Los parámetros, como en el caso de las interrupciones DOS, se facilitan en registros. A pesar de que los invoquemos a través de una interrupción, los servicios del núcleo de Linux están pensados para ser utilizados principalmente desde el lenguaje C, lo cual no es extraño ya que la mayor parte de este sistema operativo se encuentra escrita en dicho lenguaje. Cada uno de los servicios se corresponde con una función de C que toma una cierta lista de parámetros. Éstos se entregarán, al trabajar desde ensamblador, en los registros ebx, ecx, edx, es i, edi y ebp. Lógicamente, dependiendo del número de parámetros que tome cada función, parte de esos registros quedarán sin uso, pero siempre se usarán en ese orden. Si el servicio necesita tres parámetros irán en ebx, ecx y edx, si requiere sólo uno se facilitará en ebx.

Nota

Como podrá ver en el citado archivo unistd.h, el núcleo de Linux ofrece casi 250 servicios diferentes, por lo que en los puntos siguientes conoceremos sólo algunos de ellos.

Devolución del control al sistema

Lo primero que necesitamos saber, a la hora de escribir programas en ensamblador para Linux, es cómo devolver el control al sistema.

En el ejemplo escrito en el segundo capítulo nos limitábamos a utilizar la instrucción ret, pero éste no es el método habitual.

El primer servicio de la interrupción 8 Oh, el número Olh, corresponde a la instrucción exit que, como puede imaginar, pone fin a la ejecución del proceso actual y devuelve el control al que le invocó, que será el sistema si se trata de un programa puesto en marcha desde una consola.

La forma de usar este servicio es similar a la empleada con el servicio 4Ch de la interrupción 2 lh del DOS. Pondremos en el registro eax el código del servicio, en este caso 0 lh, y en ebx el código de salida.

Verá de inmediato cómo utilizar este servicio en el primer ejemplo que se desarrolla en los puntos siguientes.

Entrada y salida por consola

Operando siempre en modo texto, en una consola típica de Linux, la consola aparece a ojos del sistema como una pareja de archivos o flujos de información. El primero, del que sólo puede leerse, tiene asociado el identificador 0 y correspondería a la entrada por teclado, mientras que el segundo, en el que sólo podemos escribir, tiene el identificador 1 y sería la pantalla.

Estos dos canales se conocen habitualmente como `stdin` y `stdout`, o entrada estándar y salida estándar.

Los servicios usados para recuperar o mostrar datos por consola, por tanto, son los mismos que usaríamos para escribir o leer datos de un archivo, con la diferencia de que los identificadores ya están preestablecidos y, además, son archivos que no necesitamos abrir ni cerrar.

Para leer una secuencia de caracteres desde la consola se usa el servicio 03h, correspondiente a la función `read`.

Los parámetros necesarios son:

- eax: El código del servicio, 03h en este caso.
- ebx: Identificador del archivo del que va a leerse. En este caso sería 0.
- ecx: Puntero a un área de memoria en la cual se dejarán aquellos caracteres que son obtenidos.
- edx: Número máximo de caracteres a leer.

Hay que tener en cuenta que este servicio no restringirá el número de caracteres que el usuario puede introducir por la consola, sino que se limitará a tomar como máximo el número que se haya indicado. El resto quedarán disponibles para la siguiente lectura, como ocurriría en un archivo cualquiera.

Si todo va bien, en eax recibiremos el número de caracteres que se han leído de manera efectiva, número que puede ser menor al solicitado pero nunca mayor.

En caso de que eax sea negativo significará que se ha producido un error. Por regla general, Linux siempre notifica los errores facilitando números negativos que hacen referencia a una determinada situación que identifica el problema.

El servicio complementario, que usaremos para enviar información a la consola, es el 04h, correspondiente a la función write. Los parámetros son básicamente los mismos: el identificador del archivo, que será 1 en este caso; el puntero al área donde está la cadena y el número de caracteres a escribir. El valor de retorno, en eax, también será el número de caracteres transferidos.

Conociendo únicamente estos dos servicios, podríamos construir un programa que solicitase el nombre del usuario por la consola y lo devolviese a continuación precedido de un saludo. Es algo sencillo, pero nos permitirá ver nuestro primer programa interactivo en Linux. El código sería el siguiente:

```
; Constantes de acceso a los servicios
#define __NR_exit 1
#define __NR_read 3
#define __NR_write 4

; Punto de entrada a la aplicación
global main

; Datos con valor inicial
section .data

Pregunta db '¿Cómo te llamas?',10,0
Saludo db 'Hola ',0

; Datos sin valor inicial
section .bss

Nombre resb 128

; Código del programa
section .text
main:
    ; Escribimos la cadena de texto con
    ; la pregunta
    mov eax,__NR_write
    mov ebx, 1 ; salida por consola
    mov ecx, Pregunta
    mov edx, 17 ; longitud
    int 80h

    ; Leemos como máximo 127 caracteres
    mov eax,__NR_rread
    mov ebx, 0
    mov ecx, Nombre
    mov edx, 127
    int 80h

    ; Guardamos el número
    ; de caracteres introducidos
    push eax

    ; Mostramos el saludo
    mov eax,__NR_write
```

```

mov ebx, 1
mov edx, 5
mov ecx, Saludo
int 80h

; y el dato recogido de la consola
mov eax, __NR_write
mov ebx, 1

pop edx ; recuperamos el número de caracteres
mov ecx, Nombre
int 80h

; Devolvemos el control al sistema
mov eax, __NR_exit
xor ebx, ebx ; con el código 0
int 80h

```

El comportamiento del programa es el que puede verse en la figura 29.2. Observe que el campo en el que va a recogerse la información leída desde la consola, Nombre, está definido en una sección llamada bss en lugar de en la sección data. Esto es habitual en Linux.

La sección bss tiene la finalidad de contener aquellos campos que no tienen un contenido inicial, siendo simplemente una reserva de memoria.

```

root@mdk nasm]# ./nasm -f elf HolaInt80.asm
[root@mdk nasm]# gcc HolaInt80.o -oHolaInt80
[root@mdk nasm]#
[root@mdk nasm]#
[root@mdk nasm]# ./HolaInt80
¿Cómo te llamas?
Francisco
Hola Francisco
[root@mdk nasm]#
[root@mdk nasm]#
[root@mdk nasm]# █

```

Figura 29.2. El programa nos solicita el nombre y lo vuelve a mostrar por la consola.

Macros de ayuda

Como se aprecia en el programa anterior, la mayor parte del código escrito al crear una aplicación para Linux se dedica a preparar los valores en los registros correspondientes. Para simplificar este trabajo lo más fácil es definir algunas macros a las que, facilitándoles los parámetros, generen las instrucciones pertinentes. Un ejemplo podrían ser las tres macros siguientes:

```
; Definimos una macro genérica para
; invocar a la interrupción 80
; recibiendo como máximo 4 parámetros
%macro INT80 4
    mov eax, %1
    mov ebx, %2
    mov ecx, %3
    mov edx, %4
    int 80h
%endmacro

; Macro para leer de consola
%macro LeeConsola 2
    TNTRO____NR_vaad, 0, 9;1 , %2
%endmacro

; Macro para escribir en consola
%macro EscribeConsola 2
    INT80____NR_write, 1, %1, Y;
?.endmacro
```

La primera es una macro de tipo genérico que invoca a la interrupción 80 siempre que los parámetros a facilitar, aparte del servicio en eax, no sean más de tres. Las otras dos macros simplifican las operaciones de lectura y escritura por consola, sirviéndose de la macro anterior. Tras incluir estas definiciones al inicio del módulo de código, las sentencias que seguían a la etiqueta main pueden sustituirse en su totalidad por éstas:

```
main:
    ; Escribimos la cadena de texto con la pregunta
    EscribeConsola Pregunta, 17

    ; Leernos como máximo 127 caracteres
    LeeConsola Nombre, 127

    ; Guardamos el número
    ; de caracteres introducidos
    push eax

    ; Mostramos el saludo
    EscribeConsola Saludo, 5
    ; y el dato recogido de la consola
    pop edx
    EscribeConsola Nombre, edx

    ; Devolvemos el control al sistema
    TNT80____NR_exit, 0,0,0
```

Como puede verse, el programa es, aparte de más breve, mucho más fácil de comprender y, lo más importante, el resultado que se genera al ensamblarlo es exactamente el mismo que en la versión anterior, ya que las macros son preprocesadas por NASM y generan exactamente el mismo código.

Trabajo con archivos

En los sistemas Unix, y Linux no es una excepción, todos los dispositivos aparecen a ojos del sistema como si fuesen archivos, por lo que sabiendo cómo abrir un archivo y escribir y leer de él podemos, virtualmente, acceder a cualquier dispositivo. Las funciones para leer y escribir ya las conocemos, son las dos que hemos empleado para operar sobre la consola. Además necesitaremos saber cómo abrir el archivo, entre otras tareas.

A la hora de hacer referencia a un archivo es posible usar caminos relativos o absolutos, es decir, usar como base el camino desde el que está ejecutándose el programa o, por el contrario, facilitar un camino completo y absoluto que parte desde la raíz.

Nota

Recuerde que Linux no existen letras de unidad, como en DOS o Windows, sino que los volúmenes insertados en unidades de disquete, CD-ROM u otros discos duros se montan como ramas del sistema de archivos principal. Utilice los comandos `man mount` y `man umount` para obtener más información sobre el proceso de montado y desmontando de unidades.

Apertura y creación de archivos

Para operar sobre un archivo cualquiera, o dispositivo si es el caso, lo primero que necesitamos es abrirlo. Para ello usaremos el servicio 05h de la interrupción 8 Oh, equivalente a la función `open`. Los parámetros necesarios, aparte del número de servicio facilitado en `eax`, son los siguientes:

- `ebx`: Contendrá la dirección de una cadena de caracteres terminada con nulo, indicando el nombre del archivo o dispositivo a abrir. Como se ha dicho antes, puede ser una referencia relativa o absoluta, según nos interese.
- `ecx`: Modo de acceso al archivo. Será un conjunto de bits mediante el cual se indicará el modo de apertura del archivo.
- `edx`: Permisos del archivo en caso de que se abra creándolo.

Si todo va bien, y el archivo indicado puede abrirse sin problemas, obtendremos en el registro `eax` el descriptor de archivo, identificador que, por ejemplo, utilizaríamos con los servicios vistos antes para escribir o leer del archivo de igual forma que los hemos usado sobre la consola.

El segundo parámetro determina si el archivo se abrirá o creará, así como si tras la apertura podrá leerse, escribirse o añadirse información. Para obtener todos los detalles la mejor opción es consultar las páginas de manual de la instrucción open en su propia distribución de Linux.

En la figura 29.3, por ejemplo, puede verse el inicio de la documentación ofrecida por man open en mi propio sistema.

The screenshot shows a terminal window with the title bar "root@fedoral: /root/NASM/nasm - Konsole - Konsole". The menu bar includes "Archivo", "Sesiones", "Opciones", "Ayuda". The main area displays the man page for open(2). The title is "Llamadas al Sistema" and the section is "OPEN(2)".

NOMBRE
open, creat - abren y posiblemente crean un fichero o dispositivo

SINOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *camino, int flags);
int open(const char *camino, int flags, mode_t modo);
int creat(const char *camino, mode_t modo);
```

DESCRIPCIÓN
La llamada al sistema open() se utiliza para convertir una ruta en un descriptor de fichero (un pequeño entero no negativo que se utiliza en las operaciones de E/S posteriores como en read, write, etc.). Cuando la llamada tiene éxito, el descriptor de fichero devuelto será el descriptor de fichero más pequeño no abierto actualmente para el proceso. Esta llamada crea un nuevo fichero abierto, no

lines 1-25

Figura 29.3. Documentación de la instrucción open ofrecida por Linux.

Puede encontrar todas las constantes relativas a modo de apertura en el archivo /usr/include/asm/fcntl.h. Puede copiar aquellos que le interesen en su propio módulo de código para facilitar el acceso.

En la tabla 29.1 se enumeran algunas de las constantes de modo de apertura. Para crear un archivo, por ejemplo, usaríamos 0_CREAT, mientras que para abrir uno existente añadiéndole información utilizaríamos 0_APPEND. Recuerde que estos indicadores en realidad representan bits individuales que pueden ser combinados mediante una operación lógica or, salvo los tres primeros que son exclusivos, es decir, tan sólo puede aparecer uno de ellos.

Tabla 29.1. Constantes que representan los valores a introducir en ecx.

Constante	Significado
O_RDONLY	El archivo se abre sólo para leer de él.
O_WRONLY	El archivo se abre sólo para escribir en él.
O_RDWR	El archivo se abre para lectura y escritura.
O_CREAT	Crear el archivo en caso de que no exista.
O_EXCL	Combinado con el anterior, requiere que el archivo a crear no exista.
O_TRUNC	Inverso al anterior, eliminando el contenido actual del archivo a crear en caso de que ya exista.
O_APPEND	El archivo se abre sólo para escritura al final, añadiendo información.

En cuanto al parámetro facilitado en el registro edx, su objetivo es establecer los permisos en caso de que se cree el archivo.

Éstos determinarán quién puede leer y escribir en el archivo, siendo necesarios sólo en caso de que se cree el archivo. Algunos de esos valores están representados por las constantes de la tabla 29.2.

Tabla 29.2. Constantes que representan los permisos a aplicar al archivo.

Constante	Significado
S_IRUSR	El archivo puede ser leído por el dueño.
S_IWUSR	El archivo puede ser escrito por el dueño.
S_IRGRP	Los usuarios del grupo al que pertenece el dueño pueden leer el archivo.
S_IWGRP	Los usuarios del grupo al que pertenece el dueño pueden escribir en el archivo.
S_IROTH	El archivo puede ser leído por otros usuarios.
S_IWOTH	El archivo puede ser escrito por otros usuarios.

Como alternativa al servicio que acaba de describirse, siempre que deseemos crear un archivo podemos usar en su lugar el servicio 08h. Éste representa la función `creat` y toma en ebx la dirección donde está el camino y nombre del archivo y en ecx los bits de permiso.

El modo de apertura se asume que es O_CREAT or O_WRONLY or O_TRUNC, es decir, el archivo se crea, eliminando su contenido si es necesario, y se deja abierto solamente para escritura.

El parámetro devuelto, en eax como siempre, será el descriptor de archivo que emplearíamos para operar sobre él.

Una vez que se haya finalizado el trabajo sobre un archivo o dispositivo, como en cualquier otro sistema es necesario ejecutar la operación de cierre. El servicio es el 0 6h, equivalente a la función cióse, debiendo facilitarse como único parámetro el descriptor del archivo, en ebx.

El puntero de lectura/escritura

Cuando se abre o crea un archivo, el puntero de lectura y escritura se encuentra siempre al inicio a menos que se haya usado el modo 0_APPEND, caso en el que ese puntero siempre se llevará al final antes de ejecutar cualquiera operación de escritura. Para manipular ese puntero, alterando la posición del archivo en que nos encontramos, tenemos a nuestra disposición el servicio 13h, correspondiente a la función lseek. Al llamar a este servicio deberemos facilitar los parámetros enumerados a continuación:

- ebx: Descriptor del archivo cuyo puntero va a manipularse, obtenido previamente con una llamada a open o creat.
- ecx: Número de bytes a desplazarse desde el punto indicado como referencia.
- edx: Punto de referencia para el desplazamiento. Puede ser uno de los indicados en la tabla 29.3.

Si todo va bien, tras invocar a este servicio encontraremos en el registro eax la posición actual en el archivo, expresada en número de bytes desde el principio. Como puede ver, se trata de un comportamiento similar al del servicio equivalente de la interrupción 2 lh del DOS que aprendimos a manejar en un capítulo previo.

Tabla 29.3. Puntos de referencia para la función lseek.

Constante	Valor	Punto de referencia
SEEK_SET	0	Inicio del archivo.
SEEK_CUR	1	Posición actual en el archivo.
SEEK_END	2	Final del archivo.

Constantes y maeros

Para facilitar el uso de los servicios que acaban de describirse en los puntos previos, lo más apropiado es definir las constantes y maeros que después se utilizarán en los programas en un archivo independiente, de tal forma que pueda incluirse allí donde sea necesario.

A continuación puede encontrar una primera propuesta de lo que podría ser el contenido de este archivo, al que se ha denominado Archivos.inc. Lógicamente puede ampliarse con otras constantes adicionales y macros más elaboradas que, por ejemplo, faciliten la longitud de un archivo, la posición actual, etc.

```
; Este archivo contiene constantes y
; macros que simplifican el trabajo con
; archivos.

; Constantes de acceso a los servicios
#define __NR_exit 1
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_creat 8
#define __NR_lseek 19

; Constantes de modo de apertura
#define O_RDONLY 0000h
#define O_WRONLY 00001h
#define O_RDWR 00002h
#define O_APPEND 02000h
#define O_CREAT 00100h

; Y permisos
#define S_IRUSR 00400h
#define S_IWUSR 00200h

; Constantes para movernos
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2

; Definimos una macro genérica para
; invocar a la interrupción 80
; recibiendo como máxima 4 parámetros
%macro INT80 4
    mov eax, %1
    mov ebx, %2
    mov ecx, %3
    mov edx, %4
    int 80h
%endmacro

; Macro para leer de consola
%macro LeeConsola 2
    INT80 __NR_read, 0, %1, %2
%endmacro

; Macro para escribir en consola
%macro EscribeConsola 2
    ,INT80 __NR_write, 1, %1, 12
%endmacro
```

```

; Macro para leer de un archivo
%macro LeeArchivo 3
    INT80    NR read,  II,  %2,  %3
%endmacro

; Macro para escribir en un archivo
%macro EscribeArchivo 3
    INT80____NR_write,  %1,  %2,  %3
%endmacro

; Macro para abrir archivo
%macro AbreArchivo 2
    INT80____NR_open,  %1,  %2,  0
%endmacro

; Macro para crear archivo
%macro CreaArchivo 2
    INT80____NR_creat, %i,  %2,  0
%endmacro

; Macro para cerrar un archivo
%macro CierraArchivo 1
    INT80____NR_close, %1,  0,  0
%endmacro

; Macro para desplazar el puntero
%macro MuevePuntero 3
    INT80____NR lseek,  %1,  %2,  %3
%endmacro

```

En el punto siguiente podrá ver un ejemplo de uso de estas constantes y macros que, como se ha dicho, puede usar en cualquier programa propio.

Acceso a la memoria de pantalla

Linux es un sistema operativo de 32 bits que opera en modo protegido, impidiendo que los procesos que se ejecutan en el nivel de usuario efectúen ciertas operaciones potencialmente peligrosas. Una de esas operaciones es el acceso directo a memoria, usando para ello direcciones como las conocidas en capítulos previos y que nos permitían, por ejemplo, manipular directamente el contenido de la pantalla. Al trabajar en modo protegido, como lo hace Linux, los registros de segmento no contienen direcciones de segmento, que es el modo de usarlos que conocemos, sino índices de descriptores. Es algo que conocerá, aunque básicamente, en el capítulo siguiente. No podemos, por tanto, asignar el valor OB8 0Oh al registro ES y esperar con ello poder leer o escribir en la pantalla.

Dispositivos ves y versa

Los controladores de dispositivos de Linux se ejecutan en el nivel de privilegios del núcleo, por lo que pueden acceder directamente a posiciones de memoria o puertos de entrada y salida sin ningún problema. Como se indicaba anteriormente, los controladores

de dispositivos de Linux se comportan, desde el punto de vista del programados como si fuesen archivos, por lo que podemos abrirlos y leer y escribir en ellos como lo haríamos en cualquier otro archivo.

Nota

Si es usuario habitual de Linux ya sabrá que los elementos que representan a los controladores de dispositivos se encuentran siempre en la rama /dev del sistema de archivos de Linux.

Si cuenta con una distribución de Linux actual, entre sus dispositivos encontrará varios con los nombres vcsN y vcsaN. Cada uno de ellos representa una de las consolas virtuales con que cuenta el sistema, ves1 corresponde a la primera consola virtual, vcs2 a la segunda y así sucesivamente. El dispositivo ves correspondiente a la consola actual.

Como se puede ver en la figura 29.4, por cada dispositivo vcsN existe un vcsaN. La diferencia es que los del primer tipo dan acceso sólo al texto contenido en la pantalla, mientras que los del segundo contienen caracteres y atributos, como se almacenarían en el segmento de pantalla que conocemos en DOS. La estructura de los dispositivos vcsaN es la siguiente:

- 1 byte que indica el número de líneas que tiene actualmente la pantalla.
- 1 byte que indica el número de columnas que tiene la pantalla.
- 1 byte indicando la línea en la que se encuentra el cursor.
- 1 byte indicando la columna en la que se encuentra el cursor.
- Una secuencia de bytes conteniendo caracteres y atributos. La longitud de esta secuencia será el número de líneas multiplicado por el de columnas y multiplicado por 2.

