Programación Avanzada en Lenguaje Ensamblador

Ramón Medina

Dedicatoria

A mis cuatro más grandes amores: DIOS, Marilys, Luis Alejandro y Cristian Adrián.

Prefacio

Los programadores se pasan la vida argumentando acerca de cual es el mejor lenguaje. Si se le pregunta a un programador de C acerca del PASCAL y dirá que este nunca será lo suficientemente flexible. Si a un partidario del PASCAL se le comenta acerca de la facilidad del BASIC, seguramente traerá a colación los GOTO. Y si a un fanático del FORTH, se le dice que el suyo es un lenguaje raro y obscuro, poco adecuado para trabajos serios, probablemente sea la última vez que le dirija la palabra.

Supongamos, que en lugar de programadores, tenemos a un grupo de chef de cocina, discutiendo acerca de si las recetas deben ser escritas en francés, inglés o español. Por supuesto que resulta tonto. El sabor de la comida será el mismo independientemente del idioma en que haya sido escrita la receta.

Lo mismo ocurre en programación. Todos los lenguajes de alto nivel, debe convertir sus instrucciones en código de máquina para poder ser ejecutado por el microprocesador instalado en la computadora. Como podemos ver, todos los lenguajes de programación hablan el mismo idioma.

Por todo esto, tiene sentido aprender a programar en lenguaje ensamblador, sea cual sea el lenguaje de alto nivel de su preferencia, ya que es el único lenguaje de programación que le permite hablar en el idioma nativo del microprocesador y explotar todo su potencial.

En los próximos capítulos trataremos de explicar desde los conceptos básicos hasta tópicos avanzados de la programación en lenguaje ensamblador, desde los sistemas de numeración y arquitectura interna de un computador, hasta librerías de aplicaciones y programas residentes.

Para los principiantes, añado lo siguiente: Si alguna vez ha oído que el lenguaje ensamblador es difícil, no lo crea. Con las características y herramientas de programación disponibles hoy en día, en poco tiempo se sorprenderá desarrollando aplicaciones en este lenguaje con toda libertad.

Contenido

Conceptos Básicos

Los sistemas basados en microprocesadores están englobados dentro de lo que en electrónica se conoce como sistemas digitales. Estos actúan bajo el control de variables discretas, entendiéndose por estas, las variables que pueden tomar un número finito de valores. Por ser de fácil realización los componentes físicos con dos estados diferenciados, es éste el número de valores utilizado usualmente para dichas variables que, por lo tanto, son binarias.

Tanto si se utilizan en proceso de datos como en control industrial, los sistemas digitales han de efectuar operaciones con números discretos. Los números pueden representarse en varios sistemas de numeración, que se diferencian por su base. La base de un sistema de numeración es el número de símbolos distintos utilizados para la representación de las cantidades en el mismo. El sistema de numeración utilizado en la vida cotidiana es el de base diez, en el cual existen diez símbolos distintos, del 0 al 9.

Por la razón expuesta el sistema de numeración más utilizado en la realización de los sistemas digitales es el de base dos, o binario, en el cual existen solamente dos símbolos, que son el 0 y el 1.

Sistemas de Numeración.

En un sistema de base *b*, un número *N* cualquiera se puede representar mediante un polinomio de potencias de la base, multiplicadas por un símbolo perteneciente al sistema. En general tendremos:

$$N = a_n b^n + a_{n-1} b^{n-1} + ... + a_1 b^1 + a^0 b_0 + ... + a_{-p} b^{-p_i}$$

siendo *b* la base del sistema de numeración y a_i un número perteneciente al sistema y que, por lo tanto, cumple la condición $0 <= a_i <= b$ donde $n+1$ y p representan respectivamente el número de dígitos enteros y fraccionarios.

Si el sistema es de base diez o decimal, tendremos que $b=10$ y $0<=a_i<10$. Por ejemplo, el número 87,54 en base diez se representa por:

$$87,54 = 8*10^1 + 7*10^0 + 5*10^{-1} + 4*10^{-2}$$

De igual forma, en el sistema de base dos o binario, se tiene que $b=2$ y $0<=a_i<2$ y el número 1011,11 en este sistema se representa por el polinomio:

$$1011,11 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 1*2^{-1} + 1*2^{-2}$$

Para identificar el sistema al cual pertenece un número se suele indicar la base como subíndice. De esta forma los números mencionados como ejemplo se representarán:

$$87,54_{10};\ 1011,11_2$$

Cambios de base de numeración.

El cambio de la base de numeración consiste en la representación de un valor en otro sistema numérico, distinto del original. Existen algunos sistemas de numeración, entre los cuales la conversión es directa, por ser sus bases múltiplos y submúltiplos entre si lo cual crea una correspondencia como la que se muestra en la siguiente tabla de conversión entre números binarios y hexadecimales.

Tabla de conversión de binario a hexadecimal.

Sistema Binario	Sistema Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Sistema Binario	Sistema Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Tabla 1-1

Para convertir un número del sistema hexadecimal al binario se sustituye cada símbolo por su equivalente en binario, según la Tabla 1-1. Sea por ejemplo el número $9A7E_{16}$. El equivalente de cada símbolo es:

$9_{16} = 1001_2$
$A_{16} = 1010_2$
$7_{16} = 0111_2$
$E_{16} = 1110_2$

Por lo tanto resulta:

$$9A7E_{16} = 1001101001111110_2$$

La conversión de un número en sistema binario natural a hexadecimal se realiza a la inversa agrupando los dígitos enteros y fraccionarios en grupos de cuatro a partir de la coma decimal, y convirtiendo cada grupo independientemente. Para completar el último grupo se añaden los ceros que sean necesarios. Sea por ejemplo el número 100111,10101 en base dos. Añadiendo dos ceros a la izquierda y tres a la derecha resulta:

$0010_2 = 2_{16}$
$0111_2 = 7_{16}$
$1010_2 = A_{16}$
$1000_2 = 8_{16}$

Resulta por lo tanto:

$$100111{,}10101_2 = 27{,}A8_{16}$$

Sin embargo, en la gran mayoría de los casos, esta correspondencia no existe, para lo cual disponemos entonces de un método general que permite efectuar conversiones entre dos bases cualquiera.

La conversión de un número en un base *b* a decimal se realiza fácilmente representando el número mediante su polinomio equivalente y operando éste en base diez. Por ejemplo:

$$1101,11_2 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 1*2^{-2}$$
$$= 8 + 4 + 1 + 0,5 + 0,25$$
$$= 13,75_{10}$$

Por tanto:

$$1101,11_2 = 13,75_{10}$$

Para efectuar la conversión en sentido contrario, nos basamos en una simple regla que dice que si un número entero expresado en un sistema de base *b1* (que se desea convertir a una base *b2*) por la base *b2*, y el cociente se vuelve a dividir por *b2* y así sucesivamente, el último cociente y los restos obtenidos forman el número en el sistema de base *b2*. Por ejemplo, convirtamos el número 3524_{10} a hexadecimal.

	Cociente	Residuo
3524/16	220	4
220/16	13	12

D C 4

Luego:

$$3524_{10} = DC4_{16}$$

Para convertir un número fraccionario en la base *b1* a la base *b2*, multiplicamos el número por la base *b2*. La parte entera obtenida representa la cifra más significativa del número *N* en base *b2*. Si la parte fraccionaria se multiplica nuevamente por *b2* obtendremos la segunda cifra. Continuando el proceso se obtienen todas las cifras de *N* en base *b2*. Este proceso termina cuando la parte fraccionaria obtenida sea nula.

Aplicando el método expuesto convertiremos el número 0,625 en base diez a base dos.

0,625 * 2	=	**1**,25
0,25*2	=	**0**,5
0,5* 2	=	**1**

de donde resulta:

$$0,625_{10} = 0,101_2$$

Estructura de la Memoria del Computador.

Bit.

La memoria del computador se compone de unidades de almacenamiento llamadas bits, que tienen dos estados posibles (representados por 0 y 1), es decir, sirven para almacenar números expresados en binario. La palabra bit proviene de la contracción *binary digit* (dígito binario). Así pues, todo lo que reside en la memoria del computador (códigos de instrucción y datos) están expresados por números binarios, a razón de un dígito binario por bit.

Byte.

Los bits de la memoria se agrupan en bytes (u octetos), a razón de 8 bits por byte. Un byte es realmente la unidad de direccionamiento, es decir, podemos referirnos a cada byte mediante un número que es su dirección.

La cantidad de memoria de un computador se mide en Kilobytes (cuya abreviatura es Kbyte, Kb o simplemente K), siendo:

1K = 1024 bytes

Un byte puede almacenar números binarios de hasta ocho dígitos, lo cual corresponde a un rango de valores en decimal desde 0 hasta 255 inclusive.

Nibble.

La agrupación de los cuatro bits (superiores o inferiores) de un byte se llama nibble. Por lo tanto, un byte contiene dos nibbles. El que corresponde a los bits 0 al 3 se llama nibble inferior y el que corresponde a los bits 4 al 7 nibble superior.

Estructura de un byte de información.

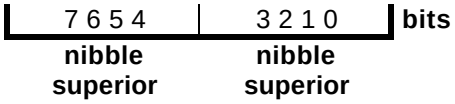


Figura 1-1

El *nibble* es una unidad de trabajo mucho más cómoda que el bit. En cada *nibble* se almacena un dígito hexadecimal.

Suma de números binarios.

Los números contenidos en bytes, expresados en modo binario, se suman de forma análoga que los de base decimal, es decir, de derecha a izquierda, arrastrando el acarreo a la columna inmediata de la izquierda. Por ejemplo:

a= 11010010b
b= 00010111b
a+b= 11101001b

Números negativos.

Hasta ahora hemos visto la representación de números sin signo. Teniendo en cuanta el carácter cíclico de los valores de un byte en operaciones de suma, podríamos, por convenio, definir el negativo de un número como aquel que sumado a dicho número nos da como resultado cero. Este número puede ser hallado mediante un artificio matemático llamado complemento a dos, el cual dice que el complemento a dos de *N*= *~N+1*. Por ejemplo, el complemento a dos de 4F es:

4Fh = 01001111b
~4F = 10110000b

~4F+1 = 10110001b = B1h

si efectuamos la operación 4F + B1 tenemos:

```
4F      =       01001111b
B1      =       10110001b

4F+B1   =       00000000b       =       00h
```

lo cual corrobora que B1 es el complemento a dos de 4F, lo que quiere decir es que de acuerdo con la convención anterior, B1 = -4F.

Numeración BCD.

BCD es un acronismo para *Binary Coded Digit* , el cual es una representación de los número decimales en la que cada dígito ocupa cuatro bits. Existen dos variaciones principales de los números BCD:

BCD empaquetado.

Los números son almacenados a razón de dos dígitos por byte, donde los dígitos individuales están en orden de mayor a menor, aún cuando los bytes estén organizados de menor a mayor.

BCD desempaquetados.

Los números son almacenados a razón de un byte por dígito.

Caracteres ASCII.

En los sistemas de numeración estudiados en apartados anteriores solamente es posible representar información numérica. Pero en muchos sistemas digitales, tanto de control como de proceso de datos, es necesario representar información alfabética y además algunos signos especiales, lo que ha dado lugar a la existencia de códigos alfanuméricos.

De entre los diversos códigos alfanuméricos existentes, ha sido definido como código internacional el ASCII (American Standard Code for Information Interchange). En el mismo se representan todos los caracteres numéricos y alfabéticos así como ciertos caracteres de control, existiendo un código para cada carácter.

Funciones Lógicas.

Función AND o producto lógico.

La función AND de dos variables es aquella que toma el valor uno cuando las dos variables toman el valor uno, tal y como lo muestra la siguiente tabla de verdad:

Tabla de Verdad de la Función AND

a	b	a.b
0	0	0
1	0	0
1	1	0
1	1	1

Tabla 1-2

Función OR o suma lógica.

La función OR de dos variables es aquella que toma el valor uno cuando al menos una de las variables toma el valor uno.

Tabla de Verdad de la Función OR

a	b	a+b
0	0	0
1	0	1
1	1	1
1	1	1

Tabla 1-3.

Función NOT o inversión.

La función NOT de una variable adopta como salida el estado inverso de la variable, es decir:

Tabla de Verdad de la Función NOT

a	$\bar{a}$
0	1
1	0

Tabla 1-4

Función XOR u O-exclusiva.

La función XOR de dos variables es aquella que adopta un valor uno, cuando una de las variables toma el valor uno y la otra cero o viceversa.

Tabla de Verdad de la Función OR

a	b	a+b
0	0	0
1	0	1
1	1	1
1	1	1

Tabla 1-5

Esta función puede ser representada en términos de las funciones básicas (AND, OR y NOT) de acuerdo con la siguiente ecuación lógica:

a+b= a.~b + ~a.b

Organización Interna del Computador

La operación de la mayoría de los sistemas de computación, incluyendo los IBM PCs y compatibles, están basados en un sencillo concepto: almacenar instrucciones y datos en memoria y usar el CPU para ejecutar repetitivamente estas instrucciones y manipular la información almacenada. Las computadoras basadas en este principio son conocidas como máquinas de Von Neumann. De esto se deduce, que el CPU y la memoria son los componentes más importantes de un computador.

En un computador, usualmente vamos a encontrar dos tipos de memoria: memoria de acceso al azar (RAM) la cual permite operaciones de escritura y lectura en cualquier posición de la misma, y memoria de sólo lectura (ROM), la cual contiene información que puede ser leída pero no alterada. La ROM es usada para almacenar rutinas de bajo nivel diseñadas para la ejecución de tareas específicas, tales como el manejo de los dispositivos de E/S. La RAM es usada por el sistema operativo y los programas del usuario.

El sistema operativo es un componente crucial en un sistema. Este programa se encarga de cargar y ejecutar otros programas, proveer acceso a los archivos del sistema, controlar los periféricos e interactuar con el usuario. El sistema operativo es quien le da a un sistema su personalidad. El MS-DOS, UNIX y OS/2 son ejemplos de sistemas operativos para la familia de PCs.

Los componentes del hardware de un sistema están interconectados. El CPU, la memoria y los dispositivos de E/S están unidos por un conjunto de conductores llamados buses. El propósito de cada conductor está claramente definido. Los buses establecen estándares acerca de los niveles y sincronización de las señales, que son entendidos por el CPU y la circuitería de soporte. Los buses puede ser divididos según su propósito en tres: bus de dato, bus de direcciones y bus de control.

Arquitectura fundamental de un computador.

Todo computador digital consta de cuatro partes bien definidas: Unidad Central de Procesamiento (CPU), Memoria, Periféricos y Buses.

Arquitectura fundamental de un computador.

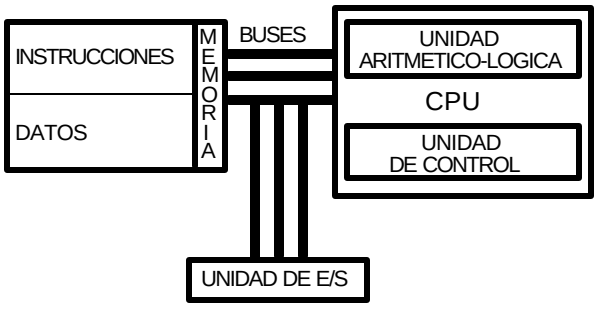


Figura 2-1

Unidad Central de Procesamiento.

La *CPU* es quien crea y controla el flujo de datos, que circula por el computador a partir de las instrucciones recibidas de la memoria, que sirven para indicar las operaciones o tratamiento a realizar sobre los datos recibidos desde el exterior o previamente almacenados en la memoria. La misma consta de dos partes: la Unidad de Control y la Unidad Aritmético-Lógica (ALU).

Unidad de Control.

La Unidad de Control recibe secuencialmente las instrucciones desde la memoria, a través del bus de datos, almacenándolas en el registro de instrucciones (IR). Desde IR las instrucciones pasan al decodificador de instrucciones, el cual se encarga de interpretarlas y producir una serie de impulsos de gobierno y control. Estos impulsos regulan a los elementos de la máquina, que participan en la ejecución de la instrucción.

La Unidad de Control, además de descodificar las instrucciones y de generar los impulsos de control, incrementa sincrónicamente un contador, llamado contador de programa (PC) cada vez que se ejecuta una instrucción, con objeto de que quede señalando a la siguiente instrucción.

Estructura interna de la Unidad de Control.

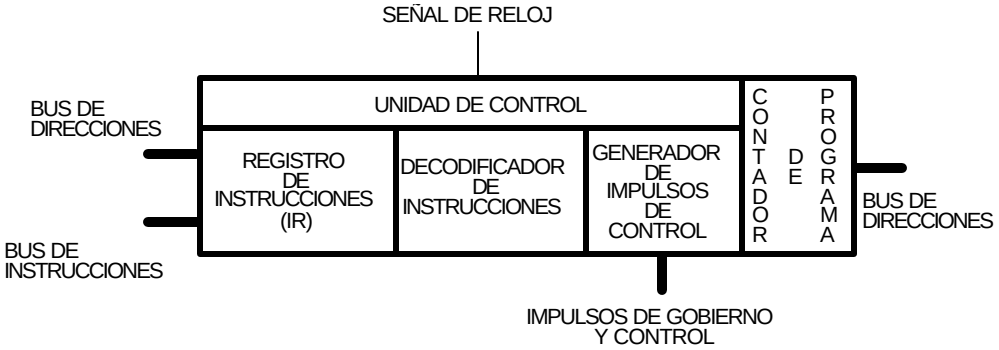


Figura 2-2

Unidad aritmético-lógica.

La Unidad aritmético-lógica (ALU) es la encargada del procesamiento lógico y aritmético de los datos, según el carácter que determine cada instrucción.

Esquema interno de la Unidad Aritmético-Lógica.

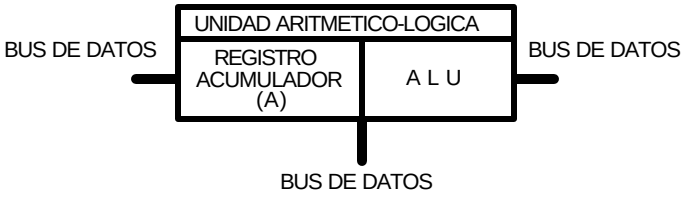


Figura 2-3

Memoria.

El programa o secuencia de instrucciones, que debe seguir la máquina para realizar el procesamiento de los datos, está almacenado en una parte de la memoria, denominada **memoria o segmento de instrucciones** para diferenciarla del resto de la misma, que se emplea para guardar datos y resultados en forma temporal.
La información, que recibe la memoria a través del **bus de direcciones**, es un conjunto de bits lógicos, tantos como líneas tenga el bus, que seleccionan la posición de memoria a la que se accede. El decodificador de direcciones se encarga de elegir una posición de la matriz de la memoria, descodificando la información que

ha llegado por el **bus de direcciones**. Como generalmente la memoria está constituida físicamente por mas de un *chip*, será tarea del decodificador de direcciones habilitar al *chip* correspondiente.

En una computadora encontraremos dos tipos de memoria: de sólo lectura (*ROM*) y de acceso al azar (*RAM).*

Memoria *ROM.*

La memoria *ROM* (*Read Only Memory*) o memoria de sólo lectura también permite el acceso directo a cada uno de los elementos que la componen, pero la información en ella contenida puede ser leída pero no alterada. Debido a que conserva la información, aún en el caso de ausencia de energía, se usa para almacenar las rutinas de mas bajo nivel, que sirven para el arranque del sistema.

Memoria *RAM.*

La memoria *RAM* (*Random Access Memory*) o memoria de acceso al azar debe su nombre al hecho de permitir el acceso a cualquiera de las localidades de memoria en forma directa, en contraste con las memorias de acceso secuencial en las cuales para acceder al N-ésimo elemento, era necesario acceder previamente a los N-1 elementos anteriores; pero su característica más importante es la de que la información contenida en cada una de las localidades puede ser leída y/o alterada. En ella se va a almacenar, por lo tanto, el sistema operativo y los programas del usuario, así como la información temporal que estos manejen.

A la memoria *RAM* se le suele llamar memoria volátil, por el hecho de que la información en ellas almacenada, se pierde en ausencia de energía.

Periféricos.

Son los encargados de enviar y/o recoger información del mundo externo a la computadora e intercambiarla con la unidad central de procesamiento a través de la unidad de entradas y salidas.

Buses

Los buses no son más que los conductores que interconectan cada una de la partes que componen al computador. A través de ellos viaja información que según su función permite clasificarlos en tres tipos: bus de Datos, bus de Direcciones y bus de Control.

Bus de datos.

El bus de datos se encarga de transferir información entre el CPU, la memoria y los periféricos. Es bidireccional, ya que la información puede fluir en ambos sentidos, es decir, desde o hacia el microprocesador.

Bus de direcciones.

El bus de direcciones permite seleccionar la localidad de memoria o el periférico que el CPU desea accesar. Este bus es unidireccional ya que la información a través de él siempre fluye desde el microprocesador.

Bus de control.

En el Bus de Control se encuentran las diferentes señales encargadas de la sincronización y control del sistema. Su naturaleza es unidireccional aun cuando existen señales que salen del microprocesador así como otras que entran al microprocesador. Ejemplos de las señales de control son:

WR	(escritura)
RD	(lectura)
WAIT	(espera)
READY	(listo), etc.

La familia de microprocesadores 80x86

El 8086 representa la arquitectura base para todos los microprocesadores de 16 bits de Intel: 8088, 8086, 80188, 80186 y 80286. Aunque han aparecido nuevas características a medida que estos microprocesadores ha ido evolucionando, un entendimiento básico de la arquitectura del 8086 es suficiente para nuestros propósitos. La base del conocimiento acerca de como programar un 8086 es válida para un 8086, 80386 u 80486. Cambian algunos detalles, pero los principios básicos permanecen inalterados.

Características Generales

Todos los procesadores Intel, usados en la actualidad en los PC y compatibles, son miembros de la familia 80x86. El conjunto de instrucciones, registros y otras características son similares, a excepción de algunos detalles.

Todos los microprocesadores de la familia 80x86 poseen dos características en común:

Arquitectura segmentada: esto significa que la memoria es dividida en segmentos con un tamaño máximo de 64k.

Compatibilidad: Las instrucciones y registros de las anteriores versiones son soportadas por las nuevas versiones. Las nuevas versiones contienen registros e instrucciones adicionales a las presentes en anteriores versiones.

La familia de microprocesadores 80x86 consta de los siguientes microprocesadores:

8088: Es un microprocesador de 16 bits, usado en las primeras PCs (XT compatibles. Soporta solamente el modo real. Es capaz de direccionar hasta 1 megabyte de memoria y posee un bus de datos de 8 bits.

8086: Similar al 8088, con la excepción de que el bus de datos es de 16 bits.

80188: Similar al 8088, pero con un conjunto de instrucciones extendido y cierta mejoras en la velocidad de ejecución. Se incorporan dentro del microprocesador algunos *chips* que anteriormente eran externos, consiguiéndose una mejora en el rendimiento del mismo.

80186: Igual al 80188 pero con un bus de datos de 16 bits.

80286: Incluye el conjunto de instrucciones extendido del 80186, pero además soporta memoria virtual, modo protegido y multitarea.

80386: Soporta procesamiento de 16 y 32 bits. El 80386 es capaz de manejar memoria real y protegida, memoria virtual y multitarea. Es más rápido que el 80286 y contiene un conjunto de instrucciones ampliado.

80386SX: Similar al 80386, pero con un bus de datos de sólo 16 bits.

80486: Incorpora un caché interno de 8K y ciertas mejoras en velocidad con respecto al 80386. Incluye un coprocesador matemático dentro del mismo *chip*.

80486SX: Similar al 80486, con la diferencia de que no posee coprocesador matemático.

80486DX2: Similar al 80486, pero con la diferencia de que internamente trabaja al doble de la frecuencia externa del reloj.

Arquitectura Interna Básica.

El 8086 tiene dos procesadores en el mismo chip. Estos son: **La unidad de Ejecución** y **La Unidad de Interface con los Buses**. Cada uno de ellos contiene sus propios registros, su propia sección aritmética, sus propias unidades de control y trabajan de manera asíncrona el uno con el otro para proveer la potencia total de cómputo. La unidad de interface de bus se encarga de buscar las instrucciones para adelantar su ejecución y proporciona facilidades en el manejo de las direcciones. Luego, la unidad de interface se responsabiliza del control de la adaptación con los elementos externos y la CPU central. Dicha unidad de interface proporciona una dirección de 20 bits o un dato de 16 para la unidad de memoria o para la unidad de E/S en la estructura externa del computador.

Una de las características de la unidad de interface de bus es la instrucción de ordenación y tratamiento de colas. Esta cola o búsqueda anticipada de instrucciones proporciona una arquitectura *pipe-line* muy sofisticada. La unidad de interface de bus busca instrucciones en el flujo de cola antes de ser ejecutadas por la unidad de ejecución. Aún cuando dos o más bytes del flujo de cola de instrucciones estén vacantes, la unidad de interface de bus fuerza un ciclo de búsqueda de instrucción automáticamente para hacer que la cola se mantenga tan llena como sea posible. La unidad de interface de bus, carga 6 bytes de RAM (FIFO) con las instrucciones siguientes a la que se ejecuta.

En otras palabras, la unidad de ejecución no tiene que esperar mientras se encuentra una instrucción y puede operar continuamente con su velocidad máxima. En el caso de saltos en el programa, una instrucción que sigue los saltos de flujo a través de la cola, en un ciclo de búsqueda de instrucción libera la cola, de instrucciones innecesarias.

La unidad de ejecución del 8086 tiene la responsabilidad de ejecutar instrucciones y se puede considerar como CPU clásica. Contiene una ALU con un registro de banderas asociados y un conjunto de registros de propósito general. Esta unidad es la responsable de la ejecución de las instrucciones.

Modos Real y Protegido.

En el modo real, sólo se puede ejecutar una tarea a la vez. Este es el único modo soportado por las actuales versiones del MS-DOS. El microprocesador es capaz de direccionar hasta un 1 megabyte de memoria.

En modo protegido, el microprocesador es capaz de ejecución multitarea. Este modo permite que el sistema operativo conserve siempre el control del recursos del sistema. Se pueden direccionar 16 megabytes o más de memoria. El término protegido implica que la memoria usada por una de las tareas está protegida contra la intromisión de otra aplicación que se esté ejecutando al mismo tiempo.

Registros Internos.

Todos los microprocesador emplean localidades internas de memoria llamadas registros. El número y tamaño de los registros es un factor indicativo de cuan poderoso es el microprocesador. Los registros llevan a cabo numerosas funciones durante la ejecución de un programa. Ellos pueden ser usados para almacenar valores temporalmente. La mayor parte de las instrucciones en ensamblador son capaces de manipular información contenida en la memoria externa, para la velocidad de ejecución alcanzada cuando los datos se encuentran en los registros internos es significativamente mayor.

Arquitectura básica de un microprocesador 8086.

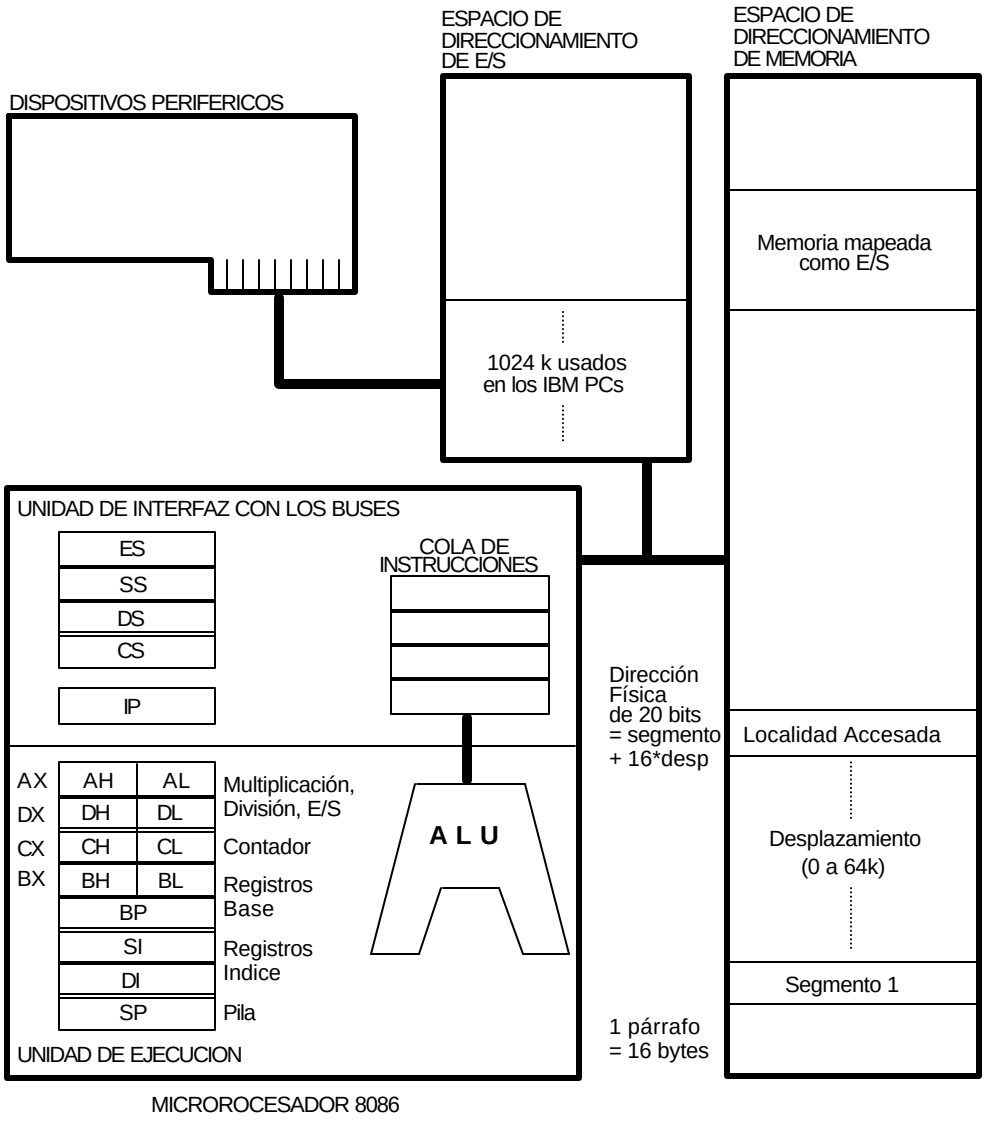


Figura 3-1

Los registros internos permiten también el direccionamiento indexado de datos en la memoria lo cual es equivalente a la técnica empleada para tener acceso a los campos de un arreglo en un lenguaje de alto nivel.

Todos los registros del 8086 son de 16 bits, aunque algunos de ellos pueden ser empleados en unidades de 8 bits. A partir del 80386 están disponibles registros de 32 bits.

El empleo de los registros para cualquier operación no es arbitrario. Algunos registros están dedicados a operaciones aritméticas, otros a labores de direccionamiento, etc.

Registros de Datos.

Los registros de datos o de propósito general (AX, BX, CX y DX), son usados para transferencia de datos y almacenamiento de parámetros en operaciones matemáticas. Cada uno de estos registros pueden ser manipulados como un dato de 16 bits, o como dos registros de 8 bits cada uno. Estas mitades son identificadas por las letras H (*Hi*) y L (*Lo*). Las letras son agregadas al nombre base del registro de 16 bits. De esta manera el registro AX, está divido en los registros AH y AL. Estos registros de 8 bits pueden ser

manipulados independientemente. Sin embargo, si se asigna un nuevo valor a AX, por ejemplo, los contenidos de AH y AL son alterados en consecuencia.

Registros internos del microprocesador 8086

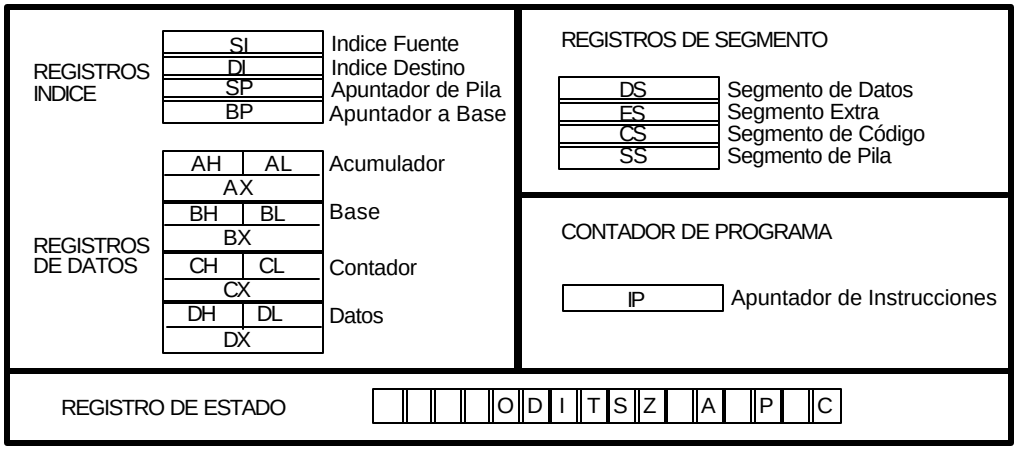


Figura 3-2

Los nombres de los registros, describen la forma como algunas instrucciones en ensamblador los emplean:

AX: El registro **A**cumulador es probablemente el más frecuentemente usado. Es empleado para operaciones aritméticas de cualquier tipo. También es usado en las instrucciones de entrada y salida y en las de manipulación de cadenas.

BX: El registro **B**ase obtiene su nombre de la manera como direcciona la memoria, partiendo de una dirección base. También puede ser empleado para operaciones aritméticas.

CX: EL registro **C**ontador es usado para el control de lazos y el desplazamiento de *bits*. También puede ser empleado en operaciones aritméticas.

DX: El registro de **D**atos es de mucha importancia en las operaciones de multiplicación. También es empleado en las instrucciones de entrada y salida.

Registros Índice.

En el 8086, además del registro BX, existen cinco registros específicamente diseñados para el direccionamiento indexado de la memoria. Estos son los registros IP, SI, DI, SP y BP.

Los registros SP y BP son empleados para direccionamiento en el área de la pila, la cual es una estructura especial de memoria en la cual los datos son *apilados* operando según la convención LIFO.

Los registros SI y DI, en adición al direccionamiento indexado, son empleados en las instrucciones para manipulación de cadenas donde sirven de apuntadores para los rangos de memoria manipulados.

El registro IP o apuntador de instrucciones es quien señala la posición de memoria donde se encuentra la próxima instrucción a ser ejecutada. El mismo es automáticamente incrementado por el microprocesador tras la ejecución de cada instrucción.

Registro de Estado.

El registro de estado, contiene el estado del microprocesador tras la ejecución de ciertas instrucciones. Este es almacenado en forma de banderas las cuales corresponden cada una a un bit. De los 16 bits disponibles sólo se emplean 9 los cuales se dividen en banderas de control y banderas de estado.

Las banderas de control registran el modo de funcionamiento del microprocesador:

Dirección (DF): Controla la dirección de los registros índice en las instrucciones de manejo de cadenas.

Interrupción (IF): Permite habilitar o deshabilitar las interrupciones enmascarables.

Trap (TF):Controla la operación en modo paso-a-paso del microprocesador.

Las banderas de estado registran el estado del microprocesador, normalmente asociado a una comparación o instrucción aritmética:

Acarreo (CF): Indica la ocurrencia de acarreo en operaciones aritméticas.

Desbordamiento (OF): Su activación indica que ocurrió un desbordamiento como producto de una operación aritmética.

Cero (ZF): Se activa para indicar que el resultado de una operación aritmética o lógica fue cero.

Signo (SF): Se activa para indicar que el resultado de una operación aritmética fue un número negativo.

Paridad (PF): Al estar activada indica que como resultado de una operación aritmética o lógica resultó un número con paridad par.

Acarreo Auxiliar (AF): Indica la ocurrencia de un acarreo a nivel del cuarto bit y se emplea para señalar la necesidad de ajuste en operaciones aritméticas con número BCD.

Registros de Segmento.

Los registros de segmento constituyen un grupo muy importante. Para entender su funcionamiento se requiere conocer la manera como los microprocesadores 80x86 direccionan la memoria.

La familia de microprocesadores 80x86 son capaces de manejar hasta 1 Mb de memoria en modo real. Para ello se requiere de un bus direcciones de 20 bits. Si embargo, los registros del 80x86 empleados para hacer referencia a posiciones de memoria son de 16 bits de longitud, lo cual los imposibilita para direccionar a cada posición disponible, por si solos. Los diseñadores de este microprocesador resolvieron el problema, dividiendo la memoria disponible en segmentos. Cuando se accesa a un segmento, la dirección base del segmento (de 16 bits) es desplazada cuatro bits hacia la izquierda, expandiéndola por lo tanto a 20 bits. Para alcanzar a una localidad específica dentro del segmento, el 80x86 añade un desplazamiento a dicha dirección base. Este desplazamiento indica la diferencia entre la dirección base del segmento y la dirección de la localidad de memoria direccionada. El desplazamiento es también de 16 bits, lo cual limita el tamaño de un segmento a 65536 bytes. Los segmentos, por lo tanto tienen un tamaño de 64 K.

Aun cuando teóricamente un segmento puede empezar en cualquier localidad de memoria, el desplazamiento de 4 posiciones a la derecha obliga a que la dirección base de un segmento siempre sea un múltiplo de 16. De acuerdo con esto, el segmento 0 comienza en la dirección 0, el segmento 1 en la dirección 16, y así sucesivamente.

Función de los registros de segmento.

En 1 Mb de memoria, puede existir hasta 65536 segmentos diferentes. Sin embargo sólo cuatro segmentos pueden estar activos en un momento dado. Cada uno es representado por uno de los cuatro registros de segmento.

CS: Este registro contiene la dirección del segmento de código, en el cual están almacenadas las instrucciones ejecutables del programa.

DS: El registro DS contiene la dirección del segmento de datos. En este segmento se almacenan las variables empleadas por un programa.

ES: El segmento extra, puede ser usado para almacenar información. Es frecuentemente usado para operaciones de transferencia de contenidos de bloques de memoria. Las instrucciones para manipulación de cadenas del 80x86, emplean este segmento.

SS: Este registro señala al segmento de pila. La pila es una estructura del tipo LIFO y es empleada para almacenar direcciones de retornos y datos temporales y para pase de parámetros a funciones y procedimientos. Esta estructura es manejada por el SS en conjunto con el registro apuntador de pila (SP).

Estructura de memoria usando la direcciones de segmento y desplazamiento.

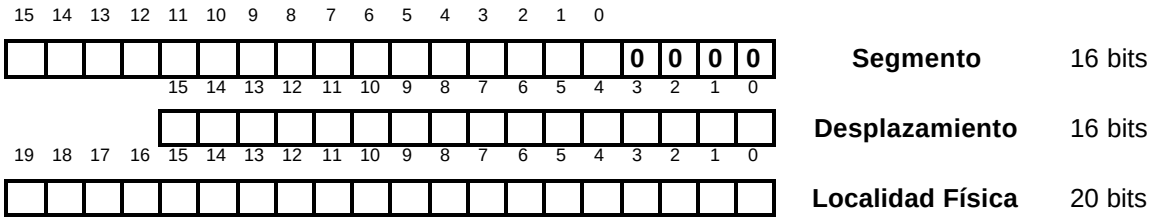


Figura 3-3

Modos de Direccionamiento.

El 8086 ofrece una multitud de vías para direccionar la información: registro a registro, direccionamiento inmediato, direccionamiento directo y varios tipos diferentes de direccionamiento indirecto. Cada modo tiene siempre un operando fuente y uno destino. El operando destino está ubicado a la izquierda de la coma; y el fuente a la derecha. Adicionalmente, los modos directo e indirecto involucran el uso de un registro de segmento.

Direccionamiento Registro.

Es aquel en el cual la operación se lleva a cabo entre los contenidos de dos registros. Por ejemplo, la instrucción

mov AX,BX

indica que el contenido del registro BX sea copiado en el registro AX.

Direccionamiento Inmediato.

En este modo de direccionamiento, uno de los operandos está presente en el o los bytes siguientes al código de operación. Por ejemplo, la instrucción

add AX,3064h

indica que el número 3064h sea sumado al contenido del registro AX y el resultado almacenado en dicho registro.

Direccionamiento Directo.

EL 8086 implementa el direccionamiento directo a memoria, sumando un desplazamiento de 16 bits, indicado por los dos bytes que siguen al código de operación, al contenido del registro de segmento de datos. La suma es pues, la posición de memoria direccionada. Por ejemplo, la instrucción

mov AH,TABLA

señala que el contenido de la posición de memoria cuya dirección está indicada por el identificador TABLA, sea copiado en el registro AH.

Direccionamiento Indirecto.

El modo de direccionamiento indirecto es el más difícil de comprender, pero también el más poderoso. Existen cuatro métodos de direccionamiento indirecto: indirecto a registro, relativo a base, indexado e indexado a base.

Indirecto a Registro.

En el modo de direccionamiento indirecto a registro, la dirección de memoria donde se encuentra uno de los operandos es indicada a través del contenido de los registros BX, BP, SI o DI. La instrucción

mov AX,[DI]

establece que el contenido de la palabra de memoria cuya dirección está indicada por el contenido del registro DI, sea copiado en el registro AX.

Relativo a Base.

El direccionamiento a la memoria de datos, relativo a base simplemente usa el contenido del registro BX o BP como base para la posición efectiva de memoria. La instrucción

mov CL,[BP]+DESP

copia el contenido de la posición de memoria cuya dirección está determinada por la suma del contenido de BP y DESP, en el registro CL.

Indexado.

El direccionamiento indexado directo está permitido especificando los registros SI o DI como índices. Empleando este modo de direccionamiento es posible acceder a los elementos de un vector. La instrucción

sub AH,MATRIZ[SI]

resta del contenido del registro AH, el valor contenido en la posición de memoria especificada por la suma del desplazamiento indicado por el identificador MATRIZ y el contenido del registro SI.

Indexado a Base.

Resulta de la combinación de los modos de direccionamiento Relativo a Base e Indexado Directo. La instrucción

mov DH,VECTOR[BX][DI]

señala que el contenido de la posición de memoria cuya dirección viene indicada por la suma de los contenidos de los registros BX y DX y del desplazamiento establecido por el identificador VECTOR, sea copiado en DH.

Modos de Direccionamiento.

No	Modo	Operando	Segmento	Ejemplo
1	Registro	registro	-	mov ax,bx
2	Inmediato	valor	-	mov ax,500
3	Directo	variable	DS	mov ax,TABLA
4	Inmediato a registro	[BX]	DS	mov ax,[bx]
		[BP]	SS	mov ax,[bp]
		[DI]	DS	mov ax,[di]
		[SI]	DS	mov ax,[si]
5	Relativo a base	[BX]+desp	DS	mov ax,[bx+4]
		[BP]+desp	SS	mov ax,[bp+6]
6	Indexado	[DI]+desp	DS	mov ax,TABLA[di]
		[SI]+desp	DS	mov ax,TABLA[si]¡
7	Indexado a base	[BX][SI]+desp	DS	mov ax,TABLA[bx][si]
		[BX][DI]+desp	DS	mov ax,TABLA[bx][di]
		[BP][SI]+desp	SS	mov ax,TABLA[bp][si]
		[BP][DI]+desp	SS	mov ax,TABLA[bp][di]

Tabla3-1

Contrarrestación de Segmentos.

Los modos de direccionamiento que hacen referencia a posiciones de memoria, pueden estar precedidos por un registro de segmento, lo cual permite sustituir el segmento de operación por defecto Por ejemplo

mov AX,ES:TABLA[SI]

implica que el registro de segmento correspondiente a esta instrucción, que por defecto es DS, será sustituido por ES. El código que se genera para la instrucción está precedido por un byte, que se llama código de prefijo de segmento. A esta acción se le llama contrarrestación de segmentos. En la Tabla 32 podemos ver las posibles combinaciones entre los cuatro registros de segmentos y los seis registros de desplazamiento.

Combinaciones de los Registro de Segmento y Desplazamiento

	CS	SS	DS	ES
IP	si	no	no	no
SP	no	si	no	no
BP	prefijo	por defecto	prefijo	prefijo
BX	prefijo	prefijo	por defecto	prefijo
SI	prefijo	prefijo	por defecto	prefijo
DI	prefijo	prefijo	por defecto	cadenas

Tabla 3-2

Interrupciones.

Esta sección estudia el concepto base de la programación de PCs: el uso de las interrupciones. Básicamente, las interrupciones son las puertas de acceso a las funciones del sistema operativo, el cual controla los recursos del sistema.

El término *interrupción* evolucionó como una descripción de lo que pasa dentro del microprocesador. Una interrupción detiene la ejecución de un programa, para invocar otra rutina. La rutina invocada por efecto de la interrupción es llamada rutina manejadora de interrupción (*interrupt handler*).

La rutina manejadora de interrupción es un programa que puede usar cualquiera de las instrucciones disponibles en el microprocesador. El final de la rutina es marcado por la instrucción **iret** (***i****nterrupt* ***ret****urn*). Tal y como lo señala el nombre de la instrucción, en este punto el control es retornado al programa original. Este mecanismo es similar a la invocación de una subrutina. La diferencia radica en que la interrupción es generada por ente externo al microprocesador y no por el propio programa.

Un programa puede ser interrumpido en cualquier momento, pero debe ser reactivado posteriormente sin modificaciones que afecten su funcionamiento. Esto incluye la restauración del contenido de los registros del microprocesador. Sin embargo es muy difícil que una instrucción no altere al menos uno de estos registros, por lo que la rutina manejadora de interrupción deberá encargarse del proceso de salvar y restaurar el contenido de los mismos.

Tabla de Vectores de Interrupción.

En el 8086 existen un total de 256 interrupciones disponibles, numeradas del 0 al 255.Cada interrupción tiene su correspondiente rutina asociada. La *tabla de vectores de interrupción* especifica la interrupción y ubicación de la rutina manejadora. Esta tabla ocupa el primer Kbyte de la memoria. Para cada interrupción, existe una localidad en la tabla que contiene la dirección de arranque de la rutina manejadora correspondiente.

Cuando se genera una interrupción, el microprocesador determina la localidad de la rutina manejadora leyendo la posición correspondiente en la tabla. Ya que las rutinas manejadoras se encuentran generalmente en otros segmento distintos al del programa interrumpido, los apuntadores establecidos en la tabla son del tipo *FAR*.

Estructura de la tabla de vectores de interrupción

	CS	
	IP	255: Libre
	:	
0000:000E	**CS**	
0000:000C	**IP**	3: Breakpoint
0000:000A	**CS**	
0000:0008	**IP**	2: NMI
0000:0006	**CS**	
0000:0004	**IP**	1: Paso a paso
0000:0002	**CS**	
0000:0000	**IP**	0: División por cero.

Figura 3-4

Estos apuntadores *FAR* son magnitudes de 32 bits donde los 16 menos significativos indican la dirección del segmento y los más significativos el desplazamiento dentro del segmento. Al ser estos apuntadores de 32 bits, es evidente que ocupan cuatro bytes dentro de la tabla.

Tipos de Interrupciones.

Las interrupciones se dividen en dos tipos: *hardware* y *software*.

Interrupciones por software.

Una *interrupción por software* es una interrupción activada por una instrucción especial: la instrucción **Int**. Esta es usada siempre en conjunto con el número de la interrupción a ser invocada. Esto permite que un programa invoque una rutina de la cual no conoce la ubicación exacta en la memoria. Las rutinas pertenecientes al *BIOS* y al núcleo del *DOS* (*kernel*), pueden ser accesadas por este mecanismo.

Tipos de interrupciones.

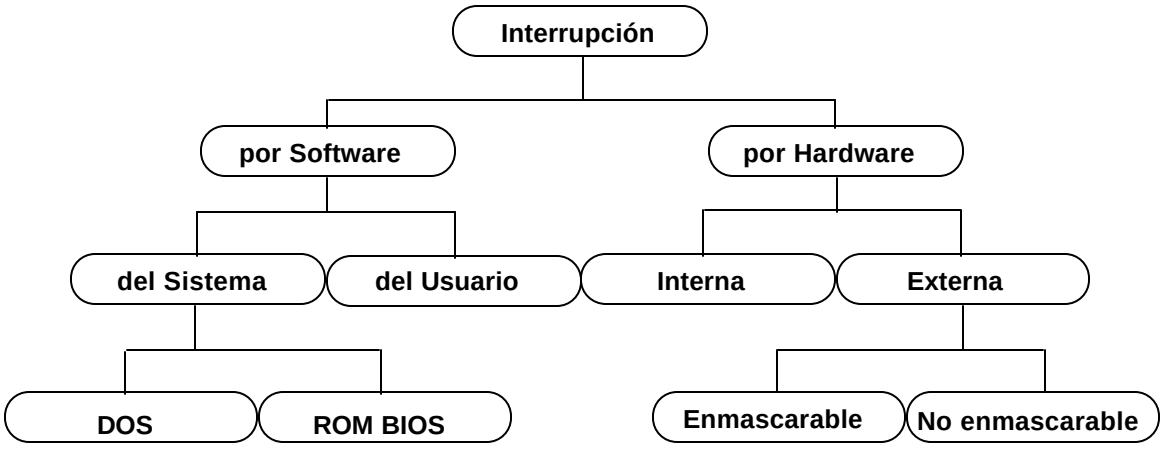


Figura3-5

El propósito más importante de dichas rutinas, es hacer que las diferencias en el *hardware* entre una computadora y otra, sea transparente a los programas de aplicación. Las rutinas del *BIOS* y del *DOS* actúan como intermediarios entre las aplicaciones y el *hardware* del equipo. Esto garantiza cierta independencia de los programas con respecto a la diversidad de configuraciones de *hardware* disponibles.

Las rutinas manejadoras de interrupciones por software generalmente requieren de parámetros que le proporcionen instrucciones específicas sobre el trabajo a realizar. Estos son normalmente transferidos por medio de los registros internos del microprocesador.

Interrupciones por hardware.

Las interrupciones por hardware, son procesos activados por los distintos componentes del sistema. Con ello, el dispositivo solicita la atención del microprocesador. La dirección de la rutina correspondiente es obtenida de la tabla de vectores de interrupción. Estos valores son inicializados por el *BIOS*.

Entre las interrupciones por *hardware* existen dos tipos: la enmascarables y las no enmascarables. Las interrupciones enmascarables son aquellas que pueden ser suprimidas por medio de la instrucción **cli** (***cl****ear* ***i****nterrupt flag*). Esto significa que la rutina correspondiente no será invocada al presentarse la interrupción. Las interrupciones no enmascarables no pueden ser suprimidas y son empleadas para servir eventos críticos del sistema.

Instrucciones.

Las instrucciones del 8086 pueden ser agrupadas en 7 renglones según sus funciones:

Instrucciones de Transferencia de Datos.

Son aquellas llevan a cabo funciones de transferencia de información y pueden ser divididas en dos categorías generales: instrucciones que mueven datos de un registro a otro registro, o entre localidades de memoria y registros e instrucciones que mueven datos hacia o desde la pila. Ejemplos de estas instrucciones son: mov, xchg, push, pop, etc.

Instrucciones Aritméticas.

Llevan a cabo operaciones aritméticas y de comparación. Se dividen en cinco tipos: Instrucciones de adición, substracción, multiplicación, división y comparación. Entre las instrucciones pertenecientes a este grupo tenemos: add, sub, mul, div, cmp, etc.

Instrucciones Lógicas.

El 8086 provee instrucciones para llevar a cabo las cuatro funciones lógicas básicas. Estas son AND, NOT, OR y XOR. Adicionalmente incluye la función TEST, la cual ejecuta una operación AND sin alterar los operandos.

Instrucciones para Manejo de Cadenas.

Las instrucciones de manejo de cadenas llevan a cabo funciones ejecutadas normalmente por un conjunto de instrucciones dentro de un lazo. Cada una de estas instrucciones actualiza los registros apuntadores involucrados en la misma. En cada iteración los registros apuntadores pueden ser incrementados o decrementados por uno o por dos. Los registros apuntadores serán incrementados si el valor del bit de dirección del registro de estado está en cero y decrementados en el caso contrario. Los registros apuntadores afectados serán modificados por uno o por dos ya sea que la operación involucre bytes o palabras. Existen cinco instrucciones para manejo de cadenas:

MOVS: Mueve 8 o 16 bits de datos desde una localidad de memoria hasta otra.
LODS: Carga un dato de 8 o 16 bits desde una localidad de memoria al registro AL o AX.
STOS: Almacena el contenido de AL (operación de 8 bits) o de AX (operación de 16 bits) en una localidad de memoria.
SCAS: Compara el contenido de AL (operación de 8 bits) o AX (operación de 16 bits) con el de una localidad de memoria.
CMPS: Compara el contenido de una localidad de memoria con el de otra.

El prefijo REP.

El prefijo REP es un byte de código que convierte a una instrucción de manejo de cadena en un lazo auto ejecutante. La instrucción correspondiente se ejecutará en cada iteración del lazo. Los registros índice SI y DI se asumen como apuntadores a las cadenas fuente y destino; estos apuntadores son incrementados o decrementados automáticamente luego de la ejecución de la instrucción. Esto causa que los apuntadores señalen a la siguiente posición de memoria. El prefijo REP especifica la condición de terminación que debe ser satisfecha para que la instrucción finalice su ejecución. Para las instrucciones MOVS, LODS y STOS, existe una única condición de terminación. El registro CX es tratado como contador; y la instrucción se repetirá tantas veces como lo indique el contenido del registro CX. Las instrucciones CMPS y SCAS también emplean el registro CX de la manera anteriormente descrita, pero adicionalmente el estado del bit cero (ZF) puede ser usado como condición de terminación. Para ello se emplean las variaciones REPZ o REPE y REPNZ o REPNE. Por ejemplo, si el prefijo empleado es REPZ se indica que la iteración finalizará cuando CX sea 0 o cuando como producto de la comparación efectuada por la instrucción de manejo de cadena, el bit de cero sea activado.

Instrucciones para Control del Contador de Programa.

Este grupo de instrucciones alteran incondicionalmente el contenido del contador de programa, y en algunos casos el del registro de segmento de código.

Las instrucciones CALL son empleadas para transferir el control a una subrutina que puede o no encontrarse en el segmento de código actual. La dirección de la subrutina puede ser indicada en forma inmediata, directa o indirecta.

Las instrucciones RET son empleadas para retornar el control de la subrutina al programa que la invocó. La instrucción RET que finaliza una subrutina opera a la inversa de la instrucción CALL. Cuando se ejecuta la instrucción RET, la información es extraída de la pila y depositada en el contador de programa, y opcionalmente en el registro de segmento de código. Adicionalmente, la instrucción RET puede añadir un desplazamiento al apuntador de pila. Esto permite el ajuste el apuntador de pila para eliminar los parámetros que se encontraban en la pila, para el funcionamiento de la subrutina.

Las instrucciones JUMP, al igual que las instrucciones CALL permiten transferir el control de la ejecución a otra sección del programa que puede o no encontrarse en el mismo segmento. La diferencia radica en el hecho de que en el caso de la instrucción JUMP, la dirección de retorno no es almacenada en la pila.

Instrucciones de Salto Condicional.

Son instrucciones que alteran el contenido del contador de programa basado en ciertas condiciones, dadas por el registro de estado. En general mayor (*greater*) o menor (*less*) son adjetivos empleados para operaciones de números con signo y por encima (*above*) y por debajo (*below*) son aplicados a operaciones de números sin signo. La sintaxis general de las instrucciones de salto condicional es j*XX* donde *XX* es sustituido por el identificador de las condición a chequear.