```
root@ndk nasn]# ls /dev/vcs*
/dev/vcs0  /dev/vcs30  /dev/vcs6#  /dev/vcsa2#  /dev/vcsa5#
/dev/vcs1#  /dev/vcs1#  /dev/vcsa0#  /dev/vcsa3#  /dev/vcsa6#
/dev/vcs2#  /dev/vcs5#  /dev/vcs1#  /dev/vcsa4#
[rooot@ndk nasn]# _
```

Figura 29.4. Dispositivos vcs y vcsa.

Hay que tener en cuenta que una consola no tiene que contar, necesariamente, con 80 columnas y 25 líneas, siendo posibles otras configuraciones. Por eso los dos primeros bytes del dispositivo contienen las dimensiones. Los dos bytes siguientes pueden servir tanto para saber la posición actual del cursor como para modificarla. Establecer el cursor en una cierta posición sería tan fácil como abrir el archivo y escribir la línea y columna en esas posiciones, no siendo necesaria la interrupción 10h ni nada parecido.

Guardar el contenido de la pantalla en un archivo

Conociendo la existencia y estructura de los dispositivos ves y vesa, podemos crear un programa que, haciendo uso de las constantes y macros previamente incluidas en Archivos.inc, guarde el contenido actual de la pantalla en un archivo. Si queremos que dicho archivo contenga sólo texto, para poder abrirlo desde cualquier editor, usaríamos el dispositivo ves, mientras que si estamos interesados también en los atributos la opción sería el dispositivo vesa.

El código siguiente, perteneciente al programa GuardaPantalla.asm, toma el contenido de la consola actual, representada por el dispositivo /dev/vesa, y lo guarda en un archivo en el directorio actual. Para ello tiene que crear un archivo, leer para lectura el dispositivo, obtener sus dimensiones para conocer el número de bytes que ocupa la pantalla, leer esa información del dispositivo y guardarla en el archivo. Observe que el número de bytes leídos se guarda también en el archivo para facilitar su posterior lectura.

```
%include "Archivos.inc"

; Punto de entrada a la aplicación
global main

; Datos con valor inicial
section .data

; Nombre del archivo destino
NombreArchivo db 'Pantallas.dat',0

; Nombre del dispositivo de pantalla
NombrePantalla db '/dev/vesa',ü

; Mensaje de error
MsgError db 'Se produce un error.',0

; Datos sin valor inicial
section .bss

; Para guardar los descriptores
; de archivo
fdPantalla resd 1
fdFinalida resd 1

; Datos a recuperar de la cabecera
; de la pantalla
Lineas resb 1
Columnas resb 1
```

```
Linea    resb 1
Columna   resb 1

; Para calcular los bytes que
; ocupa la pantalla
Bytes resd 1

; Espacio para la pantalla
Contenido resb 16384

; Código del programa
ssction .text
main:
; Abrimos el archivo de destino
CreaArchivo NombreArchivo, 0

; Comprobamos si hay error
or eax, eax
js Error

; guardamos el descriptor de archivo
mov [fdSalida], eax

; Abrimos la memoria de consola
AbreArchivo NombrePantalla,0_RDONLY
; guardamos el identificador
mov [fdPantalla],eax

; Leemos las dimensiones y posición del cursor
LeeArchivo [fdPantalla],Lineas,4

; Calculamos el número de bytes que ocupa
; la pantalla
xor eax, eax
mov al, [Lineas]
mov bl, [Columnas]
mov ex, 2
muí bl
muí ex
; y lo guardamos
mov [Bytes],eax

; Escribimos el tamaño de la pantalla
; en el archivo
EscribeArchivo [fdSalida],Bytes,4

; Leemos el contenido de la pantalla
LeeArchivo [fdPantalla],Contenido,[Bytes]
; y lo escribimos en el archivo
EscribeArchivo [fdSalida],Contenido,[Bytes]

; Cerramos ambos archivos
CierraArchivo [fdPantalla]
CierraArchivo [fdSalida]

jmp Salir ; terminar
```

```
Error:  
; Mostramos el mensaje de error  
EscribeConsola MsgError,20  
  
Salir:  
; Devolvemos el control al sistema  
INT80____NR_exit, 0,0,0
```

Tras ensamblarlo y enlazarlo, ejecute el programa y compruebe que en el directorio aparece un archivo llamado Pantallas.dat. Éste se crea cada vez que ejecute el programa, perdiéndose el contenido anterior.

Si vuelca su contenido por consola, mediante la orden cat, comprobará que están los caracteres de texto conjuntamente con los atributos.

Manipulación del contenido de fa pantalla

Si para obtener el contenido de la pantalla lo que hacemos es usar los servicios de lectura de archivos, para manipular ese mismo contenido utilizaríamos los servicios de escritura. Sabiendo la línea y columna donde queremos escribir el carácter o atributo, una simple multiplicación nos permitiría saber la posición en que deberíamos situarnos, mediante la macro MuevePuntero, para a continuación efectuar el cambio.

Puesto que, tras ejecutar el programa de ejemplo del punto anterior, tenemos el contenido completo de la pantalla alojado en un archivo, lo más lógico es escribir un programa que tome dicha información y la devuelva a la pantalla. Esto es, precisamente, lo que hace el programa mostrado a continuación.

```
%include "Archivos.inc"  
  
; Punto de entrada a la aplicación  
global main  
  
; Datos con valor inicial  
section .data  
  
,- Nombre del archivo destino  
NombreArchivo db 'Pantallas.dat',0  
  
; Nombre del dispositivo de pantalla  
NombrePantalla db '/dev/vcsa1', 0  
  
; Mensaje de error  
MsgError db 'Se produce un error.', 0  
  
; Datos sin valor inicial  
section .bss  
  
; Para guardar los descriptores  
fdPantalla resd 1  
fdEntrada resd 1  
  
; Bytes que ocupa la pantalla  
Bytes resd 1
```

```

; Pantalla
Contenido resb 16384

; Código del programa
section .text
main:
    ; Abrimos el archivo de origen
    AbreArchivo NombreArchivo, 0_RDONLY

    ; Comprobamos si hay error
    or eax, eax
    js Error

    ;,- guardamos el descriptor de archivo
    mov [fdEntrada], eax

    ; Abrimos la memoria de consola
    AhreArchivo NombrePantalla, 0_RDWR
    or eax, eax
    ja Error

    ; Guardamos el descriptor
    mov [fdPantalla], eax

    ; Leemos el número de bytes que ocupa la
    ; pantalla
    LeeArchivo [fdEntrada], Bytes, 4

    ; Saltamos en la pantalla las posiciones que
    ; indican tamaño y posición del cursor
    LooArchivo [fdPantalla], ConI:enido, 4

    ; Leemos los datos del archivo
    LeeArchivo [fdEntrada], Contenido, [Bytes]

    ; y lo escribimos en la pantalla
    EscribeArchivo [fdPantalla], Contenido, [Bytes]

    ; Cerramos ambos archivos
    CierraArchivo [fdPantalla]
    CierraArchivo [fdEntrada]

    jmp Salir ; terminar

Error:
    ; MosLiamos el mensaje de error
    EscribeConsola MsgError, ?0

Salir:
    ; Devolvemos el control al sistema
    INT80____NR_exit, 0,0,0

```

Cada vez que ejecute este programa, verá aparecer en pantalla el contenido que tenía la consola en el momento en que invocó al programa GuardaPantalla. Puede modificar ambos programas para que el primero guarde también la posición en la que estaba el

cursor y el segundo la restaure, consiguiendo así que el cursor no aparezca, en ocasiones, por encima del texto que hay en pantalla.

Estos programas funcionarán únicamente en consolas de texto reales, no en emulación de terminal bajo X. Asimismo, es posible que necesite permisos de superusuario (root) para acceder a los dispositivos de pantalla, especialmente para escribir.

Acceso a discos

Además de los archivos propiamente dichos, ya sabe que los discos mantienen otros datos, como los directorios, tablas de asignación de archivos, nodos y supernodos, etc. Linux reconoce multitud de sistemas de archivos y, como puede suponer, cada uno de ellos cuenta con su estructura y características específicas.

Al igual que la pantalla correspondiente a cada consola, las distintas unidades de disco, disquetes y CD/DVD aparecen a ojos del sistema como archivos. Las unidades de disquetes son /dev/fd0 y /dev/fd1, asumiendo que existan dos; los discos duros aparecen como hdXX o sdXX, dependiendo de que sean IDE o SCSI. Cada disco se identifica mediante una letra, por ejemplo hda para el primer disco duro, mientras que cada partición se asocia con un número, por ejemplo hdal para la primera partición del primer disco.

Nada nos impide abrir cualquiera de dichos dispositivos y tratarlo como un flujo de bytes, leyendo o escribiendo en él. Lógicamente, deberemos saber lo que estamos haciendo sino queremos perder información e, incluso, dañar irreversiblemente el sistema de archivos. No necesitamos más funciones que las conocidas en los puntos previos.

Usando sólo las funciones de apertura de archivos, lectura, escritura y cierre podemos crear un programa que nos permite copiar disquetes.

El código sería el siguiente:

```
"#include <Archivos.inc>

; Punto de entrada a la aplicación
global main

; Datos con valor inicial
section .data

; Nombre del dispositivo
NombrieDisco db '/dev/fd0',0

; Mensaje de error
MsgError db 'Se produce un error.',0

; Mensajes indicativos
```

```
MsgDiscoOrigen db 'Inserte disco de origen:',0
MsgDiscoDestino db 'Inserte disco de destino:',

MsgFin db 'El proceso de copia ha terminado'
        db 10,13,0

; Datos sin valor inicial
section .bss

; Para guardar el descriptor
; de archivo
fdDisco    resd 1

,- Espacio para contener el disco
"sdefine Bytes 80*2*18*512
Contenido resb Bytes

Espera resb 10 , - Para esperar Intro

,* Código del programa
section .text
main:
; Solícitamos el disco de origen
cali SolicitaOrigen

; Copiamos el contenido en archivo
cali LeeDisco

; Solicitamos el disco de destino
cali SolicitudDestino

; Copiamos en el disco
cali EscribeDisk 0

; Indicamos que ha finalizado el proceso
cali IndicaFin

; Devolvemos el control al sistema
INT80____NR_exit, 0,0,0

SolicitaOrigen:
; Mostramos el mensaje
EscribeConsola MsgDiscoOrigen,24
; y esperamos la pulsación de <Intro>
LeeConsola Espera,2

ret ; volver

LeeDisco:
; Abrimos el disco
AbreArchivo NombreDisco,0_RDONLY
; comprobando un posible error
or eax, eax
js Error
; guardamos el identificador
mov [fdDisco],eax
```

```

    ;• Leemos el contenido del disco
LeeArchivo [fdDisco],Contenido,Bytes

    ; Y lo cerramos
CierraArchivo [fdDisco]

    ret ; volver

SolicitaDestino:
    ; Mostramos el mensaje
EscribeConsola MsgDiscoDestino,25
    ; y esperamos la pulsación de <Tntrn>
LeeConsola Espera,2

    ret ; volver

EscribeDisco:
    ; Abrimos el disco para escritura
AbreArchivo NomhreDiseo, 0_WRONLY,Y
    ; comprobando un posible error
or cax, eax
js Error

    ; Guardamos el identificador
mov [fdDisco], eax
    ; escribimos los datos en el disco
EscribeArchivo [fdDisco],Contenido,Bytes

    ; Y lo cerramos
CierraArchivo [fdDisco]

    ret ; volver

IndicaFin:
    ; Mostramos el mensaje
EscribeConsola MsqFin, 34

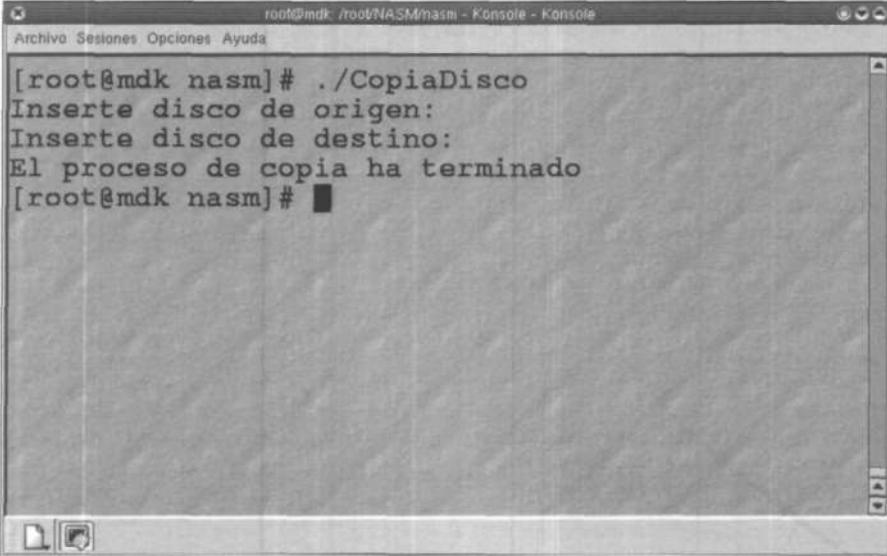
    ret ; volver

Error:
    ; Mostramos el mensaje de error
F.soríbeConsolaMsqrror, 20

    ; Devolvemos el control al sistema
INT80____NR_exit, 0,0,0

```

Observe que no se utiliza ningún archivo intermedio, sino que se lee el disco entero en memoria usando un campo definido a tal efecto. Esto es algo que, por ejemplo, no podría hacer nunca en DOS, puesto que no puede reservar esa cantidad de memoria y usarla directamente con un servicio del DOS o la BIOS. Puesto que el programa lee el disco completo en una sola instrucción, mientras dura ese proceso no hay ninguna notificación o respuesta. Puede modificar el programa para que vayan leyéndose pistas individuales, aún cuando siga almacenándolas directamente en memoria, y ofrecer un retorno por consola indicando el curso del proceso.



The screenshot shows a terminal window titled "root@mdk: /root/NASM/nasm - Konsole - Konsole". The window has a menu bar with "Archivo", "Sesiones", "Opciones", and "Ayuda". The main area displays the following text:

```
[root@mdk nasm]# ./CopiaDisco
Inserte disco de origen:
Inserte disco de destino:
El proceso de copia ha terminado
[root@mdk nasm]# █
```

Figura 29.5. El programa copia un disco completo.

La biblioteca de funciones de Linux

Linux, como se decía al inicio del capítulo, es un sistema operativo en el que se desarrolla principalmente en C y, por ello, una gran parte de las funciones del sistema, más allá de las que son accesibles mediante la interrupción 8 Oh, forman parte de lo que se conoce como la biblioteca estándar de funciones de Linux.

Programar en ensamblador empleando la biblioteca de funciones de Linux es, básicamente, como programar en lenguaje C. La mayor parte del programa se dedicará a poner parámetros en la pila y llamar a esas funciones, moviendo datos el resto del tiempo entre la pila y los campos de datos propios, poco más. Lo único que precisamos, como en el caso de Windows, es la información de referencia de esas funciones, para saber en qué módulos se encuentran, cuál es su nombre o su lista de parámetros.

Una de las diferencias que encontramos a la hora de usar funciones externas, alojadas en la biblioteca estándar de Linux (conocida como `libc` o `glibc` según versiones) o bibliotecas externas, respecto a la interrupción 8 Oh es que los parámetros no se entregan en registros, sino a través de la pila. Además, el orden en que deben insertarse es el inverso al definido en el prototipo de la función, como ocurría en Windows.

Otro aspecto a tener en cuenta es que los responsables de extraer dichos parámetros de la pila somos nosotros, el código de nuestro programa, y no el sistema. Esto es una

diferencia respecto a Windows debida a que Linux usa la convención de llamada de C, mientras que Windows emplea, como se indicó en su momento, la convención *stdcall*.

La extracción de los datos de la pila, tras la llamada, puede efectuarse mediante instrucciones pop, lo cual implica en ocasiones múltiples instrucciones y la pérdida del valor de al menos un registro, o bien modificando directamente el puntero de pila. Sabiendo que la pila crece desde las direcciones más altas hacia el inicio, bastará con sumar a esp el número de bytes que quieren eliminarse para ajustar apropiadamente la pila.

Servicios disponibles

Con los servicios de la biblioteca estándar puede efectuar cualquier operación de entrada o salida hacia archivos y dispositivos, gestionar la memoria del sistema asignando o liberando bloques, obtener información sobre fecha y hora, comunicarse mediante el uso de *sockets* TCP/IP, efectuar operaciones aritméticas, de manipulación de cadenas, búsqueda de patrones, etc.

Gracias a estos servicios podemos, por ejemplo, dar formato a una serie de datos generando una cadena de caracteres, sin tener que preocuparnos por la conversión de binario a ASCII mediante la codificación de rutinas como EnteroCadena (creada en un capítulo previo). El programa siguiente utiliza la función printf para mostrar dos parámetros que se facilitan en la pila. En realidad los dos parámetros son idénticos, el número entero 115, pero, como se aprecia en la figura 29.6, la función printf le da formato como cadena ASCII o como carácter, según se haya indicado en la cadena de formato.