Existe un conjunto de instrucciones, dentro de este grupo, que decrementan el contenido de CX para luego, opcionalmente, alterar el contenido del contador de programa, las cuales son llamadas estructuras LOOP.

Instrucciones de Control del Procesador.

Son instrucciones que operan sobre algunas de las banderas del registro de estado y controlan varios aspectos de la operación del microprocesador. Como ejemplo de estas instrucciones tenemos clc, stc, cld, etc.

Instrucciones de Entrada y Salida.

Son aquellas que llevan a cabo funciones de entrada y salida. La dirección de un puerto puede ser especificada de manera inmediata o a través del registro DX. Como ejemplo de estas instrucciones tenemos in y out.

Instrucciones para generación de interrupciones por software.

Las instrucciones para generación de interrupciones por software, transfieren el control a una subrutina cuya dirección está ubicada en una tabla que ocupa el primer Kbyte del mapa de memoria del 8086 (la cual es llamada tabla de vectores de interrupción). Tienen la ventaja, sobre las instrucciones CALL, de ocupar solo dos bytes, en comparación con cinco bytes usados por la instrucción CALL en llamadas intersegmento. Adicionalmente, las interrupciones por software salvan automáticamente el contenido del registro de estado en la pila. La desventaja está en que una subrutina llamada por esta vía debe culminar con una instrucción iret en lugar de ret, la cual toma más tiempo en ejecutarse.

Instrucciones de Rotación y Desplazamiento.

Estas instrucciones permiten desplazar los bits que componen un dato dentro de un registro o memoria. Si el último bit es realimentado, tenemos entonces una operación de rotación. Las instrucciones de rotación y desplazamiento son frecuentemente usadas en operaciones de chequeo de bits. Pueden ser empleadas solas, o en conjunto con operaciones lógicas para la verificación de diversos patrones de bits. Las instrucciones de rotación pueden también ser usadas para llevar a cabo operaciones aritméticas tales como multiplicación y división. Como ejemplo de estas instrucciones tenemos rol, shr, ror, etc.

Introducción a la Programación en Lenguaje Ensamblador

Seguramente usted habrá oído que el lenguaje ensamblador es tan difícil que tan sólo unos pocos superdotados son capaces de utilizar. ¡No lo crea!. En lenguaje ensamblador no es más que una forma en que los humanos nos comunicamos con los computadores, en su lenguaje nativo. Además, el lenguaje ensamblador es muy poderoso, tanto que es la única manera de sacarle todo el provecho a los microprocesadores.

Usted podrá escribir programas totalmente en lenguaje ensamblador, o mezclarlo con lenguajes de alto nivel. De cualquier forma, en ensamblador es posible escribir programas compactos y muy rápidos. Tan importante como la velocidad, también lo es la habilidad de controlar cada aspecto de la operación del computador, hasta el más mínimo detalle.

En este capítulo nos introduciremos en la programación en lenguaje ensamblador y podremos explorar las características únicas que nos brinda.

Escribiendo el primer Programa en Lenguaje Ensamblador.

En el mundo de la programación, el primer programa es tradicional un programa que muestre el mensaje "Hola Mundo" y justo con el empezaremos.

Haciendo uso del editor de textos de su preferencia (editor ASCII), tipee las siguientes líneas:

```
MODEL   Small
Stack           100h
DATASEG
Mensaje         DB 'Hola, Mundo',13,10,'$'
CODESEG
Inicio:         mov ax,@data
                mov ds,ax               ; DS ahora apunta al segmento de datos.
                mov ah,9                ; Función del DOS para impresión de cadenas.
                mov dx,OFFSET Mensaje   ; Apuntador al mensaje 'Hola Mundo'.
                int 21h
                mov ah,4Ch              ; Transferencia del control al DOS.
                int 21h
END             Inicio
```

Tan pronto como lo finalice, salve HOLA.ASM en disco.

Si usted está familiarizado con algún lenguaje de alto nivel, pensará que la versión en ensamblador de 'Hola, Mundo' es un poco larga. En realidad los programas en ensamblador tienden a ser largos ya que cada instrucción por si misma, hace menos que una instrucción en un lenguaje de alto nivel. Por otra parte, usted podrá combinar las instrucciones de la manera que lo requiera. Esto significa, que al contrario de un lenguaje de alto nivel, con lenguaje ensamblador usted puede ordenarle al microprocesador que haga cualquier cosa de la cual sea capaz.

Ensamblando el primer programa.

Antes de poder ejecutar el programa, es necesario convertirlo en un archivo ejecutable. . Esto requiere de dos pasos adicionales: ensamblaje y enlace. La Figura 4-1 muestra el ciclo de desarrollo de un programa en lenguaje ensamblador.

Ciclo de desarrollo de un programa en lenguaje ensamblador.

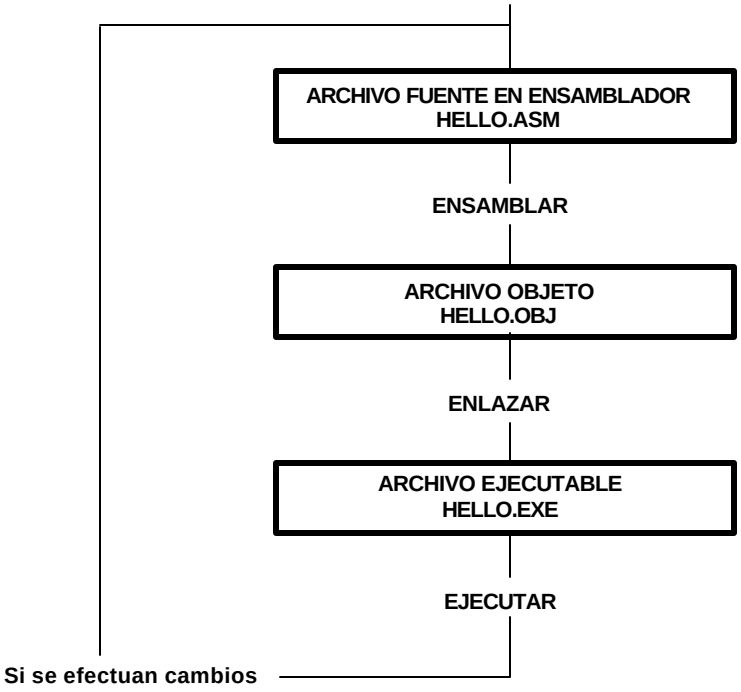


Figura 4-1

El ensamblaje convierte el programa fuente en una forma intermedia llamada programa objeto, y el paso de enlace combina uno o más módulos objeto en un archivo ejecutable.

Para ensamblar HOLA.ASM tipee desde la línea de comandos los siguiente:

TASM HOLA

El archivo HOLA.ASM será ensamblado y convertido en HOLA.OBJ (Fíjese que no es necesario indicar la extensión del archivo fuente ya que el ensamblador adopta por defecto la extensión .ASM). No recibirá ningún mensaje de error si tipeo el programa exactamente como aparece en el listado anterior. Cualquier mensaje de error aparecerá en pantalla, acompañado del número de línea donde este se encuentre. Si recibió algún mensaje de error, revise el código con su editor de textos y compílelo de nuevo.

Enlazando el primer programa.

Después que HOLA.ASM ha sido ensamblado, sólo falta un paso para obtener un archivo ejecutable. Una vez enlazado el código objeto obtenido, usted podrá ejecutar su programa.

Para enlazar el programa, emplearemos el programa TLINK. Tipee desde la línea de comandos lo siguiente:

TLINK HOLA

Note que no es necesario indicar la extensión del archivo. El enlazador asume, por defecto, que la extensión es .OBJ. Cuando finaliza el proceso de enlace, se ha creado un programa con el mismo nombre del módulo objeto, pero con extensión .EXE.

Es posible que ocurran errores durante el proceso de enlace, pero es muy poco probable que ocurra en este caso. Si recibió algún mensaje de error, corrija el código fuente de tal manera que sea un copia fiel del listado dado anteriormente, ensámblelo y enlácelo nuevamente.

Ejecutando el primer programa.

Ahora que el programa está listo para ser ejecutado, tipee HOLA desde la línea de comandos. El siguiente mensaje

Hola, Mundo

aparecerá en pantalla. Y eso es todo. Hemos creado el primer programa en lenguaje ensamblador.

Modificaciones al primer programa.

Regresemos al editor de texto, y efectuemos algunas modificaciones en el programa fuente de tal manera que nuestro programa pueda tener alguna información del mundo exterior (por ejemplo, del teclado). Cambie el código de acuerdo al listado siguiente:

```
MODEL           Small
STACK           100h
DATASEG
Tiempo          DB ' ¿ Es de mañana o de tarde (M/T) ?$'
BuenosDias      DB 'Buenos días, Mundo',13,10,'$'
BuenasTardes    Db ´Buenas tardes, Mundo',13,10,'$,
CODESEG
Inicio:         mov ax,@data
                mov ds,ax                   ; DS ahora apunta al segmento de datos.
                mov dx,OFFSET Tiempo        ; Apuntador al mensaje Tiempo.
                mov ah,9                    ; Función de impresión de cadenas.
                int 21h
                mov ah,1                    ; Función de lectura de un carácter desde el teclado.
                int 21h
                cmp al,'m'                  ; ¿ Es de mañana ?
                jz AM                       ; si
                cmp al,'M'
                jz AM
                mov dx,OFFSET BuenasTardes ; no, Es de tarde
                jmp MostrarMensaje
AM:             mov dx,OFFSET BuenosDias
MostrarMensaje: mov ah,9                    ; Función de impresión de cadenas.
                int 21h
                mov ah,4Ch                  ;  Transferencia del control al DOS.
                int 21h
END             Inicio
```

Al programa se le han añadido dos cosas muy importantes: entrada de datos y toma de decisiones. El programa pregunta si es de mañana o de tarde y lee un carácter del teclado. Si el carácter tipeado es una m mayúscula o minúscula, mostrará el mensaje **Bueno días, Mundo** y en el caso contrario, el mensaje **Buenas tardes, Mundo**. Todos los elementos esenciales de cualquier programa, entrada de datos, despliegue de información y toma de decisiones, están presente en este código.

Salve la versión modificada en disco (reemplazará a la versión anterior), recompile y enlácelo nuevamente tal y como se hizo en el ejemplo anterior. Ejecute el programa tipeando HOLA desde la línea de comandos. El mensaje:

¿ Es de mañana o de tarde (M/T) ?

es mostrado, con el cursor parpadeando junto al signo de interrogación, esperando por su respuesta. Presione M. El programa responderá:

Buenos días, Mundo

HOLA.ASM es ahora un programa interactivo y capaz de tomar decisiones.

El curso de la escritura de programas en lenguaje ensamblador, es posible cometer una gran variedad de errores de tipeo y sintaxis. El ensamblador los detecta durante el proceso de conversión del programa fuente a objeto y los reporta. Los errores reportados pueden ser de dos tipos: *warning* y *errors*. El ensamblador mostrará un *warning* si detecta algo sospechoso, que si bien no impide la compilación del programa, puede causar problemas en la ejecución del mismo. Algunas veces estos pueden ser ignorados, pero lo mejor es verificar el código hasta que nos aseguremos de haber comprendido el problema. El ensamblador mostrará un mensaje de *error* cuando detecta algún problema en el código que impide la compilación del programa fuente y por ende la generación del archivo objeto.

En otras palabras, los *warning* son de precaución, más no fatales mientras que los *errores* deben necesariamente ser corregidos antes de poder ejecutar el programa.

Al igual que con cualquier otro lenguaje de programación, el ensamblador no será capaz de detectar errores de lógica en los algoritmos implementados.

El segundo programa en lenguaje ensamblador: *REVERSE.ASM*.

Ahora ya estamos listo para escribir nuestro segundo programa en lenguaje ensamblador: *REVERSE.ASM*. Vayamos al editor de textos e introduzcamos las siguientes líneas:

```
MODEL           Small
STACK                   100h
DATASEG
LongitudMaxima          EQU 1000
CadenaAInvertir         DB LongitudMaxima DUP (?)
CadenaInvertida         DB LongitudMaxima DUP (?)
CODESEG
Inicio:                 mov ax,@data
                        mov ds,ax               ; DS apunta al segmento de datos.
                        mov ah,3Fh                      ; Función de lectura.
                        mov bx,0                        ; Entrada estándar.
                        mov cx,LongitudMaxima           ; Máximo # de caracteres a ser leídos.
                        mov dx,OFFSET CadenaAInvertir   ; Apuntador a la cadena a invertir.
                        int 21h
                        and ax,ax                       ; ¿Fue leído algún carácter?.
                        jz Salir                        ; no, salir
                        mov cx,ax               ; Almacenar longitud en CX.
                        push cx                         ; Salvar la longitud en la pila.
                        mov bx,OFFSET CadenaAInvertir
                        mov si,OFFSET CadenaInvertida
                        add si,cx                       ; Apuntador al final del buffer
                        dec si                          ; de la cadena invertida.
Lazo:                   mov al,[bx]                     ; Obtener carácter siguiente.
                        mov [si],al                     ; Almacenar en orden contrario.
                        inc bx                          ; Apuntar al siguiente carácter.
                        dec si                          ; Apuntar a localidad anterior.
                        loop Lazo                       ; Repetir.
                        pop cx                          ; Recuperar longitud.
                        mov ah,40h                      ; Función de escritura.
                        mov bx,1                        ; Salida estándar.
                        mov dx,OFFSET CadenaInvertida   ; Apuntador a la cadena invertida.
                        int 21h
Salir:                  mov ah,4Ch                      ; Retornar control al DOS.
                        int 21h
END Inicio
```

Pronto veremos que hace el programa. Lo primero es salvarlo en disco.

Para ejecutar REVERSE.ASM primero debemos ensamblarlo. Tipee

TASM REVERSE

y luego

TLINK REVERSE

para crear el archivo ejecutable.

Tipee REVERSE desde la línea de comandos para ejecutar el programa. Si el ensamblador o el enlazador reportaron algún error, revise cuidadosamente el código fuente, corrija los errores y recompile y enlace de nuevo el programa.

Al ejecutar el programa, el cursor aparecerá parpadeando en la pantalla. El programa está esperando a que usted tipee algo. Pruebe por ejemplo con la cadena siguiente.

ABCDEFGHIJK

luego presione *ENTER*. El programa mostrará en pantalla:

KJIHGFEDCBA

y finaliza. Tipee REVERSE nuevamente desde la línea de comandos. Esta vez tipee:

0123456789

y presione ENTER. El programa responde con

9876543210

Ahora está claro lo que hace REVERSE: invierte el orden de los caracteres en una cadena introducida desde el teclado. La manipulación de caracteres y cadenas es uno de los puntos fuertes de la programación en lenguaje ensamblador.

Felicitaciones. Usted a escrito, ensamblado, enlazado y ejecutado varios programas en lenguaje ensamblador, y ha estudiado los fundamentos de la programación en lenguaje ensamblador: entrada de datos, procesamiento y despliegue de resultados.

Programando para MS-DOS.

El conocimiento de la estructura general del MS-DOS es de mucha utilidad para la comprensión del funcionamiento del sistema. En este capítulo se discutirá acerca de como está organizado el MS-DOS y como resulta estructurada la memoria al encender el computador.

Estructura del MS-DOS.

El MS-DOS está distribuido en varios niveles, que permiten aislar el núcleo lógico del sistema operativo (*kernel*) y la percepción del usuario del sistema, del *hardware* en el que esta ejecutándose. Estos niveles son:

- BIOS (*Basic Input/Output System*).
- El núcleo del DOS (*kernel*).
- El procesador de comandos (*COMMAND.COM*).

BIOS.

El BIOS es específico para cada sistema y proporcionado por el fabricante. Contiene los manejadores (*drivers*) dependientes del *hardware* pre instalados:

- Consola y teclado (*CON*).
- Impresora (*PRN*).
- Dispositivo auxiliar (*AUX*).
- Hora y Fecha (*CLOCK$*).
- Disco de arranque (*block device*).

El núcleo del MS-DOS se comunica con estos manejadores de dispositivos a través de mensajes de requerimiento de entrada/salida; los manejadores traducen estos mensajes a los comandos apropiados para los diversos controladores de *hardware*. En la mayor parte de los sistemas MS-DOS, las partes más primitivas de los manejadores de *hardware* se encuentran en memoria de sólo lectura (*ROM*). de tal manera que puedan ser usados por las rutinas de arranque del sistema.

Los términos residente e instalable, son usados para diferenciar entre los manejadores incluidos en el BIOS y los instalados durante la inicialización del sistema por medio del comando *DEVICE* en el archivo *CONFIG.SYS*.

El núcleo del DOS (*kernel*).

El núcleo del DOS (*kernel*) suministra el interfase entre los programas de aplicación y el MS-DOS. . El *kernel* es un programa suplido por *Microsoft Corporation* y proporciona una serie de servicios (independientes del *hardware*) llamados funciones del sistema (*system functions*).

Los programas pueden tener acceso a las funciones del sistema cargando los registros con los parámetros adecuados y transfiriéndolos al sistema operativo por medio de una interrupción por *software*.

El procesador de comandos (*COMMAND.COM*).

El procesador de comandos, es el interfase de usuario con el sistema operativo. Es el responsable de interpretar y ejecutar los diversos comandos, incluyendo la carga y ejecución de los programas de aplicación.

El *shell* por defecto suministrado con el MS-DOS se encuentra en un archivo llamado *COMMAND.COM*. Aunque el *COMMAND.COM* constituye ordinariamente, la total percepción que usuario tiene del MS-DOS, es importante aclarar que el mismo no es el sistema operativo, sino un tipo especial de programa ejecutándose bajo el control del MS-DOS.

El *COMMAND.COM* está dividido en tres partes:

- La residente.
- La de inicialización.
- La temporal (*transient*).

La porción residente es almacenada en memoria, por encima del núcleo, *buffers* y tablas del sistema operativo. Contiene las rutinas encargadas de manejar los procesos **Ctrl-C**, **Ctrl-Break**, **Errores críticos** y la **terminación** de los programas de aplicación. También contiene el código necesario para recargar la porción temporal del *COMMAND.COM* cuando sea necesario.

La sección de inicialización es cargada por encima de la porción residente cuando el sistema arranca, procesa el archivo *AUTOEXEC.BAT* y luego es descartado.

La parte temporal se encarga de leer, interpretar y ejecutar los comandos

Los comandos aceptados por el *COMMAND.COM* están repartidos en tres categorías:

- Comandos internos.
- Comandos externos.
- Archivos *batch*.

Los comandos internos, también llamados intrínsecos, son aquellos llevados a cabo por el código contenido en el propio *COMMAND.COM*. Entre estos tenemos los comandos **Copy**, **Rename**, **Directory** y **Delete**. Las rutinas para los comandos internos están incluidos en la porción temporal del *COMMAND.COM*.

Los comandos externos, son los nombres de programas almacenados en archivos en disco. Antes de que puedan ser ejecutados deben ser cargados desde disco a el área temporal de programas (*transient program área*). Algunos de estos comandos son **Chkdsk**, **Backup** y **Restore**. Tan pronto como terminan su labor, son descartados de la memoria

Los archivos *batch***,** son archivos de texto que contienen listas de comandos internos, externos o de procesamiento por lotes. Estos archivos son procesados por un interpretador especial localizado en la porción temporal del *COMMAND.COM*.

Al momento de interpretar un comando del usuario, el *COMMAND.COM* determina primero si el mismo es un comando interno que pueda ser llevado a cabo directamente. Si no, buscará un comando externo (archivo ejecutable) o un archivo *batch*. La búsqueda se efectúa comenzando por el directorio actual y luego en cada uno de los directorios especificados por el comando *PATH*. En cada directorio inspeccionado, el *COMMAND.COM* buscará el archivo con extensión *COM*, luego con extensión *EXE* y finalmente *BAT*. Si encuentra un archivo con extensión *COM* o *EXE*, el *COMMAND.COM* emplea la función *EXEC* del sistema operativo para cargar y ejecutar el archivo correspondiente. La función *EXEC* crea una estructura especial llamada **Prefijo de Segmento de Programa** (*PSP*), la cual contiene varios enlaces y apuntadores requeridos por los programas de aplicación. Luego carga el programa en si y efectúa todos los ajustes y re localizaciones necesarias para finalmente transferir el control al punto de entrada (*entry point*) del programa. Cuando el programa ha finalizado su ejecución se invoca a una función especial que libera la memoria por él ocupada y retorna el control al proceso que originó su ejecución (*COMMAND.COM* en este caso).

Los programas de aplicación tiene el casi absoluto control de los recursos del sistema mientras se están ejecutando. Las únicas tareas ejecutadas paralelamente son aquellas llevadas a cabo o los manejadores de interrupción y las operaciones requeridas por el programa de aplicación al sistema operativo. El MS-DOS no soporta la ejecución concurrente de varias tareas.

Estructura de los programas de aplicación para MS-DOS.

Los archivos ejecutables en MS-DOS pueden ser de dos tipos: *COM* los cuales tienen un tamaño máximo de aproximadamente 64K y *EXE,* los cuales pueden ser tan largos como la memoria disponible. En el argot *Intel* un programa *COM* cabe en el modelo *TINY*, en el cual todos los registros de segmento contienen el mismo valor, lo cual implica que el código y los datos están mezclados. En contraste, los programas *EXE* se ajustan a los modelos *SMALL*, *MEDIUM* o *LARGE*, en los cuales los registros de segmento contienen valores diferentes, lo cual significa que el código, los datos y la pila se encuentran en segmentos diferentes. Los programas *EXE* pueden tener varios segmentos de código o datos los cuales son manejados a través de saltos largos (*long jumps*) o manipulando el contenido del registro *DS*.

Un programa *COM* reside en disco como una copia fiel del programa, sin encabezado ni información identificadora. Un programa *EXE* está contenido en un tipo especial de archivo con un encabezado individual, un mapa de reasignación de localidades, un código de detección de errores (*checksum*) e información adicional que puede ser usada por el *MS-DOS*.

Ambos tipos de programadas son cargados y ejecutados a través del mismo mecanismo: la función *EXEC*.

El Prefijo de Segmento de Programa (*PSP*).

Una completa comprensión del prefijo de segmento de programa es vital para la escritura de exitosa de programas para MS-DOS. Es una área reservada de 256 bytes ubicada al comienzo del bloque de memoria reservado para un programa de aplicación. El *PSP* contiene ciertos enlaces con el sistema operativo, alguna información que será usada por el propio MS-DOS y otra que será transferida al programa de aplicación.

En las primeras versiones del MS-DOS, el *PSP* fue diseñado para mantener compatibilidad con el área de control de programas, que manejaba el sistema operativo *CP/M*, líder del mercado para ese entonces, de tal forma que los programas pudieran ser exportados a MS-DOS sin demasiadas modificaciones. Aunque desde entonces, el MS-DOS a evolucionado considerablemente, aún la resemblanza de esta estructura con su contraparte *CP/M* es fácilmente reconocible. Por ejemplo, la dirección 0000h dentro del *PSP* contiene un apuntador a la rutina manejadora de terminación de proceso, la cual efectúa todos los ajustes pertinentes al finalizar la ejecución de un programa de aplicación. De manera similar, la dirección 0005h contiene un apuntador la *despachadora de funciones* del DOS, la cual es capaz de realizar una cantidad de servicios, tales como manejo de archivos, consola, teclado, etc. que pudieran ser requeridos por el programa de aplicación.. Estas llamadas, *PSP:0000* y *PSP:0005* tienen el mismo efecto que *CALL 0000* y *CALL 0005* en *CP/M*. Estos enlaces, sin embargo, no son los más recomendados en la actualidad.

La palabra de datos contenida en el desplazamiento 0002h en el *PSP*, contiene la dirección del segmento del último bloque de memoria reservado para el programa de aplicación. El programa puede usar este valor para determinar si requiere asignar más memoria para realizar su trabajo o si por el contrario está en capacidad de liberar alguna de tal manera que pueda ser usada por otros procesos.

En la zona desde 000Ah hasta 0015h dentro del *PSP* se encuentran las direcciones de las rutinas manejadoras de los eventos **Ctrl-C**, y **Error crítico**. Si el programa de aplicación altera estos valores para sus propios propósitos, el MS-DOS los restaura al finalizar la ejecución del mismo.

Estructura del prefijo de segmento de programa.

0000h	Int 20h
0002h	Segmento, final del bloque de asignación
0004h	Reservado
0005h	Invocación FAR a la función despachadora del MS-DOS
000Ah	Vector de Interrupción de terminación (Int 22h)
000Eh	Vector de Interrupción Ctrl-C (Int 23h)
0012h	Vector de Interrupción de error crítico (Int 24h)
0016h	Reservado
002Ch	Segmento del bloque de variables de ambiente
002Eh	Reservado
005Ch	Bloque de control de archivo por defecto (#1)
006Ch	Bloque de control de archivo por defecto (#2)
0080h	Línea de Comandos y Area de transferencia de disco
00FFh	Final del PSP

Figura 5-1

La palabra de datos en el desplazamiento 002Ch contiene la dirección del segmento del bloque de variables de ambiente (*Environment block*), el cual contiene una serie de cadenas *ASCIIZ*. Este bloque es heredado del proceso que causo la ejecución del programa de aplicación. Entre la información que contiene tenemos, el paso usado por el *COMMAND.COM* para encontrar el archivo ejecutable, el lugar del disco donde se encuentra el propio *COMMAND.COM* y el formato del *prompt* empleado por este.

La *cola de comandos*, la cual está constituida por los caracteres restantes en la línea de comandos, después del nombre del programa, es copiado a partir de la localidad 0081h en el *PSP*. La longitud de la esta cola, sin incluir el carácter de retorno al final, está ubicada en la posición 0080h. Los parámetros relacionados con redireccionamiento o *piping* no aparecen en esta porción de la línea de comandos, ya que estos procesos son transparentes a los programas de aplicación.

Para proporcionar compatibilidad con *CP/M*, el MS-DOS coloca los dos primeros comandos en la cola, dentro de los bloques de control de archivo (*FCB*) por defecto en las direcciones *PSP:005Ch* y *PSP:006Ch* asumiendo que puedan ser nombres de archivos. Sin embargo, si alguno de estos comandos son nombres de archivos que incluyen especificaciones del paso, la información colocada en los *FCB* no será de utilidad ya que estas estructuras no soportan el manejo de estructuras jerárquicas de archivos y subdirectorios. Los *FCB* son de muy escaso uso en los programas de aplicación modernos.

El área de 128 bytes ubicado entre las direcciones 0080h y 00FFh en el *PSP* pueden también servir cómo área de transferencia de disco por defecto (*DTA*), la cual es establecida por el MS-DOS antes de transferir el control al programa de aplicación. A menos que el programa establezca de manera explícita otra *DTA*, este será usado como *buffer* de datos para cualquier intercambio de información con disco que este efectúe.

Atención: *Los programas de aplicación no deben alterar la información contenida en el PSP a partir de la dirección 005Ch.*

Estructura de un programa con extensión *COM*.

Los programa con extensión *COM* están almacenados en archivos que contienen una copia fiel del código a ser ejecutado. Ya que no contienen información para reasignación de localidades, son más compactos y son cargados más rápidamente que sus equivalentes *EXE*. EL MS-DOS no tiene manera de saber si un archivo

con extensión *COM* es un programa ejecutable válido. Este simplemente lo carga en memoria y le transfiere el control.

Debido al hecho de que los programa *COM* son siempre cargados inmediatamente después del *PSP* y no contienen encabezado que especifique el punto de entrada al mismo, siempre debe comenzar en la dirección 0100h. Esta dirección deberá contener la primera instrucción ejecutable. La longitud máxima de un programa *COM* es de 65536 bytes, menos la longitud del *PSP* (256 bytes) y la longitud de la pila (mínimo 2 bytes).