```
global main

; Funciones que vamos a usar
extern printf
extern exit

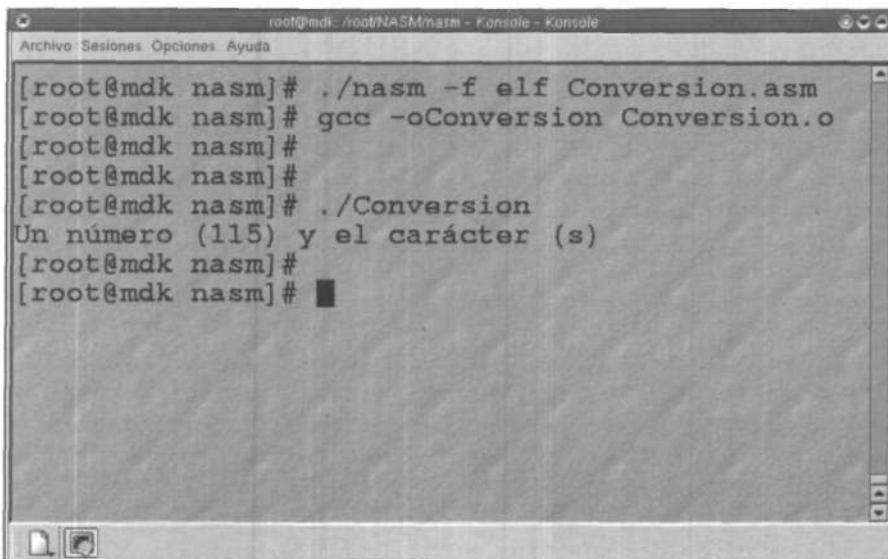
section .data

; Cadena de caracteres
Formato db "Un número %d y el carácter (%c)",0ah,0

section .text
main:
    ; introducimos tsn la pila
    ; dos enteros
    push 11b
    push 115
    ; y la dirprinn de la cadena
    push dword Formato
    ; para llamar a printf
    ; cali printi
    cali printi

    ; eliminamos los parámetros
    add esp,12

    push 0 ; Salimos con el código 0
    cali exit '
```

A screenshot of a Linux terminal window titled "root@mdk: /root/NASM/nasm - Konsole - Konsole". The window has a menu bar with "Archivo", "Sesiones", "Opciones", and "Ayuda". The terminal content shows the following command sequence and output:

```
[root@mdk nasm]# ./nasm -f elf Conversion.asm
[root@mdk nasm]# gcc -oConversion Conversion.o
[root@mdk nasm]#
[root@mdk nasm]#
[root@mdk nasm]# ./Conversion
Un número (115) y el carácter (s)
[root@mdk nasm]#
[root@mdk nasm]#
```

Figura 29.6. Resultado que produce la función printf.

Recorriendo a la documentación de la biblioteca de funciones, que encontrará completa en el apartado Software Libraries del sitio GNU Manuals Online (<http://www.gnu.org/manual>), podrá invocar a cualquier otra función. Esa documentación está enfocada a los usuarios de lenguaje C, pero recuerde que lo único que debe hacer es incluir una declaración extern seguida del nombre de la función, facilitar los parámetros en orden inverso en la pila y recoger cualquiera resultado que devuelva la función en el registro eax. También le serán de utilidad los archivos de cabecera mencionados en la documentación, ya que puede localizar en su propio sistema y recuperar definiciones de constantes y estructuras.

Resumen

Al igual que Windows, Linux es un sistema operativo de 32 bits en el que, por tanto, tenemos acceso a los registros de 32 bits y todas las instrucciones mejoradas que aparecieron a partir del 80386. También trabajamos en un modelo de memoria plano, sin tener que preocuparnos de registros de segmento ni limitaciones a la hora de reservar un área de memoria. Como sistema que trabaja en modo protegido, no tenemos acceso directo al hardware, a elementos como la memoria, los puertos de entrada y salida o los servicios que ofrece la BIOS. Tenemos, a cambio, los servicios del núcleo de Linux, accesibles mediante la interrupción 8 Oh, y toda la biblioteca de funciones estándar. De igual forma podemos acceder a funciones alojadas en otras bibliotecas sin más que enlazarlas con nuestro programa.

30

32 bits en DOS

En los dos capítulos anteriores ha creado distintas aplicaciones para dos sistemas operativos, Windows y Linux, que se caracterizan por facilitar el uso de los registros e instrucciones de 32 bits con que cuentan los procesadores de Intel y compatibles, haciendo posible de esta forma el aprovechamiento de posibilidades como el acceso a una cantidad de memoria mucho mayor que la que puede direccionarse en cualquier programa DOS.

Incluso existe la posibilidad, en las versiones más actualizadas de dichos sistemas, de operar en el modo de 64 bits, aunque éste por el momento no es el más usual.

Para los programadores que usan el lenguaje ensamblador, sin embargo, la creación de aplicaciones para DOS sigue teniendo mucho atractivo, ya que es un sistema operativo que, en cierta forma, les permite controlar totalmente el sistema. Una aplicación completa creada en ensamblador, conjuntamente con el sistema operativo, pueden incluirse en un disquete y sobrar aún espacio para incluir gráficos y otros datos, pudiendo utilizarse prácticamente en cualquier equipo por antiguo que sea.

Las limitaciones que tiene este sistema, principalmente la capacidad para acceder a toda la memoria y los mecanismos de protección, pueden superarse activando el modo protegido cuando sea necesario, ya sea mediante código propio o a través de los servicios DPMI o un extensor de DOS.

El objetivo de este capítulo es servirle como introducción a los fundamentos del modo protegido, mostrándole cómo detectar en qué modo está trabajando, como pasar de uno a otro y efectuar algunas operaciones básicas.

También conocerá el uso de un anfitrión DPMI, que se encargará de automatizar el mecanismo de activación del modo protegido.

El modo protegido

Desde la aparición de los primeros PC, a principios de la década de los ochenta, son millones de aplicaciones las que se han desarrollado para DOS, el sistema operativo que incorporaban en principio los ordenadores de IBM y todos los compatibles aparecidos con posterioridad.

Ese volumen de software, como es lógico, no podía cambiarse de la noche a la mañana por la presentación, por parte de Intel, de un nuevo procesador con más posibilidades: el 80286. Era obligatorio mantener la compatibilidad hacia atrás, haciendo que ese nuevo procesador se comportase, en principio, como si fuese un 8086.

Ya que no podía conocerse de antemano si el sistema operativo que iba a instalarse en el ordenador equipado con un 80286 reconocería o no este nuevo procesador, la única solución era que, al conectarse, el 80286 funcionara exactamente igual que un 8086. A ese modo de funcionamiento se le conoció a partir de entonces como *modo real*, entendiendo por *real* el del primer microprocesador de la familia x86.

¿Qué sentido tendría crear un nuevo microprocesador con más capacidad, por ejemplo con un bus de direcciones de 24 bits en lugar de 20, si no había forma de aprovecharlo? Trabajando en modo real no era posible acceder a esa capacidad, así que era necesario habilitar un nuevo modo de funcionamiento, modo al que se denominó *protegido* por el hecho de que contaba con mecanismos de protección de la memoria. Cualquier programa, ya sea una aplicación o un sistema operativo, que quisiera trabajar en dicho modo debía activarlo explícitamente.

El 80286 contaba con un mecanismo documentado para pasar del modo real, el existente al poner en marcha el ordenador, al modo protegido, pero no para dar el paso inverso, de tal manera que, una vez se había entrado en modo protegido, difícilmente podía volverse al modo real. No obstante, empresas como IBM ingenieraron diversos trucos para hacerlo posible dentro de las limitaciones existentes.

La posterior aparición del procesador 80386, en 1985, se encontró con el mismo problema ya existente: la práctica totalidad de las aplicaciones funcionaban en DOS en modo real. Por eso dicho procesador, al igual que los 80486 y todos los Pentium, Core y sucesores, funcionan en modo real cuando se ponen en marcha. Hasta el microprocesador de Intel más reciente en el momento de escribir estas líneas, el Core i7 con múltiples núcleos e *HyperThreading*, se comporta como un 8086 cuando conectamos nuestro ordenador, aunque eso sí, ejecutando instrucciones a velocidades de aproximadamente 3 Ghz, en lugar de hacerlo a 4,77 Mhz como el 8086 original, y con la capacidad de procesar múltiples instrucciones paralelamente.

Tendremos, por lo tanto, un sistema mucho más rápido pero con el mismo límite de los 640 kilobytes.

El 80386 mejoró la operación en modo protegido, haciendo posible la vuelta al modo real si era necesario, y añadió un nuevo modo de trabajo conocido como V86. En dicho modo, realmente una extensión del modo protegido, el procesador puede emular varios 8086 funcionando de manera concurrente. Es el modo que hace posible, por ejemplo, abrir desde Windows una ventana de consola y tener aplicaciones DOS funcionando en ella. Realmente esa consola no está operando en modo real sino en modo virtual o V86.

Las explicaciones de los puntos siguientes se centran en el mecanismo de gestión del modo real de los 80386 y posteriores, obviando la herencia de los 80286 ya que, actualmente, es difícil encontrar equipos con dicho procesador.

Registros de control del procesador

Los microprocesadores 80386, y por tanto posteriores, cuentan, además de con los registros de segmento y propósito general que ya conocemos, con un importante número adicional de registros, entre los que se encuentran los registros del control del procesador. Se trata de cuatro registros, de 32 bits cada uno, denominados CRO, CR1, CR2 y CR3. Nos interesa especialmente el primero, el registro CRO, ya que varios de sus bits controlan si el procesador opera en modo real o protegido, si está habilitada la unidad de paginación de memoria del procesador, si está cambiándose de tarea, etc.

El bit 0 del registro CRO, conocido como bit PE, nos indica si está operándose en modo real, caso en el que estará a 0, o en modo protegido, estando entonces a 1. Teniendo en cuenta que podemos acceder al contenido de los registros de control del procesador como lo haríamos con cualquier otro tipo de registro, con instrucciones mov, nada nos impide hacer lo siguiente para saber si estamos o no en modo protegido:

```
segment Pila stack
    resw 512
FinPila:

segment Datos
; Mensajes informativos
MsgReal db 'Está en modo real.$'
MsgProtegido db 'Está en modo protegido.$'

; Segmento de código
*****
segment Código
..start:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila

    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax

    ; Asumir que estamos en modo real
    mov dx, MsgReal

    ; Tomar el contenido de CRO
    mov eax, cr0
```

```

; y comprobar el estado del bit 0
test al, 1
; Si está a 0 estamos en modo real
jz Salir

; En caso contrario en modo protegido
mov dx, MsqProtegido

Salir:
; Mostramos el mensaje
mov ah, 9
int 21h

; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

```

Si ejecuta este programa en un sistema que esté ejecutando DOS, sin otro sistema y sin estar usando ciertos gestores de memoria como EMM386, el resultado sería el de la figura 30.1. El procesador está funcionando en modo real. Tanto los citados gestores de memoria como Windows ponen el procesador en modo virtual V86, de tal forma que la consola en la que operamos es, aparentemente, un sistema 8086 trabajando en modo real. El resultado que genera este programa, sin embargo, es distinto como se aprecia en la figura 30.2. El microprocesador está en modo protegido, aunque no nos lo parezca.

En caso de que el procesador esté en modo virtual V86, lo que ocurre en las consolas de Windows o tras iniciar controladores de memoria como EMM386, el procedimiento que va a explicarse en los puntos siguientes para entrar en modo protegido no será válido ya que, de hecho, el procesador ya está en modo protegido, aunque emulando un 8086. En estos casos habrá que usar los servicios DPMI que ofrecen estos sistemas y que, aunque básicamente, conocerá más adelante.

```

C:\>keyb sp
Keyboard layout sp loaded for codepage 850
C:\>
C:\>
C:\>modo
Está en modo real.
C:\>_

```

Figura 30.1. En un sistema DOS el procesador trabaja en modo real.

D:\Ejemplos\30>nasm -f obj modo.asm
D:\Ejemplos\30>alink modo
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved
Loading file modo.obj
matched Externs
matched ComDefs
D:\Ejemplos\30>
D:\Ejemplos\30>modo
Está en modo protegido.
D:\Ejemplos\30>
D:\Ejemplos\30>

Figura 30.2. En una consola DOS de Windows el procesador se encuentra en modo protegido.

Modificación de los registros de control

Los registros de control del procesador, y más concretamente el registro CRO, no son registros sólo de lectura, sino que un programa puede tomar su contenido, modificarlo y devolverlo a dicho registro. Esto significa, por ejemplo, que podemos cambiar el mencionado bit PE para activar el modo protegido, o desactivarlo para activar el modo real. ¿Puede ser tan fácil pasar a modo protegido? Compruébelo por sí mismo con el programa siguiente:

```
segment Pila staok
    resw 512
FlnPila:

segment Datos
,- Mensajes informativos
MsgProtegido db 'Estoy en modo protegido.$'

; Segmento de código
*A ****
    segment Código
..start:
Inicio:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp', FinPila
```

```

; y del segmento de datos
mov ax, Datos
mov ds, ax

; Mostramos el mensaje antes
; de entrar en modo protegido
mov dx, MsgProtegido

; Mostramos el mensaje
mov ah, 9
int 21h

; Tomar el contenido de CRO
mov eax, cr0
; cambiar el estado del bit 0
or eax, 1
; y ponerlo de nuevo en CRO
mov cr0, eax
; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

```

Asumiendo siempre que trabaja con un sistema DOS, no desde una ventana de consola de Windows, al ejecutar el programa anterior verá que aparece el mensaje y que, a continuación, en efecto el sistema pasa al modo protegido, quedándose totalmente bloqueado en ese punto. El programa funciona, porque el procesador está en modo protegido, lo que ocurre es que a partir de ese punto el procesador se ha encontrado con una configuración de registros y memoria inadecuada y el sistema se para.

Dependiendo del sistema en particular donde ejecute el programa, existe la posibilidad de que el ordenador se reinicie y, por lo tanto, devuelva el procesador al modo real de inmediato.

Segmentos y selectores

Para poder activar adecuadamente el modo protegido necesitaremos, aparte de modificar un bit del registro CRO, comprender muchos otros aspectos, entre ellos el nuevo papel que juegan los registros de segmento en ese modo, muy diferente al que tienen en modo real. Como ya sabe, en modo real los registros de segmento hacen referencia a direcciones físicas de memoria, combinándose con un desplazamiento para generar una dirección completa. El contenido de los registros de segmento se expresa en párrafos, de tal manera que cada 16 bytes comienza un nuevo segmento, cuya longitud máxima será siempre de 64 kilobytes. Este esquema permite componer direcciones de 20 bits y acceder a cualquier posición dentro del megabyte con cierta facilidad ya que, tomada

cualquier dirección lineal, basta con dividirla por 16 y asignar el cociente a un registro de segmento y el resto como desplazamiento para acceder a esa dirección.

Al pasar al modo protegido, el significado de los valores que contienen los registros de segmento es totalmente distinto. Cada uno de los seis registros de segmento, es, ds, es, fs, gs y ss, contiene lo que se denomina un *selector*, un índice que hace referencia o selecciona un elemento de una tabla. No se trata, por tanto, de una dirección física de memoria. Hay que tener en cuenta que los registros de segmento siguen siendo de 16 bits, mientras que el bus de direcciones de los 80386 es de 32 bits, por lo que la forma de direccionar la memoria debe, necesariamente, ser distinta.

El modo protegido no sólo facilita el acceso a una mayor cantidad de memoria sino que, además, ofrece mecanismos de protección e intercambio de tareas. La protección impide, por ejemplo, que pueda escribirse en un segmento dedicado a código o que una tarea de usuario pueda escribir en zonas de memoria de una tarea de sistema. Por ello la descripción de cada uno de los segmentos en que puede dividirse la memoria es algo más compleja que una simple dirección física expresada en párrafos.

Cada uno de los registros de segmento, por tanto, actúa ahora como selector de un cierto segmento. Los descriptores de segmentos se alojan en una tabla que es preciso definir antes de entrar en el modo protegido.

Descriptores de segmentos

Para comprender cómo se accede a la memoria en modo protegido es indispensable saber qué es un descriptor de segmento, qué son las tablas de descriptores y cuál es el formato de un descriptor. Toda esta información puede encontrarla en la documentación de referencia que ofrece Intel sobre sus procesadores con mucho mayor detalle del que va a describirse a continuación, donde se abordan sólo los fundamentos básicos.

La memoria a la que accede un 80386, o posterior, puede dividirse en tantos segmentos como se desee, teniendo cada uno de ellos cualquier longitud y comenzando en cualquier dirección lógica perteneciente al espacio de direccionamiento. En realidad existen una serie de límites, pero son tan extremos que difícilmente los alcanzaremos programando en ensamblador. El número máximo de segmentos simultáneos es de 16384, teniendo cada uno un máximo de 4 gigabytes. Una simple multiplicación da como resultado una capacidad virtual de direccionamiento de hasta 64 terabytes, si bien la capacidad de direccionamiento físico viene limitado por el bus de direcciones de 32 bits: 4 gigabytes en total.

En los microprocesadores actuales, como Pentium y Core, existe un bit de extensión de direccionamiento que permite generar direcciones de hasta 36 bits en modo protegido, ampliando así el límite físico de 4 gigabytes. También existe la posibilidad de utilizar el nuevo modo *Long* para operar con 64 bits, pero es un aspecto que no va a tratarse en este capítulo.

Cada uno de los segmentos se describe con los datos siguientes:

- **Dirección base:** Una dirección de 32 bits que indica la posición en la que comienza el segmento.
 - **Longitud:** Un campo de 20 bits que indica la longitud del segmento.
 - **Atributos:** Un conjunto de bits que definen diversos atributos del segmento.

La longitud del segmento, a la cual se dedican 20 bits, puede venir expresada en bytes o en páginas de 4 kilobytes, dependiendo del estado de uno de los bits en el campo de atributos. Si se usa una longitud en bytes, el segmento tendrá como máximo 2^{20} bytes, es decir, 1 megabyte, mientras que si se usan páginas la longitud sería de 2^{20} páginas, por 4 kilobytes cada una sería 4 gigabytes.

Nota

A partir del Pentium las páginas también pueden ser de 4 megabytes, ampliando aún más el tamaño que puede tener cada segmento.

En la figura 30.3 puede ver la estructura física que tiene un descriptor de segmento. Observe que ocupa 4 palabras, es decir, 8 bytes completos. Fíjese también en que la dirección base del segmento, así como su longitud, no aparecen como datos completos sino que están partidos en varios bytes. Esto hace algo más compleja su composición, como verá más adelante.

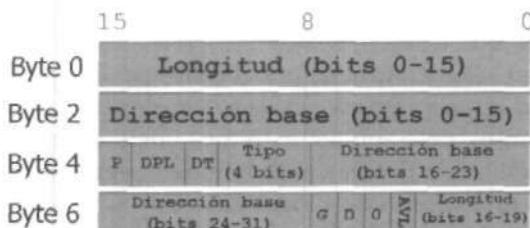


Figura 30.3. Estructura de un descriptor de segmento.

El significado de los distintos bits de atributo que se encuentran repartidos en los bytes 4 y 7 del descriptor es el siguiente:

- P: *Presence*. Indica si un segmento está o no disponible en memoria. Es útil para la implementación de gestores de memoria virtual capaces de intercambiar páginas a disco cuando es necesario, ofreciendo más memoria de la que existe físicamente. Es lo que hacen, por ejemplo, tanto Windows como Linux.
 - DPL: *Descriptor Privilege Level*. Son dos bits que establecen el nivel de privilegio asociado al segmento. Éste, como podrá suponer, será un valor entre 0 y 3, ya que

éos son los valores representables con dos bits. El valor 0 indica el mayor nivel de privilegio y el valor 3 el menor. Los segmentos de código que tienen un privilegio X no pueden acceder a segmentos que tienen un privilegio mayor, es decir, X-1, X-2 o X-3, según los casos. De intentarlo, se produciría una excepción.

- DT: *Descriptor Type*. Este bit indica si el descriptor de segmento pertenece al sistema, caso en el que estará a 0, o bien un segmento de aplicaciones, estando entonces a 1.
- Tipo: Son cuatro bits que determinan el tipo del segmento. Se explica con mayor detalle a continuación.
- G: *Granularity*. Si este bit está a 0 los 20 bits de longitud se interpretarán como un número de bytes, mientras que si está a 1 se interpretarán como un número de páginas de 4 kilobytes. Afecta, por tanto, a la longitud del segmento.
- D: *Default*. Normalmente estará siempre a 1, a menos que el segmento contenga código compatible con 80286 caso en el que estaría a 0. En realidad afecta al tamaño por defecto de ciertas instrucciones.

Eos demás bits o están reservados o no tienen aplicación, al menos en los procesadores 80386.

Tipos de segmentos

Los cuatro bits de menor peso del cuarto bit del descriptor, señalados como Tipo en la figura 30.3, determinan el tipo del segmento que está describiéndose. En realidad estos cuatro bits se dividen, a su vez, en tres grupos. El significado individual de cada bit es el que se muestra en la figura 30.4. El bit de más peso, o bit 3, es el señalado como *Código/Datos*.

Código/ Datos	Ajustable/ Sentido	Lectura/ Escritura	Accedido
------------------	-----------------------	-----------------------	----------

Figura 30.4. Bits que afectan al tipo de segmento.

El bit de menor peso, el bit 0 del byte 4, indica si el procesador ha accedido al segmento, caso en el que se pondrá a 1. Es útil para determinar qué segmentos están siendo menos usados y, por tanto, pueden descargarse de la memoria y llevarse a disco para dar cabida a segmentos de otras aplicaciones.

El bit de mayor peso, el 3, indica si el segmento es de código o de datos. En el primer caso el bit estará a 1 y en el segundo a 0. Tenga en cuenta que por datos se entiende cualquier segmento que no contengan código ejecutable, incluidos aquellos que, por ejemplo, se dediquen a actuar como áreas de pila. Dependiendo de que el segmento sea de código o de datos, los dos bits restantes tendrán un significado u otro.

Si el segmento es de código, el bit 1 indicará si puede leerse de él o no. Todos los segmentos de código pueden ejecutarse y, si este bit está a 1, además pueden leerse. Nunca

puede escribirse en un segmento de este tipo y, de intentarlo, se generaría una excepción. En cuanto al bit 2, determina si el privilegio del segmento se ajusta automáticamente al privilegio del segmento que le haya invocado o no.

En caso de que el segmento sea de datos, el bit de mayor peso a 0 lo revelará, el bit 1 indicará si además de leerse puede escribirse en dicho segmento. Con el bit a 0 el segmento de datos sólo puede leerse, mientras que a 1 también puede escribirse en él. En cuanto al bit 2, determinará el sentido del crecimiento del segmento. Si está a 0 el segmento crece en sentido normal, incrementando direcciones, mientras que si está a 1 lo hace en sentido inverso, como ocurre con la pila.

Obviamente el bit de menor peso, que indica si se ha accedido o no al segmento, las combinaciones de los otros tres darían como resultado ocho tipos distintos de segmento que son los enumerados en la tabla 30.1.

Tabla 30.1. Tipos de segmento posibles.

Combinación de bits	Tipo del segmento
000	De datos y sólo lectura, con expansión positiva.
001	De datos y lectura/escritura, con expansión positiva.
010	De datos y sólo lectura, con expansión negativa.
011	De datos y lectura/escritura, con expansión negativa.
100	De código, sólo ejecutable y sin ajuste de privilegio.
101	De código, ejecutable y de lectura y sin ajuste de privilegio.
110	De código, sólo ejecutable y con ajuste de privilegio.
111	De código, ejecutable y de lectura y con ajuste de privilegio.

Recuerde que el tipo de segmento puede provocar que ciertas operaciones provoquen una excepción. Si, por ejemplo, definimos un segmento de datos en el que no se permite la escritura e intentamos modificar uno sólo de sus bytes, obtendremos esa excepción.

Tablas de descriptores

Los descriptores no se asignan directamente a registro alguno, al menos de manera explícita, sino que se utilizan cuando se asigna un índice a un registro de segmento. Ese índice hace referencia a uno de los descriptores existentes en una tabla. En el momento en que se efectúa la asignación el procesador busca el descriptor en la tabla que corresponda y asigna su contenido a unos registros ocultos, de acceso muy rápido, que serán los usados a partir de ese momento cada vez que se intente acceder al segmento.

En el sistema pueden existir dos tipos diferentes de tablas de descriptores: una global y varias locales. La tabla global de descriptores, conocida como GDT (*Global Descriptor Table*), es la que utiliza normalmente el sistema operativo para sus propios segmentos,

mientras que las tablas locales o LDT (*Local Descriptor Table*) se crean asociadas a tareas. La capacidad que tiene el procesador para el cambio de contexto entre tareas, cambiando de la LDT de la tarea que se abandona por la de la que se inicia, es la que hace posible que cada tarea cuente con un espacio de direcciones totalmente independiente.

Cada tabla de descriptores, tanto la GDT como las LDT que pudieran existir, puede contener como máximo 8192 descriptores que, a razón de 8 bits por descriptor, ocuparían 64 kilobytes cada una. En la práctica no es habitual usar más de una decena de descriptores, especialmente al escribir aplicaciones directamente en ensamblador.

La GDT se aloja en la memoria del sistema y su posición se facilita en un registro llamado gdtr. El valor de dicho registro se establece mediante la instrucción lgdt. Teniendo en cuenta que la GDT debe estar definida y alojada en memoria antes de cambiar al modo protegido, es fácil suponer que esta instrucción se ejecutará cuando aún nos encontramos en modo real. El operando a facilitar será la dirección en memoria de una estructura compuesta de una palabra, indicando la longitud de la GDT, y una doble palabra indicando la dirección lineal en la que se encuentra.

IMOio

El primer descriptor de la GDT siempre tiene todos sus campos a 0, siendo utilizado por el microprocesador para interceptar la situación inválida que se da cuando un registro de segmento o selector tiene el valor 0.

En cuanto a las LDT, como se ha dicho antes pueden existir varias en caso de que el procesador esté ejecutando múltiples tareas, la dirección de la que corresponde a la tarea actual se encontrará siempre en el registro ldtr. Éste no contiene una dirección de 32 bits y una longitud, sino un índice que hace referencia a un elemento de la GDT. Deberá existir, por tanto, un descriptor de segmento en la GDT por cada LDT o tarea activa.

De vuelta a los selectores de segmento

Ahora que conocemos la estructura de un descriptor de segmento, las tablas de descriptores que usa el microprocesador y los registros que las mantienen, es el momento de volver a analizar el contenido de los registros selectores de segmento, es decir, de los registros es, ds, es, fs, gs y ss.

Anteriormente decíamos que estos registros, de 16 bits, contenían un selector, un índice que hace referencia a una tabla. Su estructura, sin embargo, es algo más compleja y se divide en los elementos representados en la figura 30.5.

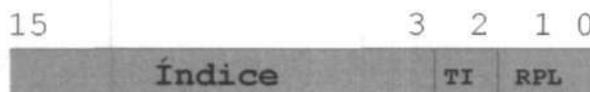


Figura 30.5. Estructura de un selector de segmento.

Los dos bits de menor peso, conocidos como RPL (*Requested Privilege Level*), establecen el nivel de privilegio del segmento donde se encuentra el código que ha provocado el acceso al segmento al que corresponde este selector. Parece un juego de palabras, pero tan sólo tiene que imaginar el proceso: el código de un determinado segmento, por ejemplo perteneciente al sistema, solicita la ejecución del código que está en otro segmento, perteneciente a una aplicación. El selector almacenaría el nivel de privilegio del primer segmento, a fin de compararlo con el del segmento al que va a accederse y ver si es posible. Es algo que forma parte del mecanismo de protección de memoria de estos microprocesadores.

Los bits 3 a 15, un total de 13 bits, componen el índice del descriptor correspondiente al segmento al que quiere accederse. Con 13 bits pueden componerse 8192 valores distintos que, como ya sabe, es el número máximo de descriptores que puede contener la GDT o una LDT.

Por último, el bit 2 es el que determina si el índice del selector está haciendo referencia a la GDT o a la LDT actual. En el primer caso estaría a 0 y en el segundo a 1. De esta forma es posible acceder a todos los descriptores existentes en ambas tablas desde cualquier tarea.

Direccionamiento en modo protegido

Relacionando toda la información de los puntos previos, veamos cómo se accede a una dirección de memoria en modo protegido en comparación con el modo real. Suponga que tiene las instrucciones siguientes:

```
mov ax, 40h
mov ds, ax
mov bx, 1Ah
mov si,[bx]
```

Operando en modo real, la instrucción `mov si, [bx]` estaría accediendo a la posición de memoria 1050, o 41Ah si lo expresamos en hexadecimal. El proceso para llegar a esa dirección, aunque ya lo conoce, es el representado en la figura 30.6. El valor de `ds` se desplaza cuatro bits a la izquierda, lo cual equivale a multiplicar por 16, y después se le suma el valor de `bx`, obteniendo la dirección física de 20 bits.

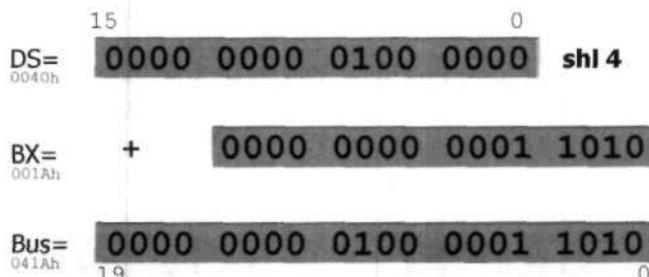


Figura 30.6. Traducción del valor de un registro de segmento en modo real.

Al trabajar en modo protegido el valor de ds se interpreta de manera totalmente distinta. Sus dos primeros bits indican que el nivel de privilegio del solicitante es 0, y el bit 2 que el índice corresponde a la GDT. Los demás bits, los que actúan como índice, tienen el valor 1000, lo cual corresponde a seleccionar el noveno descriptor de la GDT. Tomando la dirección base de dicho descriptor, siempre que los mecanismos de protección verifiquen que ello es posible, se aplicará el desplazamiento sumándolo directamente. Puede producirse un fallo, por ejemplo, si el desplazamiento resulta en una dirección más allá del límite del segmento. Suponiendo que el estado de la GDT fuese el representado en la figura 30.7, con el descriptor seleccionado por ds teniendo el valor 0 como dirección base y el valor OFFFFFh como longitud, la dirección física a la que se accedería sería la 26, o 1Ah en hexadecimal. Además, y asumiendo que el bit G del descriptor está a 1, la longitud del segmento sería de 4 gigabytes, por lo que podríamos acceder a cualquier posición de memoria utilizando registros de 32 bits como ebx, esi o edi.

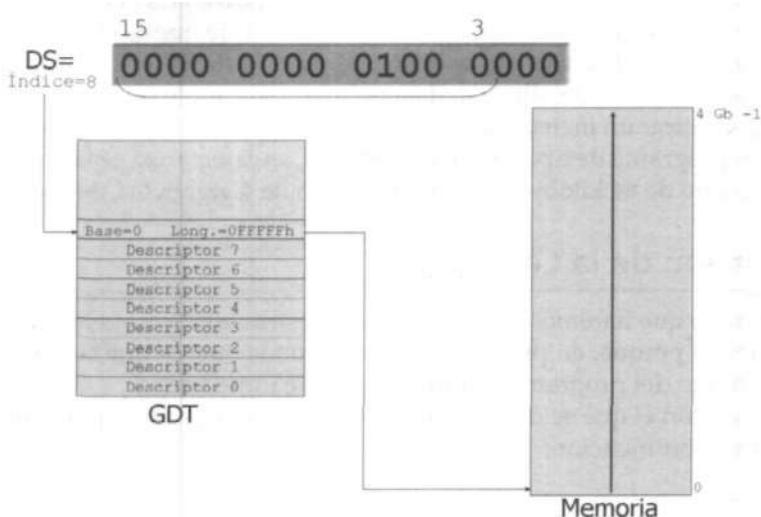


Figura 30.7. Traducción del valor de un registro de segmento en modo protegido.

Entrada y salida del modo protegido

A pesar de toda la información vertida en las páginas previas, el cambio a modo protegido cuenta, además, con algunos trucos o trampas que es necesario tener en cuenta antes de ponerse a crear su programa ya que, de lo contrario, puede usar una gran cantidad de tiempo intentando descubrir por qué el sistema se queda parado continuamente.

Para activar el modo protegido debe ejecutar un programa que, al ponerse en marcha, está en modo real, en un entorno de 16 bits. Todas las instrucciones de preparación de la GDT, incluyendo la asignación al registro gdtr, y modificación del registro cr0 se ejecutan en modo real. Tras activar el indicador de modo protegido el procesador espera encontrar instrucciones de 32 bits, pero en ese momento la cola de ejecución interna del micro, en la que se han ido descodificando instrucciones con cierta antelación, estará

llena de instrucciones del modo real. Es necesario dejar vacía esa cola en cuanto se entra en el modo protegido y, para ello, uno de los mecanismos más inmediatos es provocar un salto a la dirección donde se encuentre la primera instrucción de modo protegido, incluso aunque ésta se encuentra justo a continuación.

Ese salto, mediante una instrucción jmp, debe introducir en el registro es el índice correspondiente al descriptor del segmento que contiene el código de 32 bits que, como es lógico, habremos preparado previamente.

Otro aspecto a tener en cuenta es el de las interrupciones. En el caso de que nuestro programa no prepare una tabla de selectores de interrupciones, y facilite el código para su gestión, deben desactivarse antes de entrar en modo protegido.

Si tras actuar en modo protegido pretendemos volver al modo real, la GDT deberá contener descriptores de los segmentos de código y datos con sus atributos ajustados a las características del modo real, lo que significa que los segmentos deberían tener 64 kilobytes de longitud. El truco del salto, introduciendo en es el índice del descriptor correspondiente al segmento de código de 16 bits, será de nuevo el medio empleado para dejar vacía la cola y llenarla con las instrucciones de 16 bits.

Veamos todos estos detalles escribiendo un programa cuyo objetivo es pasar al modo protegido, mostrar un mensaje en pantalla y volver al modo real. Usaremos NASM para generar un programa de tipo COM que, ya sabe, se caracteriza por ofrecer un modelo de memoria plano de 64 kilobytes, en modo real, o de 4 gigabytes, en modo protegido.

Preparación de la GDT

Lo primero que haremos será componer la estructura o esqueleto de la GDT, y decidimos esqueleto porque, en principio, dejará algunos huecos que será preciso completar desde el código del programa, como veremos de inmediato.

El código con el que se define la GDT, que situaremos al final del módulo, es el que se muestra a continuación:

```
; Componemos la GDT

; El primer descriptor es nulo
SelectorNulo dd 0
        dd 0 ; 8 bytes a cero

; El segundo descriptor será el de
; código en modo protegido
SelectorCodigo32
        dw OFFFFh      ; bits 0-15 longitud
        dw 0           ; bits 0-15 dirección base
        db 0           ; bits 16-23 dirección base
        db 10011010b   ; bits P,DPL,DT y tipo
        db 11001111b   ; bits G,D y bits 16-19 longitud
        db 0           ; bits 24-31 dirección base

; El tercer descriptor será el de
; datos en modo protegido
```

```

SelectorDatos32
    dw OFFFFh      i bits 0-15 longitud
    dw 0           ; bits 0-15 dirección base
    db 0           ; bits 16-23 dirección base
    db 10010010b  ; bits P,DPL,DT y tipo
    db 11001111b  ; bits G,D y bits 16-19 longitud
    db 0           ; bits 24-31 dirección base

; El cuarto descriptor será el de
; código en modo real
SelectorCódigo16
    dw OFFFFh      ; bits 0-15 longitud
    dw 0           ; bits 0-15 dirección base
    db 0           ; bits 16-23 dirección base
    db 10011010b  ; bits P,DPL,DT y tipo
    db 00000000b  ; bits G,D y bits 16-19 longitud
    db 0           ; bits 24-31 dirección base

; El quinto descriptor será el de
; datos en modo real
SeleectorDatos16
    dw OFFFFh      ; bits 0-15 longitud
    dw 0           ; bits 0-15 dirección base
    db 0           ; bits 16-23 dirección base
    db 10010010b  ; bits P,DPL,DT y tipo
    db 00000000b  ; bits G,D y bits 16-19 longitud
    db ü           ; bits 24-31 dirección base

; El sexto y último descriptor nos
; permitirá tratar con 4 Gb planos de memoria
SelectorPlano
    dw OFFFFh      ; bits 0-15 longitud
    dw 0           ; bits 0-15 dirección base
    db 0           ; bits 16-23 dirección base
    db 10010010b  ; bits P,DPL,DT y tipo
    db 11001111b  ; bits G,D y bits 16-19 longitud
    db 0           ; bits 24-31 dirección base

; El campo siguiente tendrá el tamaño
; y dirección física donde está la GDT,
; datos que se introducirán en el registro GDTR

DaLosGDTR
    ; El tamaño lo conocemos en este momento
    TamanoGDT dw NumDescriptoros*8
    ; Id dirección física hay que calcularla
    DireccionGDT dd 0

```

Lo primero que tenemos es el descriptor nulo con el que debe comenzar la GDT, seguido de cinco descriptores más: dos para los segmentos de código y datos de 32 bits, dos para los segmentos de código y datos de 16 bits y uno más que permitirá el acceso a toda la memoria disponible. Observe que la dirección base, repartida en una palabra y dos bytes independientes, está inicialmente a cero en todos los descriptores, pero sólo permanecerá así en el último.

Repasemos la estructura de los descriptores, comenzando por SelectorCodigo32, para comprender cuál es su significado. La longitud es de OFFFFFh o, lo que es lo mismo, 1.048.575 bytes o páginas, según los casos. Los 16 bits de menor peso de este dato, la longitud del segmento, se encuentra en la primera palabra de SelectorCodigo32, mientras que los 4 de mayor peso están en el penúltimo byte, expresado en binario. Los primeros cuatro bits, puestos a 1, es ese dígito F que añadimos a las primeras FFFF.

Del primer byte de indicadores tenemos, por una parte, los cuatro bits de mayor peso: 1001, en el que el bit más significativo está a 1 para indicar que el segmento está presente en memoria. Los demás tienen valores fijos. Los otros cuatro bits, 1010 en este caso, indican que se trata de un segmento de código que puede ejecutarse y leerse y con privilegio no ajustable.

En el penúltimo byte tenemos cuatro bits, los de mayor peso, con los indicadores 1100. Con ellos indicamos que el tamaño del segmento está expresado en páginas de 4 kilobytes, no en bytes, y que se trata de un segmento de 32 bits, no compatible con 80286. Los otros cuatro bits del byte pertenecen al tamaño.

Pasemos al siguiente descriptor: SelectorDatos32. La única diferencia respecto al anterior la encontramos en el primer byte de indicadores, concretamente en los bits que indican el tipo de segmento que, en este caso, es de datos y no de código.

Los dos descriptores que siguen pertenecen a los segmentos de código y datos en modo real, es decir, de 16 bits. Por eso la longitud de los segmentos no es OFFFFFh sino OFFFFh, puede ver que los cuatro bits de mayor peso, que están en el penúltimo byte, están a 0 en lugar de a 1. Además, el indicador G también está a 0 de tal manera que la longitud se interpreta como número de bytes y no de páginas, es decir, los segmentos tienen un tamaño de 64 kilobytes, no de 4 gigabytes.

Por último tenemos el descriptor SelectorPlano que, como se aprecia, es idéntico, al menos en principio, a SelectorDatos32. Después veremos que este último obtiene una dirección base que permitirá acceder a los datos desde el modo protegido, mientras que el primero quedará como está, facilitando el acceso a toda la memoria existente en el sistema.

Tras los descriptores encontramos una estructura de datos compuesta de una palabra, con el tamaño de la GDT, y una doble palabra que deberá almacenar la dirección física de la GDT y que calcularemos posteriormente. Esta estructura, 48 bits con el tamaño y dirección de la GDT, será la que se introduzca en el registro gdtr.

Para calcular el tamaño de la GDT se usa una constante, NumDescriptores, que definimos conjuntamente con otras que representarán los valores de selección de los descriptores. Estas constantes son:

```
*****  
;  
; Constantes  
*****  
  
. Número de descriptores  
%define NumDescriptores 6  
  
;  
; Selectores de esos descriptores  
;  
; para asignar a los registros de  
; segmento
```

```
%define SELNulo 0
"«define SELCod32 8
%define SELDat32 16
%define SELCod16 24
%define SELDat16 32
%define SELPlano 40
```

Si descarta los tres bits de menor peso de cada uno de los selectores, bits que indican el nivel de privilegio y que el selector hace referencia a la GDT, verá que los índices son el 0, 1, 2, 3, 4 y 5.

Cálculo de direcciones físicas

Nuestro programa se pondrá en marcha al invocarlo desde DOS, momento en el cual el sistema reservará un bloque de memoria convencional a partir de la dirección que esté disponible, y asignará esa dirección de segmento al es, configurando el registro ip con la dirección que se hubiese indicado con la directiva org. A partir de ese momento se pone en marcha la ejecución del programa que, para empezar, configurará la GDT calculando las direcciones físicas de los segmentos e introduciéndolas en los descriptores correspondientes.

Actualmente los descriptores correspondientes a los segmentos de código de 16 y 32 bits tienen la dirección base 0, pero el código realmente no está en esa dirección, sino en la que haya utilizado el sistema operativo para alojarlo y proceder a su ejecución. ¿Qué dirección es esa? La que contiene es en este momento.

El registro es contiene una dirección de segmento, no una dirección física real. No obstante, es fácil obtener esa dirección física: basta con desplazar cuatro bits hacia la izquierda el contenido de dicho registro. Puesto que esa operación no puede efectuarse directamente en es, ni en ningún otro registro de 16 bits, emplearemos un registro de 32 bits como es eax:

```
xor eax, eax
mov ax, ds
shl eax, 4
```

La dirección física, que tenemos en este momento en eax, hay que introducirla en los descriptores, de tal forma que al entrar en modo protegido el procesador pueda recuperarla. Ya sabe que hay que partirla en tres trozos: la palabra de menor peso por una parte y la de mayor peso dividida en dos bytes.

Teniendo el valor completo en eax, es fácil dividirlo usando ax, a1 y a h y las operaciones de desplazamiento de bits.

El caso de los descriptores correspondientes a los segmentos de datos es similar, pero usando ds en lugar de es como punto de partida.

Además, tenemos que calcular la dirección física donde se encuentra la GDT, sumando a la dirección física del segmento de datos el desplazamiento donde se encuentra esa estructura.

Todo este proceso lo hemos aislado en la rutina ConfiguraDescriptores, cuyo código puede ver a continuación:

```
*****  
; Esta rutina se encarga de  
; configurar la tabla de  
; descriptores de segmentos  
*****
```

ConfiguraDescriptores:

```
; Calculamos la dirección física del  
; segmento de datos

xor eax, eax ; ponemos a 0 EAX
mov ax, ds ; y obtenemos el segmento
Shl eax,4 ; lo desplazamos 4 bits

; Lo que tenemos en este momento en
; F.AX es la dirección física del segmento
; de datos. La usamos para establecer la
; dirección base de los segmentos de
| datos de 16 y 32 bits

push eax ; guardamos temporalmente

; los bits 0-15 de la dirección base
mov [SelectorDatos32+2],ax
mov [SelectorDatos16+2],ax

; tomamos en AX los otros 16 bits
shr eax, 16

; colocamos los bits 16-23
mov [SelectorDatos32+4j],al
mov [SelectorDatos16+4],al

; y los bits 24-31
mov [SelectorDatos32+7],ah
mov [SelectorDatos16+7],ah

; recuperamos EAX
pop eax

; Calculamos la dirección física de
; la GDT

; tomamos el desplazamiento donde
; se encuentra el primer selector
mov ebx, SelectorNulo
; y lo sumamos
add eax, ebx

; En este momento tenemos en EAX una
; dirección física de 32 bits, que
; establecemos como inicio de la GDT
mov [DireccionGDT], eax
```

```

; Calculamos la dirección física del
; segmento de código

xor eax, eax ; ponemos a 0 EAX
mov ax, es    ; y obtenemos el segmento
shl eax,4     ; lo desplazamos 4 bits

; los bits 0-15 de la dirección base
mov [SelectorCodigo32+2],ax
mov [SelectorCodigo16+2],ax
; tomamos en AX los otros 16 bits
shr eax, 16

; colocamos los bits 16-23
mov [SelectorCodigo32+4],al
mov [SelectorCodigo16+4],al
; y los bits 24-31
mov[SelectorCodigo32 + 7] , ah
mov [SelectorCodigo16+7],ah

ret ; hemos terminado

```

Núcleo del programa

Teniendo ya la tabla global de descriptores completamente dispuesta, podemos centrarnos en la parte principal del código que se encargará de pasar al modo protegido, mostrar el mensaje y volver. Los pasos se van describiendo individualmente para facilitar su comprensión. Fíjese en el uso de las directivas BITS16 y BITS32 donde es necesario para indicar las partes del código que se ejecutarán en modo real y en modo protegido.