Imagen de memoria de un programa COM típico

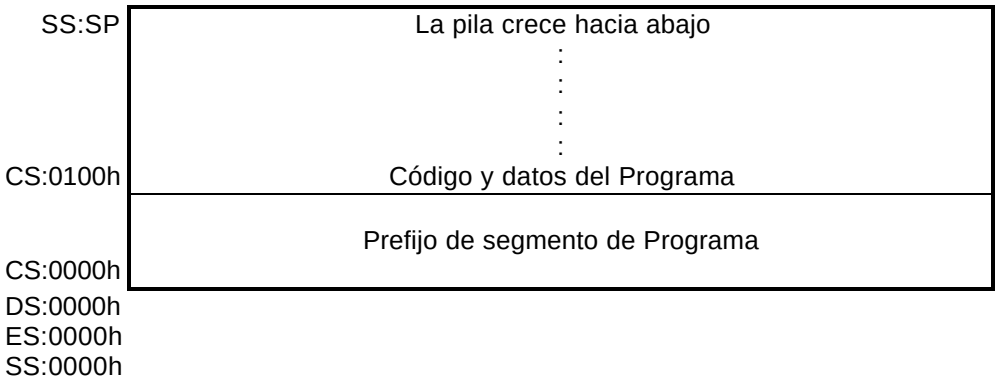


Figura 5-2

Cuando el sistema operativo le transfiere el control a un programa *COM* ,todos los registros de segmento apuntan al *PSP*. El registro apuntador de pila (*SP*), contiene el valor 0FFFEh si la memoria lo permite. En otro caso adopta el máximo valor posible menos dos bytes (El MS-DOS introduce un cero en la pila ante de transferir el control al programa).

Aún cuando la longitud de un programa *COM* no puede exceder los 64Kb, las versiones actuales del MS-DOS reservan toda la memoria disponible. Si un programa *COM* debe ejecutar otro proceso, es necesario que el mismo libere la memoria no usada de tal manera que pueda ser empleada por la otra aplicación.

Cuando un programa *COM* termina, puede retornar el control al sistema operativo por varios medios. El método preferido es el uso de la función **4Ch** de la **Int 21h,** la cual permite que el programa devuelva un código de retorno al proceso que lo invocó. Sin embargo, si el programa está ejecutándose bajo la versión 1.00 del MS-DOS, el control debe ser retornado mediante el uso de la **Int 20h** o de las función **00h** de la **Int 21h**.

Un programa *COM* puede ser ensamblado a partir de varios módulos objeto, con la condición de todos ellos empleen los mismo nombres y clases de segmentos y asegurando que el módulo inicial, con el punto de entrada en 0100h sea enlazado primero. Adicionalmente, todos los procedimientos y funciones deben tener el atributo *NEAR*, ya que todo el código ejecutable estará dentro del mismo segmento.

Al enlazar un programa *COM* en enlazador mostrará el siguiente mensaje:

Warning: no stack segment.

Este mensaje puede ser ignorado, ya que el mismo se debe a que se ha instruido al enlazador para que genere un programa con extensión *EXE* donde el segmento de pila debe ser indicado de manera explícita, y no así en los *COM* donde esta es asumida por defecto.

Ejemplo de un programa con extensión *COM*.

El programa *HOLA.COM* listado en la figura 5-3 demuestra la estructura de programa sencillo en lenguaje ensamblador, destinado a ser un archivo ejecutable con extensión *COM*.

Listado del programa HOLA.COM

```
;           HOLA.COM:       Demuestra la estructura de un programa .COM escrito en lenguaje ensamblador.
StdOut              EQU     1                   ; Dispositivo estándar de salida
CR                  EQU     0Dh                 ; Retorno de carro
LF                  EQU     0Ah                 ; Salto de línea
                    ASSUME CS:_TEXT,DS:_TEXT,ES:_TEXT,SS:_TEXT
_TEXT               SEGMENT WORD PUBLIC 'CODE'
                    ORG 100h            ; Los programas COM siempre comienzan en la dirección 100h.
Inicio:             jmp Continuar               ; Punto de entrada al programa. Primera instrucción ejecutable.
Mensaje             DB CR,LF,'Hola',CR,LF       ; Mensaje a mostrar.
LongMens            = $ - Mensaje               ; Longitud del mensaje en bytes.
Continuar:          mov ah,40h                  ; Función 40h: escribir.
                    mov bx,StdOut               ; Manejador del dispositivo estándar de salida.
                    mov cx,LongMens             ; Longitud del mensaje.
                    mov dx,OFFSET Mensaje       ; Dirección de la variable que contiene el mensaje.
                    int 21h                     ; transferir al MS-DOS.
                    mov ax,4C00h                ; Terminar; código de retorno cero.
                    int 21h                     ; Transferir al MS-DOS.
_TEXT               ENDS
                    END Inicio
```

Figura 5-3

Estructura de un programa con extensión *EXE*.

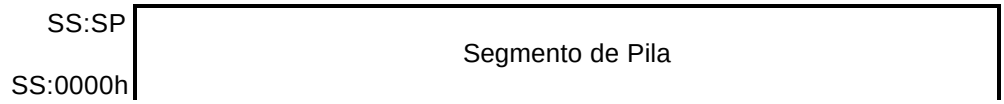
En el punto anterior hemos discutido la estructura de un programa en lenguaje ensamblador, de tal forma que pueda ser convertido en un archivo ejecutable *COM*. La estructura del mismo es tan simple, que el programador puede concentrarse en la lógica del mismo sin mayores preocupaciones en el control del ensamblador. Sin embargo, las limitaciones de los programas *COM* son tales que la gran mayoría de las aplicaciones escritas en lenguaje ensamblador se hacen de tal manera que puedan ser convertidas en archivos ejecutables *EXE*.

A diferencia de los programas *COM,* que se encuentran restringidos a un espacio de memoria de sólo 64K donde deben coexistir código, datos y pila, los programas *EXE* son prácticamente ilimitados en tamaño (el límite lo dictamina la memoria disponible en el equipo). Además, los programas *EXE* pueden colocar el código, datos y pila en distintos segmentos de la memoria. La oportunidad de colocar las diversas partes de un programa en fragmentos diferentes de memoria y la establecer segmentos de memoria con solamente código que pudieran ser compartidos por varias tareas, es un significativa para ambientes multitarea tales como el *Microsoft Windows*.

El cargador del MS-DOS, sitúa al programa EXE, inmediato después del *PSP*, aunque el orden de los segmentos que lo constituyen puede variar. El archivo *EXE* contiene un encabezado, o bloque de información de control, con un formato característico (Figura 5-5). El tamaño de dicho encabezado puede variar dependiendo del número de instrucciones que deben ser re localizadas al momento de la carga del programa, pero siempre será un múltiplo de 512.

Antes de que el MS-DOS transfiera el control al programa, se calculan los valores iniciales del registro de segmento de código (*CS*) y el apuntador de instrucciones (*IP*) basados en la información sobre el punto de entrada al programa, contenida en el encabezado del archivo *EXE.* Esta información es generada a partir de la instrucción **END** en el módulo principal del programa fuente. Los registro de segmento de datos y segmento extra son inicializados de tal manera que apunten al *PSP* de tal manera que el programa pueda tener acceso a la información allí contenida.

Imagen de Memoria de un programa EXE típico.



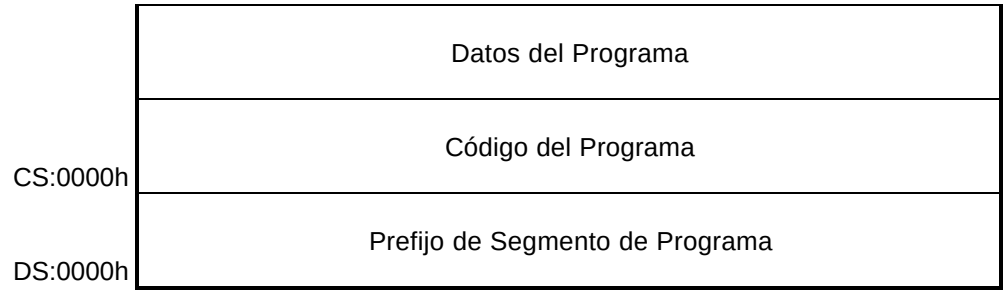


Figura 5-4

Formato de un archivo de carga EXE.

0000h	Primera parte del identificador de archivo EXE (4Dh)
0001h	Segunda parte del identificador de archivo EXE (5Ah)
0002h	Longitud del archivo MOD 512
0004h	Tamaño del archivo, en páginas de 512 bytes, incluyendo encabezado
0006h	Número de items en la tabla de relocalizaciones
0008h	Tamaño del encabezado en párrafos (16 bytes)
000Ah	Número mínimo de párrafos requeridos para el programa
000Ch	Máximo número de párrafos deseables para el programa
000Eh	Desplazamiento del segmento del módulo de pila
0010h	Contenido del apuntador de pila al comenzar el programa
0012h	Suma de chequeo
0014h	Contenido del apuntador de instrucciones al comenzar el programa
0016h	Desplazamiento del segmento del módulo de código
0018h	Desplazamiento del primer item en la tabla de relocalizaciones
001Ah	Número de *overlay* (0 para la parte residente del programa)
001Bh	Espacio reservado (longitud variable)
	Tabla de relocalizaciones
	Espacio reservado (longitud variable)
	Segmentos de programa y datos
	Segmento de pila

Figura 5-5

El contenido inicial del registro de segmento de pila y del apuntador de pila provienen también del encabezado del archivo. Esta información es derivada de la declaración del segmento de pila efectuada mediante la sentencia **STACK.** El espacio reservado para la pila puede ser inicializado o no dependiendo de la manera como este haya sido declarado. Puede ser conveniente en muchos casos inicializar el segmento de pila con un patrón de caracteres predeterminado que permitan su posterior inspección.

Cuando el programa *EXE* finaliza su ejecución debe retornar el control al sistema operativo mediante la función 4Ch de la Int 21h. Existen otros métodos, pero no ofrecen ninguna ventaja y son considerablemente menos convenientes (generalmente requieren que el registro *CS* apunte al segmento del *PSP*).

Un programa *EXE* puede ser construido a partir de varios módulos independientes. Cada módulo puede tener nombres diferentes para el segmento de código y los procedimientos puede llevar el atributo *NEAR* o *FAR*, dependiendo del tamaño del programa ejecutable. El programador debe asegurarse de que los módulos a ser

enlazados sólo tengan una declaración de segmento de pila y que haya sido definido un único punto de entrada (por medio de la directiva *END*). La salida del enlazador es un archivo con extensión *EXE* el cual puede ser ejecutado inmediatamente.

Segmentos de Programa.

El término *segmento* se refiere a dos conceptos individuales de programación: segmentos físicos y segmentos lógicos.

Los *segmentos físicos* son bloques de memoria de 64K. Los microprocesadores 8086/8088 y 80286, poseen cuatro registros de segmento, los cuales son empleados como apuntadores a estos bloques (los microprocesadores 80386 y 80486 contienen seis registros de segmento). Cada registro de segmento puede apuntar a un bloque de memoria diferente. Cualquier programa puede direccionar cualquier localidad de memoria manipulando el contenido de los registros de segmento pero sólo puede señalar a 256K simultáneamente.

Como se ha discutido anteriormente, los programas *COM* asumen que todos los registros de segmento apuntan al mismo bloque de memoria. Esta es la razón por la cual están limitados a 64K. Los programas *EXE* pueden direccionar diversos segmentos físicos, por lo que su tamaño máximo práctico está limitado sólo por la cantidad de memoria disponible.

Los *segmentos lógicos* son los componentes del programa. Cualquier programa *EXE* tiene un mínimo de tres segmentos lógicos declarados: un segmento de código, un segmento de datos y un segmento de pila. Los programas que posean más de 64K de código o datos, podrán tener más de un segmento lógico de código o datos.

Los segmentos son declarados mediante las directivas *SEGMENT* y *ENDS* de la siguiente forma:

```
nombre      SEGMENT atributos
.
.
.
nombre      ENDS
```

Entre los atributos de un segmento tenemos el tipo de alineación (*BYTE*, *WORD* o *PARA*), combinación (*PUBLIC*, *PRIVATE*, *COMMON* O *STACK*), y la clase. Los atributos de segmento son usados por el enlazador para crear los segmentos físicos de un programa ejecutable.

Los programas son categorizados dentro de un *modelo de memoria*, dependiendo del número de segmentos de código y de segmentos de datos que contenga. El modelo de memoria más usado cuando se programa en lenguaje ensamblador es el modelo *SMALL*., el cual contiene un segmento de código y un segmento de datos, pero también están disponibles los modelos *MEDIUM*, *COMPACT* y *LARGE* (Figura 5-6).

Modelos de Memoria empleados en programación en lenguaje ensamblador.

Modelo	***Segmentos de Código***	***Segmentos de Datos***
Small	Uno	Uno
Medium	Varios	Uno
Compact	Uno	Varios
Large	Varios	Varios

Figura 5-6

Existen para cada modelo de memoria, convenciones acerca de los nombres de los segmentos, los cuales son adoptados por la mayoría de los compiladores de lenguajes de alto nivel. Es conveniente adoptar estas convenciones al escribir programas en lenguaje ensamblador, ya que esto facilitará la labor de integración entre estas y rutinas escritas en un lenguaje de alto nivel tal como C o PASCAL..

Herramientas de Programación

Para escribir programas en lenguaje ensamblador, se requieren al menos tres herramientas: un editor de textos ASCII con el cual escribir el programa fuente, un ensamblador, con el cual transformamos el programa fuente en código objeto y un enlazador con el cual finalmente obtenemos un archivo ejecutable. En la actualidad, contamos con algunas herramientas adicionales, que hacen más eficiente la escritura de programas confiables, tales como los depuradores, los cuales facilitan la detección de errores de lógica, otras que permiten el desarrollo de aplicaciones modulares, tales como la herramienta *MAKE*, la cual automatiza el proceso de ensamblaje y enlace de programas con varios módulos fuente y las que nos permiten agrupar y manejar cómodamente, grupos de rutinas tales como los manejadores de librerías. En este capítulo estudiaremos estas herramientas antes mencionadas.

Editor de Textos.

Para la preparación del archivo fuente, se requiere el uso de un editor ASCII. Se conocen como editores ASCII a aquellos que generan archivos que representan un copia fiel de lo escrito por el usuario, lo cual quiere decir, que el editor no incluye tipo alguno de caracteres de control o formato.

Entre los editores de textos ASCII más conocidos tenemos el Editor Norton, Brief, QEdit, SideKick (*notepad*), WordStar en modo no documento y Edit (del MS-DOS 5.0).

Ensamblador.

El ensamblador, es un programa que procesa un archivo fuente en lenguaje ensamblador y produce código objeto. El ensamblador es capaz de detectar y notificar los errores de sintaxis y deletreo de comandos.

Turbo Assembler.

El Turbo Assembler (*TASM*) es un programa ensamblador de línea de comandos, multi-pase, con resolución de referencias hacia adelante (*forward reference*), capaz de ensamblar hasta 48000 líneas por minuto (en una IBM PS/2 modelo 60), 100% compatible con Macro Assembler y el modo Ideal que extiende las capacidades de ciertas sentencias. Soporta los microprocesadores 8086, 286, 386 y 486. Proporciona un conjunto de directivas simplificadas, extensiones para las instrucciones *PUSH* , *POP* y *CALL*, etiquetas locales, estructuras y uniones, directivas anidadas e inclusión de información para depuración con Turbo Debugger

El formato general para ejecutar el TASM desde la línea de comandos es el siguiente:

TASM conjunto de archivos [; conjunto de archivos]...

El punto y coma (;) después del primer corchete ([) permite ensamblar varios grupos de archivos separados entre si, de una sola vez. Es posible incluso, especificar parámetros de ensamblaje diferentes para cada grupo de archivos. Por ejemplo:

TASM /e Archivo1; /a Archivo2

ensamblará el Archivo1 con la opción **/e** y el Archivo 2 con la opción **/a**.

La forma general de un conjunto de archivos es:

[opción]... archivo fuente [[+] archivo fuente]...,[,archivo objeto] [,archivo listado], [,archivo de referencia cruzada]

La sintaxis muestra que cada grupo de archivos puede comenzar con las opciones que se desean aplicar al grupo, seguida de los nombres de los archivos que se desean ensamblar. Un nombre de archivo puede incluir los comodines (*,?) aceptados por el MSDOS. Si el nombre de archivo no incluye extensión el TASM asume la extensión .ASM. Por ejemplo, para ensamblar todos los archivos en el directorio actual, tipee:

TASM *

Si desea ensamblar varios archivos, puede separarlos con el signo más (+):

TASM Archivo1 + Archivo2

Es posible especificar opcionalmente, el nombre del archivo objeto, del archivo de listado y del archivo de referencia cruzada. Si no se especifica el nombre del archivo objeto o del archivo de listado, el TASM nombra al archivo objeto con el mismo nombre del fuente, pero con extensión .OBJ.

El archivo de listado no se generará a menos que se explícitamente solicitado. Si no se especifica el nombre del archivo de listado, el TASM lo nombra con el mismo nombre del archivo fuente, pero con extensión .LST.

De igual forma, el archivo de referencia cruzada no será generada a menos que sea solicitado de manera explícita. Si no se da el nombre, se adopta el mismo del archivo fuente, con extensión .XRF.

Si decide aceptar el nombre por defecto del archivo objeto, pero desea especificar el nombre del archivo de listado, debe tipear el comando de la siguiente forma:

TASM Archivo1,,Prueba

Serán generados los archivos Archivo1.obj y Prueba.lst.

Opciones.

Las opciones del TASM, le permiten controlar la manera como trabajará el ensamblador y como será será mostrada la información en pantalla, archivo de listado y archivo objeto. En el Apéndice D, se describen todos las opciones disponibles.

Archivo de Listado.

El archivo de listado se refiere a la opción de generar un archivo donde se mezcla el código fuente con el código de máquina resultante, expresado en hexadecimal. Este listado provee además información relacionada con los símbolos y segmentos empleados en el programa.

Archivo de Referencia Cruzada.

El archivo de referencia cruzada (extensión .XRF), contiene información acerca de cada símbolo empleado en el programa mostrando el número de línea donde fue declarado y todos donde se haga referencia a él.

Enlazador.

El programa enlazador, se encarga de combinar uno o más archivos objeto en uno ejecutable. Además lleva a cabo otras tareas, que permiten organizar grandes programas en pequeños módulos fuente. . En estos casos, el código de un programa fuente, frecuentemente hace referencia a variables y funciones definidas en otros módulos fuente. Cuando el ensamblador genera un archivo objeto, inserta en el mismo, información que

permite al enlazador determinar las referencias externas de cada módulo y resolverlas en el proceso de enlace.

Turbo Link.

El programa Turbo Link (*TLINK*) es un enlazador de línea de comandos, veloz y compacto. TLINK combina módulos objeto y librerías para producir archivos ejecutables.

La sintaxis general de TLINK es:

TLINK archivos objeto, ejecutable, mapa, librerías, definición de módulo

Los archivos deben ser dados en el orden especificado, separados por comas. Por ejemplo:

TLINK /c mainline wd ln tx, fin, mfin, work\lib\comm work\lib\support

hace que TLINK enlace los módulos objeto MAINLINE.OBJ, WD.OBJ, LN.OBJ y TX.OBJ con las librerías COMM.LIB y SUPPORT.LIB que se encuentran en el subdirectorio WORK\LIB. El archivo ejecutable tendrá como nombre FIN.EXE y el archivo mapa MFIN.EXE. No se suministró archivo de definición de módulo. La opción **/c** hace que el enlazador diferencia entre minúsculas y mayúsculas en los símbolos procesados.

Sin no se especifica el nombre del archivo ejecutable, TLINK lo construye a partir del nombre del primer archivo objeto y la extensión acorde con el formato especificado. Si no se indica el nombre del archivo mapa, este es obtenido a partir del nombre del archivo ejecutable, añadiéndole la extensión MAP.

TLINK maneja las siguientes extensiones por defecto:

OBJ para archivos objeto
EXE para archivos ejecutables. Con las opciones **/t** y **/Tdc** las extensión es **COM**.
DLL para librerías de enlace dinámicas.
MAP para los archivos mapa.
LIB para las librerías.
DEF para los archivos de definición de módulo

Todos los nombres de archivo son opcionales, excepto los módulos objeto. Por ejemplo:

TLINK dosapp dosapp2

enlaza los archivos objeto DOSAPP.OBJ y DOSAPP2.OBJ, crea un archivo ejecutable llamado DOSAPP.EXE, un archivo mapa DOSAPP.MAP, no considera librerías y no emplea archivo de definición de módulo

Opciones.

Las opciones pueden ser incluidas en cualquier punto de la línea de comandos. Consisten de un carácter *slash (/*), un carácter (-) y la letra correspondiente a la opción. Estas modifican y configuran ciertos aspectos del procesamiento que realiza el enlazador. Estas opciones son descritas en el apéndice F.

MAKE.

Es una herramienta, que ayuda a mantener actualizado el archivo ejecutable. Muchas aplicaciones constan de varios módulos fuente que deben ser sometidos a preprocesamiento, ensamblaje y enlace. La herramienta MAKE ayuda a la manutención de proyectos de este tipo, sometiendo cada archivo fuente al procesamiento adecuado, cuando haga falta, de tal forma que se mantenga actualizado el archivo ejecutable. Esta es invocada mediante el comando

MAKE makefile

donde *makefile* es el nombre de un archivo que contiene los comandos para MAKE. Para un programa que consista de varios archivos combinados en uno sólo, es posible establecer comandos MAKE que generen los códigos objeto correspondientes e invoquen al enlazador para producir la versión ejecutable. Además de la automatización del proceso, con MAKE se consigue que en cada invocación sean procesados nuevamente sólo aquellos archivos que hayan sido modificados. Esto lo hace verificando las fechas de los archivos involucrados. Un archivo fuente es procesado sólo si fue modificado posteriormente a su correspondiente archivo objeto.

Para usar MAKE es necesario preparar un archivo de texto con los comandos correspondientes. Si a este archivo lo llamamos MAKEPROG, por ejemplo, entonces dichos comandos pueden ser ejecutados de la siguiente manera:

MAKE makeprog

Los comandos de MAKE consisten en reglas de dependencia, que muestran para cada archivo, los archivos del cual el depende. La lista de dependencia es seguida por otra línea que explica como preparar dicho archivo. Por ejemplo, sin un programa llamado TXTIO.ASM depende de un archivo de inclusión llamado MYMACROS.INC, y se desea ensamblarlo con las opciones /v /z /zi, los comandos del archivo MAKE deberán ser especificados como sigue:

```
TXTIO.OBJ:   TXTIO.ASM  MYMACROS.INC
TASM /V /Z /ZI TXTIO;
```

Aquí TXTIO.OBJ es el archivo destino y TXTIO.ASM y MYMACROS.INC los archivos dependientes. La segunda línea explica como obtener el archivo destino a partir de sus dependientes.

Tenemos ahora otro ejemplo donde se ensambla y enlaza un archivo compuesto de varios módulos:

```
##########################################
#           Los comentarios comienzan con #
#

# Opciones del Ensamblador
AOPTS=/ml /m9 /c /z /zi
TASM=tasm $(AOPTS)

# Reglas Generales de Inferencia
# Reglas para obtener los archivos .OBJ a partir
# de los archivos .ASM
.ASM.OBJ:
$(TASM) $*.ASM;

# Ensamblar los Archivos
PROG.OBJ:  PROG.ASM LOCAL.INC COMMON.INC
FILE1.OBJ: FILE1.ASM COMMON.INC
FILE2.OBJ: FILE2.ASM LOCAL.INC

# Crear el archivo ejecutable
PROG.EXE: PROG.OBJ FILE1.OBJ FILE2.OBJ
TLINK /v $**, $@;
```

Este archivo MAKE construye un programa llamado PROG.EXE el cual tiene tres archivos fuente: PROG.ASM, FILE1.ASM y FILE2.ASM. Hay además dos archivos de inclusión, LOCAL.INC y COMMON.INC.

En nuestro ejemplo, PROG.OBJ depende de los archivos fuente PROG.ASM, LOCAL.INC y COMMON.INC; PROG.EXE depende de los archivos objeto PROG.OBJ, FILE1.OBJ y FILE2.OBJ. Nótese el uso de los símbolos predefinidos $** y $@: Existen tres símbolos predefinidos que tienen el siguiente significado:

$* - nombre del archivo destino sin la extensión
$@ - nombre completo del archivo destino

$** - todos los nombres que aparecen en la lista de dependencia.

Depurador.

El programa depurador es un herramienta que facilita la localización de errores en la lógica de un programa.

Turbo Debugger.

El Turbo Debugger (*TD*) es un avanzado depurador a nivel de archivo fuente, diseñado para ser usado en archivos generados por compiladores y enlazadores de la empresa Borland (Turbo Assembler, Turbo PASCAL, Turbo C++, Borland C++, etc.).

Entre las características más importantes tenemos que puede emplear memoria expandida para depurar programas largos, evalúa sentencias y expresiones en Assembler, PASCAL y C, posibilidad de acceso al CPU, macros, ejecución paso a paso, ejecución hacia atrás (*back tracing*), depuración remota (vía canal serial), soporte para tarjetas de depuración por *hardware*, soporte de lenguajes orientados al objeto, capacidad de depuración para programas residentes, *device drivers* y de programas para ambiente Windows.

Para que un programa en lenguaje ensamblador pueda ser depurado empleando Turbo Debugger a su máxima capacidad, es necesario prepararlo, suministrando ciertas opciones de compilación y enlace. A nivel del ensamblador se requiere indicar la opción **/zi** la cual indica que se incluya en el archivo objeto la información para depuración:

TASM programa /zi

Al enlazar es necesario indicar la opción **/v**:

TLINK /v programa

De esta manera, el archivo ejecutable contiene toda la información requerida por Turbo Debugger, para efectuar la depuración del mismo a nivel de archivo fuente.

Una vez que el programa ha sido completamente depurado, se le puede eliminar la información de depuración mediante el empleo del programa TDSTRIP:

TDSTRIP programa

Librerías.

Las librerías son colecciones de archivos objetos, agrupados en una sola unidad. Los programas para manejo de librerías, permiten la creación y mantenimiento de las mismas.

Las ventajas de emplear librerías se resumen en los siguientes puntos: se acelera el proceso de enlace al incluirse todos los módulos objeto en un único archivo, se optimiza el código resultante debido al hecho de que el enlazador extrae de las librerías, sólo aquellas porciones de código realmente empleadas en la aplicación dada y se reduce el espacio consumido en disco debido a que las librerías son de menor tamaño que los módulos fuente u objeto correspondientes.

Turbo Librarian.

El Turbo Librarian (*TLIB*) es una herramienta que permite el manejo de librerías Con TLIB es posible crear librerías, añadir, eliminar, extraer y reemplazar módulos objeto y mostrar el contenido de una librería.

La sintaxis de TLIB es la siguiente:

TLIB librería [/C] [/E] [/Ptamaño] [operaciones] [,archivo de listado]

donde **librería** es el paso y nombre de la librería a procesar, **/C** es la bandera para diferenciación de minúsculas y mayúsculas, **/E** indica la creación de un diccionario extendido, **/Ptamaño** especifica el tamaño de la página, **operaciones** es el conjunto de operaciones que TLIB puede llevar a cabo y **archivo de listado** es el nombre del archivo en el que se mostrará el contenido de la librería.

Operaciones.

TLIB reconoce tres símbolos de acción: **-**, **+** y *****. Estos individualmente y combinados producen las siguientes operaciones:

Símbolo	Nombre	Descripción
+	Añadir	Añade el archivo indicado a la librería. Si no se indica la extensión, TLIB asume OBJ. Si el archivo es una librería, su contenido es añadido a la librería especificada.
-	Remover	TLIB elimina el modulo indicado de la librería. Si el módulo no existe en la librería, TLIB muestra el mensaje correspondiente.
*****	Extraer	TLIB crea el archivo especificado copiando el contenido del mismo de la librería al archivo. Si el módulo no existe TLIB muestra el mensaje correspondiente y el archivo no es creado
-* ***-**	Extraer y Remover	TLIB copia el módulo especificado de la librería al archivo correspondiente y lo elimina de la librería.
-+ **+-**	Reemplazar	TLIB reemplaza el módulo especificado por el archivo correspondiente.

Por ejemplo:

TLIB mylib +x +y -+z

añade los módulos **x** y **y** a la librería **mylib** y reemplaza el módulo **z.**

Programación en Lenguaje Ensamblador

Archivos Fuente, Objeto y Ejecutable.

Después que es creado un programa en ensamblador, este debe ser almacenado en un archivo. Este archivo, que recibe el nombre de **archivo fuente**, no es más que archivo de texto que contiene sentencias en lenguaje ensamblador. Generalmente, el nombre de un archivo fuente en lenguaje ensamblador tiene la extensión **ASM**. Generalmente, los ensambladores asumen esta extensión.

El resultado de ensamblar un archivo fuente es un archivo binario con código de máquina e instrucciones para el enlazador. Este archivo es llamado **archivo objeto** y por defecto tiene la extensión .OBJ.

Uno o más archivos objeto son combinados por el enlazador para formar un archivo ejecutable, cuya extensión por defecto es .EXE. La rutina de carga del MS-DOS reconoce esta extensión. este programa puede ser ejecutado tecleando su nombre, sin extensión, desde la línea de comandos del DOS.

Contenido de un Archivo Fuente.

El programa fuente esta constituido por un archivo de texto, generalmente con extensión ASM, que contiene sentencias dirigidas al microprocesador y al ensamblador.

Tipos de Sentencias Fuente.

Un programa fuente en lenguaje ensamblador contiene dos tipos de sentencias:

INSTRUCCIONES: Son representaciones simbólicas del juego de instrucciones del microprocesador.

DIRECTIVAS, SEUDO-OPERACIONES o SEUDO-OPS: Indican al ensamblador que hacer con las instrucciones y los datos.

Las instrucciones se aplican en tiempo de ejecución. Las directivas se aplican en tiempo de ensamblaje del programa.

Instrucciones.

El formato de una instrucción es el siguiente:

[etiqueta] nombre [operandos] [comentario]

Los corchetes indican que el campo correspondiente es opcional. De los cuatro campos, sólo **nombre** es obligatorio. La instrucción se especifica en un sola línea, y los campos se separan ente si por al menos un espacio en blanco. Por ejemplo,

```
mover:  mov ax,300      ; poner contador

etiqueta                : mover:
nombre instrucción      : mov
```

```
operandos       : ax y 300
comentario      : ; poner contador
```

Campo Etiqueta.

Es el nombre simbólico de la primera posición de la instrucción. Puede tener hasta 31 caracteres. Los caracteres pueden ser:

Letras desde la A hasta la Z (por defecto, no hace distinción entre mayúsculas y minúsculas).
Números del 0 al 9.
Los caracteres especiales '@', '-', '.' y '$'

El primer carácter no puede ser un dígito. El carácter sólo puede ir precediendo el nombre de la etiqueta. No está permitido el uso de los símbolos:

AX, AH, AL, BX, BH, BL, CX, CH, CL, DX, DH, DL, DI, SI, DS, ES, SP, BP.

Tampoco es permitido el uso de los nombres de las instrucciones como etiquetas.

Si la etiqueta tiene como sufijo el carácter ':', significa que la misma tiene atributo NEAR. Esto quiere decir que va a referenciar a la instrucción dentro del mismo segmento de código. Si por el contrario, el sufijo ':' no está presente, significa que el atributo de la etiqueta es FAR lo cual quiere decir que va a hacer referencia a la instrucción desde otro segmento de código.

Campo Nombre.

Es el nombre simbólico de la instrucción. Consta de dos a seis letras. Por ejemplo:

in, add, push, pushf, loopne

Campo Operandos.

Indica donde se encuentran los datos con los que ha de operar la instrucción señalada en el campo nombre. Pueden haber 0, 1 o 2 operandos. En el caso de que existan dos operandos, el primero se llama destino y el segundo fuente y deben ir separados por una coma. Por ejemplo:

```
pushf       ; ningún operando
push ax     ; un operando
mov  ax,bx  ; dos operandos (destino=ax, fuente=bx)
```

Tipos de Operandos.

Los operandos de las instrucciones pueden ser:

Registro:	r8	indica registro de 8 bits.
	r16	indica registro de 16 bits.
	r	indica registro de 8 o 16 bits.
Memoria:	m8	indica byte (8 bits).
	m16	indica palabra (16 bits).
	m	indica byte o palabra.
Valor Inmediato:	inm8	indica valor que puede almacenarse en 8 bits
	inm16	indica valor que puede almacenarse en 16 bits.
	inm	indica valor que puede almacenarse en 8 o 16 bits.

Campo Comentario.

Debe empezar por el carácter ';' y contiene cualquier texto que el programador desee añadir al programa con la finalidad de aclarar la lectura del mismo. Por ejemplo:

```
mov ax,0        ; borrar registro
```

También es permitido especificar un línea completa como comentario, iniciándola con el carácter ';'.

Directivas.

El formato de una sentencia directiva es el siguiente:

```
[etiqueta]  nombre  [operandos]  [comentario]
```

De los cuatro campos, sólo **nombre** es obligatorio. La directiva se especifica en una sola línea. Los campos se separan entre si por al menos un espacio en blanco. Por ejemplo:

```
CR      EQU 13          ; retorno de carro (asigna a CR el valor 13)
```

La sintaxis del campo **etiqueta** difiere del correspondiente en el caso de las instrucciones en el hecho de que no existe el sufijo ':'. Los campos **nombre** y **operandos** contendrán identificadores de directivas y operandos de directivas respectivamente. El campo **comentario** es análogo al de las instrucciones.

Constantes y Operadores en Sentencias Fuente.

Las sentencias fuente (tanto instrucciones como directivas) pueden contener constantes y operadores.

Constantes.

Existen cinco tipos:

Binario	Ej.	1011b
Decimal		Ej. 129d (la letra d es opcional)
Hexadecimal	Ej.	0E23h (no puede empezar por letra)
Octal	Ej.	1477q (puede usarse o en vez de q)
Carácter	Ej.	"ABC" (comillas simples o dobles)
		'Dije "Bueno Días"'

El sufijo puede ser en mayúscula o minúscula. Es posible especificar números negativos. Si el número es decimal se precede del signo menos. Si es binario, hexadecimal u octal se especifica su complemento a dos.

Operadores.

Un operador es un modificador que se usa en el campo de operandos de una sentencia ensamblador. Se pueden utilizar varios operandos y combinaciones de ellos en una misma sentencia. Hay cinco clases de operadores:

Aritméticos: Operan sobre valores numéricos.

Lógicos: Operan sobre valores binarios bit a bit.

Relacionales: Comparan dos valores numéricos o dos direcciones de memoria dentro del mismo segmento y produce 0 si la relación es falsa y 0FFFFh si la relación es verdadera.

De Retorno de valores: Son operadores pasivos que suministran información acerca de las variables y de las etiquetas del programa.

De Atributos: Permiten redefinir el atributo de una variable o de una etiqueta. Los atributos para variables de memoria son:

BYTE = byte
WORD = palabra
DWORD = doble palabra
TBYTE = diez bytes

Los atributos para etiquetas son:

NEAR = se puede referenciar sólo dentro del segmento donde está definida.
FAR = se puede referenciar desde fuera del segmento donde está definida.

Tipos de Directivas

Las directivas o seudo operaciones se pueden dividir en cuatro grupos funcionales:

Directivas de Datos.
Directivas Condicionales.
Directivas de Listado.
Directivas de Macros.

Directivas de Datos.

Las directivas de datos están divididas en seis grupos.

Definición de Símbolos: Sirven para asignar nombres simbólicos a expresiones. Una vez definido el símbolo, puede ser usado en lugar de la expresión equivalente. Dentro de este grupo tenemos las directivas = y EQU.

Definición de Datos: Permiten reservar memoria para las variables del programa. Opcionalmente se puede dar un valor inicial a cada variable. En este grupo tenemos a DB, DW, DD, DQ y DT.

Referencias Externas: Hacen referencia a información que se encuentra en módulos o archivos distintos. Dentro de esta categoría tenemos las directivas PUBLIC, EXTRN e INCLUDE.

Control del Ensamblador: Controlan aspectos diversos del ensamblaje de un archivo fuente. Aquí encontramos a END, ORIGIN, EVEN y .RADIX.

Definición de Segmentos y Procedimientos: Permiten declarar segmentos y procedimientos dentro de un archivo fuente. En este grupo encontramos las directivas SEGMENT, ENDS, ASSUME, PROC y ENDP.

Definición de Bloques: Permiten definir bloques funcionales. En esta categoría tenemos las directivas GROUP, NAME, LABEL, RECORD y STRUC.

Directivas Condicionales.

Sirven para condicionar la compilación de ciertas porciones de un programa fuente. La sección del programa sometida a compilación condicional comenzará con una directiva **IF** y terminará con una **ENDIF**.

En este grupo tenemos a las directivas IFxxx, ENDIF y ELSE, donde xxx denota la condición a la cual estará sujeta la compilación de la porción de programa fuente involucrada.

Directivas de Listado.

Le indican al ensamblador la información que se desea aparezca en el listado de salida y el formato de dicha información.

Formato de Listado: Permiten indicar el formato de las páginas del listado de salida. En este grupo tenemos a las directivas PAGE, TITLE y SUBTTL.

Listado de Macros: Informan al ensamblador acerca de como deben ser mostradas las expansiones de los macros en los listados de salida. En esta categoría encontramos a .LALL, .SALL y .XALL.

Control del Listado: Establecen la información que debe aparecer en el listado de salida. Aquí tenemos a .XCREF, .CREF, XLIST y .LIST.

Comentarios: Permiten incluir comentarios en el archivo fuente. COMMENT es la única directiva en esta categoría.

Mensajes: Le indican al ensamblador, que emita mensajes durante el proceso de ensamblaje. %OUT es la única directiva en este grupo.

Control del Listado de los Bloques Asociados a una Condición Falsa: Están relacionadas con la aparición en el listado de salida de porciones del programa fuente no compiladas por estar asociadas a condiciones de compilación falsas. En este grupo tenemos a .LFCOND, .SFCOND y .TFCOND.

Directivas de Macros.

En este grupo encontramos las directivas disponibles para definición de macros. Las directivas de macros se dividen en dos categorías: definición de macros y operadores de macros.

Definición de Macros: En este grupo encontramos las directivas MACRO, ENDM, LOCAL, EXITM, PURGE, REPT, IRP e IRPC.

Operadores de Macros: En esta categoría encontramos las directivas &, ;;, ! y %.

Macros, Procedimientos y Herramientas para Programación Modular.

Los lenguajes de alto nivel son muy populares debido en gran parte al hecho de que facilitan la programación haciéndola más fácil y rápida. En lenguaje ensamblador, aun la tarea más simple, requiere de muchas instrucciones, y el código de un programa completo puede resultar intimidante. El Turbo Assembler proporciona facilidades que permiten manejar la codificación por bloques.

Entre las características más importantes que ofrece el Turbo Assembler tenemos los **macros**, la **compilación condicional**, la **combinación de archivos** y los **procedimientos**. Estos cuatro aspectos se relacionan con el manejo del código por bloques (programación estructurada). Algunos bloques se repetirán a lo largo de todo el programa; para este trabajo deben considerarse los macros. Es posible establecer instrucciones *IF THEN ELSE* que habiliten determinadas secciones de código de acuerdo con una condición particular: compilación condicional. Un programa puede ser desarrollado en varios archivos, cada uno con un propósito específico, y luego incluir o enlazar estas rutinas con el programa principal. Finalmente, el código puede ser escrito en procedimientos, ubicados en un sólo lugar, e invocándolo desde distintas partes del programa.

Macros.

Usando la directiva **MACRO** es posible asignar un nombre simbólico a un bloque de código y usar ese nombre cada vez que se necesite incluir el bloque de código correspondiente. Cuando el programa es compilado, el ensamblador reemplaza cada ocurrencia del nombre por el bloque de código asociado. Pero la capacidad de un macro va mucho más alla de simplemente servir como abreviatura para secciones de código. La clave de la versatilidad de los macros está en el hecho de que puedan aceptar argumentos, y de que estos argumentos sean reemplazados en cada invocación por los valores actuales de los mismos. Esto quiere decir, que el código generado por un macro puede cambiar dependiendo de los argumentos. Para definir un macro use un identificador, seguido de la directiva MACRO y de los argumentos necesarios, luego escriba el bloque de código correspondiente y finalice con la directiva ENDM.

En un programa en lenguaje ensamblador escrito para MS- DOS, una situación típica puede ser el uso repetitivo de funciones del DOS tales como leer el teclado, o escribir un carácter por pantalla. Por ejemplo, el leer un carácter del teclado involucra almacenar un 1 en AH y ejecutar la interrupción 21h del DOS. Luego que la interrupción retorna, AL contiene el código ASCII de la tecla presionada. Ya que es muy probable que se tengan que efectuar múltiples lecturas del teclado en un programa, pudiéramos crear un macro que contenga el código necesario para efectuar dicha tarea:

```
@getchr     MACRO
            mov ah,01h      ; función 01h del DOS
            int 21h         ; para leer el teclado
            ENDM
```

Ahora, cada vez que sea necesario leer del teclado, se puede emplear el identificador @getchar para efectuar el trabajo. Durante la compilación, el ensamblador reemplazará cada aparición de @getchar, por el código correspondiente. Este ejemplo ilustra el uso más simple que podemos hacer de un macro. Pero los macro pueden hacer mucho más. Consideremos el caso de un macro que llamaremos @putchar, para imprimir un carácter por pantalla, usando la función 2 del DOS. El macro luce simple: almacenar el número 2 en AH, el carácter a imprimir en DH e invocar la interrupción 21h. Pero si nos detenemos a pensar por un momento, llegamos a la conclusión de que debemos prever alguna manera de informar al macro sobre el carácter a imprimir. Afortunadamente, los macros en Turbo Assembler pueden tomar argumentos. De hecho, la sintaxis general para la declaración de un macro es la siguiente:

```
nombre_macro    MACRO  arg_1, arg_2, ..., arg_n
                ; cuerpo del macro
                ENDM
```

donde **nombre_macro** es el identificador del macro, y la directiva macro es seguida por una lista de argumentos separados por comas. Estos argumentos pueden ser usados en el cuerpo del macro de acuerdo a la necesidad. Los argumentos actúan como casilleros. En la expansión del macro, el ensamblador sustituye cada argumento por su valor actual especificado en la invocación del macro. De esta forma, si nombre_macro toma dos argumentos, una invocación como esta

```
nombre_macro    X, 2
```

resultaría en el uso de X y 2 en los lugares de arg_1 y arg_2 respectivamente. Macros sencillos podrían no tomar argumentos (como el caso de @getchar), pero la mayoría de ellos no serían de mucha utilidad sin la presencia de argumentos. Ahora que sabemos de argumentos, el macro @putchar puede ser escrito de la siguiente manera:

```
@putchar    MACRO c         ; el argumento c es el carácter a imprimir
            mov dl,c        ; mueve el carácter a DL
            mov ah,02h      ; función 02h del DOS
            int 21h         ; invocación del DOS
            ENDM
```

Una vez que @putchar ha sido definido de esta manera, es posible usarlo de la siguiente forma. Para imprimir el carácter contenido en AL seguido de la letra A mayúscula se escribe:

```
@putchar al     ; imprimir el carácter en AL
@putchar "A"    ; imprimir una A
```

Como se puede ver, los argumentos proporcionan un mecanismo poderoso que permite escribir bloques de código que alteran su funcionamiento para satisfacer las necesidades del momento. Sin embargo es necesario ser cuidadoso de no emplear nombres reservados como identificadores de los argumentos.

Etiquetas Locales.

Hasta los momentos hemos estudiado macros que facilitan la escritura de secciones de código. Existe sin embargo un problema, al usar etiquetas dentro de un macro. Consideremos el macro @getchar que

escribimos anteriormente. Para poder leer todas las teclas disponibles en el teclado de una PC, se requiere efectuar ciertas mejoras. En particular, las teclas que no tienen asociado un código ASCII (tales como las teclas de función y las de cursor), retornan un 0 en AL, cuando se invoca la función 1 de la interrupción 21h. Cuando ocurre esto, es necesario llamar una vez más a la función 1 del DOS, para obtener el código extendido de la tecla. Es deseable que nuestro macro considere esta situación. Para ello debe alterarlo de la siguiente manera:

```
@getchr     MACRO
            mov ah,01h      ; función 01h del DOS
            int 21h         ; para leer el teclado
            cmp al,0        ; ¿ es el código es igual a 0 ?
            jnz done        ; si no, listo
            int 21h         ; es igual a 0, leer otra vez.
done:
            ENDM
```

Si este macro es empleado más de una vez, el ensamblador indicará el error **Redefinition of a Symbol**. Esto ocurre porque la etiqueta ***done*** aparece donde quiera que el macro @getchar haya sido invocado. El ensamblador señala error debido a que no puede manejar múltiples definiciones de un símbolo. Afortunadamente, este problema puede ser resuelto fácilmente con un pequeño cambio en el macro. Simplemente insertemos una segunda línea, la sentencia **LOCAL done** en la definición del macro @getchar:

Esto le dice al ensamblador que el símbolo ***done*** es usado localmente dentro del macro y que debe ser reemplazado por un símbolo único en cada expansión del macro. El ensamblador construye estos símbolos únicos añadiendo 4 dígitos hexadecimales a dos signos de interrogación. En la primera invocación del macro @getchar el símbolo será ??0000, en el lugar de ***done***; el segundo será ??0001 y así sucesivamente.

```
@getchr     MACRO
            LOCAL done
            mov ah,01h      ; función 01h del DOS
            int 21h         ; para leer el teclado
            cmp al,0        ; ¿ es el código igual a 0 ?
            jnz done        ; si no, listo
            int 21h         ; es igual a 0, leer otra vez.
done:
            ENDM
```

Directivas de Repetición.

Los macros pueden hacer muchas cosas interesantes. Cuando se estudian las directivas relacionadas con la definición de macros, se descubre que con ellos es posible controlar la mayoría de los aspectos relacionados con disposición de datos y generación de código necesarios para programar en ensamblador. Se puede pensar en los macros como en una especie de lenguaje de programación para el ensamblador. En otras palabras, los macros son usados para dar instrucciones al ensamblador tales como reservar memoria para un cierto número de variables, o incluir un bloque de código determinado.

Las directivas de repetición son útiles cuando se requiere repetir un bloque de código, un número específico de veces, por cada ocurrencia de un argumento. Por ejemplo, supongamos que necesitamos reservar un arreglo de tamaño dado, inicializado con los valores 0 hasta número de bytes - 1. Se puede escribir un macro que lleve a cabo esta tarea, usando la directiva REPT:

```
byte_array  MACRO n
            value = 0
                    REPT n
                    DB value
                    value = value + 1
                    ENDM
            ENDM
```

Si invocamos este macro como **byte_array 4**, por ejemplo, el ensamblador comienza inicializando el símbolo **value** a 0. Entonces el grupo de instrucciones entre REPT y ENDM se repite cuatro veces, resultando la siguiente secuencia de directivas DB:

```
DB      0
DB      1
DB      2
DB      3
```

Este es un arreglo de 4 bytes inicializados con los valores del 0 al 3.

Para usar este arreglo en un programa, es necesario un nombre; para ello se emplea la directiva **LABEL** justo antes de la invocación al macro. Por ejemplo, el código siguiente reserva espacio e inicializa un arreglo de 128 bytes llamado **array128**:

```
array128        LABEL BYTE
                byte_array 128
```

Aun cuando en este caso la directiva REPT está incluida dentro de un macro, ella puede ser usada por si sola.

Mientras que REPT indica al ensamblador que repita incondicionalmente un conjunto de instrucciones, las directivas IRP e IRPC repiten el bloque por cada miembro de una lista de argumentos. IRP puede ser usado de la siguiente manera:

```
baudrates LABEL WORD                ; para accesar las
          IRP x,<300,1200,2400>     ; variables
          DW  x
          ENDM
```

Este ejemplo crea tres palabras inicializadas con los valores 300, 1200 y 2400 respectivamente. La directiva IRP asigna valores al argumento **x**, uno por uno, de la lista de parámetros, separados por comas, que están encerrados entre los signos de mayor y menor que (<...>) y repite el bloque de código, hasta la directiva ENDM, para cada valor. Nótese, que los signos mayor y menor que, son parte de la sintaxis de la directiva IRP.

La directiva IRPC es similar a IRP, pero repite el bloque de instrucciones por cada carácter de una cadena. He aquí un ejemplo. Supongamos que escribimos un programa que toma 17 comandos de un sólo carácter: las letras desde la a hasta la q. Se requiere comparar un carácter de entrada con cada uno de los comandos y tomar la acción adecuada en cada caso. El código que identifique el comando puede ser escrito con la directiva IRPC, de la siguiente manera:

```
                    ; asumamos que en AL está el carácter de entrada
          mov bx,-1
          IRPC x,<abcdefghijklmnopq>
          cmp al,"&x"
          jz is&x
          inc bx
is&x:
          ENDM
          inc bx
                    ; ahora BX contiene el índice del comando
          cmp bx,-1
          jnz command_ok
                    ; comando desconocido
command_ok:
```

Al final de esta comparación, BX tiene la posición (comenzando en cero) de la letra de comando correspondiente al carácter de entrada. Este puede ser usado como índice para una tabla de procedimientos. Hágase notar, que la cadena de caracteres de la directiva IRPC debe estar encerradas entre signos mayor y menor que.

El operador **&** identifica a un argumento incluido en una instrucción, de tal manera que el ensamblador lo sustituya correctamente por su valor. Por ejemplo, en este ejemplo, **cmp al,"&X"** se convierte en **cmp al,"a"** cuando **x** toma el valor **a** y la etiqueta **is&x** es reemplazada por **isa**.

Otras Directivas de Macros.

Hasta los momentos hemos estudiado las directivas básicas para definición de macros y repetición de bloques de código. Existen dos directivas adicionales que pueden ayudar en la construcción de macros.

La primera es **EXITM**. Esta directiva se emplea, en combinación con una directiva condicional, para finalizar prematuramente, la expansión de un macro. Supongamos por ejemplo, que deseamos suprimir la expansión de un macro si su argumento excede al valor 10. El macro se escribiría como sigue:

```
allocate        MACRO howmany
                        IF (howmany GT 10)
                        EXITM
                        ENDIF
                DB howmany DUP (?)
                ENDM
```

La directiva IF ensambla el bloque de instrucciones hasta ENDIF sólo mientras el argumento **howmany** no exceda el valor 10. La directiva EXITM garantiza que el macro no generará código si el argumento toma un valor superior a 10.

La directiva restante, PURGE, hace que el ensamblador olvide la definición de un macro, de tal manera que el mismo pueda ser redefinido. Para eliminar la definición de los macros @getchar y @putchar se procede de la siguiente manera:

```
        PURGE       @getchar, @putchar
```

Directivas de Compilación Condicional.

Los argumentos de los macros y la directivas de repetición ofrecen muchas facilidades, pero es necesario también, verificar que el macro haya sido invocado con los argumentos correctos. Por ejemplo, si el macro @putchar se invoca sin argumentos, el ensamblador mostrará el siguiente error:

warning A4101: Missing data; zero assumed

Lo que quiere decir el ensamblador, es que ya que no fue indicado el argumento, asumirá el valor 0 en la expansión del mismo. Esto no es conveniente para nuestro macro. El ensamblador proporciona las herramientas para verificar que el argumento haya sido o no especificado y tomar la acción correspondiente al caso. Si decidimos imprimir un espacio en blanco, cuando @putchar sea invocado sin argumentos, podemos reescribir nuestro macro de la siguiente manera:

```
@putchar        MACRO c                 ; c es el carácter a imprimir
                        IFB <c>         ; si c está no fue dado
                        mov dl,' '      ; imprimir un espacio en blanco
                        ELSE
                        mov dl,c        ; si no, imprimir c
                        ENDIF
                mov ah,2                ; función 2 del DOS
                int 21h                 ; invocación al DOS
                ENDM
```

Aquí hemos usado la construcción **IFB...ELSE...ENDIF** para detectar si fue dado el argumento **c** y en caso de que no haya sido así, copiar el carácter **espacio en blanco** en el registro DL. SI el argumento fue especificado, el mismo es copiado en DL para ser imprimido. Es de hacer notar, que la sintaxis de la directiva IFB exige la presencia de los signos mayor y menor que alrededor del argumento a chequear. De manera

similar, con la directiva IFNB podemos verificar que un argumento no está en blanco, lo que significa que fue especificado en la invocación del mismo.

Otra situación que requiere de verificación, es el macro **byte_array** que definimos anteriormente. Ya que un byte sólo puede contener valores entre 0 y 255, debemos chequear si el argumento del mismo excede el valor 256. Primero veremos que pasa si se invoca dicho macro de la siguiente manera:

```
byte_array 257
```

Durante el ensamblaje, el ensamblador mostrará el siguiente mensaje de error:

error A2050: Value out of range

Esto ocurre, porque la directiva DB no puede ser usada para inicializar un byte con un valor por encima de 255. Para corregir el problema, es necesario verificar el argumento para reservar el arreglo sólo cuando este no exceda 256. Reescribimos el macro de la siguiente manera:

```
byte_array      MACRO n
                        IF (n LE 256)
                        value = 0
                                REPT n
                                DB value
                                value = value + 1
                                ENDM
                        ELSE
                                IF1
                                %OUT  Valor muy grande: n
                                ENDIF
                        ENDIF
                ENDM
```

Si la condición que sigue a la directiva IF es verdadera (resulta un valor diferente de cero), el ensamblador procesa el conjunto de instrucciones hasta el próximo ELSE (o ENDIF si no se usó ELSE). Si la condición es falsa, se ensamblan las sentencias entre el ELSE y el ENDIF. En este caso, si el argumento sobrepasa el valor 256, el macro imprime un mensaje empleando la directiva %OUT la cual muestra el texto que tiene a continuación. Nótese que la directiva %OUT fue colocada entre dos instrucciones IF1 y ENDIF, para evitar que el mensaje **Valor muy grande:** aparezca durante en los múltiples pases de ensamblaje. La directiva IF1 indica al ensamblador que procese las instrucciones que los siguen, si está ejecutándose el primer pase de compilación. Durante los pases sucesivos, el ensamblador ignora las instrucciones encerradas entre estas dichas directivas.

La directivas condicionales pudieran también ser de utilidad, fuera de los macros. Por ejemplo, podríamos condicionar la ejecución de rutinas empleadas con fines de depuración. Este podría ser el caso:

```
IFDEF DEBUG
call prdebug
ENDIF
```

Si el símbolo **DEBUG** está definido, se produce la invocación a la subrutina **prdebug**. El símbolo DEBUG no tiene por que estar definido en el programa fuente. En vez de esto, el mismo puede ser definido durante el ensamblaje usando la opción /D del ensamblador. Para hacerlo, ensamble el archivo empleando el comando:

TASM /DDEBUG programa

Archivos de Inclusión.

A medida que vaya desarrollando programas en ensamblador, se dará cuenta que existen macros que usará frecuentemente. Típicamente esto incluye macros que ayudan a usar los recursos del DOS y del BIOS, o que sirven de interfaz con lenguajes de alto nivel. No hay necesidad de que estos macros se escriban una y otra

vez en cada archivo. Estos pueden estar recopilados en un sólo archivo llamado, por ejemplo, MYMACROS.INC, y simplemente *incluirlos* cada vez que los necesitemos, con una simple instrucción

```
INCLUDE         MYMACROS.INC
```

La extensión del archivo puede ser cualquiera. La misma idea sirve para constantes (definidas con EQU) y estructuras.