```

; Creamos un COM
ORG 100h
; Zona de código de 16 bits
BITS 16

*****  

; Segmento de código
*****  

.  

Inicio:
; Configuramos los descriptores
Cali ConfiguraDescriptores

; Desactivamos interrupciones
Cli
; cargamos la GDT
Igdт [DatosGDTR]

; activamos el modo protegido
mov eax,cr0
or al, 1
mov cr0, eax

```

```

; y descartamos todo el contenido
; de la cola de instrucciones del
; procesador
jmp SF,LCod32 ;CódigoProtegido

; Zona de código de 32 bits
BITS 32
CódigoProtocido:

; Tomamos el selector de datos
mov ax,SELDat32
; y lo ponemos en DS
mov ds,ax

; Tomamos el selector correspondiente
; al segmento plano de 4 Oh
mov ax,SELPlano
; y lo ponemos en ES
mov es,ax

; Tomamos la dirección del segmento
mov es,i, MsgProtegido

; introducimos en DI la dirección
; física de la memoria de pantalla
mov edi, 0B8000h
cid

; Borramos todo
mov ax, 7020h
mov ecx, 2000
rep stosw

; Tomamos de nuevo la dirección
mov edi, 0B8000h

Bucle: ; y mostramos el mensaje

lodsb      ; Leer un carácter
or al, al  ; Si es 0
jz FinBucle ; hemos terminado

stosb      ; En caso contrario escribir
mov al, 70h ; junto con el atributo
stosb

jmp Bucle  ; repetir hasta el fin

FinBucle: ; Hemos terminado

; Descartamos el contenido de la cola
; de instrucciones y saltamos usando
; el descriptor de segmento de código
; de 1S bits
jmp SELCod16:SalirModoProtegido

```

```

; Zona de código de 16 bits
BITS 16
SalirModoProtegido:
    ; Tomamos el descriptor del segmento
    ; de datos de 16 bits
    mov ax,SELDat16
    ; y lo ponemos en DS
    mov ds,ax

    ; Desactivamos el modo protegido
    mov eax,cro
    and eax,0Feh
    mov cr0,eax

    ; Termi namos
    jmp Salir

Salir:
    sti ; reactivando las interrupciones

    ; y devolviendo el control al sistema
    mov ah, 4ch
    int 21h

```

Una vez se ha cargado la dirección de la GDT en el registro gdtr, modificamos el primer bit del registro cr0 para activar el modo protegido y saltamos al segmento de código de 32 bits, usando como dirección el selector de ese descriptor, que se introduce en es, y la dirección de la etiqueta `CodigoProtegido`, que pasa a eip. Funcionando ya en modo protegido, configuramos adecuadamente los registros ds y es seleccionando los segmentos de datos y el segmento plano que parte desde la dirección 0.

Sirviéndonos de ese segmento, y con un registro de 32 bits como puede ser edi, podemos acceder a cualquier dirección física de memoria. Al asignar el valor 0B8000h a edi, por ejemplo, estaremos accediendo a la memoria de pantalla. Observe que la dirección es 0B8000h, no es 0B8 0 0h, ya que ahora estamos hablando de direcciones reales, no direcciones de segmento que después se multiplican.

Asumiendo que hemos iniciado el programa desde DOS, y que no estamos usando ningún gestor de memoria ni aplicación que acceda a la memoria extendida, podríamos sin ningún problema usar toda la memoria que tenga el equipo, ya sean 16 megabytes o 1 gigabyte, da igual. En este programa nos limitamos a borrar la pantalla y mostrar la cadena.

Finalizado el proceso, saltamos a un segmento de 16 bits que aún se ejecutará en modo protegido, puesto que el bit 0 de cr0 sigue a 1. En dicho segmento asignamos a ds el selector del segmento de datos, desactivamos el bit de cr0 para volver al modo real y, finalmente, reactivamos las interrupciones para terminar devolviendo el control al sistema.

Al ejecutar el programa, siempre que lo haga desde DOS y no desde una ventana de Windows u otro sistema que use el modo virtual V86, obtendrá el resultado que se aprecia en la figura 30.8. Pruebe a ampliar el código que se ejecuta en modo protegido efectuando otras operaciones y experimentar con los resultados.

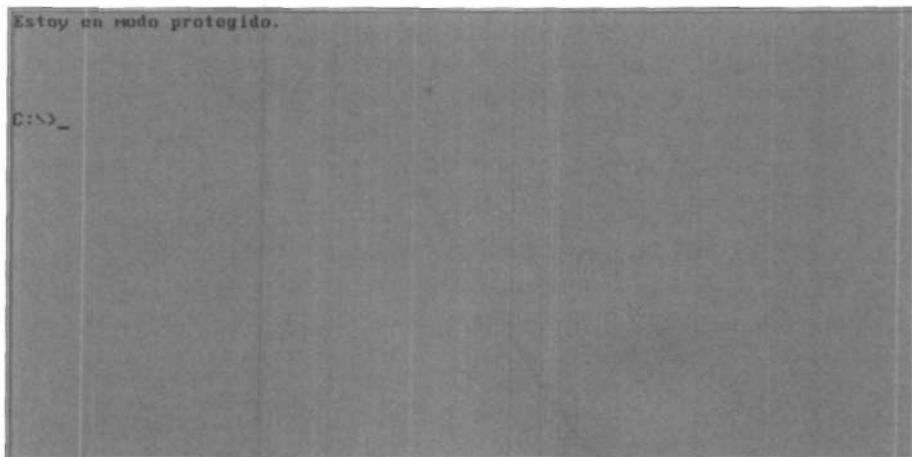


Figura 30.8. El programa accede a la pantalla desde el modo protegido.

Interrupciones en modo protegido

Como ha podido ver, el programa de ejemplo anterior desactiva las interrupciones al inicio del código y no las reactiva prácticamente hasta devolver el control al sistema. De no hacerlo así el programa difícilmente llegaría a funcionar, ya que las interrupciones se gestionan en el modo protegido de forma totalmente diferente a la que conocemos hasta ahora.

La tabla de vectores en modo real ocupa el primer kilobyte de memoria física, desde la dirección 0, con un total de 256 entradas compuestas cada una de ellas de dos palabras. Una de ellas contiene el segmento y la otra el desplazamiento correspondiente al código que debe gestionar la interrupción. Como acaba de verse en los puntos previos, al trabajar en modo protegido las direcciones de segmento no sirven de nada, son necesarios selectores, algo que no contiene la tabla de vectores inicial.

Incluso aunque la tabla de vectores pudiese modificarse, a fin de que lograr el acceso al código de gestión de cada interrupción, también hay que tener en cuenta que todo ese código, tanto el de la BIOS como el de las interrupciones DOS, es código de 16 bits y, por tanto, no puede ser ejecutado en un entorno de 32 bits, al menos no directamente transfiriéndole el control sin más.

En modo protegido existe una tabla de vectores de interrupción, llamada IDT (*Interrupt Descriptor Table*), que puede encontrarse en cualquier posición de memoria, no necesariamente en el primer kilobyte físico. Esto es posible gracias a la disponibilidad de un registro, llamado `idtr`, cuya dirección se introduce mediante la instrucción `lidt`, de forma similar a como se configura el registro `gdtr`.

A pesar de contar con el mismo número de entradas que en el modo real, la IDT ocupa justo el doble ya que cada entrada se compone de ocho bytes en lugar de cuatro. Con ellos se indica el selector de segmento y dirección de 32 bits donde está el código de gestión de la interrupción, así como los atributos asociados a cada entrada.

Iniciando la operación en modo protegido con un programa desde DOS, tal y como lo hemos hecho en el ejemplo previo, no existe IDT ni rutinas de servicio a las interrupciones. Tendríamos, por tanto, que preparar nosotros mismos la IDT, como hemos hecho con la GDT, y escribir el código de las interrupciones que necesitásemos. Tendríamos que duplicar, por ejemplo, las interrupciones de acceso al adaptador de vídeo o las unidades de disco escribiendo básicamente sólo instrucciones in y out.

Una solución, costosa en tiempo pero que ahorra mucho trabajo en principio, consiste en cambiar a modo real cada vez que se necesita hacer uso de un servicio de la BIOS, volviendo a continuación al modo protegido. Cada cambio de este tipo requiere que se guarde el estado actual de los selectores de 32 bits para pasar a los de 16 y vuelta.

DPMI

^ ^

Acaba de comprobar que la programación en modo protegido no es precisamente una tarea fácil, ya que una vez ha activado dicho modo se encuentra, por decirlo así, directamente sobre el hardware del ordenador, sin ningún servicio que le facilite trabajo alguno. Por regla general, cuando un programa va a crear una cierta aplicación su objetivo es centrarse en la funcionalidad de dicha aplicación, y no en todos los detalles relativos a cómo debe acceder a cada uno de los recursos que precise.

Ésta es la razón, por ejemplo, de que, con el tiempo, aparezcan especificaciones como XMS (*Extended Memory Specification*), VCPI (*Virtual Control Program Interface*) o DPMI (*Dos Protected Mode Interface*). La primera, cuyos servicios conocíó en un capítulo previo, facilita el uso de la memoria extendida desde aplicaciones en modo real, mientras que las otras dos lo que hacen es llevar un programa a esa memoria extendida y ejecutarlo desde allí, usando para ello el modo V86 de los procesadores 80386 y posteriores.

La aparición de DPMI, cuya versión más extendida es la 0.9, propició la aparición de aplicaciones para DOS capaces de aprovechar todos los recursos disponibles, mejorando así la capacidad de hojas de cálculos, bases de datos y aplicaciones similares.

Anfitriones DPMI

Para poder usar DPMI lo primero que necesitamos es una *anfitrión DPMI*, es decir, el software que ofrezca los servicios definidos en la especificación. Las versiones más usadas del DOS, como el MS-DOS 6.2 o el DR-DOS 6.0, no incluyen un anfitrión DPMI, pero sí lo hacían versiones posteriores como DR-DOS 7. Por ello aparecieron anfitriones independientes que podían instalarse en cualquier versión del DOS, como cwsppmi que puede obtenerse de empaquetado como csdpmi4b.zip en distintos sitios, por ejemplo http://site.n.mi.org/info/_msdos-devel, y aplicaciones comerciales más complejas que, aparte de actuar como anfitriones DPMI, ofrecían la posibilidad de tener multitarea en DOS con múltiples consolas, como en Linux. Uno de los productos más conocidos, de este tipo, era QEMM/DESQview de Quarterdeck. La mayoría de sistemas operativos que facilitan consolas DOS, como es el caso de Windows u OS/2, actúan

directamente como anfitriones DPMI, de tal forma que es posible usar los servicios de la especificación sin necesidad de instalar nada más. En este sentido, nos resultará más fácil generalmente acceder a servicios DPMI desde una consola DOS de Windows que desde un sistema que realmente ejecute sólo DOS como sistema operativo.

Aprenderá de inmediato a detectarla presencia o ausencia de un anfitrión DPMI desde aplicaciones propias con la simple ejecución de un servicio de la interrupción 2Fh.

Clientes DPMI

Se conoce como *clientes DPMI* a las aplicaciones que usan los servicios que ofrecen los anfitriones DPMI. Dichas aplicaciones, aunque pueden ser programas corrientes como se verá en los ejemplos mostrados a continuación, por regla general son extensores del DOS, programas cuyo objetivo es facilitar la ejecución de otros programas en modo protegido. El uso de un extensor, como se explica en el último punto del capítulo, comporta muchas ventajas para el desarrollador.

El primer paso que debe dar un cliente DPMI es comprobar la presencia del anfitrión ya que, de lo contrario, no podrá usar servicio alguno. El anfitrión le facilitará diversos datos, entre ellos el tamaño del bloque de memoria que precisa para almacenamiento de parámetros locales y la dirección a la que debe transferir el control la aplicación para entrar en modo protegido. Una vez se ha activado el modo protegido, el cliente hará uso de los servicios del anfitrión DPMI, principalmente a través de la interrupción 31h. Con ellos podrá definir descriptores locales, reservar y liberar memoria, acceder a vectores interrupción del modo real, etc.

Al finalizar la ejecución, el cliente devolverá el control al anfitrión y éste, a su vez, retornará al modo real y devolverá el control al DOS.

Detectar la presencia de un anfitrión DPMI

Lo primero que debe hacer una aplicación que pretenda usar los servicios DPMI es comprobar la presencia de un anfitrión que los ofrezca, para lo cual se servirá del servicio 1687h de la interrupción 2Fh. No se necesita ningún parámetro, devolviéndose información en los registros siguientes:

- ax: Contendrá el valor 0 en caso de que exista un anfitrión DPMI, teniendo cualquier otro valor en caso contrario.
- bx: El primer bit estará a 1 si el anfitrión DPMI contempla la ejecución de aplicaciones de 32 bits o bien a 0 en caso contrario. El estado de los demás bits es indeterminado.

- *el*: Contendrá un código que indica el tipo de procesador con que cuenta el sistema. La tabla 30.2 enumera algunos de los documentados en la especificación.
- *dx*: Contiene la versión de la especificación DPMI a la que se ajusta el anfitrión. El registro dh contendrá el número más significativo y di la revisión menor, estando ambos datos expresados en binario.
- *si*: Indica el número de párrafos de memoria que necesita el anfitrión por parte del cliente para poder hacer el cambio al modo protegido.
- *es : di*: Dirección de la rutina a la que hay que transferir el control para entrar en modo protegido.

Tabla 30.2. Códigos que identifican el tipo de microprocesador.

Código	Procesador
2	80286
3	80386
4	80486
5	Pentium
6	Pentium Pro/Pentium II
>6	Otros procesadores

Conociendo tan sólo este servicio podríamos codificar un programa que, como el siguiente, nos indicase si está presente o no un anfitrión DPMI y, en caso afirmativo, mostrase algunos datos acerca de él.

En la figura 30.9 puede observar cómo se ejecuta el programa en un sistema DOS sin DPMI y tras iniciar el anfitrión cwsdpmi mencionado anteriormente.

```

segment Pila stack
    resw 512
FinPila:

segment Datos
; Mensajes informativos
MsgNoDPMI db 'No hay servicios DPMI.$'
MsgProccsarioor db 'El tipo de procesador es '
    Procesador db '     ,13,10,10,'$
MsgSi32Bits
    db 'Puede ejecutar programas de 32 bits.'
    db 13,10,10,'$'
MsgNo32Bits
    db 'No puede ejecutar programas de 32 bits.'
    db 13,10,10,'$'
MsgVersion db 'La versión de ÜPMI es la '
    Vers1 db '0.'
    Vers2 db '00',13,10,10,'$'

```

```
MsgParrafos db 'Se necesitan '
Párrafos db '
db ' párrafos de memoria para el acceso.'
db 13,10,10,'?'

; Segmento de código
. *****
    segment Código
..start:
    ; Preparar los registros de pila
    mov ax, Pila
    mov ss, ax
    mov sp, FinPila
    ; y del segmento de datos
    mov ax, Datos
    mov ds, ax
    mov es, ax

    ; Comprobamos si hay un anfitrión
    ; DPMI instalado en el sistema
    mov ax,1687h
    int 2fh

    ; en caso de no ser así
    or ax, ax
    ; salir directamente
    jnz Error

    ; Tomar el número de versión
    add dh, '0'
    mov [Vers1], dh
    ; y convertir en ASCII
    mov al, di
    mov di, 10
    div di
    add ah, '0'
    add al, '0'
    ; para componer el mensaje
    mov [Vers2], al
    mov [Vars2+1], ah
    ; y mostrarlo en pantalla
    mov dx, MsgVersion
    mov ah, 9
    int 21h

    ; Convertimos el código de procesador
    ; para mostrarlo
    mov al, el
    xor ah, ah
    mov di, Procesador+2
    cali Enterocadena

    mov dx,MsgProcesador
    mov ah, 9
    int 21h
```

```

; Convertimos también el número de
; párrafos de memoria que necesita
mov ax, si
mov di, Parrafos+4
cali Enterocadena
mov dx, MsgParrafos
mov ah, 9
int 21h