Este mecanismo permite pues, mezclar distintas módulos fuentes para ser procesados por el ensamblador como uno sólo.

Procedimientos.

Los macros le añaden flexibilidad a la tarea de repetir bloques de código. Otra forma de mejorar la productividad es escribiendo procedimientos los cuales van a ser módulos de código que llevan a cabo tareas específicas. A diferencia de los macros, el cuerpo de un procedimiento no es repetido cada vez que se le invoca. En lugar de esto, los procedimientos basan su funcionamiento en las instrucciones **call** y **ret** del microprocesador 80x86.

Los macros, cuando son usados en varias partes de un programa, expanden en cada lugar, incrementando la cantidad de código (el tamaño del programa es mayor). Un procedimiento tiene un único bloque de código, pero involucra el uso de las instrucciones **call** y **ret** lo cual añade tiempo de ejecución al programa (lo hacen más lento). La selección de cual herramienta usar estará de parte del programador y representa una situación de compromiso entre tamaño de código y velocidad de ejecución.

Analicemos el siguiente procedimiento, el cual recibe en AX un número entre 0 y 99 y lo muestra por pantalla en notación decimal.

```
prdigit         PROC NEAR
                push ax         ; Salva los registros usados
                push bx         ; el procedimiento
                push dx
                aam             ; convierte a dígitos BCD
                add ax,3030h    ; convierte a ASCII
                mov bx,ax       ; salva el dígito en BX
                mov ah,2        ; función 02 del DOS
                mov dl,bh       ; dígito más significativo
                int 21h         ; imprime el dígito
                mov dl,bl       ; dígito menos significativo
                int 21h         ; imprime el dígito
                mov dl," "      ; añade un espacio en blanco
                int 21h
                pop dx          ; Recupera los registros usados
                pop bx
                pop ax
                ret
prdigit         ENDP
```

La primera y última lineas son requeridas por el ensamblador para delimitar el inicio y el final del procedimiento, así como el atributo del mismo: NEAR o FAR (NEAR significa que tanto el procedimiento invocado como el invocador residen en el mismo segmento de memoria y FAR el caso contrario). El cuerpo del procedimiento contiene el código encargado de llevar a cabo la función deseada y consta de instrucciones del procesador (al igual que los macros). Este procedimiento emplea la instrucción **aam** para convertir el valor en AX a dos dígitos BCD. Los dígitos BCD son convertidos en sus códigos ASCII correspondientes sumándole 3030h a AX. Esto convierte los dos dígitos a la vez. Luego se emplea la función 02 del DOS, para imprimir los dígitos, uno a la vez. Nótese que los registros usados dentro del procedimiento son salvados al principio y restaurados al final. Esta práctica aísla al procedimiento del resto del programa y permite que sea usado tomando en cuenta el sólo conocimiento acerca de sus entradas y salidas.

Pasos para Escribir un Procedimiento.

El primer paso para escribir un procedimiento es identificar lo que este habrá de hacer. Debe ser algo que sea útil en muchas situaciones diferentes. Rutinas que impriman un *retorno de carro* y un *salto de línea*, o que tomen una cadena ASCIIZ y determine su longitud, o que inicialice el puerto serial para comunicaciones, son candidatos típicos para ser escritos como procedimientos. Luego es necesario determinar las variables de entrada y de salida. Los procedimientos reciben argumentos también, pero el programador decide la manera.

El uso de símbolos y etiquetas en un procedimiento es simple. Sin embargo, hay dos aspectos importantes de tomar en cuenta. La primera es recordar que segmento está asociado con los datos y asegurarse de que DS apunte a dicho segmento. El segundo, es que si el procedimiento requiere de nombres de símbolos nuevos, estos deben ser únicos ya que no deben repetirse en ninguna otra parte de nuestro programa. En este caso, al igual que con los macros, la directiva **LOCAL** puede ayudarnos en la generación de símbolos únicos para cada procedimiento escrito.

Una vez que se haya definido el propósito, las entradas y las salidas, podemos entonces comenzar a escribir el procedimiento. Comience dándole un nombre *descriptivo de su función* y estableciendo un atributo para el mismo (NEAR o FAR), los cual dependerá del modelo de memoria empleado. En algunos modelos de memoria tales como SMALL, sólo existe un segmento de código por lo que todo el código del programa residirá en un único segmento. Esto obliga a que todas las invocaciones de procedimientos sean del tipo NEAR, por lo que éste deberá ser el atributo de los procedimientos en este caso. Si se usa, por el contrario, un modelo de memoria que soporte el uso de múltiples segmentos de código, será necesario asignar el atributo FAR a aquellos procedimientos que puedan ser invocados por una porción del programa ubicado en un segmento de código distinto del correspondiente al procedimiento en cuestión. A continuación se muestra un *template* para procedimientos del tipo NEAR:

```
procname        PROC NEAR
                .
                .       ;cuerpo del procedimiento
                .
                ret     ; retorno
procname        ENDP
```

El procedimiento debe comenzar con instrucciones que salven el contenido de los registros a ser usados en el mismo. Esto se logra almacenándolos temporalmente en memoria. Generalmente se emplea una zona de memoria llamada pila para este almacenamiento temporal. Si este es el caso, la instrucción **push** permite almacenar un valor en la pila y **pop** recuperarlo. El resto del código se encarga de llevar a cabo la tarea encomendada al procedimiento, la cual depende en gran parte de la forma como hayan sido transferidos los argumentos. La mayoría de los procedimiento toman los argumentos, los manipulan y retornan los resultados de acuerdo con convenciones establecidas.

Pase de Parámetros.

Al escribir una rutina, sea esta una función o un procedimiento, debemos resolver el problema de como transferirle los parámetros con los cuales ella ha de trabajar.

A través de Registros.

El método más común y fácil de emplear es el pase de parámetros es a través de los registros del microprocesador. Sus principales virtudes son acceso instantáneo y alta velocidad. Casi todas las rutinas de servicio del DOS intercambian su información de esta manera. Rutinas cortas en ensamblador que sirvan de interfaz con los servicios del DOS usualmente emplean, para manipular la información, los mismo registros requeridos por las funciones que invocan.

Una desventaja de este método está en el número y tamaño limitados de registros. Tendremos inconvenientes al emplear esta técnica, si el número o tamaño de los parámetros excede la cantidad o tamaño de los registros disponibles.

A través de Variables Globales.

Otra alternativa está en el uso de áreas comunes de la memoria, para transferencia de datos. La rutina A coloca su información en la zona de memoria llamada **DATA** y la rutina B busca la suya en dicha área. **DATA** es por lo tanto, área común de datos.

Esta técnica resuelve el problema de limitación de cantidad y tamaño de los parámetros transferidos. El límite los establece ahora, la cantidad de memoria disponible. Adicionalmente, de esta manera la información está siempre disponible para cualquier módulo que la necesite.

Sin embargo, posee una gran desventaja: el uso de zonas comunes de memoria restringe la generalidad y reusabilidad de los módulos escritos conforme a esta técnica. La presencia de un determinado procedimiento dentro de un programa exigiría la existencia del área común de datos. Adicionalmente los distintos procedimientos que compartan el área común deberán establecer mecanismo para no producir corrupción de la información allí almacenada. Otra desventaja, más importante aún que las anteriores, es que no permite la escritura de rutinas recursivas ni reentrantes.

A través de la Pila.

Este es el método más comúnmente usado por los compiladores de lenguajes de alto nivel para pase de parámetros a procedimientos. En este método, todos los parámetros requeridos son almacenados en la pila antes de la invocación. Al ejecutarse el procedimiento, este accesa a la información contenida en la pila. Los diseñadores de la familia de microprocesador 80x86 facilitaron la aplicación de esta técnica al suministrar el registro BP (*base pointer*). El registro BP tiene la maravillosa cualidad de direccionar datos dentro del segmento de pila. Esto quiere decir, que si cargamos dicho registro con el valor apropiado, podemos tener acceso a la información de la pila usando direccionamiento indexado.

Pero, ¿Cual es el valor apropiado para BP?. No será el contenido de SP, ya que este señala a la dirección de retorno de la subrutina sino SP+2 o SP+4. ¿Por qué mas dos o mas cuatro?. Porque en las llamadas a subrutinas NEAR, el procesador almacena en la pila sólo el desplazamiento de la rutina dentro del segmento actual (2 bytes), mientras que las invocaciones FAR se almacena en la pila el desplazamiento y el segmento (4 bytes). El procedimiento o función debe ser codificado de tal manera que haga acceso de las localidades correctas (dependiendo del atributo de la rutina):

NEAR	**FAR**
mov bp,sp	mov bp,sp
mov <1er arg>,[bp+2]	mov <1er arg>,[bp+4]

Es de hacer notar, que si es necesario preservar el contenido del registro BP, como ocurre normalmente, este debe salvado en la pila también, con lo que la dirección del primer argumento cambiará a [BP+4] para rutinas NEAR y a [BP+6] para rutinas FAR. Si no salvamos el contenido de BP dentro del procedimiento, debe hacerlo la rutina invocadora. Sin embargo, por razones de compatibilidad, esta última opción no es recomendable, por lo que nuestro procedimiento debe ser quien preserve el contenido de este registro. A continuación se muestra una estructura recomendable para el pase de parámetros:

```
                                    ; Rutina Invocadora
                :
                push <argumento N>  ; almacena el último argumento
                :
                push <argumento 2> ; almacena el segundo argumento
                push <argumento 1> ; almacena el primer argumento
                call <myproc>       ; invoca al procedimiento
                add  sp,2N          ; limpia la pila
                :

                                    ; Procedimiento Invocado
<myproc>        PROC NEAR           ; ejemplo con rutina NEAR
                push bp             ; salva el contenido de BP
                mov  bp,sp          ; punto de referencia en pila
                :
                mov <dummy>,[bp+4]  ; primer parámetro
```

```
                    mov <dummy>,[bp+6]          ; segundo parámetro
                    :
                    mov <dummy>,[bp+2+2N]       ; último parámetro
                    :
                    mov sp,bp                   ; restaura SP
                    pop bp                      ; recupera BP
                    ret                         ; retorno
<myproc>            ENDP
```

El uso de esta estructura, que ha sido adoptada por muchos lenguajes de alto nivel, garantiza la producción de rutinas reusables y portables. Estas rutinas pueden ser aglutinadas en librerías que puedan ser usadas en muchos programas reduciendo la carga de codificación y mejorando la productividad.

Cuando la rutina retorne, los parámetros que fueron introducidos en la pila deben ser removidos. La rutina invocadora puede remover los parámetros extrayéndolos de la pila mediante operaciones **pop** o sumándole a SP el tamaño de los mismos, como en **add SP,2N** donde **2N** es el número de bytes ocupado por los **N** parámetros. Este método corta efectivamente la pila en el lugar donde estaba originalmente. Otra alternativa consiste en que la rutina invocada limpie la pila por medio de una instrucción **ret 2N** donde **2N** nuevamente es el número de bytes ocupado por los **N** parámetros. En cualquier caso, **N** es el número de palabras almacenadas en la pila.

La diferencia entre los dos métodos radica en el hecho de que cuando se emplea la instrucción **ret 2N**, las rutinas deben ser invocadas con el número de parámetros exacto sin omitir ni agregar ninguno. Si el número de parámetros almacenados en la pila no es el apropiado, la instrucción **ret 2N** desalineará la pila pudiendo causar una caída del sistema. Usando el otro método, la rutina invocadora puede decidir cuantos parámetros almacenar en la pila, y luego encargarse de limpiarla adecuadamente.

El primer método es empleado casi exclusivamente por los compiladores de lenguaje "C". El segundo método es empleado por los compiladores de BASIC, FORTRAN y PASCAL entre otros.

Parámetros por Valor y por Referencia.

Un *argumento* es el valor que se le da a un determinado parámetro de un procedimiento o función. Este valor puede ser un dato en si mismo, o la dirección (referencia) del dato.

Parámetros por Valor.

La mayoría de los parámetros en ensamblador son transferidos por valor. En este método, el dato actual es pasado a la rutina invocada. Dicha rutina recibe un número que puede estar almacenado en un registro del microprocesador, o en la pila, de acuerdo con la técnica empleada.

Cuando la información está almacenada en un área común (variable global), tenemos un caso especial, ya que en cierto sentido se trata de un direccionamiento por referencia debido a que tanto la rutina invocadora como la invocada hacen uso de un dirección común pero por otra parte, la información almacenada en el área común pudiera ser datos en si o direcciones, de tal manera que para simplificar las cosas, basaremos la decisión en la naturaleza de la información almacenada. De esta forma diremos, que si la información son datos en si, el pase de parámetros es por valor y si son direcciones, es por referencia.

El tamaño de la información pasada por valor tiene las restricciones de los registros y la pila: 16 bits máximo. Cuando necesitemos transferir estructuras de datos mayores, es más fácil y conveniente pasar su dirección en lugar del dato en si.

Parámetros por Referencia.

En el **pase por referencia** la rutina invocada recibe sólo la dirección de la localidad de memoria donde se encuentra la información. Podemos enumerar una gran cantidad de ventajas inmediatas. Primero, a menos que la información se encuentre diseminada en varios segmentos de datos, toda la dirección puede ser almacenada en un valor de 16 bits, conveniente para el uso de registros o de la pila. Segundo, la rutina se hace completamente general, ya que suministrando una dirección diferente tenemos acceso a un nuevo

conjunto de datos. Tercero, la información puede ser manipulada indirectamente de tal forma que la rutina invocada retorne un valor a la invocadora a través de las mismas localidades de memoria donde residen los parámetros.

Funciones versus Procedimientos.

Frecuentemente se requiere que la rutina invocada retorne datos a la rutina invocadora. Las rutinas que retornan valores se llaman funciones y las que no procedimientos. En los lenguajes de alto nivel, las funciones están generalmente restringidas a retornar un sólo valor. Cualquier otra información que se desee retornar, es pasada modificando uno o más parámetros. En lenguaje ensamblador no existen estas restricciones.

Retornando Datos en Registros.

De nuevo, la vía más simple para retornar un valor es a través de los registros del microprocesador. Al igual que el pase de parámetros, esta opción está limitada por el número de registros disponibles y el tamaño de la información a ser retornada. Pero desde el punto de vista positivo, la información puede ser verificada y manipulada con facilidad.

Esta es una buena opción para rutinas que sean invocadas frecuentemente. No se requieren configuraciones especiales ni anticipación de *buffers* de memoria. Casi todas las rutinas de servicio del MS-DOS retornan información de esta manera. La mayoría de los lenguajes de alto nivel emplean esta técnica para retornar valores. Usualmente se emplea el registro AX para retornar valores de 16 bits y el par DX, AX para valores de 32 bits.

Retornando Datos en la Pila.

El otro método para retornar valores consiste en colocarlos en la pila. Esta operación requiere el uso del registro BP para direccionar la pila (en la misma forma que para pasar parámetros a través de la pila). Para retornar un valor, este es cargado en una de las posiciones de memoria por encima de la dirección de retorno. Si el procedimiento fue invocado con parámetros, la localidad ocupada por uno de ellos puede servir para almacenar el valor a retornar. Si el procedimiento se invoca sin parámetros, la rutina invocadora debe almacenar en la pila un parámetro *dummy* para reservar el espacio necesario para el valor a ser retornado. Si el valor a ser retornado es muy grande para caber en la pila, se regresa en ella un apuntador al mismo. Entonces, dicha zona de memoria contendrá el valor devuelto actual.

Reporte por Excepción.

En muchas aplicaciones es una opción deseable tener funciones o procedimientos que proporcionen algún tipo de indicación de error. Muchas de las funciones del MS-DOS retornan un código de estado al finalizar su ejecución. Frecuentemente se emplea el bit de acarreo para señalar una condición de error y uno o más registros, usualmente AX, para detallar dicho error.

El bit de acarreo es usado por múltiples razones. Es fácil de verificar (con JC o JNC), de manipular (con STC, CMC o CLC) y de salvar y recuperar (con PUSHF y POPF). En la arquitectura del 8086, este es el bit de estado más fácil de manipular. Esto proporciona un mecanismo ideal para indicación de excepciones. El programador correrá con la responsabilidad de mantener dicho bit a cero cuando no ocurra ningún error, ya que el mismo puede ser activado por ciertas operaciones normales.

Una vez que la rutina invocadora ha detectado la existencia de un error (verificando el estado del bit de acarreo), puede proceder a un análisis más detallado del mismo examinando los registros que proporcionan información adicional.

Aplicaciones

En MS-DOS existen dos tipos de archivos ejecutables, los cuales son identificados por la extensión: **COM** y **EXE**. En lenguaje ensamblador, es posible escribir cualquiera de los dos tipos de programas, aunque en este libro se muestre una clara preferencia por los archivos **EXE**.

Esta preferencia se debe entre otras cosas al hecho, de que el formato **EXE** proporciona mayor libertad en la programación y de que no existen garantías de que el formato **COM** continúe siendo soportado por el MS-DOS en futuras versiones ya que Microsoft (la empresa que diseño el MS-DOS) ha expresado reiteradamente sus intenciones de eliminar dicho formato.

Los programas tipo **COM** sólo pueden ocupar un segmento lógico y son almacenados en disco como una imagen binaria del mismo. Esto trae como consecuencia, que la longitud total del programa, incluyendo datos y pila no puede exceder los 64K y que la primera instrucción ejecutable del mismo deberá estar ubicada siempre en la posición 100h (justo después del PSP). Además, es obligatorio el empleo de un nombre único para los segmentos declarados en todos los módulos fuente que constituyan la aplicación. Por su parte el sistema operativo inicializa los registros de segmento de tal forma que apunten al inicio del PSP, el apuntador de pila (SP) señala a la posición de memoria más alta disponible dentro del segmento y la pila es inicializada con un valor cero. Las funciones y procedimientos empleados tendrán atributo **NEAR** ya que todas las invocaciones son intrasegmento.

El formato **EXE** es flexible y poderoso. Soporta el empleo de múltiples segmentos, por lo que su tamaño sólo es limitado por la cantidad de memoria disponible. El punto de entrada al programa y el tamaño de la pila son definidos por el programador. El sistema operativo inicializa CS de tal manera que apunte al segmento de código, los registros DS y ES para que señalen al segmento del PSP y SS para que apunte al segmento de pila. SP indicará el tope de la pila. Las funciones y procedimientos podrán tener atributos **NEAR** o **FAR** dependiendo del modelo de memoria y de la ubicación de cada una dentro del programa. El archivo en disco contiene un encabezado que permite la relocalización del programa al momento de carga.

Templates para programas COM y EXE.

Los *templates* son plantillas que contienen las sentencias mínimas necesarias (fundamentalmente directivas) para escribir un programa en lenguaje ensamblador. Debido a las diferencias entre programas **COM** y **EXE**, es necesario escribir *templates* diferentes para cada tipo de aplicación. Estos *templates* servirán como base para la escrituras de aplicaciones en lenguaje ensamblador.

Template para programas COM.

El *template* que se muestra permite la escritura de programas **COM** usando directivas simplificadas. Nótese la presencia de la directiva **ORG** que obliga a que la primera instrucción ejecutable se ubique en la dirección 100h.

```
MODEL       Tiny

                        ; espacio para inclusión de archivos.
                        ; espacio para definición de constantes.

DATASEG
                        ; espacio para declaración de variables

CODESEG
            ORG 100h; primera instrucción en 100h
Inicio:

                        ; cuerpo principal del programa.

            mov ax,4C00h    ; retorno al sistema operativo.
            int 21h

                        ; funciones y procedimientos

END Inicio
```

Template para programa EXE.

El *template* presentado permite escribir programas **EXE** en modelo **SMALL**. Este modelo admite la existencia de un segmento para código y otro para datos, lo cual proporciona una capacidad teórica de 64K para código y 64K para datos. Se establece un tamaño para la pila de 256 bytes. Nótese que al arrancar el programa es necesario ajustar el contenido del registro DS de tal manera que este señale al segmento de datos.

```
MODEL       Small
STACK       256

                        ; espacio para inclusión de archivos.
                        ; espacio para definición de constantes.

DATASEG
                        ; espacio para declaración de variables
CODESEG
Inicio:     mov ax,@data    ; ajustar el registro de segmento de datos
            mov ds,ax

                        ; cuerpo principal del programa.

            mov ax,4C00h    ; retorno al sistema operativo.
            int 21h

                        ; funciones y procedimientos

END Inicio
```

Estructuras de Control.

La implementación de algoritmos conlleva la utilización de estructuras de control. Estas permiten la selección e iteración condicional dentro del flujo de un programa. Entre las más comunes tenemos **IF-THEN-ELSE**, **WHILE-DO**, **REPEAT-UNTIL**, **FOR-DO** y **CASE-OF**. Estas generalmente están incorporadas a lenguajes de alto nivel como el **PASCAL** y el **C**.. El lenguaje ensamblador, al ser de bajo nivel, carece de dichas sentencias, pero con el empleo de macros y valiéndonos de la compilación condicional, es posible la implementación de algunas de ellas. En esta sección se muestra la implementación de las estructuras **IF-THEN-ELSE** y **WHILE-DO**, las cuales fueron escogidas por ser base de las restantes.

Los listados a continuación, muestran macros auxiliares para la implementación de las estructuras antes mencionadas. La labor de ellos es la generación dinámica de etiquetas.

```
; @@Save,@@@Save,@@Load,@@@Load

@@Save      MACRO
            IFNDEF @@Level
                  @@Level= 0              ; inicializa la pila
            ELSE
                  @@Level= @@Level+1      ; incrementa el apuntador de pila
            ENDIF
            @@@Save %@@Level              ; salva la etiqueta en la pila
            ENDM
@@@Save     MACRO Sym
            @@_Sym_&Sym= THIS NEAR        ; genera una etiqueta dinámicamente
            ENDM

@@Load      MACRO
            @@@Load %@@Level              ; recupera un etiqueta de la pila
            ENDM

@@@Load     MACRO Sym
            IFNDEF @@Sym_&Sym
                  %OUT >>>Error de estructura.  ; error en el uso de la estructura
            ELSE
                  @@Symbol= @@Sym_&Sym    ; determina el nombre de la etiqueta
                  @@Level = @@Level-1     ; actualiza el apuntador de pila
            ENDIF
            ENDM
```

Macros para implementación de la estructura **IF-THEN-ELSE**. Nótese el empleo de instrucciones NOP para reserva de espacio para código y del manejo de la directiva **ORG** para el posicionamiento del apuntador de sentencias.

```
; @If,@Else,@EndIf

@If         MACRO Op1,T,Op2
            LOCAL @@IfOk
            cmp Op1,Op2                   ; comparación de los operandos
            j&T @@IfOk                    ; salto condicional
            @@Save                        ; genera una etiqueta dinámica
            nop                           ; reserva espacio para código
            nop
            nop
@@IfOk:     ENDM

@Else       MACRO
            LOCAL @@IfAddr,@@End
            @@Load                        ; obtiene la posición de la sentencia IF
            @@IfAddr= @@Symbol
            @@Save                        ; genera una etiqueta dinámicamente
            nop                           ; reserva espacio para código
            nop
            nop
            @@End= THIS NEAR
            ORG @@IfAddr                  ; genera el salto a la cláusula alternativa
            jmp @@End
            ORG @@End
            ENDM

@EndIf      MACRO
            LOCAL @@End
            @@Load                        ; obtiene la dirección de la cláusula IF
            @@End= THIS NEAR
            ORG @@Symbol
            jmp @@End                     ; genera el salto al final de la estructura
```

```
                ORG @@End
                ENDM
```

Macro para la implementación de la estructura **WHILE-DO**.

```
; @DoWhile, @EndDo

@DoWhile MACRO Op1,T,Op2
                LOCAL @@TOk
                @@Save                  ; genera una etiqueta dinámicamente
                cmp Op1,Op2             ; comparación
                j&T @@TOk               ; salto condicional
                @@Save                  ; genera etiqueta dinámicamente
                nop                     ; reserva espacio para código
                nop
                nop
@@TOk:          ENDM

@EndDo          MACRO
                LOCAL @@RetAddr,@@End
                @@Load                  ; obtiene la posición de la sentencia NOP
                @@RetAddr= @@Symbol
                @@Load                  ; obtiene la posición de la sentencia WHILE
                jmp @@Symbol            ; genera un salto al inicio de la estructura
                @@End= THIS NEAR
                ORG @@RetAddr
                jmp @@End               ; genera un salto al final de la estructura
                ORG @@End
                ENDM
```

Aplicaciones.

En esta sección de aplicaciones, se desarrollan rutinas para la ejecución de diversas tareas, que incluyen manejo de teclado, video, archivos, directorios, impresora, ratón así como conversión de datos y manipulación de cadenas.

Todas las rutinas escritas son en formato EXE, reciben sus argumentos a través de la pila y retornan datos en los registros DX y AX de acuerdo con el siguiente criterio:

AX: si el valor devuelto es de menos de 16 bits.
DX:AX: si el valor devuelto es un dato de 32 bits.
DX:AX: si el dato devuelto es un apuntador, en cuyo caso DX contiene el segmento y AX el desplazamiento.

El formato general de las rutinas, (que pueden ser incluidas en una librería) es el siguiente:

```
                PUBLIC _Rutina          ; permitir el acceso desde otro módulo
_Rutina         PROC NEAR               ; atributo NEAR (modelo de memoria Small)
                ARG Lista de argumentos ; pase de argumentos a través de la pila
                LOCAL variables locales ; variables locales
                LOCALS                  ; símbolos y etiquetas locales

                push bp                 ; permitir acceso a los argumentos
                mov bp,sp

                sub sp,variables locales ; reservar espacio para variables locales

                                        ; salvar registros

                                        ; cuerpo del procedimiento

                                        ; recuperar registros
```

```
                                        ; ajuste de la pila

                    ret argumentos      ; retorno al programa invocador
                                        ; la rutina invocada limpia la pila
_Rutina             ENDP
```

Todos los nombres de las rutinas contiene como primer carácter el *underscore* (_) para permitir la escritura de macros de acceso a las rutinas con el mismo nombre de la rutina, eliminando dicho carácter.

Para la compilación e inclusión en librería de las rutinas presentadas se deben emplear los siguientes comandos:

TASM *rutina* **/la /ml /m9 /zi**
TLIB *librería* **/C +***módulo objeto*

Para la compilación y enlace de un programa de aplicación que emplee dicha librería es necesario usar los comandos siguientes:

TASM *programa* **/la /ml /m9 /zi**
TLINK /c /v *programa,,librería*

Teclado.

El medio fundamental mediante el cual el usuario introduce datos a un computador operando en MS-DOS, es el teclado. Esto es una consecuencia natural del interfaz de línea de comandos que proporciona el sistema operativo.

Existen tres métodos básicos para tener acceso al teclado:

Funciones tradicionales del MS-DOS: son un conjunto de rutinas, heredadas del CP/M, incluidas originalmente para facilitar la adaptación de las aplicaciones existentes al MS-DOS. Estas funciones han recibido mejoras con la evolución del MS-DOS, tales como el redireccionamiento.

Funciones orientadas a manejadores del MS-DOS: Las funciones orientadas a manejadores (tipo *handle*) tienen su origen en el sistema operativo UNIX y fueron incluidas por primera vez en la versión 2.00 del MS-DOS. Un programa hace uso de estas funciones suministrando un manejador para el dispositivo deseado y la dirección y longitud de un *buffer* de memoria.
Cuando el programa es ejecutado, el MS-DOS suministra manejadores predefinidos para los dispositivos más comúnmente usados, incluyendo el teclado.
Estos manejadores pueden ser empleados para operaciones de lectura y escritura. Estas funciones soportan el redireccionamiento.

Funciones de manejo de teclado del BIOS: El uso de funciones del BIOS presupone que el programa se está ejecutando en una máquina compatible IBM. Las rutinas del BIOS operan a un nivel más bajo que las del sistema operativo, lo que permite un manejo más directo del *hardware* involucrado.

En este apartado se muestran siete rutinas para acceso al teclado, en las cuales se emplean las técnicas antes mencionadas:

Rutina	Descripción
_GetC	Lee un carácter del dispositivo estándar de entrada, retornando en AX el código ASCII del mismo.
_GetChe	Lee un carácter del dispositivo estándar de entrada, lo envía al dispositivo estándar de salida y retorna en AX el código ASCII del mismo.
_GetCBios	Lee un carácter del buffer del teclado retornando el código ASCII y el código de ubicación del mismo.

_Input	Lee una cadena de caracteres del dispositivo estándar de entrada, hasta encontrar e incluyendo un retorno de carro (0Dh), y la coloca en la variable Var. La cadena es enviada al dispositivo estándar de salida. La variable Var tiene la siguiente estructura: MaxLen DB ? Longitud máxima de la cadena Length DB ? Caracteres leídos (incluyendo 0Dh). String DB n DUP(0) Cadena leída.
_GetS	Lee una cadena de caracteres del dispositivo estándar de entrada y la almacena en la variable Var. Retorna en AX el número de caracteres efectivamente leídos.
_GetKeybStatus	Devuelve el estado del teclado y lee un carácter si está disponible. Si el carácter está disponible, el bit de cero estará apagado, en AH el código de búsqueda y en AL el código ASCII del carácter. Si no hay carácter disponible, el bit de cero se enciende.
_GetKeybFlags	Devuelve en AX el estado de las teclas shift, alt, ctrl, num lock, caps lock e insert.

Rutina _GetC: Lee un carácter del dispositivo estándar de entrada, retornando en AX el código ASCII del mismo.
Invocación: call _GetC

```
                PUBLIC _GetC
_GetC           PROC NEAR
                mov ah,08h              : función 08h
                int 21h                 ; del DOS
                xor ah,ah               ; ah= 0
                ret
                _GetC   ENDP
```

Rutina _GetChe: Lee un carácter del dispositivo estándar de entrada, lo envía al dispositivo estándar de salida y retorna en AX el código ASCII del mismo.
Invocación: call _GetChe

```
                PUBLIC _GetChe
_GetChe PROC NEAR
                mov ah,01h              : función 01h
                int 21h                 ; del DOS
                xor ah,ah               ; ah= 0
                ret
_GetChe ENDP
```

Rutina _GetCBios: Lee un carácter del buffer del teclado retornando el código ASCII y el código de ubicación del mismo. El buffer del teclado usualmente se encuentra localizado en 0040:001Ah.
Invocación: call _GetCBios

```
                PUBLIC _GetCBios
_GetCBios       PROC NEAR
                mov ah,00h              : función 00h
                int 16h                 ; del BIOS (teclado)
                ret
_GetCBios       ENDP
```

Rutina _Input: Lee una cadena de caracteres del dispositivo estándar de entrada, hasta encontrar e incluyendo un retorno de carro (0Dh), y la coloca en la variable Var. La cadena es enviada al dispositivo estándar de salida. La variable Var tiene la siguiente estructura:

MaxLen DB ?	Longitud máxima de la cadena
Length DB ?	Caracteres leídos (incluyendo 0Dh).
String DB n DUP(0)	Cadena leída.

Invocación: mov BYTE PTR buffer,<longitud máxima>
push SEG <buffer>
push OFFSET <buffer>
call _Input

```
                PUBLIC _Input
_Input          PROC NEAR
                ARG Var:DWORD= ArgLen
                push bp                 : salvar BP
                mov bp,sp               ; establecer acceso a los argumentos
                push ax                 ; salvar registros
                push dx
                push ds

                mov ah,0Ah              ; función 0Ah
                lds dx,Var
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop ax
                pop bp
                ret ArgLen
_Input          ENDP
```

Rutina _GetS: Lee una cadena de caracteres del dispositivo estándar de entrada y la almacena en la variable Var. Retorna en AX el número de caracteres efectivamente leídos.
Invocación: push SEG <buffer>
push OFFSET <buffer>
push <longitud máxima>
call _GetS

```
                PUBLIC _GetS
_GetS           PROC NEAR
                ARG Length:WORD,Var:DWORD= ArgLen
                push bp                 ; salvar registro BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ds                 ; salvar registros
                pushf

                mov ax,StdIn            ; dispositivo estándar de entrada
                push ax
                lds ax,Var
                push ds
                push ax
                push Length
                call _FRead             ; invocar a FRead (ver sección de manejo de archivos)

                popf                    ; recuperar registros
                pop ds
                pop bp
                ret ArgLen
_GetS           ENDP
```

Rutina _GetKeybStatus: Devuelve el estado del teclado y lee un carácter si está disponible. Si el carácter está disponible, el bit de cero estará apagado, en AH el código de búsqueda y en AL el código ASCII del carácter. Si no hay carácter disponible, el bit de cero se enciende.
Invocación: call _GetKeybStatus

```
                PUBLIC _GetKeybStatus
_GetKeybStatus  PROC NEAR
                mov ah,01h              ; función 01h
                int 16h                 ; del BIOS (teclado)
```

```
                ret
_GetKeybStatus  ENDP
```

Rutina _GetKeybFlags: Devuelve en AX el estado de las teclas shift, alt, ctrl, num lock, caps lock e insert.
Invocación: call _GetKeybFlags

```
                PUBLIC _GetKeybFlags
_GetKeybFlags   PROC NEAR
                mov ah,02h              ; función 02h
                int 16h                 ; del BIOS (teclado)
                xor ah,ah               ; ah= 0
                ret
_GetKeybFlags   ENDP
```

Video.

La presentación visual de un programa de aplicación es uno de los elementos más importantes. El MS-DOS proporciona algunas funciones para la transferencia de texto a la pantalla incluyendo las orientadas a manejador. No suministra facilidades para el manejo gráfico ni para la programación del adaptador de video.

El BIOS permiten escribir textos o pixels individuales, seleccionar modos de video y manejar la paleta de colores. El soporte de gráficos, sin embargo, es primitivo.

En esta sección se presentan 25 rutinas agrupadas en tres áreas: despliegue de textos, manejo de video y modo gráfico.

Rutina	Descripción
_PutC	Escribe un carácter en el dispositivo de salida estándar.
_PutCR	Escribe un CR/LF usando _PutC.
_PutCBios	Escribe un carácter ASCII en la posición actual del cursor en la página y con el atributo especificados.
_Print	Envía una cadena de caracteres a la consola. La cadena debe finalizar en un carácter '$'.
_PutS	Envía una cadena de caracteres al dispositivo estándar de salida.
_PutH	Imprime un entero corto sin signo (ocho bits) como un número hexadecimal de dos dígitos.
_PutW	Imprime un entero sin signo como un número hexadecimal de 4 dígitos.
_PutI	Imprime un entero como un número decimal.
_PutU	Imprime un entero sin signo como un número decimal.
_PutL	Imprime un valor de 32 bits como un entero largo con signo.
_PutUL	Imprime un valor de 32 bits como un entero largo sin signo.
_PutASCIIZ	Envía una cadena ASCIIZ al dispositivo estándar de salida.

Rutina _PutC: Escribe un carácter en el dispositivo de salida estándar.
Invocación: push <carácter>
call _PutC

```
                PUBLIC _PutC
_PutC           PROC NEAR
                ARG Character:BYTE= ArgLen
                push bp                 : salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push dx

                mov ah,02h              ; función 02h
                mov dl,Character
                int 21h                 ; del DOS
```

```
                pop dx                  ; recuperar registros
                pop ax
                pop bp
                ret ArgLen
_PutC           ENDP
```

Rutina _PutCR: Escribe un CR/LF usando _PutC.
Invocación: call _PutCR

```
                PUBLIC _PutCR
_PutCR          PROC NEAR
                push CR                 ; imprimir retorno de carro
                call _PutC
                push LF                 ; imprimir salto de línea
                call _PutC
                ret
_PutCR          ENDP
```

Rutina _PutCBios: Escribe un carácter ASCII en la posición actual del cursor en la página y con el atributo especificados.
Invocación: push <carácter>
push <atributo>
push <número de caracteres>
push <página de video>
call _PutCBios

```
                PUBLIC _PutCBios
_PutCBios       PROC NEAR
                ARG DisplayPage:BYTE,No:WORD,Attr:BYTE,Character:BYTE= Arglen
                push bp                 : salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx

                mov ah,09h              ; función 09h
                mov al,Character
                mov bh,DisplayPage
                mov bl,Attr
                mov cx,No
                int 10h                 ; del BIOS (video)

                pop cx                  ; recuperar registros
                pop bx
                pop ax
                pop bp
                ret ArgLen
_PutCBios       ENDP
```

Rutina _Print: Envía una cadena de caracteres a la consola. La cadena debe finalizar en un carácter '$'.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _Print

```
                PUBLIC _Print
_Print          PROC NEAR
                ARG  Var:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push dx
                push ds
```

```
                mov ah,9                ; función 09h
                lds dx,Var
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros.
                pop dx
                pop ax
                pop bp
                ret ArgLen
_Print          ENDP
```

Rutina _PutS: Envía una cadena de caracteres al dispositivo estándar de salida.
Invocación: push SEG <cadena>
push OFFSET <cadena>
push <longitud cadena>
call _PutS

```
                PUBLIC _PutS
_PutS           PROC NEAR
                ARG Length:WORD,Var:DWORD= ArgLen
                push bp                 : salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push ds
                pushf

                mov  ax,StdOut          ; dispositivo estándar de salida
                push ax
                lds ax,Var
                push ds
                push ax
                push Length
                call _FWrite            ; invocar FWrite (ver manejo de archivos)

                popf                    ; recuperar registros.
                pop ax
                pop ds
                pop bp
                ret ArgLen
_PutS           ENDP
```

Rutina _PutH: Imprime un entero corto sin signo (ocho bits) como un número hexadecimal de dos dígitos.
Invocación: push <valor>
call _PutH

```
                PUBLIC _PutH
_PutH           PROC NEAR
                ARG UnsignedShortInt:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx

                mov al,UnsignedShortInt
                mov cx,2                ; 2 dígitos
                mov bl,16               ; hexadecimales
                xor dh,dh
@@1:            xor ah,ah
                div bl                  ; divide por 10
                mov dl,ah
```

```
                add dl,'0'              ; convierte el residuo en su código ASCII
                cmp dl,'9'
                jbe @@2
                add dl,'A'-'9'-1

@@2:            push dx                 ; salvar dígito en la pila
                loop @@1
                mov cx,2

@@3:            call _PutC              ; imprimir dígitos
                loop @@3

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_PutH           ENDP
```

Rutina _PutW: Imprime un entero sin signo como un número hexadecimal de 4 dígitos.
Invocación: push <valor>
call _PutW

```
                PUBLIC _PutW
_PutW           PROC NEAR
                ARG  Unsigned:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros

                mov ax,Unsigned
                xchg al,ah
                push ax
                call _PutH              ; imprimir los dos dígitos más significativos
                mov al,ah
                push ax
                call _PutH              ; imprimir los dos dígitos menos significativos

                pop ax                  ; recuperar registros
                pop bp
                ret ArgLen
_PutW           ENDP
```

Rutina _PutI: Imprime un entero como un número decimal.
Invocación: push <valor>
call _PutI

```
                PUBLIC _PutI
_PutI           PROC NEAR
                ARG Integer:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push dx

                mov ax,Integer
                test ax,8000h           ; si es positivo
                jnz @@1
                mov dx,' '              ; imprimir ' '
                jmp SHORT @@2

@@1:            mov  dx,'-'             ; si no, imprimir el signo '-'
```

```
                neg  ax

@@2:            push dx
                call _PutC
                push ax
                call _PutU              ; imprimir entero

                pop  dx                 ; recuperar registros
                pop ax
                pop bp
                ret ArgLen
_Putl           ENDP
```

Rutina _PutU: Imprime un entero sin signo como un número decimal.
Invocación: push <valor>
call _PutU

```
                PUBLIC _PutU
_PutU           PROC NEAR
                ARG  Integer:WORD= ArgLen
                LOCALS
                push bp                 : salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx

                mov ax,Integer
                mov bx,10
                xor cx,cx
@@1:            xor dx,dx
                div bx                  ; divide por 10
                add dx,'0'              ; convierte número a código ASCII
                push dx                 ; y lo guarda en la pila
                inc cx                  ; incrementa contador de dígitos
                or ax,ax
                jnz @@1

@2:             call _PutC              ; imprime dígito a dígito
                loop @@2

                pop dx                  ; recupera registros
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_PutU           ENDP
```

Rutina _PutL: Imprime un valor de 32 bits como un entero largo con signo.
Invocación: push <16 bits menos significativos del valor>
push <16 bits m s significativos del valor>
call _PutL

```
                PUBLIC _PutL
_PutL           PROC NEAR
                ARG HiWord:WORD,LoWord:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push dx
```

```
                mov ax,LoWord
                mov dx,HiWord
                test dx,8000h           ; si es positivo
                jnz @@1
                mov bx,20h              ; imprimir espacio
                jmp SHORT @@2

@@1:            mov bx,'-'              ; si no, imprimir el signo '-'
                not dx
                not ax
                add ax,1
                adc dx,0

@@2:            push bx
                call _PutC
                push ax
                push dx
                call _PutUL             ; imprimir el entero largo

                pop dx                  ; recuperar los registros
                pop bx
                pop ax
                pop bp
                ret ArgLen
_PutL           ENDP
```

Rutina _PutUL: Imprime un valor de 32 bits como un entero largo sin signo.
Invocación: push <16 bits menos significativos del valor>
push <16 bits más significativos del valor>
call _PutUL

```
                PUBLIC _PutUL
_PutUL          PROC NEAR
                ARG HiWord:WORD,LoWord:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx

                xor cx,cx
                mov ax,LoWord
                mov dx,HiWord

@@1:            mov bx,10               ; divide por 10
                call Ldiv               ; rutina Ldiv (ver rutinas misceláneas)
                add bx,'0'              ; convierte número a código ASCII
                push bx                 ; almacena en la pila
                inc cx                  ; incrementa contador de dígitos
                mov bx,ax
                or bx,dx
                jnz @@1

@@2:            call _PutC              ; imprime dígitos
                loop @@2

                pop dx                  ; recupera registros
                pop cx
                pop bx
                pop ax
                pop bp
```

```
                ret ArgLen
_PutUL          ENDP
```

Rutina _PutASCIIZ: Envía una cadena ASCIIZ al dispositivo estándar de salida.
Invocación: push SEG <cadena ASCIIZ>
push OFFSET <cadena ASCIIZ>
call _PutASCIIZ

```
                PUBLIC _PutASCIIZ
_PutASCIIZ      PROC NEAR
                ARG Str:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push dx
                push ds

                lds dx,Str              ; determina longitud de la cadena ASCIIZ
                push ds
                push dx
                call _StrLen            ; función StrLen (manejo de cadenas)

                push ds
                push dx
                push ax
                call _PutS              ; imprimir la cadena

                pop ds                  ; recuperar registros
                pop dx
                pop ax
                pop bp
                ret ArgLen
_PutASCIIZ      ENDP
```

Rutinas	Descripción
_CLS	Borra la pantalla en modo texto.
_GotoXY	Permite establecer la posición del cursor en modo texto.
_WhereX	Devuelve en AX la posición horizontal actual del cursor en modo texto.
_WhereY	Devuelve en AX la posición vertical actual del cursor en modo texto.
_GetVideoMode	Devuelve en AX el modo de video activo.
_SetVideoMode	Establece el modo de video activo. Para prevenir el borrado de pantalla en adaptadores EGA, MCGA y VGA, el bit 7 del modo de video debe estar encendido.
_SetCursorType	Permite establecer el tipo de cursor en modo texto.
_ScrollUp	Permite desplazar hacia arriba una ventana en modo texto, el número de líneas especificadas, rellenando el área vacía con el atributo indicado.
_ScrollDown	Permite desplazar hacia abajo una ventana en modo texto, el número de líneas especificadas, rellenando el área vacía con el atributo indicado.

Rutina _CLS: Borra la pantalla en modo texto.
Invocación: call _CLS

```
                PUBLIC _CLS
_CLS            PROC NEAR
                push ax                 ; salvar registros
                push bx
                push cx
```

```
                push dx

                mov ax,0600h            ; función 06h
                mov bh,07h
                mov cx,0000h
                mov dx,184Fh
                int 10h                 ; del BIOS (video): borrar pantalla

                mov ah,02h              ; función 02h
                mov bh,00h
                mov dx,0000h
                int 10h                 ; del BIOS (video): colocar cursor en 0,0

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                pop ax
                ret
_CLS            ENDP
```

Rutina _GotoXY: Permite establecer la posición del cursor en modo texto.
Invocación: push <columna>
push <fila>
call _GotoXY

```
                PUBLIC _GotoXY
_GotoXY         PROC NEAR
                ARG Y:BYTE,X:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push dx

                mov ah,02h              ; función 02h
                mov bh,00h
                mov dh,Y
                mov dl,X
                int 10h                 ; del BIOS (video)

                pop dx                  ; recuperar registros
                pop bx
                pop ax
                pop bp
                ret ArgLen
_GotoXY         ENDP
```

Rutina _WhereX: Devuelve en AX la posición horizontal actual del cursor en modo texto.
Invocación: call _WhereX

```
                PUBLIC _WhereX
_WhereX         PROC NEAR
                push bx                 ; salvar registros
                push cx
                push dx

                mov ah,03h              ; función 03h
                mov bh,00h
                int 10h                 ; del BIOS (video)
                mov al,dl
                xor ah,ah               ; AX= coordenada X del cursor

                pop dx                  ; recuperar registros
                pop cx
```

```
                pop bx
                ret
_WhereX         ENDP
```

Rutina _WhereY: Devuelve en AX la posición vertical actual del cursor en modo texto.
Invocación: call _WhereY

```
                PUBLIC _WhereY
_WhereY         PROC NEAR
                push bx                 ; salvar registros
                push cx
                push dx

                mov ah,03h              ; función 03h
                mov bh,00h
                int 10h                 ; del BIOS (video)
                mov al,dh
                xor ah,ah               ; AX= coordenada Y del cursor

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                ret
_WhereY         ENDP
```

Rutina _GetVideoMode: Devuelve en AX el modo de video activo.
Invocación: call _GetVideoMode

```
                PUBLIC _GetVideoMode
_GetVideoMode   PROC NEAR
                push bx                 ; salvar registros

                mov ah,0Fh              ; función 0Fh
                int 10h                 ; del BIOS (video)

                pop  bx                 ; recuperar registros
                ret
_GetVideoMode   ENDP
```

Rutina _SetVideoMode: Establece el modo de video activo. Para prevenir el borrado de pantalla en adaptadores EGA, MCGA y VGA, el bit 7 del modo de video debe estar encendido.
Invocación: push <modo de video>
call _SetVideoMode

```
                PUBLIC _SetVideoMode
_SetVideoMode   PROC NEAR
                ARG Mode:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a argumentos
                push ax                 ; salvar registros

                mov ah,00h              ; función 00h
                mov al,Mode
                int 10h                 ; del BIOS (Video)

                pop ax                  ; recuperar registros
                pop bp
                ret ArgLen
_SetVideoMode   ENDP
```

Rutina _SetCursorType: Permite establecer el tipo de cursor en modo texto.

Invocación: push <línea superior>
push <línea inferior>
call _SetCursorType

```
                PUBLIC _SetCursorType
_SetCursorType  PROC NEAR
                ARG Bottom:BYTE,Top:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a argumentos
                push ax                 ; salvar registros
                push cx

                mov ah,01h              ; función 01h
                mov ch,Top
                mov cl,Bottom
                int 10h                 ; del BIOS (Video)

                pop cx                  ; recuperar registros
                pop ax
                pop bp
                ret ArgLen
_SetCursorType  ENDP
```

Rutina _ScrollUp: Permite desplazar hacia arriba una ventana en modo texto, el número de líneas especificadas, rellenando el área vacía con el atributo indicado.
Invocación: push <número de líneas>
push <atributo>
push <columna izquierda>
push <fila superior>
push <columna derecha>
push <fila inferior>
call _ScrollUp

```
                PUBLIC _ScrollUp
_ScrollUp       PROC NEAR
                ARG BottomY:BYTE,RightX:BYTE,TopY:BYTE,LeftX:BYTE,Attr:BYTE,NumLines:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx

                mov ah,06h              ; función 06h
                mov al,NumLines
                mov bh,Attr
                mov ch,TopY
                mov cl,LeftX
                mov dh,BottomY
                mov dl,RightX
                int 10h                 ; del BIOS (Video)

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_ScrollUp       ENDP
```

Rutina _ScrollDown: Permite desplazar hacia abajo una ventana en modo texto, el número de líneas especificadas, rellenando el área vacía con el atributo indicado.
Invocación: push <número de líneas>

```
push <atributo>
push <columna izquierda>
push <fila superior>
push <columna derecha>
push <fila inferior>
call _ScrollDown
```

```
                PUBLIC _ScrollDown
_ScrollDown     PROC NEAR
                ARG BottomY:BYTE,RightX:BYTE,TopY:BYTE,LeftX:BYTE,Attr:BYTE,NumLines:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx

                mov ah,07h              ; función 07h
                mov al,NumLines
                mov bh,Attr
                mov ch,TopY
                mov cl,LeftX
                mov dh,BottomY
                mov dl,RightX
                int 10h                 ; del BIOS (Video)

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_ScrollDown             ENDP
```

Rutina	Descripción
_GetPixel	Devuelve el color (en AX) del pixel en las coordenadas especificadas.
_SetPixel	Enciende el pixel especificado en el color indicado.
_DrawLine	Traza un linea entre los dos puntos especificados, en el color indicado. Emplea el algoritmo de Bresenham.
_DrawRectangle	Dibuja un rectángulo en el color especificado

Rutina _GetPixel: Devuelve el color (en AX) del pixel en las coordenadas especificadas.
Invocación:
```
push <coordenada X>
push <coordenada Y>
call _GetPixel
```

```
                PUBLIC _GetPixel
_GetPixel       PROC NEAR
                ARG y:WORD,x:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push dx

                mov ah,0Dh              ; función 0Dh
                mov bh,0
                mov cx,x
                mov dx,y
                int 10h                 ; del BIOS (Video)
```

```
                xor ah,ah               ; AH= 0

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                pop bp
                ret ArgLen
_GetPixel       ENDP
```

Rutina _PutPixel: Enciende el pixel especificado en el color indicado.
Invocación: push <coordenada X>
push <coordenada Y>
push <color>
call _SetPixel

```
                PUBLIC _SetPixel
_SetPixel  PROC NEAR
                ARG color:BYTE,y:WORD,x:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx

                mov ah,0Ch              ; función 0Ch
                mov al,color
                mov bh,0
                mov cx,x
                mov dx,y
                int 10h                 ; del BIOS (Video)

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_SetPixel  ENDP
```

Rutina _DrawLine: Traza un linea entre los dos puntos especificados, en el color indicado. Emplea el algoritmo de Bresenham.
Invocación: push <x1>
push <y1>
push <x2>
push <y2>
push <color>
call _DrawLine

```
                PUBLIC _DrawLine
_DrawLine       PROC NEAR
                ARG Color:WORD,y2:WORD,x2:WORD,y1:WORD,x1:WORD= ArgLen
                LOCAL dirX:WORD,dirY:WORD,p:WORD,k1:WORD,k2:WORD= LocalLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                sub sp,LocalLen         ; reservar espacio para variables locales
                push ax                 ; salvar registros
                push bx
                push cx
                push dx
                push si
                push di
```

```
                xor bx,bx                       ; BX= 0
                mov ax,y2                       ; AX= y2-y1
                sub ax,y1
                jge @@1                         ; si (y2<y1)
                neg ax                          ; complemento a dos de AX
                dec bx                          ; decrementar BX
                jmp SHORT @@2
@@1:            je @@2                          ; si (y2=y1)
                inc bx                          ; incrementar BX
@@2:            mov dx,ax                       ; si (y2>y1) deltaY= abs(y2-y1)
                mov dirY,bx                     ; dirY = abs(y2-y1)

                xor bx,bx                       ; BX= 0
                mov ax,x2
                sub ax,x1                       ; AX= x2-x1
                jge @@3                         ; si (x2<x1)
                neg ax                          ; complemento a dos de AX
                dec bx                          ; decrementar BX
                jmp SHORT @@4
@@3:            je @@4                          ; si (x2=x1)
                inc bx                          ; incrementar BX
@@4:            mov  cx,ax          ; si (x2>x1) deltaX= abs(x2-x1)
                mov dirX,bx                     ; dirX = sgn(x2-x1)

                cmp cx,dx                       ; si (deltaX>deltaY)
                jle  @@5

                mov ax,dx
                shl ax,1
                mov k1,ax                       ; k1= 2*deltaY
                sub ax,cx
                mov p,ax                        ; p = 2*deltaY-deltaX
                mov ax,dx
                sub ax,cx
                shl ax,1
                mov k2,ax                       ; k2= 2*(deltaY-deltaX)
                jmp SHORT @@6

@@5:            mov ax,cx                       ; si no
                shl ax,1
                mov k1,ax                       ; k1= 2*deltaX
                sub ax,dx
                mov p,ax                        ; p = 2*deltaX-deltaY
                mov ax,cx
                sub ax,dx
                shl ax,1
                mov k2,ax                       ; k2= 2*(deltaX-deltaY)

@@6:            mov si,x1                       ; x= x1
                mov di,y1                       ; y= y1

                jmp @@putpixel                  ; putpixel

@@13:           cmp cx,dx                       ; si (deltaX>deltaY)
                jle  @@7

                add si,dirX                     ; x+= sgn(x2-x1)
                jmp SHORT @@8

@@7:            add di,dirY         ; si no, y+= sgn(y2-y1)

@@8:            mov ax,p                        ; si (p<0)
                cmp ax,0
                jge @@9
```

```
                add ax,k1               ; p+= k1
                mov p,ax
                jmp SHORT @@putpixel

@@9:            cmp cx,dx               ; si no,
                jle @@11                ; si (deltaX>deltaY)

                add di,dirY             ; y+= sgn(y2-y1)
                jmp SHORT @@15

@11:            add si,dirX             ; si no, x+= sgn(x2-x1)

@@15:           mov ax,p                ; p+= k2
                add ax,k2
                mov p,ax

@@putpixel:     push si                 ; PutPixel(x,y,Color)
                push di
                push Color
                call _SetPixel

                cmp cx,dx               ; si (deltaX>deltaY)
                jle @@12

                cmp si,x2               ; y (x!=x2) entonces iterar
                jne @@13
                jmp SHORT @@14

@@12:           cmp  di,y2              ; si no, si (deltaX<=deltaY) y (y!=y2)
                jne SHORT @@13          ; entonces iterar

@@14:           pop di                  ; recuperar registros
                pop si
                pop dx
                pop cx
                pop bx
                pop ax

                mov sp,bp               ; restablecer la pila
                pop bp
                ret ArgLen
_DrawLine ENDP
```

Rutina _DrawRectangle: Dibuja un rectángulo en el color especificado
Invocación:
```
push <coordenada X superior izquierda>
push <coordenada Y superior izquierda>
push <coordenada X inferior derecha>
push <coordenada Y inferior derecha>
push <color>
call _DrawRectangle
```

```
                PUBLIC _DrawRectangle
_DrawRectangle  PROC NEAR
                ARG Color:BYTE,Bottom:WORD,Right:WORD,Top:WORD,Left:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push cx
                push dx
                push si
                push di
```

```
                mov al,Color            ; AX= color
                mov cx,Left             ; CX= extremo izquierdo
                mov dx,Top              ; DX= tope
                mov si,Right            ; SI= extremo derecho
                mov di,Bottom           ; DI= borde inferior

                push cx                 ; trazar línea superior
                push dx
                push si
                push dx
                push ax
                call _DrawLine

                push si                 ; trazar línea vertical derecha
                push dx
                push si
                push di
                push ax
                call _DrawLine

                push si                 ; trazar línea inferior
                push di
                push cx
                push di
                push ax
                call _DrawLine

                push cx                 ; trazar línea vertical izquierda
                push di
                push cx
                push dx
                push ax
                call _DrawLine

                pop di                  ; recuperar registros
                pop si
                pop dx
                pop cx
                pop ax
                pop bp
                ret ArgLen
_DrawRectangle  ENDP
```

Manejo de Archivos.