; Y por último comprobamos si contempla o
; no la ejecución de aplicaciones de
; 32 bits
mov dx, MsgSi32Bits

test bx, 1
jnz Salir

mov dx, MsgNo32Bits
jmp Salir

Error:
; No hay un anfitrión DPMT instalado
mov dx, MsgNoDPMI

Salir:
; Mostrar el último mensaje
mov ah, 9
int 21h

; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

; Necesitamos la rutina de conversión
%include "Convert.inc"

```

```

C:\>30>
C:\>30>haydpmi
No hay servicio DPMI.
C:\>30>C:\>30>
C:\>30>haydpmi
La versión de DPMI es la 0.90
El tipo de procesador es 04
Se necesitan 06 párrafos de memoria para el acceso.
Puede ejecutar programas de 32 bits.
C:\>30>_

```

Figura 30.9. Información aportada por el programa sobre el anfitrión DPMI.

Activación del modo protegido

En caso de que la llamada al servicio 1687h nos indique que existe un anfitrión DPMI, y que contempla la ejecución de aplicaciones de 32 bits en caso de que la nuestra lo sea, podemos dar el paso siguiente: cambiar del modo real al modo protegido. Para ello deberemos dar los pasos siguientes:

- Guardar la dirección facilitada en es : di para poder usarla posteriormente.
- En caso de que el valor devuelto en si por el servicio 1687h sea distinto de cero, deberemos reservar un bloque de memoria de ese tamaño usando para ello el habitual servicio 48h de la interrupción 21h.
- Introducir en es la dirección de segmento correspondiente al bloque de memoria reservado.
- Indicar mediante ax si la aplicación es de 16 o 32 bits. En el primer caso el valor a asignar será 0 y 1 en el segundo.
- Transferir el control a la dirección del punto de entrada indicado por el anfitrión mediante una instrucción cali.

Si todo va bien, el anfitrión DPMI devolverá el control a la instrucción que sigue al cali y el indicador de acarreo estará desactivado. Si está activo es que no se ha podido completar el cambio a modo protegido.

Recuerde que al iniciar un programa, desde la línea de comandos, el DOS reserva toda la memoria disponible y se la asigna a dicho programa. Antes de poder utilizar la función 48h, por tanto, tendríamos que ajustar el tamaño del bloque de memoria como hacíamos en ejemplos de capítulos previos.

El cambio a modo protegido implica, por parte del anfitrión DPMI, la creación de una tabla de descriptores ajustada a las necesidades de nuestro programa. Al volver de la instrucción cali los registros de segmento contendrán los siguientes selectores:

- es: índice de un descriptor cuya base será la dirección física que tenía el código del programa en modo real y un límite de 64 kilobytes.
- ds: índice de un descriptor cuya base será la dirección física que tenía el registro ds en modo real y una longitud máxima de 64 kilobytes.
- es: índice de un descriptor con la dirección base apuntando al área de memoria donde se encuentra el PSP del programa y una longitud de 256 bytes.
- ss: índice de un descriptor cuya base será la dirección física que tenía el registro ss en modo real y una longitud de 64 kilobytes.

El anfitrión preparará, además, una IDT que permita al cliente acceder a las distintas interrupciones de servicio. Por ejemplo, para terminar el programa y volver al modo real, devolviendo el control al DOS, usaremos el servicio 4Ch de la interrupción 21h. En realidad esa interrupción apunta a una rutina de gestión de servicio del propio anfitrión DPMI que cambiará al modo real y remitirá la interrupción para que pueda ser ejecutada por el DOS. Si intenta usar cualquier otro servicio de la interrupción 21h, o cualquier interrupción DOS o BIOS, simplemente no ocurrirá nada. El anfitrión gestiona esas interrupciones pero se limita a devolver el control, sin más.

Servicios DPMI

Una vez nos encontramos en modo protegido, tenemos a nuestra disposición todos los servicios del anfitrión a través de la interrupción 31h. El código del servicio a usar se facilita siempre en ax, registro que también suele usarse para devolver el resultado que corresponda. El número de servicios disponibles ronda el centenar. Encontrará los códigos de cada uno de ellos, así como los parámetros de entrada y salida, en la documentación de la especificación DPMI. A continuación sólo van a mencionarse algunos de ellos. Mediante el servicio 500h podemos obtener información sobre la memoria con que cuenta el sistema, sabiendo así el tamaño del bloque mayor, el número de megabytes totales, el espacio de intercambio en disco, etc. Los servicios 501h y 502h facilitan la asignación y liberación, respectivamente, de bloques de memoria, obteniendo siempre una dirección física de 32 bits.

Puesto que estamos operando en modo protegido, no podemos usar sin más una dirección física. Tendremos que crear un nuevo descriptor en la LDT, mediante el servicio 000h, estableciendo su dirección base con el servicio 007h y su tamaño con el servicio 008h. En el caso de ser necesario, también se fijarán los atributos del descriptor con el servicio 009h. Todos los descriptores creados con el servicio 000h deben ser liberados antes de devolver el control al sistema, para lo cual se usa el servicio 001h.

Un servicio especialmente útil cuando se está acostumbrado a trabajar en modo real es el 002h, ya que nos permite obtener un selector asociado a una dirección de segmento. No hay más que facilitar en el registro bx la dirección de segmento y obtendremos en ax el selector correspondiente al descriptor creado por el anfitrión. Este descriptor, además, no hay que liberarlo.

Además hay servicios para gestión de interrupciones, simulación de interrupciones en modo real, llamadas a rutinas con código de modo real, etc.

Un ejemplo

Demos en la práctica los pasos descritos en los puntos anteriores creando un programa cuyo objetivo será activar el modo protegido, obtener un selector que permite acceder a la memoria de vídeo, llenar la pantalla de asteriscos y, finalmente, devolver el control al modo real y luego salir al DOS. El código, que compilaríamos mediante el comando nasm -o dpmisimple.com dpmisimple.asm, sería el siguiente:

```

cpu 386 ; Vamos a usar instrucciones 386

; Generaremos un COM
org 100h

. *****
; Segmento de código

Tnicio:
; Ajustamos el bloque de memoria
;- ocupado por el programa
mov bx, FinPrograma

; Convertimos a parrales
shr bx, 4
inc bx
; ajustamos
mov ah, 4ah
int 21h

; Comprobamos si hay un anfitrión
I DPMI instalado en el sistema
mov ax,1687h
int 2fh

; en caso de no ser así
or ax, ax
; salir directamente
jnz Error

; Si no se permite la ejecución de
; aplicaciones de 32 bits
test bx, 1
; salir
jz Error

; Guardamos la dirección del punto
; de entrada al modo protegido
mov [DirModoProtegido] , di
mov [DirModoProtegido+2] , es

; ¿Hay que reservar memoria?
or si, si
; si no es así saltar
jz EntrarModoProtegido

; Reservamos el número de párrafos
; solicitado por el anfitrión
mov bx, si
mov ah, 48h
int 21h
je ErrorMemoria

; Ponemos en ES el segmento del
; área de datos
mov es, ax

```

```
EntrarModoProtegido:
    ; indicamos 32 bits
    mov dx, 1

    ; Entramos en modo protegido
    cali tar [DirModoProtegido]

    ; saltar si hay error
    je ErrorEntrada

    ; Estamos en modo protegido

    ; Solicitamos un selector para accod^i~
    ; al segmento de pantalla
    mov ax,2
    mov bx,ObñOOh
    int 31h

    ; si no se puede obtener saltar
    je SalirModoProtegido

    ; Ponemos el selector en ES
    mov es,ax

    ; Llenamos la pantalla de asteriscos
    xor di, di
    mov ax,1F2Ah
    mov ex,2000
    cid
    rep stosw

SalirModoProtegido:
    I Devolvemos el control al modo real
    mov ah, 4ch
    int 21h

    ; Apartados que muestran los
    ; distintos errores
ErrorMemoria:
    mov dx, MsgErrorMemoria
    jmp Salir

ErrorEntrada:
    mov dx, MsgErrorEntrada
    jmp Salir

Error:
    ; No hay un anfitrión DPMI instalado
    mov dx, MsgNoDPMI

Salir:
    ; Mostrar el último mensaje
    mov ah, 9
    int 21h
```

```

; devolvemos el control
; al sistema
mov ah, 4Ch
int 21h

; Para guardar la dirección de entrada a modo protegido
DirModoProtegido dd 0

; Mensajes informativos
MsgNoDPMI db 'No es posible entrar en modo protegido.$'
MsgErrorEntrada db 'Error al intentar cambiar.$'
MsgModoProtegido db 'Estoy en modo protegido.$'
MsgErrorMemoria db 'No puede asignarse la memoria.$'

```

FinPrograraa:

Para ver el funcionamiento del programa lo más aconsejable es que use un sistema con DOS y algún anfitrión instalado previamente, como cwsdpmi. Al ejecutar el programa se encontrará, si todo va bien, con el resultado que aparece en la figura 30.10. El programa ha entrado y salido del modo protegido con muchas menos operaciones, y menos complejas, que las que dábamos anteriormente al controlarlo todo nosotros mismos. Lógicamente, ahora dejamos el trabajo más difícil al anfitrión DPMI.

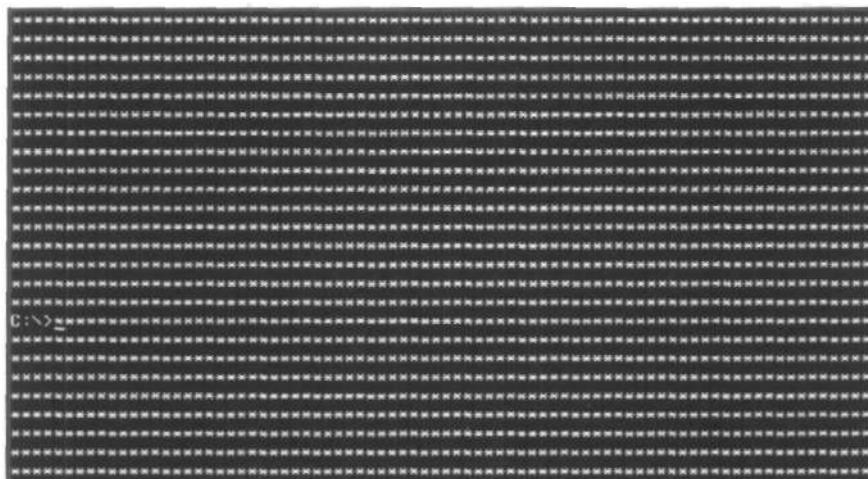


Figura 30.10. Aspecto de la pantalla tras entrar y salir del modo protegido.

Extensores DOS

A pesar de que un anfitrión DPMI puede ahorrarnos una gran cantidad de trabajo y simplificar nuestras aplicaciones en modo protegido, lo cierto es que aún hay muchos elementos que tendremos que controlar mediante código propio. Seguimos sin tener servicios que nos permitan acceder a la mayoría de dispositivos y, o lo hacemos manualmente

mediante instrucciones in y out, o bien usamos los servicios del anfitrión para cambiar al modo real, invocar la interrupción y volver al modo protegido. Tal y como se dijo anteriormente, ésta es una operación que consume mucho tiempo, por lo que siempre es recomendable ejecutar directamente desde el modo protegido todo el código posible.

Si pretendemos crear aplicaciones completas que funcionen en modo protegido, lo más recomendable, aún a pesar de programar directamente en ensamblador, es hacerse con un extensor del DOS. Éste actuará como cliente DPMI, por una parte, y por otra como un módulo que se une a nuestros programas para hacer que éstos operen directamente en modo protegido sin, por ello, perder la posibilidad de acceder a las interrupciones habituales. Para ello los extensores implementan su propio código de gestión de interrupciones, en unos casos, o bien automatizan el proceso de ejecución del código original de la interrupción en modo real, en otros.

En Internet pueden encontrarse multitud de extensores para DOS, tanto comerciales como gratuitos, e, incluso, ensambladores que ya añaden a la aplicación resultante el propio extensor. Es el caso, por ejemplo, de Pass32, un ensamblador que puede obtenerse de <http://people.freenet.de/dieterp/main.htm> junto con su extensor, llamado Pro32, y una completa documentación de unas 150 páginas sobre cómo programar en modo protegido con este ensamblador. También incorpora un depurador específico que hace posible el seguimiento de las aplicaciones en modo protegido, algo que no es posible con DEBUG, por ejemplo.

Algunos extensores ofrecen, además de apoyo para acceder a las rutinas básicas del DOS y la BIOS, bibliotecas propias que facilitan aspectos como la programación gráfica. Indistintamente de las posibilidades, programar con un extensor no difiere en exceso de programar en el entorno que hemos conocido hasta ahora.

Resumen

Los procesadores 80386 y posteriores cuentan con unas posibilidades de direccionamiento, protección e intercambio de tareas que quedan ocultas desde que se pone en marcha el sistema, ya que a partir de ese momento se opera en modo real. En este capítulo ha conocido la mayor parte de los detalles necesarios para poder cambiar a modo protegido, aunque sólo para efectuar alguna operación, más o menos simple, y volver de inmediato al modo real.

Con la documentación de referencia sobre el procesador 80386 es posible crear prácticamente un sistema operativo multitarea, habilitando mecanismos de protección entre tareas, paginación de memoria a disco para ofrecer memoria virtual, etc. La mayoría de los programadores que usan ensamblador, sin embargo, no necesitan llegar a este nivel y optan por recurrir a extensores que, mediante servicios DPMI o similares, facilitan la ejecución de aplicaciones DOS en modo de 32 bits. De esta forma se facilita el acceso a toda la memoria disponible y, por regla general, se obtiene una mayor velocidad al poder emplear registros y operaciones de 32 bits en lugar de emplear los de 16 bits.

31

Interfaz entre ensamblador y C/C++

Como ha podido comprobar al completar los distintos ejercicios propuestos en los capítulos de este libro, implementar una mínima funcionalidad utilizando exclusivamente el lenguaje ensamblador es una tarea ardua y con un cierto nivel de complejidad. No solamente es necesario escribir larguísimas listas de instrucciones, dado que cada una de ellas lleva a cabo acciones muy simples, sino que también resulta difícil la depuración y el mantenimiento del código.

No es habitual desarrollar aplicaciones completas en lenguaje ensamblador, especialmente en los actuales sistemas operativos con sofisticadas interfaces de usuario. Esto no implica, sin embargo, que no haya cabida para el uso del ensamblador en ciertos contextos, especialmente allí donde es necesario llevar a cabo tareas de muy bajo nivel o en las que alcanzar el rendimiento máximo posible justifica ese esfuerzo. En estos casos lo que se hace es codificar pequeños procedimientos en ensamblador, invocándolos normalmente desde el código de un programa principal.

Son muchos los lenguajes de programación de alto nivel que pueden ser utilizados para desarrollar aplicaciones, pero no todos ellos producen código directamente ejecutable que pueda ser enlazado con subrutinas escritas en ensamblador. Dos de los lenguajes más utilizados en la actualidad, como son Java y Visual Basic, cuentan con compiladores que producen un código intermedio (*p-code* o *bytecode*) que es ejecutado en el interior de una máquina virtual, la cual forma parte de la plataforma Java o la plataforma .NET, respectivamente.

También son muy populares ciertos lenguajes relacionados con el desarrollo Web, como es el caso de PHP, Python o JavaScript, lenguajes que, obviamente, no están enfocados al desarrollo de sistemas ni la programación a bajo nivel.

De entre los lenguajes con compiladores diseñados para generar código directamente ejecutable, los empleados con mayor asiduidad en proyectos donde se requiere velocidad y cercanía al hardware son indudablemente C y C++. Por ello en este breve capítulo nos centraremos exclusivamente en analizar los métodos a los que podemos recurrir para crear una interfaz entre ensamblador y dichos lenguajes.

Ensamblador embebido

Una de las vías más cómodas para combinar en una misma aplicación partes escritas en lenguaje ensamblador con otras implementadas en C/C++, especialmente cuando la mayor parte sea desarrollada con este último lenguaje y la aparición del ensamblador sea puntual, consiste en embeber este último en aquellos puntos donde se necesite.

La mayoría de los compiladores de C/C++ cuentan con una palabra clave que facilita la introducción de instrucciones en ensamblador, ya sea de manera aislada o bloques completos de sentencias. El compilador de GNU, disponible para múltiples sistemas operativos y especialmente popular en Linux, cuenta con la palabra clave `asm`. También Visual C++, posiblemente el compilador más empleado en Windows, dispone de un mecanismo similar, si bien en este caso la palabra clave es `__asm`.

Dependiendo del compilador de C/C++ que esté utilizándose es posible que la sintaxis de las instrucciones escritas en ensamblador no sea la que estamos habituados a usar. Las instrucciones lógicamente se siguen denominando de la misma forma, pero el orden de los operandos, y la forma de referirse a éstos, sí es posible que cambien. Es lo que ocurre, por ejemplo, con el ensamblador embebido en GCC, ya que este compilador sigue la sintaxis de AT&T en lugar de la de Intel.

La mayoría de los ensambladores para x86 se ajustan a la sintaxis establecida por el propio fabricante: Intel. Ésta es la sintaxis que ha conocido y utilizado a lo largo de todo el libro. Hay, sin embargo, algunas excepciones a esta norma, formada por los ensambladores y compiladores (como GCC) que se ajustan a la sintaxis definida por AT&T.

Con leves diferencias, la mayoría de los compiladores de C/C++ seguirán el modelo de Visual C++ o bien el de GCC, de ahí que a continuación tomemos éstos como base para el resto de las explicaciones.

Visual C++

Este compilador de Microsoft sigue la sintaxis estándar de Intel y, además, las sentencias escritas en ensamblador no precisan ningún tipo de delimitador específico. Basta con que vayan precedidas de la palabra clave `__asm` o formen parte de un bloque `__asm`.

Para escribir una única sentencia en ensamblador, por tanto, basta con precederla de esa palabra clave. Por ejemplo:

```
__asm mov cax, ebx;
```

En este caso se emplean registros de 32 bits porque el código generado por el compilador de C++ se ejecutará en una plataforma de 32 bits. Esta forma de introducir instrucciones en ensamblador es cómoda siempre que su número no sea muy grande, por lo que resulta adecuado para escribir fragmentos de procedimiento como podría ser el caso siguiente:

```
int máximo(int a, int b)
{
    __asm mov eax, a;
    __asm cmp eax, b;
    __asm jge fin;

    a - b;

fin:
    return a;
```

Hay algunos aspectos que merece la pena destacar en este código. El primero es la forma en que se utilizan en las sentencias escritas en ensamblador el contenido de variables locales o parámetros de la función, sencillamente haciendo referencia a las mismas. Esto es posible siempre que los dos operandos que aparecen en la sentencia sean del mismo tamaño, de lo contrario el compilador generaría un error. Una variable de tipo `int` en una aplicación C/C++ de 32 bits tiene precisamente ese tamaño: 32 bits. Por ello pueden llevarse a un registro como `eax` sin ningún problema.

Otro aspecto destacable es el relacionado con las instrucciones de salto en ensamblador embebido que, como puede apreciarse en este ejemplo, pueden hacer referencia a etiquetas C/C++.

En caso de que el número de instrucciones a escribir en ensamblador sea importante, y todas ellas aparezcan de manera secuencial, resultará mucho más cómo crear un bloque en vez de preceder cada sentencia con `__asm`. Para ello no hay más que utilizar las llaves, como en cualquier otro caso. La función cuyo código tiene a continuación genera exactamente el mismo resultado que el anterior, pero en este caso todo el código está escrito en ensamblador:

```
int maximo2(int a, int b)
{
    __asm (
        mov eax, a;
        cmp eax, b ;
        jge fin;
        mov eax, b;
    fin:
    }
}
```

Observe que en este caso no existe ninguna instrucción `return` con la que se indique explícitamente el valor a devolver desde la función. El valor de retorno será el que se contenga el registro `eax` en el momento de salir de la función. De hecho la instrucción `return` de C/C++, en este caso concreto, lo que haría sería asignar al registro `eax` el valor entregado como parámetro.

Por simplicidad y brevedad, en los ejemplos ofrecidos se han usado solamente tres instrucciones del lenguaje ensamblador pero, lógicamente, podría utilizar todas las que ya conoce y muchas otras que no se han tratado en este libro, como las instrucciones MMXySSE.

GCC

El compilador libre de GNU cuenta básicamente con la misma capacidad para usar ensamblador embebido que el compilador de Microsoft, si bien la sintaxis empleada sí que difiere claramente. La palabra clave a usar es `asm` o, alternativamente, `__asm__` si se quieren diferenciar claramente las sentencias en ensamblador y evitar que la palabra clave `asm` entre en conflicto con algún otro identificador.

Las instrucciones en ensamblador deben facilitarse como una cadena de caracteres que se entregaría a `asm` en forma de argumento. Es una técnica que resulta algo menos intuitiva que la descrita antes para Visual C++, especialmente cuando se trata de introducir un bloque de instrucciones.

Cada una de las instrucciones introducidas en la cadena han de seguir la sintaxis de AT&T, cuyos aspectos más destacables se resumiría en los puntos siguientes:

- En las instrucciones en las que aparece un origen y un destino, el caso más habitual es `mov`, el orden de los operandos se invierte, apareciendo primero el origen y después el destino.
- Todos los registros han de precederse con el símbolo `%`.
- Los valores literales deben ir precedidos del símbolo `$`.
- El indicador de que un valor numérico está expresado en hexadecimal no es el sufijo `h` sino el prefijo `0x`,
- Las referencias a posiciones de memoria, empleando direccionamiento indirecto e indexado, se efectúan sustituyendo los corchetes por paréntesis.
- El nombre de muchas instrucciones se completa con un sufijo de un carácter de longitud que indica el tamaño de los operandos: `b` indica *byte* (8 bits), `w` indica *word* (16 bits) y `l` indica *long* (32 bits).

Supongamos que, por ejemplo, queremos inicializar el registro de segmento es para apuntar a la memoria de pantalla, como hemos hecho en tantos ejercicios de capítulos previos con las dos sentencias siguientes:

```
mov ax, 0B800h  
mov es, ax
```

Para introducir estas sentencias como ensamblador embebido en Visual C++ bastaría con preceder cada sentencia con la palabra clave `__asm`, pero en el caso de GCC se escribiría así:

```
asm("movw $0xB800, %ax; mov %ax, Ses; ");
```

Observe especialmente la primera sentencia, ya que en ella se reflejan varios de los detalles relativos a la sintaxis indicados anteriormente. La instrucción `mov` aparece como `movw`, indicando el sufijo `w` que va a operar sobre operandos de 16 bits. El destino de la operación es el registro `ax`, pero el primer operando es el dato a asignar. Fíjese también en los prefijos, en ocasiones acumulados. El valor `B8 00h` es una literal, por eso se precede con el símbolo `$`, y está expresado en hexadecimal como indica el prefijo `Ox`.

Desde el ensamblador embebido a compilar con GCC también es posible acceder a variables locales, así como a los parámetros recibidos por una función, si bien la sintaxis resulta un tanto peculiar. Los identificadores de dichas variables locales no pueden aparecer directamente en la cadena que contiene las sentencias en ensamblador, sino que debe indicarse la posición en que se utilizarán mediante parámetros sustituibles. Éstos se designan como `%0`, `%1` y así sucesivamente en el interior de la cadena, facilitándose los parámetros reales fuera de ésta.

Imagine que quiere implementar la función máximo del punto previo, utilizando para ello la sintaxis para ensamblador embebido de GCC. El código sería el mostrado a continuación:

```
int máximo(int a, int b)  
{  
    asm ("moví %0, %%eax;"  
         "cmpl %1, %%eax;"  
         "jge fin;"  
         "moví %1, H M X ;"  
         "fin:" : : "g" (a), "g" (b) );  
}
```

Ésta es la denominada sintaxis extendida para ensamblador embebido en GCC, una sintaxis en la que, aparte de la cadena con las sentencias a ejecutar, se entregan varios argumentos más separados por el símbolo `:`.

Fíjese que al utilizar esta sintaxis los registros aparecen con un doble `%` como prefijo, distinguiéndolos así de los parámetros sustituibles.

Tras el primer carácter `:` ha de facilitarse la lista de parámetros de salida, es decir, los identificadores en los que se almacenarán resultados generados por las sentencias escritas en ensamblador. En este ejemplo dicha lista está vacía, de ahí que aparezca de manera inmediata el segundo carácter `:` que da paso a la lista de parámetros de entrada.

En ésta se enumeran los identificadores de las variables locales que se emparejarían con los parámetros sustituibles. Cada variable, dispuesta entre paréntesis, va precedida de un indicador de restricción que comunica a GCC los registros del microprocesador que puede utilizar para almacenar temporalmente esos parámetros. El indicador "g" no impone restricciones, permitiendo el uso de cualquier registro o valor inmediato.

Puede encontrar todos los detalles relativos a la sintaxis extendida del ensamblador embebido en GCC en el documento *GCC Inline Assembly HOWTO*, que está alojado en <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>.

Salvo por las diferencias en la sintaxis, descritas básicamente en este punto para el caso concreto de GCC, el uso de ensamblador embebido es una posibilidad que suelen ofrecer todos los compiladores de C/C++, así como los de muchos otros lenguajes de programación.

Procedimientos externos en ensamblador

Una alternativa a la inclusión del código ensamblador en el interior de los métodos C/C++, mediante el ensamblador embebido que ya conoce, consiste en escribir módulos exclusivamente en lenguaje ensamblador que, con posterioridad, serán enlazados con los que genere el compilador de C/C++.

Habíralo en mente los primeros contendrán procedimientos o funciones a invocar desde los segundos, es decir, escribiremos funciones en lenguaje ensamblador preparadas para que sean llamadas desde lenguaje C/C++.

Es necesario tomar en consideración, por tanto, la convención de llamada que emplean C y C++, es decir, la forma en que los compiladores de estos lenguajes facilitarán la lista de parámetros y esperarán el valor de retorno. El código que generan esos compiladores siempre da los pasos siguientes:

- Los parámetros facilitados al invocar desde C/C++ a una función se introducen en la pila de derecha a izquierda, de forma que el último parámetro ocupará la posición más profunda en la pila.
- A continuación se introduce también en la pila la dirección de retorno y se transfiere el control a la función.
- Una vez que finalice su tarea, la función debe devolver el control sin alterar el contenido de la pila y, en general, ninguno de los registros de propósito general. En caso de que deba devolver un resultado, lo habitual es que lo haga en el acumulador.

Nuestras funciones escritas en ensamblador deberán, en consecuencia, acceder a la pila para recuperar los parámetros de entrada y, usualmente, devolver el resultado que generen en el registro AL, AX o EAX, dependiendo del tamaño.

Prólogo y epílogo

Las funciones escritas en C y C++ siempre generan un código de inicialización, cuya finalidad es facilitar el acceso a los parámetros recibidos, y otro de finalización que devuelve los registros a su estado original. Esas dos porciones de código suelen denominarse *prólogo* y *epílogo* por razones obvias.

El código del prólogo, asumiendo que estamos en un entorno de 32 bits, siempre es similar al mostrado a continuación:

```
push ebp  
mov ebp, esp  
; push registros a usar
```

Puesto que los parámetros se encuentran en la pila, la dirección base es la dada por el puntero de pila: ESP.

Una forma fácil de acceder a los mismos es por medio de direccionamiento indexado a través del registro EBP, por eso el valor actual de éste se guarda en la pila y, a continuación, se le asigna el contenido de ESP. Además habrá que agregar sentencias push adicionales para preservar el contenido de los registros que vayan a ser modificados en el cuerpo de la función.

En cuanto al código del epílogo, su función es la de devolver los registros y la pila al estado en que se encontraban inicialmente, estando compuesto de un bloque de sentencias como el siguiente:

```
; pop registros usados  
mov esp, ebp  
pop ebp  
ret
```

La asignación de EBP a ESP normalmente es necesaria solamente si la propia función ha utilizado espacio en la pila para almacenar sus propios datos locales, alterando así el contenido de ESP.

Acceso a los parámetros de entrada y devolución de resultados

El acceso a los parámetros de entrada, en el cuerpo de la función, resulta especialmente sencillo una vez se haya ejecutado el código de prólogo. No hay más que recurrir al direccionamiento indexado empleando el registro EBP como base. Obviamente resulta imprescindible conocer cuántos parámetros se han entregado, cuál es su orden y su tamaño.

Imaginemos que la función recibe como argumento un valor entero de 32 bits. Al efectuar la llamada se colocará en la pila ese único valor y a continuación la dirección de retorno, también de 32 bits.

Al entrar en la función se almacena en la pila el valor actual de EBP, una vez más de 32 bits. Esto significa que en la pila existen en este momento, delante del parámetro que es lo que nos interesa, un total de 8 bits. La forma de recuperar el valor, por tanto, sería la siguiente:

```
raov eax, [ebp+8]
```

Si la función recibiese más de un parámetro, esta instrucción obtendría el contenido del último. Sumando su tamaño se tendría acceso al anterior y así sucesivamente. En sintaxis AT&T la recuperación del parámetro se escribiría así:

```
moví 8(%ebp), %eax
```

En cuanto a la devolución de resultados desde la función, como se indicó anteriormente bastará con asignar el valor a devolver al acumulador del tamaño que corresponda.

Compilación, ensamblado y enlace

Una vez escrita la función en ensamblador, así como el código en C/C++ desde la que va a ser invocada, solamente resta proceder a la compilación, ensamblado y enlazado para obtener el ejecutable final.

El procedimiento habitual sería:

- Ensamblado del módulo escrito en ensamblador, obteniendo un módulo con el código objeto pero sin enlazar.
- Compilación del módulo escrito en C/C++, generándose igualmente como resultado un módulo con el código objeto sin enlazar.
- Enlazados de los dos (o más) módulos objeto para obtener el ejecutable final.

Veamos en la práctica cómo realizaríamos todo el proceso reproduciendo el sencillo ejercicio de puntos previos, consistente en codificar en ensamblador una función que devuelva el máximo de dos números enteros dados.

Comenzaremos creando el módulo con el código en ensamblador, cuyo contenido sería el mostrado a continuación:

```
segment Código
; máximo será el nombre de la función
global _raaxirao

_maximo:
    push ebp          ; Prólogo
    mov ebp, esp
```

```

; Comparación de los parámetros recibidos
mov eax, [ebp+8]
cmp eax, [ebp+12]
jge fin

    mov eax, [ebp+12]
fin:

pop ebp           ; Epílogo
ret

```

El único elemento nuevo que hay en este código, no descrito en los capítulos previos, es la palabra clave `global` mediante la que se indica al ensamblador los símbolos del programa que serán accesibles desde el exterior. En este caso ese símbolo es la etiqueta `máximo` que marca el punto de entrada a la función. En realidad el nombre de ésta será `máximo` para el programa escrito en C/C++, el carácter de subrayado inicial lo añade el compilador.

A continuación preparamos el programa C++ que hará uso de la función, realmente simple como puede verse en el código siguiente:

```

#include <iostream>
using namespace std;

// Declaración de la función externa
extern "C" int máximo(int a, int b);

int main(int argc, char* argv[])
{
    int a = 5, b = 10;

    cout << "El máximo es: " << máximo(a, b) << endl;
    return 0;
}

```

Lo único que denota que la función `máximo` no está implementada en el propio programa (aparte de la ausencia de su código, claro está) es la declaración `extern`. Ésta le permite al compilador C++ saber cuál es el identificador de la función, su lista de parámetros de entrada y tipos y el tipo del valor de retorno.

Con esta información podrá generar el código objeto adecuado para invocarla, sin necesidad de más detalles.

Llega el momento del ensamblado, compilado y enlazado. El módulo ensamblador se procesaría introduciendo en la línea de comandos:

```
nasm -f coff -o máximo.obj máximo.asm
```

En este caso la opción `-f coff` implica que el código será de 32 bits para una plataforma Windows, debiendo cambiarse por el adecuado para otros sistemas.

Si para compilar el código C++ va a emplearse el compilador de GNU, podría utilizarse el comando siguiente:

```
g++ -o fexterna fexterna.cpp máximo.obj
```

De esta forma se compilaría el programa y el código resultante se enlazaría con el código objeto obtenido con el ensamblado previo.

Utilizando Visual C++ bastará con agregar el módulo maximo.obj generado por el ensamblador a la solución que contiene el módulo C++, tal como se puede apreciar en la figura 31.1.

La compilación del proyecto se encargará de enlazar adecuadamente los códigos objeto para obtener el ejecutable.

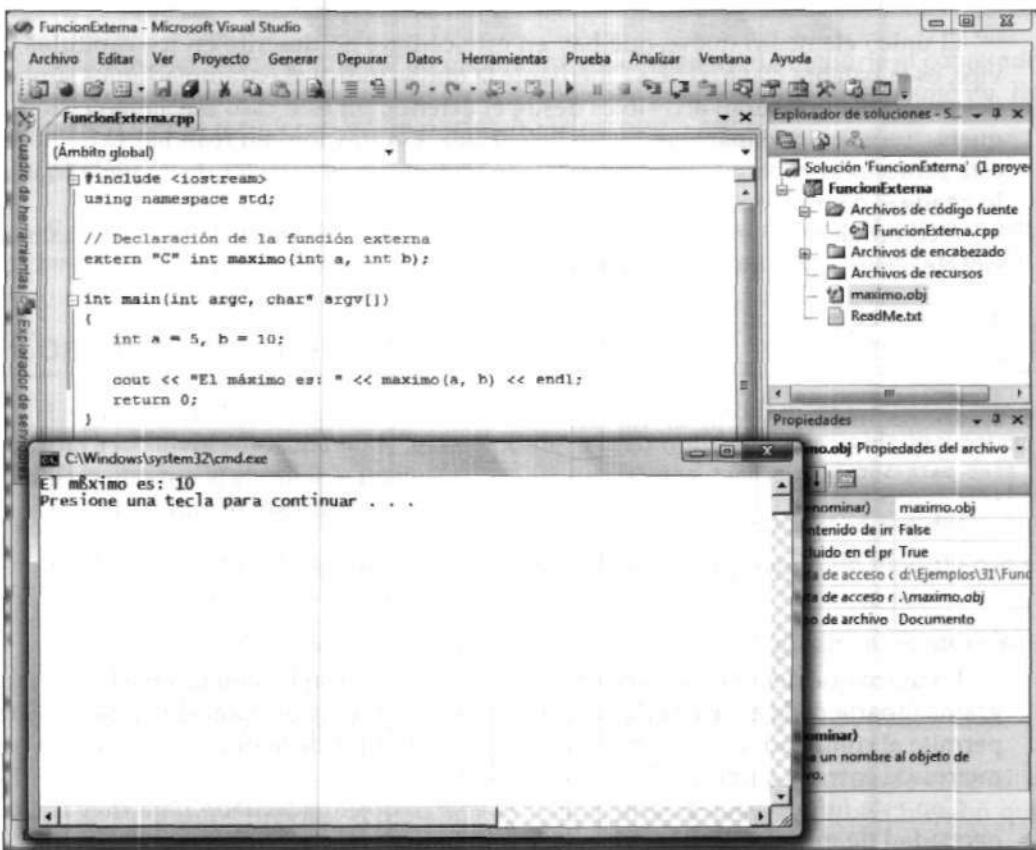


Figura 31.1. Agregamos el código objeto generado por NASM a un proyecto Visual C++.

Un método alternativo al que acaba de describirse, que también es perfectamente factible, consistiría en agregar al proyecto el módulo de código fuente ensamblador, configurando en Visual C++ las reglas necesarias para invocar a NASM y obtener el código objeto.

Resumen

Como ha tenido ocasión de comprobar en este capítulo, el uso conjunto de los lenguajes ensamblador y C/C++ a la hora de escribir un programa no conlleva dificultad alguna, lo cual permite utilizar el lenguaje de alto nivel para las tareas que no requieren ni la velocidad ni la concisión del lenguaje de bajo nivel, como puede ser, por ejemplo, la entrada/salida por consola o archivos.

La posibilidad de usar ensamblador embebido resulta, además, una forma sencilla de introducir sentencias en ensamblador directamente en el código del lenguaje de alto nivel, lo cual abre las puertas a operaciones avanzadas que dicho lenguaje podría no contemplar.

Finalmente, en este capítulo también ha conocido los aspectos más destacables de la sintaxis AT&T para ensamblador x86, una sintaxis que no solamente se aplica al uso conjunto del compilador de C/C++ de GNU, sino que es también la utilizar por algunos ensambladores.

32

Recursos de interés

Tras finalizar la lectura de este libro posiblemente se vea capacitado para crear aplicaciones utilizando el lenguaje ensamblador para cualquiera de los sistemas operativos tratados: DOS, Windows, Linux y DOS 32 bits. Al tiempo, sin embargo, también le abordarán multitud de interrogantes y una necesidad de información de referencia que le permita hacer uso de los recursos propios de cada sistema, la sintaxis del ensamblador que vaya a usar, etc.

Una simple búsqueda en Google, o bien cualquier otra herramienta similar, le facilitará miles de páginas con información relacionada con la programación en lenguaje ensamblador.

Lo que resulta difícil, en ciertas ocasiones, es separar lo útil de aquello que no lo es, evitando así perder una gran cantidad de tiempo hasta encontrar lo que realmente se necesita.

Por eso en este capítulo se facilitan diversas direcciones que pueden ser útiles al lector, ahorrándole el tiempo de búsqueda en la medida de lo posible.

Las direcciones indicadas a continuación están disponibles en el momento de finalizar la redacción de este libro, pero podría suceder que cuando el lector intente acceder a ellas hayan desaparecido o cambiado a otra dirección, por lo que no se ofrece ninguna garantía al respecto.

Tal y como podrá comprobar, prácticamente todos los recursos se encuentran en inglés puesto que la información disponible en nuestro idioma es escasa y, en ocasiones, desfasada.

NASM Manual

<http://home.comcast.net/~fbkotler/nasmdoc0.html>

Se trata de un manual en línea del ensamblador NASM. La dirección indicada da acceso al índice de contenidos, desde el cual podrá acceder a toda la información sobre esta herramienta, desde la instalación hasta el conjunto de instrucciones que pueden utilizarse.

MASM Programmer's Referente

http://webster.es.ucr.edu/Page_TechDocs/MA.SMDoc/

Como su propio nombre indica, se trata de un manual de referencia sobre MASM, el otro ensamblador usado en este libro. En él se describe la sintaxis, directivas, macros predefinidas, etc. También se explican procesos como la creación de bibliotecas de enlace dinámico para Windows o la programación de aplicaciones residentes.

Ralf Brown's Interrupt List - HTML Versión

<http://www.ctyme.com/rbrown.htm>

Independiente de que programemos para DOS, Windows o DOS en 32 bits, un recurso indispensable en la lista de interrupciones recompilada por Ralf Brown. En la dirección indicada se encuentra en formato HTML, pudiendo buscarse una cierta interrupción directamente por su código o bien por categoría. Se trata, sin duda, de la referencia más completa sobre interrupciones que hay disponible.

Linux 2.xx Syscalls

<http://www.lxhp.in-berlin.de/htmlfiles.tar.bz2>

Un recurso similar al anterior pero, en este caso, dedicado a las funciones o llamadas de sistema de Linux a través de la interrupción 8 Oh. Puede accederse a ellas mediante índices diversos o bien por su código si es que lo conocemos. Facilita enlaces directos a las páginas de manual de Linux, por lo que si la consulta desde este sistema podrá ir directamente a la documentación relativa a la función en la que esté interesado.

Art of Assembly Language Programming and HLA by Randall Hyde

<http://webster.es.ucr.edu/index.html>

Extensísima Web con información sobre programación en DOS, Windows y Linux con ensamblador, usando HLA (*High Level Assembler*), un ensamblador propio creado por el autor del sitio y que facilita construcciones de alto nivel y, por tanto, hacen más fácil la creación de aplicaciones en ensamblador. Además facilita enlaces a documentación sobre múltiples procesadores, formato de archivos y documentación de herramientas.

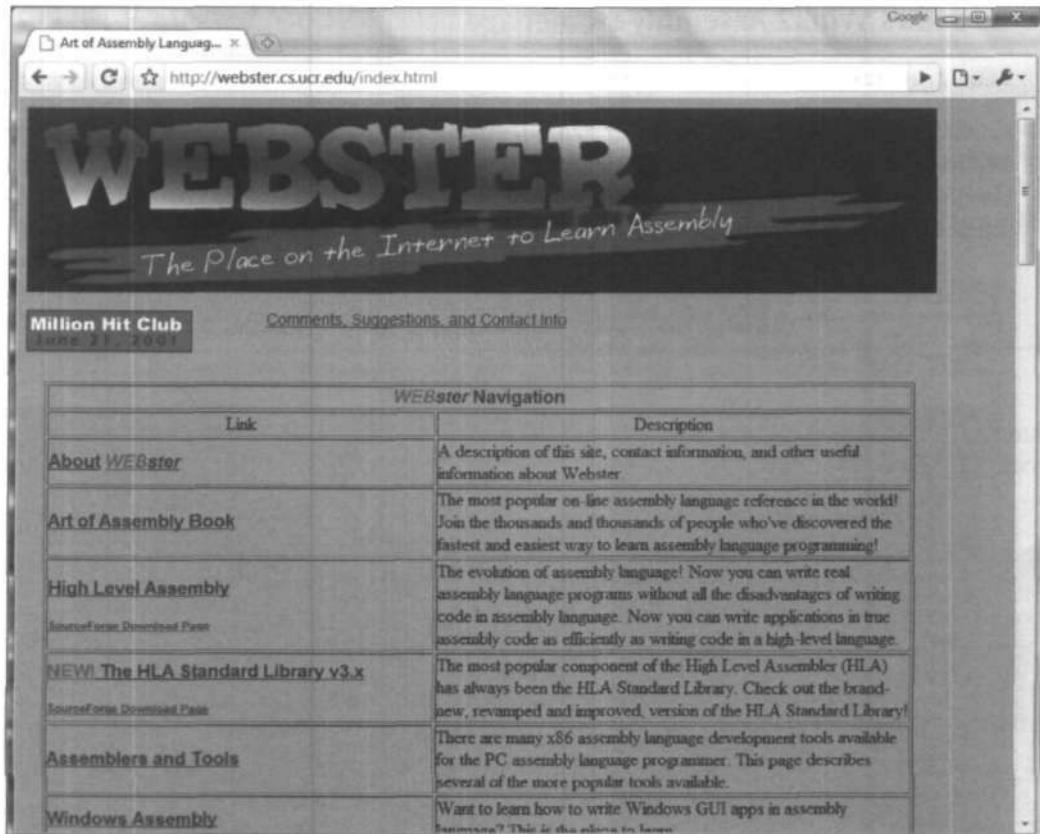


Figura 32.1. La sede de *Art of Assembly Language Programming and HLA* by Randall Hyde.

Assembly Language Journal

<http://asmjournalfr.freeservers.com/>

Se trata de una publicación periódica en la que se recogen artículos sobre programación con MASM, NASM, TASM, sobre DOS, Linux y Windows, trucos, etc. Aunque no aparece ningún número posterior a septiembre de 2001, los nueve números disponibles hasta el momento contienen información que puede resultarle de interés.

Linux Assembly

<http://linuxassembly.org/>

Una Web dedicada específicamente a la programación en ensamblador sobre Linux. Ofrece diversos documentos y guías, así como enlaces a múltiples recursos. También puede obtenerse una biblioteca prefabricada que facilita el acceso a los servicios más habituales de Linux.

Iczelion's Win32 Assembly Homepage

<http://win32assembly.onli.ne.fr/tutorials.html>

Guía completísima sobre la programación de aplicaciones Windows usando ensamblador, abarcando desde la aplicación más simple hasta la programación de controladores de dispositivos. Puede leerse directamente u obtenerse un paquete con todas las guías. En el apartado de traducción puede encontrarse parte del material en nuestro idioma.

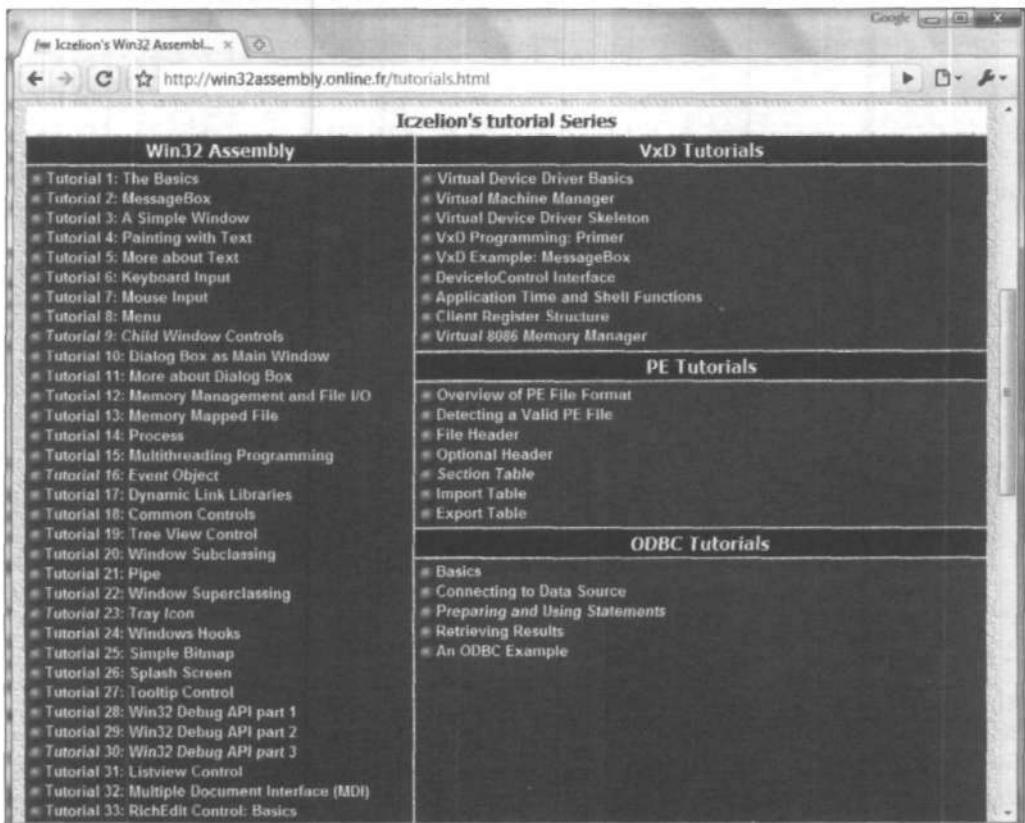


Figura 32.2. Sede de Iczelion's Win32 Assembly Homepage.

Win32NASM

<http://rsl.szif.hu/~tomcat/win32/>

Conjunto de enlaces a paquetes de herramientas que facilitan la creación de aplicaciones para Windows con NASM en lugar de MASM. En esos paquetes pueden encontrarse archivos con prototipos, bibliotecas de información y otros recursos útiles si nos interesa programar para Windows y queremos utilizar NASM.

OSRC: Protected Mode

<http://www.nondot.org/^sabre/oa/articles/ProtectedMode/>

Colección de enlaces a guías didácticas, ejemplos y documentación de referencia sobre temas relacionados con el modo protegido y su programación, ya sea directamente o mediante DPMI.

También puede encontrar direcciones de ejemplos y herramientas, como el extensoí de dos *PMode*.

Programmers Heaven - Assembler programming zone

<http://www.programmersheaven.com/tags/Assembly/>

Multitud de enlaces a recursos relacionados con el lenguaje ensamblador, desde manuales, herramientas de desarrollo y artículos hasta código fuente de ejemplo, clasificados por procesadores, sistemas operativos y temas concretos.

The screenshot shows a web browser window with the URL <http://www.programmersheaven.com/tags/Assembly/Articles>. The page title is "Assembly Articles". The main content area displays a list of articles under the "Assembly" category. The first article listed is "Ease Versioning Multiple Assemblies by Splitting Up AssemblyInfo". Other visible articles include "Call Functions with PowerPC ABI for 64-bit ELF", "Fundamentals of POWER5 Assembly Language", "Webpart Post Build .Net Automation Of Global Assembly Cache & IISRESTART", and "Satellite Assemblies". To the right of the main content, there is a sidebar with various links and advertisements related to programming, such as ".NET Architecture", "C to FPGA Compiler", "Free code examples & book", and "Programming Books".

Figura 32.3. Recursos en *Programmers Heaven*.

80x86 Instruction Set

<http://www.penguin.cz/~literakl/intel/intel.html>

Como su propio nombre indica, esta Web recoge el conjunto de instrucciones de los procesadores de Intel, concretamente hasta el 80486, facilitando información sobre cada instrucción en particular.

El universo digital del IBM PC, AT y PS/2

<http://ate.ugr.es/docencia/udigital/index.html>

Una de las pocas Web en castellano que merecen mención. Se trata, en realidad, de un completo libro que abarca la programación en ensamblador de la familia de procesadores x86, introduciendo el conjunto de instrucciones y describiendo su uso en el hardware de los PC sobre el sistema operativo DOS. Trata temas tan interesantes como la gestión de memoria, creación de programas residentes y la creación de controladores de dispositivos.

A

Contenido del CD-ROM

Este libro incorpora un CD-ROM en el que podrá encontrar todos los ejemplos que se han descrito en sus capítulos, tanto en versión fuente como ensamblada, facilitándole así el acceso a todo el código sin necesidad de tener que escribirlo personalmente. En la carpeta Ejemplos encontrará una subcarpeta por capítulo.

Tenga en cuenta que muchos de los programas son meramente ejemplos que, a simple vista, no hacen nada aparente, pero al leer los capítulos correspondientes encontrará el sentido de cada uno de ellos. En cualquier caso, debe tener en cuenta que estos programas se entregan como pruebas y nunca como aplicaciones de uso final, por lo que no se otorga garantía ni soporte alguno de su funcionamiento. Nuestro objetivo es facilitarle código sobre el que poder trabajar y crear sus propios proyectos.

En el CD-ROM también encontrará ensambladores, enlazadores, editores, depuradores y otras herramientas que le facilitarán el trabajo. En la carpeta Herramientas encontrará una subcarpeta por sistema operativo, conteniendo cada una de ellas las herramientas que correspondan a dicho sistema. Dichos productos se entregan al lector con el consentimiento, expreso o implícito según los casos, de la empresa o persona que los ha desarrollado, quedando tanto *Anuya Multimedia* como el autor de este libro exentos de toda responsabilidad sobre los perjuicios que su uso pudieran causar, al tiempo que no están obligados a ofrecer soporte algunos de esos productos.

Índice alfabético

- Símbolos 80486, 78
_asm, 770
- 33h, 426
68K, 33
4004, 30
6502, 33
8008, 32
8080, 32
8085, 33
8086, 33
8087 78
8237 94
8253, 86
8255, 89
8259, 92
8259A, 77
8284, 83
8288, 76
80286, 78
80287, 78
80386, 78
80387, 78
- A
- AAA, 110
AAD, m
AAM > HO
AAS, n0
- acumulador, 70
Acumulador, 38
ADQ110
^ADD, 110
ADnn, 75
AF, 72
ALE, 75
Altair 8800, 32
ALU, 38
ancho de palabra, 39
AND, 111
arquitectura Harvard, 32

arquitectura Von Neumann, 36
asm, 770
Atari ST, 33
AX, 69

D

BCD, 60, 217
big endian, 58
binario, 47
BIOS
 1Ah, 332
 1Oh, 321
 1lh, 323
 13h, 328
 14h, 331
 16h, 322
 17h, 331

bit, 47
BIU / 69
BP 71
BU, 68
bus de datos, 74
BX, 69

C

CALL, 113
CBW112
CF, 72
CGA, 341
Chip-Select, 41
circuito integrado, 30
CISC, 34
CLC, 116
CLD, 116
CLI, 116
CMC, 116
CMP, 111
CMPS, 112
coma fija, 60
coma flotante, 60
Commodore Amiga, 33

Commodore PET, 33
complemento a dos, 52
Contador de programa, 39

Core, 79
CPU, 36
CS, 41
CU, 67
CWD _ 112
CX, 69

D

DAA, 110
DAS, 110
Datos
 db, 178
 dd, 178
 dup, 178
 d w , 178
DWORD, 166
QVWORD, 166
 r e s b , 178
 r e s d , 178
 r e s w - 178
times, 178
WORD, 166

DEBUG
 a, 200
 d 192
 ' 197

g, 196
J 187
n j g 7
p ^94
r ; i 93
t 195
ú, 188

DEC, 110
Depuradores
 DEBUC, 132, 185
 g d b / 132 , 186
 GRDB, 185
 GRDBBL09, 132
descodificador 3 a 8, 82

DF, 73
dígito binario, 51
DIP, 74
Direcciones
 seg, 179
DIV, 110
DMA, 94
DPMI, 757

E

EAX, 69
Editores
 ASMCrunch, 120
 Assembler Editor, 121
 Bloc de notas, 120
 EDIT, 120
 emacs, 124
 SetEdit, 125
 vi, 120
 xemacs, 124
EGA, 342
EIP, 71
Enlazadores
 ALINK, 131
 LINK, 131
 TLINK, 131
Ensambladores
 A86, 130
 as, 126
 gas, 130
 MASM, 121
 NASM, 121
 TASM 121
ES 72 '
p 71

Formatos
 a.out, 155
 COFF, 155
 COM, 138, 154
 ELF, 155

EXE, 138
MZ, 154
NE, 155
PE, 155
FS, 71
Fundamentos
 AH, 139
 Complemento a dos, 52
DX, 140
end, 140
ends, 140
int, 139
interrupciones, 139
jmp, 139
modo protegido, 171
modo real, 171
OBJ, 141
org, 139
pop, 147
ret, 147

^

GDT 744
G p u ; 3 6
GS 71

H

Harvard, 32
Herramientas
 ASMEedit, 120
 NasmEdit, 122
 NASM-IDE, 121
 VASM, 123
 VisualASM, 124
hexadecimal, 47
HLT, 116

I

IBM PC, 33
IC, 30

IDIV, 110	fbstp, 229
IDT, 756	fchs, 228
IEEE-754, 60	feos, 228
IF, 73	fdiv, 228
IMUL, 110	fild, 228
IN, 115	fist, 229
INC, 110	fistp, 229
include, 149	fld, 227
includelib, 149	fldl, 228
Indicadores	fldpi, 228
carry, 208	fldz, 228
INS, 115	fmul, 228
Instrucciones	frndint, 228
aaa, 219	fSCALE, 228
aad, 222	fsqrt, 228
aam, 222	fst, 229
aas, 222	fstp, 229
ade, 209	fsub, 228
add, 206	idiv, 214
and, 244	imul, 214
bt, 246	in, 279
btc, 246	inc, 216
btr, 246	iret, 315
bts, 246	ja, 241
cali, 270	jae, 241
cbw, 225	jb, 241
ele, 235	jbe, 241
cid, 234, 264	je, 246
di, 234	jcxz, 256
eme, 235	je, 238
emp, 238	jg, 241
emps, 304	jge, 241
empsb, 305	jl, 241
empsd, 305	jle, 241
empsq, 305	jmp, 238
empsw, 305	jna, 241
cwd, 225	jnc, 246
daa, 219	jne, 238
das, 222	jnz, 238
dec, 216	jz, 238
div, 214	lds, 320
fabs, 228	les, 320
fadd, 228	lgdt, 745
fbld, 228	lidt, 756

lod, 294
lodsb, 294
lodsd, 294
lodsq, 295
lodsw, 294
loop, 257
loope, 257
loopne, 257
Iss, 606
mov,¹⁷ 3
movsb, 301
movsd, 301
movsq, 301
movsw, 301
mul,²¹²
ne[§],²²⁴
or,²⁴³
out,²⁷⁹
pop, 235
popa, 278
P_oP/¹²f_c
P^{US}I[·]*L
P^{US}A[']1⁷«
pU5hf,²³⁵
rcl,²⁴⁸
rct,²⁴^{*}*o*
repe,³⁰²
~n,,
repne, 302
reti, 272
retn, 272
rol, 248
ror, 248
sbb,²¹²
seas, 302
scasb, 302
scasd, 302
scasq, 302
scasw, 302
shl, 248
shr, 248
ste, 235
std, 234
sti, 234
stos, 299
stosb, 299
stosw, 299
sub,²¹²
test, 246
xchg, 309
xor, 245
INT, 113
i_{nt} 2Fh
168Bh, 669
1681h, 672
1682h, 672
1683h/ 669
1687h/ 758
4300h; 558
4310h, 559
int1Oh
0Ah, 352
OBh, 364
0Ch, 361
QDK 361
0Fh, 346
OOh, 348
1A00h, 342
Olh 352
muZi
02h, 352
03h, 350
4F02h, 348
'
05h, 356
06h, 359
07h, 359
08h, 353
09h, 352
1000h, 366
1001h, 366
1002h, 366
1010h, 371
1012h < 371
1015h, 371
1017h, 371
int 13h
Olh, 537
02h, 536
03h, 537

int15h	08h,471
88h, 464	09h, 472
int16h	19h, 493
OOh, 384	25h, 474
Olh, 384	31h, 476
02h, 399	33h, 632
lOh, 398	34h, 610
11h, 398	35h, 474
12h, 399	41h,493
int17h	42h, 488
OOh, 416	43h, 493
Olh, 416	47h,495
02h, 416	48h, 477
int21h	49h,477
OAh, 408	50h, 606
OBh, 472	51h, 606
OCh, 472	56h, 493
OEh, 493	59h, 627
1Ah, 496	62h, 553
Olh, 471	7305h, 536
2Ah, 473	int 33h, 426
2Bh, 473	15h, 636
2Ch, 473	16h, 636
2Dh, 473	17h, 636
2Fh, 624	int 80h
02h, 471	Olh, 711
3Bh, 495	03h, 712
3Ch, 486	04h, 713
3Dh, 488	05h, 716
3Eh, 488	06h, 719
03h, 473	08h, 718
4Ah, 477	13h, 719
4Bh, 474	INTA, 77
4Ch, 476	Intel 4004, 30
4Dh, 476	Intel 8008, 32
4Eh, 496	Intel 8080, 32
4Fh, 496	Intel 8085, 33
04h, 473	Intel 8086, 33
5Ah, 487	interrupciones, 77
5Bh, 487	Interrupciones
05h, 417	lBh, 631
06h, 471	ICh, 618
07h,471	2Fh,470,557

- 09h, 620
13h, 536
15h, 463
17h, 415
20h, 470
23h, 634
24h, 625
25h, 470
26h, 470
31h, 763
- y
- JA, 114
JAE, 114
JB, 114
JBE, 114
JC, 114
JE, 114
JG, 114
JGE, 114
JL, 114
JLE, 114
JMP, 113
JNA, 114
JNAE, 114
JNB, 114
JNBE, 114
JNC, 114
JNE, 114
JNG, 115
JNGE, 115
JNL, 115
JNLE, 115
JNO, 115
JNP, 115
JNS, 115
JNZ, 115
JO, 115
JP, 115
JPE, 115
JPO, 115
JS, 115
JZ, 115
- L
- LAHF, 116
LDS, 113
LDT, 745
LEA, 113
LES, 113
Ley de Moore, 31
Linux
 cióse, 719
 creat, 718
 exit, 711
 gcc, 148
 lseek, 719
 open, 716
 read, 712
 write, 713
little endian, 58
LOCK, 116
LODS, 112
lógica de selección, 82
LOOP, 113
LSB, 58
- M
- mainframes, 29
mantisa (coma flotante), 61
Mark-8, 32
MASM
 /AT, 140
 /c, 141
 /coff, 150
 invoke, 681
 Proto, 681
 ptr, 176
MCGA, 342
MDA, 342
microcontrolador, 36
MMU, 78
MN/MX, 76
modo máximo, 76
modo mínimo, 76
modo protegido, 78

modo real, 78, 736
modo virtual 8086, 78
MOS T 6502, 33
Motorola 68K, 33
MOUSE.COM, 425
MOUSE.SYS, 425
MOV, 113
MOVS, 112
MSB, 58
MSW, 59

N

NASM
 $d > 309$
 $^{**} 45$
NEG, 110
Nehalem, 35
nibble, 51
NMI, 77
NOP, 116

O

OF, 72
OR, 111
OUT, 115
OUTS, 115

P

PC, 71
Pentium, 78
PF, 72
PIC, 92
POP, 116
POPF, 117
PowerPC, 34
PPI, 42
Preprocesador
 %define, 286, 309
 %else, 308
 %endif, 308

%endmacro, 288
%ifdef, 308
%include, 290
%macro, 288
PSP, 478
PTI, 85
puntero de instrucción, 71
Puntero de pila, 39
PUSH, 117

R

RAX, 69
R C L m
RCR, 111
registro de estado, 71
Registro de estado, 39
Registros, 39

 AH 172
 AL'172
 AX; 172
 BH, 172
 BL, 172
 BP, 161
 BX, 162,172
 CH, 172
 CL, 172
 CRO, 737
 CX, 172
 DH, 172
 DI, 162

 DL, 172
 DX, 162
 EAX, 172
 EBX, 172
 ECX, 172
 EDX, 172
 gdtr, 745
 IP, 161
 ldtr, 745
 lgdt, 756
 Registros de segmento, 168
 SI, 162

sp, 235
SP, 161
registros de segmento, 71
Registros de segmento
 CS, 160
 DS, 160
 ES, 162
 FS, 168
 GS, 168

Registros FPU
 ST, 226

relés, 29

Reloj del sistema
 70h, 279
 71h, 279

REP 112

RET 113

Rjp 7i

RISC, 34

S

 50, 76
 51, 76
 52, 76
 SAHF, 117
 SAL, 111
 SAR, 111
 SBB, 110
 SCAS, 112
 selectores de segmento, 741
 SF, 72
 SHL, 111
 SHR, 111
 SIMD, 69
 sintaxis AT&T, 770
 SP, 71
 SS, 72
 SSE, 78
 STC, 116
 STD, 116
 STI, 116

STOS, 112
SUB, 110
sufijos (base de numeración), 57
SVGA, 342
SXGA, 342

T

TEST, 111

transistores, 29

TRS-80, 33

U

 Unidad aritmético lógica, 38
 Unidad de control, 37
 Unidades
 bit < 165
 byte, 165
 doble palabra, 166
 gigabyte, 166
 kilobyte, 166
 megabyte, 166
 palabra, 166
 párrafo, 169
 segmento, 169

user32

 BeginPaint, 703
 CréateWindowEx, 685
 DefWindowProc, 689
 DispatchMessage, 688
 EndPaint, 703
 GetMessage, 688
 LineTo, 704
 MoveToEx, 704
 PostQuitMessage, 692
 RegisterClassEx, 682
 SendMessage, 695
 SetWindowText, 696
 ShowWindow, 685
 TextOut, 704
 TranslateMessage, 702

V

válvulas de vacío, 29
VCPI, 757
VESA, 342
VGA, 342
Von Neumann, 36

W

WAIT, 117

Windows

 ExitProcess, 149
 MessageBox, 149

word, 176

X

XCHG, 113
XLAT, 112
XMS, 757
 OAh, 566
 OBh, 565
 OOh, 559
 08h, 563
 09h, 562

XOR, U1

Z8Ü, 33
 PC, 167
 SP, 167
ZF, 72
Zilog Z80, 33