La herencia dual del MS-DOS - CP/M y UNIX - queda claramente demostrada al analizar los servicios para manejo de archivos, que este proporciona. El MS-DOS suministra dos juegos de funciones para manejo de archivos, que hacen uso de técnicas diferentes.

El conjunto de instrucciones, compatibles con CP/M son conocidas como funciones FCB. Estas funciones están basadas en un estructura llamada *File Control Block*, para mantener la información referente a los archivos activos. Estas funciones permiten al programador crear, abrir, cerrar y borrar archivos, así como leer o escribir registros de cualquier tamaño y en cualquier posición de un archivo. Sin embargo, las funciones FCB no soportan la estructura jerárquica de subdirectorios.

Las instrucciones que brindan compatibilidad con UNIX, son conocidas como funciones *handle* (orientadas a manejador). Estas funciones permiten abrir y crear archivos suministrando al sistema operativo una cadena ASCIIZ con el nombre y ubicación dentro de la estructura de subdirectorios del archivo deseado. Si la operación es exitosa, el sistema operativo retorna un *handle* (manejador), que puede ser usado por la aplicación para operaciones subsecuentes con el archivo.

Cuando se emplean funciones *handle*, mantiene estructuras de datos con la información correspondiente a cada archivo activo, en su propia memoria y esta no es accesible al programa de aplicación. Estas funciones

soportan las estructuras jerárquicas de subdirectorios y permiten al programador, crear, abrir, cerrar y borrar archivos en cualquier disco y subdirectorio y leer o escribir registros de cualquier tamaño y cualquier posición de dichos archivos.

La discusión de la técnica FCB, sólo se debe a motivos históricos. Las nuevas aplicaciones deben siempre usar las funciones *handle*, las cuales son más poderosas y fáciles de manejar. En esta sección se presentan 15 rutinas para manejo de archivos, todas empleando la técnica *handle*.

Rutina	Descripción
_FCreate	Dada una cadena ASCIIZ, crea un archivo en el paso actual o en el especificado, en la unidad de disco actual o en la especificada y devuelve un manejador (handle en AX) que podrá ser usado para subsecuentes accesos al archivo. Si ya existe un archivo con el nombre indicado, el mismo es truncado. En caso de error se encenderá el bit de acarreo y en AX retornará el código de error.
_FOpen	Dada una cadenas ASCIIZ, abre el archivo especificado en el disco y subdirectorio indicados o actuales. Retorna un manejador (handle en AX) que podrá ser empleado para subsecuentes accesos al archivo. En caso de error se encenderá el bit de acarreo y AX contendrá el código de error.
_FClose	Dado un manejador válido, vacía los buffers internos, cierra el archivo y libera el manejador para su posterior uso. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.
_FWrite	Dado un manejador válido, escribe el número de bytes especificados en el archivo correspondiente y actualiza el apuntador de archivo. Retorna en AX el número de bytes transferidos. En caso de error se enciende el bit de acarreo y AX contiene el código de error.
_FRead	Dado un manejador válido, lee el número de bytes especificados del archivo correspondiente, los almacena en el buffer indicado y actualiza el apuntador de archivo. Retorna en AX el número de bytes transferidos. En caso de error se enciende el bit de acarreo y AX contiene el código de error.
_FSeek	Posiciona el apuntador de archivo en un punto relativo al inicio del archivo, final o relativo a la posición actual. Retorna la posición actual del apuntador en DX:AX. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.
_FSize	Dado un manejador válido, retorna en DX:AX el tamaño del archivo correspondiente. En caso de error se enciende el bit de acarreo y AX contiene el código de error.
_FRename	Renombra un archivo o lo mueve a un subdirectorio diferente en el mismo disco. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.
_FDelete	Borra un archivo en la unidad y directorio especificados o actuales. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.
_FTemp	Crea un archivo con un nombre único, la unidad y directorio especificados o actual y retorna una manejador en AX que podrá ser usados para subsecuentes operaciones con el archivo. El nombre del archivo es devuelto en la variable dada. En caso de error se enciende el bit de acarreo y AX contiene el código de error.
_FGetAttr	Retorna en AX el atributo del archivo especificado. En caso de error se enciende el bit de acarreo y AX contiene el código de error.

_FSetAttr	Altera el atributo del archivo especificado. En caso de error se enciende el bit de acarreo y AX contiene el código de error.
_FNew	Dada una cadena ASCIIZ, crea un archivo nuevo en el paso actual o en el especificado, en la unidad de disco actual o en la especificada y devuelve un manejador (handle en AX) que podrá ser usado para subsecuentes accesos al archivo. Si ya existe un archivo con el nombre indicado, la función falla. En caso de error se encenderá el bit de acarreo y en AX retornará el código de error.
_FGetDate	Obtiene la fecha en DX y la hora en AX del archivo especificado. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.
_FSetDate	Modifica la fecha y hora del archivo especificado. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.

Rutina _FCreate: Dada una cadena ASCIIZ, crea un archivo en el paso actual o en el especificado, en la unidad de disco actual o en la especificada y devuelve un manejador (handle en AX) que podrá ser usado para subsecuentes accesos al archivo. Si ya existe un archivo con el nombre indicado, el mismo es truncado. En caso de error se encenderá el bit de acarreo y en AX retornará el código de error.

Invocación:
```
push SEG <nombre archivo>
push OFFSET <nombre archivo>
push <atributo>
call _FCreate
```

```
                PUBLIC _FCreate
_FCreate        PROC NEAR
                ARG Attr:WORD,FileName:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push cx                 ; salvar registros
                push dx
                push ds

                mov ah,3Ch              ; función 3Ch
                mov cx,Attr
                lds dx,FileName
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bp
                ret ArgLen
_FCreate        ENDP
```

Rutina _FOpen: Dada una cadenas ASCIIZ, abre el archivo especificado en el disco y subdirectorio indicados o actuales. Retorna un manejador (handle en AX) que podrá ser empleado para subsecuentes accesos al archivo. En caso de error se encenderá el bit de acarreo y AX contendrá el código de error.

Invocación:
```
push SEG <nombre archivo>
push OFFSET <nombre archivo>
push <modo de acceso>
call _FOpen
```

```
                PUBLIC _FOpen
_FOpen          PROC NEAR
                ARG Mode:BYTE,FileName:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros
                push ds
```

```
                mov ah,3Dh              ; función 3Dh
                mov al,Mode
                lds dx,FileName
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop bp
                ret ArgLen
_FOpen          ENDP
```

Rutina _FClose: Dado un manejador válido, vacía los buffers internos, cierra el archivo y libera el manejador para su posterior uso. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.

Invocación: push <manejador>
call _FClose

```
                PUBLIC _FClose
_FClose         PROC NEAR
                ARG Handle:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros

                mov ah,3Eh              ; función 3Eh
                mov bx,Handle
                int 21h                 ; del DOS

                pop bx                  ; recuperar registros
                pop bp
                ret ArgLen
_FClose         ENDP
```

Rutina _FWrite: Dado un manejador válido, escribe el número de bytes especificados en el archivo correspondiente y actualiza el apuntador de archivo. Retorna en AX el número de bytes transferidos. En caso de error se enciende el bit de acarreo y AX contiene el código de error.

Invocación: push <manejador>
push SEG <variable>
push OFFSET <variable>
push <número de bytes>
call _FWrite

```
                PUBLIC _FWrite
_FWrite         PROC NEAR
                ARG Length:WORD,Var:DWORD,Handle:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push dx
                push ds

                mov ah,40h              ; función 40h
                mov bx,Handle
                mov cx,Length
                lds dx,Var
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bx
                pop bp
```

```
                ret ArgLen
_FWrite         ENDP
```

Rutina _FRead: Dado un manejador válido, lee el número de bytes especificados del archivo correspondiente, los almacena en el buffer indicado y actualiza el apuntador de archivo. Retorna en AX el número de bytes transferidos. En caso de error se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push <manejador>
push SEG <variable>
push OFFSET <variable>
push <# de bytes>
call _FRead
```

```
                PUBLIC _FRead
_FRead          PROC NEAR
                ARG Length:WORD,Var:DWORD,Handle:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push dx
                push ds

                mov ah,3Fh              ; función 3Fh
                mov bx,Handle
                mov cx,Length
                lds dx,Var
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bx
                pop bp
                ret ArgLen
_FRead          ENDP
```

Rutina _FSeek: Posiciona el apuntador de archivo en un punto relativo al inicio del archivo, final o relativo a la posición actual. Retorna la posición actual del apuntador en DX:AX. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push <manejador>
push <parte baja del desplazamiento>
push <parte alta del desplazamiento>
push <método>
call _FSeek
```

```
                PUBLIC _FSeek
_FSeek          PROC NEAR
                ARG Method:BYTE,HiWord:WORD,LoWord:WORD,Handle:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx

                mov ah,42h              ; función 42h
                mov al,Method
                mov bx,Handle
                mov cx,HiWord
                mov dx,LoWord
                int 21h                 ; del DOS

                pop cx                  ; recuperar registros
                pop bx
                pop bp
```

```
                ret ArgLen
_FSeek          ENDP
```

Rutina _FSize: Dado un manejador válido, retorna en DX:AX el tamaño del archivo correspondiente. En caso de error se enciende el bit de acarreo y AX contiene el código de error.

Invocación: push SEG <nombre archivo>
push OFFSET <nombre archivo>
call _FSize

```
                PUBLIC _FSize
_FSize          PROC NEAR
                ARG FileName:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push si
                push di
                push ds

                mov ah,3Dh              ; abre el archivo
                mov al,0
                lds dx,FileName
                int 21h
                jc @@exit               ; si hay error, terminar
                mov si,ax               ; si no, guardar el manejador

                mov bx,ax               ; posiciona el apuntador de archivo
                mov ax,4202h            ; al final del mismo,
                mov cx,0                ; empleando la función 42h del DOS
                mov dx,0
                int 21h
                mov di,ax
                pushf

                mov ah,3Eh              ; cierra el archivo
                mov bx,si
                int 21h

                mov ax,di
                popf

@@exit:         pop ds                  ; recuperar archivos
                pop di
                pop si
                pop cx
                pop bx
                pop bp
                ret ArgLen
_FSize          ENDP
```

Rutina _FRename: Renombra un archivo o lo mueve a un subdirectorio diferente en el mismo disco. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.

Invocación: push SEG <nombre actual>
push OFFSET <nombre actual>
push SEG <nuevo nombre>
push OFFSET <nuevo nombre>
call _FRename

```
                PUBLIC _FRename
_FRename        PROC NEAR
                ARG NewFileName:DWORD,FileName:DWORD= ArgLen
```

```
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros
                push di
                push ds
                push es

                mov ah,56h              ; función 56h
                lds dx,FileName
                les di,NewFileName
                int 21h                 ; del DOS

                pop es                  ; recuperar registros
                pop ds
                pop di
                pop dx
                pop bp
                ret ArgLen
_FRename        ENDP
```

Rutina _FDelete: Borra un archivo en la unidad y directorio especificados o actuales. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push SEG <nombre archivo>
push OFFSET <nombre archivo>
call _FDelete
```

```
                PUBLIC _FDelete
_FDelete        PROC NEAR
                ARG FileName:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros
                push ds

                mov ah,41h              ; función 41h
                lds dx,FileName
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop bp
                ret ArgLen
_FDelete        ENDP
```

Rutina _FTemp: Crea un archivo con un nombre único, la unidad y directorio especificados o actual y retorna una manejador en AX que podrá ser usados para subsecuentes operaciones con el archivo. El nombre del archivo es devuelto en la variable dada. En caso de error se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push SEG (paso del archivo)
push OFFSET (paso del archivo)
push <atributo>
call _FTemp
```

```
                PUBLIC _FTemp
_FTemp          PROC NEAR
                ARG Attr:WORD,Var:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push cx                 ; salvar registros
                push dx
                push ds

                mov ah,5Ah              ; función 5Ah
```

```
                mov cx,Attr
                lds dx,Var
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bp
                ret ArgLen
_FTemp          ENDP
```

Rutina _FGetAttr: Retorna en AX el atributo del archivo especificado. En caso de error se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push SEG <nombre archivo>
push OFFSET <nombre archivo>
call _FGetAttr
```

```
                PUBLIC _FGetAttr
_FGetAttr       PROC NEAR
                ARG FileName:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push cx                 ; salvar registros
                push dx
                push ds

                mov ax,4300h            ; función 43h, subfunción 00h
                lds dx,FileName
                int 21h                 ; del DOS
                jc @@1
                mov ax,cx               ; si no hay error, atributo en AX

@@1:            pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bp
                ret ArgLen
_FGetAttr       ENDP
```

Rutina _FSetAttr: Altera el atributo del archivo especificado. En caso de error se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push SEG <nombre archivo>
push OFFSET <nombre archivo>
push <atributo>
call _FSetAttr
```

```
                PUBLIC _FSetAttr
_FSetAttr       PROC NEAR
                ARG Attr:WORD,FileName:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push cx                 ; salvar registros
                push dx
                push ds

                mov ax,4301h            ; función 43h, subfunción 01h
                mov cx,Attr
                lds dx,FileName
                int 21h                 ; del DOS
                jc @@1
                mov ax,cx
```

```
@@1:        pop ds                  ; recuperar registros
            pop dx
            pop cx
            pop bp
            ret ArgLen
_FSetAttr   ENDP
```

Rutina _FNew: Dada una cadena ASCIIZ, crea un archivo nuevo en el paso actual o en el especificado, en la unidad de disco actual o en la especificada y devuelve un manejador (handle en AX) que podrá ser usado para subsecuentes accesos al archivo. Si ya existe un archivo con el nombre indicado, la función falla. En caso de error se encenderá el bit de acarreo y en AX retornará el código de error.

Invocación:
```
push SEG <nombre archivo>
push OFFSET <nombre archivo>
push <atributo>
call _FNew
```

```
            PUBLIC _FNew
_FNew       PROC NEAR
            ARG Attr:WORD,FileName:DWORD= ArgLen
            push bp                 ; salvar BP
            mov bp,sp               ; permitir acceso a los argumentos
            push cx                 ; salvar registros
            push dx
            push ds

            mov ah,5Bh              ; función 5Bh
            mov cx,Attr
            lds dx,FileName
            int 21h                 ; del DOS

            pop ds                  ; recuperar registros
            pop dx
            pop cx
            pop bp
            ret ArgLen
_FNew       ENDP
```

Rutina _FGetDate: Obtiene la fecha en DX y la hora en AX del archivo especificado. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push <manejador>
call _FGetDate
```

```
            PUBLIC _FGetDate
_FGetDate   PROC NEAR
            ARG Handle:WORD= ArgLen
            push bp                 ; salvar BP
            mov bp,sp               ; permitir acceso a los argumentos
            push bx                 ; salvar registros
            push cx

            mov ax,5700h            ; función 57h, subfunción 00h
            mov bx,Handle
            int 21h                 ; del DOS
            jc @@1                  ; si hay error, terminar
            mov ax,cx

@@1:        pop cx                  ; recuperar registros
            pop bx
            pop bp
            ret ArgLen
_FGetDate   ENDP
```

Rutina _FSetDate: Modifica la fecha y hora del archivo especificado. En caso de error, se enciende el bit de acarreo y AX contiene el código de error.

Invocación:
```
push <manejador>
push <fecha>
push <hora>
call _FSetDate
```

```
                PUBLIC _FSetDate
_FSetDate PROC NEAR
                ARG Time:WORD,Date:WORD,Handle:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push dx

                mov ax,5701h            ; función 57h, subfunción 01h
                mov bx,Handle
                mov cx,Time
                mov dx,Date
                int 21h                 ; del DOS

                pop dx                  ; recuperar registros
                pop cx
                pop bx
                pop bp
                ret ArgLen
_FSetDate ENDP
```

Directorios.

En MS-DOS, cada archivo es identificado de manera única por su nombre y localidad. La localidad, a su vez, está compuesta de dos partes: la unidad lógica que contiene al archivo y el directorio donde este se encuentra.

Las unidades lógicas son identificadas con una letra y dos puntos (por ejemplo, **A:**). El número de unidades lógicas en un sistema no es necesariamente igual al de unidades físicas; por ejemplo, es común que discos duros de gran tamaño estén divididos en dos o más unidades lógicas. Lo que identifica a una unidad lógica es el hecho de poseer un sistema de archivos autocontenidos con uno o más directorios y cero o más archivos en cada uno, y toda la información necesaria para localizar a cada uno de ellos y determinar cuanto espacio libre y usado posee.

Los directorios son simples listas de archivos. Cada item de las lista consta de un nombre, tamaño, localidad de inicio, atributos y fecha y hora de la última modificación efectuada al mismo. La información detallada acerca de la ubicación de cada bloque de datos asignado a un archivo o directorio, está contenida en un estructura llamada tabla de ubicación de archivos (**FAT**).

Cada unidad de disco tiene potencialmente dos tipos de directorios: directorio *raíz* y otros directorios. El directorio *raíz* siempre está presente y posee un límite en el número de items que puede albergar, determinado en el momento de ser formateada la unidad, y que no puede ser alterado. Los *otros* directorios, que pueden o no estar presentes, pueden ser anidados hasta cualquier nivel y pueden crecer hasta cualquier tamaño. Cada directorio, excepto el *raíz* tiene un nombre que es designado por el carácter *backslash* (\).

El MS-DOS mantiene un registro acerca de cual es la unidad y directorio activos y los incorpora en la especificación del archivo, cuando uno o ambos no sean indicados de manera explícita. Es posible seleccionar cualquier directorio en cualquier unidad especificando en orden los nombres de los directorios que los preceden en la estructura jerárquica, bien sea a partir del directorio *raíz* o del directorio actual. A esta secuencia de nombres de directorios, separados por caracteres *backslash* (\), se le llama paso o camino.

En esta sección se presentan 10 rutinas para obtención y selección de la unidad activa, obtención del directorio actual, creación y borrado de directorios, búsqueda de archivos y manejo del área de transferencia de disco (DTA).

Rutina	Descripción
_SetCurDrive	Establece la unidad de disco activa y devuelve en AX el número de unidades lógicas presentes en el sistema.
_GetCurDrive	Devuelve en AX la unidad de disco activa (A= 0,B= 1,etc.)
_GetDTA	Establece al área de transferencia de disco a ser usada por el sistema operativo.
_SetDTA	Devuelve en DX:AX el segmento:desplazamiento del DTA activo.
_SetCurDir	Permite establecer el directorio activo. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.
_GetCurDir	Devuelve una cadena ASCIIZ con el nombre y paso del directorio activo en la unidad de disco especificada. En caso de error se enciende el bit de acarreo y AX contiene el código del error.
_CreateDir	Permite la creación de un subdirectorio nuevo. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.
_RemoveDir	Permite la eliminación del subdirectorio especificado. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.
_FindFirst	Encuentra el primer archivo que cumpla con las especificaciones dadas en una cadena ASCIIZ, y coloca la información relacionada con dicho archivo, en el DTA activo. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.
_FindNext	Encuentra los archivos adicionales que cumplan con las especificaciones dadas en la invocación a _FindFirst. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Rutina _SetCurDrive: Establece la unidad de disco activa y devuelve en AX el número de unidades lógicas presentes en el sistema.
Invocación: push <unidad de disco>
call _SetCurDrive

```
                PUBLIC _SetCurDrive
_SetCurDrive    PROC NEAR
                ARG Drive:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros

                mov ah,0Eh              ; función 0Eh
                mov dl,Drive
                int 21h                 ; del DOS
                xor ah,ah               ; AH= 0

                pop dx                  ; recuperar registros
                pop bp
                ret ArgLen
_SetCurDrive    ENDP
```

Rutina _GetCurDrive: Devuelve en AX la unidad de disco activa (A= 0,B= 1,etc.)
Invocación: call _GetCurDrive

```
                PUBLIC _GetCurDrive
```

```
_GetCurDrive    PROC NEAR
                mov ah,19h              ; función 19h
                int 21h                 ; del DOS
                xor ah,ah               ; AH= 0
                ret
_GetCurDrive    ENDP
```

Rutina _SetDTA: Establece al área de transferencia de disco a ser usada por el sistema operativo.
Invocación: push SEG <DTA>
push OFFSET <DTA>
call _SetDTA

```
                PUBLIC _SetDTA
_SetDTA         PROC NEAR
                ARG DTA:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push dx
                push ds

                mov ah,1Ah              ; función 1Ah
                lds dx,DTA
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop ax
                pop bp
                ret ArgLen
_SetDTA         ENDP
```

Rutina _GetDTA: Devuelve en DX:AX el segmento:desplazamiento del DTA activo.
Invocación: call _GetDTA

```
                PUBLIC _GetDTA
_GetDTA         PROC NEAR
                push bx                 ; salvar registros
                push es

                mov ah,2fh              ; función 2Fh
                int 21h                 ; del DOS
                mov dx,es
                mov ax,bx

                pop es                  ; recuperar registros
                pop bx
                ret
_GetDTA         ENDP
```

Rutina _SetCurDir: Permite establecer el directorio activo. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.
Invocación: push SEG <nombre del subdirectorio>
push OFFSET <nombre del subdirectorio>
call _SetCurDir

```
                PUBLIC _SetCurDir
_SetCurDir      PROC NEAR
                ARG DirName:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros
                push ds
```

```
                mov ah,3Bh              ; función 3Bh
                lds dx,DirName
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop bp
                ret ArgLen
_SetCurDir      ENDP
```

Rutina _GetCurDir: Devuelve una cadena ASCIIZ con el nombre y paso del directorio activo en la unidad de disco especificada. En caso de error se enciende el bit de acarreo y AX contiene el código del error.

Invocación:
```
push <unidad de disco>
push SEG <Var>
push OFFSET <Var>
call _GetCurDir
```

```
                PUBLIC _GetCurDir
_GetCurDir      PROC NEAR
                ARG Var:DWORD,Drive:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros
                push si
                push ds

                mov ah,47h              ; función 47h
                mov dl,Drive
                lds si,Var
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop si
                pop dx
                pop bp
                ret ArgLen
_GetCurDir      ENDP
```

Rutina _CreateDir: Permite la creación de un subdirectorio nuevo. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Invocación:
```
push SEG <nombre del subdirectorio>
push OFFSET <nombre del subdirectorio>
call _CreateDir
```

```
                PUBLIC _CreateDir
_CreateDir      PROC NEAR
                ARG DirName:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros
                push ds

                mov ah,39h              ; función 39h
                lds dx,DirName
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop bp
                ret ArgLen
_CreateDir      ENDP
```

Rutina _RemoveDir: Permite la eliminación del subdirectorio especificado. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Invocación:
```
push SEG <nombre del subdirectorio>
push OFFSET <nombre del subdirectorio>
call _RemoveDir
```

```
                PUBLIC _RemoveDir
_RemoveDir      PROC NEAR
                ARG DirName:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros
                push ds

                mov ah,3Ah              ; función 3Ah
                lds dx,DirName
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop bp
                ret ArgLen
_RemoveDir      ENDP
```

Rutina _FindFirst: Encuentra el primer archivo que cumpla con las especificaciones dadas en una cadena ASCIIZ, y coloca la información relacionada con dicho archivo, en el DTA activo. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Invocación:
```
push SEG <especificación de archivo>
push OFFSET <especificación de archivo>
push <atributo>
call _FindFirst
```

```
                PUBLIC _FindFirst
_FindFirst      PROC NEAR
                ARG Attr:WORD,FileSpec:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push cx                 ; salvar registros
                push dx
                push ds

                mov ah,4Eh              ; función 4Eh
                mov cx,Attr
                lds dx,FileSpec
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bp
                ret ArgLen
_FindFirst      ENDP
```

Rutina _FindNext: Encuentra los archivos adicionales que cumplan con las especificaciones dadas en la invocación a _FindFirst. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Invocación:
```
call _FindNext
```

```
                PUBLIC _FindNext
_FindNext       PROC NEAR
                mov ah,4Fh              ; función 4Fh
                int 21h                 ; del DOS
                ret
_FindNext       ENDP
```

Acceso a Disco.

El sistema de almacenamiento en disco constituye el soporte externo de la información. Los datos se registran sobre la superficie del disco en una serie de circunferencias concéntricas llamadas pistas (*track*). Varias pistas, una por cada cara del disco (generalmente 2), componen un *cluster*. Cada pista está dividida en porciones iguales llamadas sectores. Un sector es la unidad básica de almacenamiento en disco. El tamaño de un sector se mide en bytes, y depende de las características del disco.

En esta sección se presentan 3 rutinas que permiten lectura y escritura absoluta de sectores, así como la determinación del espacio libre disponible en un disco.

Rutina	Descripción
_AbsoluteRead	Transfiere el contenido de un o más sectores del disco al buffer especificado, accesando directamente a los sectores lógicos. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.
_AbsoluteWrite	Transfiere el contenido del buffer especificado a uno o más sectores de disco, accesando directamente a los sectores lógicos. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.
_FreeDiskSpace	Devuelve en DX:AX el espacio libre en disco (en Kb). En caso de error, se enciende el bit de accarreo.

Rutina _AbsoluteRead: Transfiere el contenido de un o más sectores del disco al buffer especificado, accesando directamente a los sectores lógicos. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Invocación:
```
push <unidad de disco>
push <número de sectores a leer>
push <primer sector a leer>
push SEG <buffer>
push OFFSET <buffer>
call _AbsoluteRead
```

```
                PUBLIC _AbsoluteRead
_AbsoluteRead   PROC NEAR
                ARG Buffer:DWORD,Start:WORD,NumSect:WORD,Drive:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push dx
                push ds

                mov al,Drive            ; lectura absoluta de disco
                mov cx,NumSect
                mov dx,Start
                lds bx,Buffer
                int 25h
                pop bx

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bx
                pop bp
                ret ArgLen
_AbsoluteRead   ENDP
```

Rutina _AbsoluteWrite: Transfiere el contenido del buffer especificado a uno o más sectores de disco, accesando directamente a los sectores lógicos. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Invocación:
```
push <unidad de disco>
push <número de sectores a escribir>
push <primer sector a escribir>
push SEG <buffer>
push OFFSET <buffer>
call _AbsoluteWrite
```

```
                PUBLIC _AbsoluteWrite
_AbsoluteWrite  PROC NEAR
                ARG Buffer:DWORD,Start:WORD,NumSect:WORD,Drive:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push dx
                push ds

                mov al,Drive            ; escritura absoluta a disco
                mov cx,NumSect
                mov dx,Start
                lds bx,Buffer
                int 26h
                pop bx

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bx
                pop bp
                ret ArgLen
_AbsoluteWrite  ENDP
```

Rutina _FreeDiskSpace: Devuelve en DX:AX el espacio libre en disco (en Kb). En caso de error, se enciende el bit de accarreo.

Invocación:
```
push <unidad de disco>
call _FreeDiskSpace
```

```
                PUBLIC _FreeDiskSpace
_FreeDiskSpace  PROC NEAR
                ARG Drive:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx

                mov ah,36h              ; función 36h
                mov dl,Drive
                int 21h                 ; del DOS

                mul cx                  ; DX:AX= bytes por cluster
                mov cx,1024             ; CX= 1 Kb
                div cx                  ; DX:AX= Kb por cluster
                mul bx                  ; DX:AX= Kb libres en el disco

                pop cx                  ; recuperar registros
                pop bx
                pop bp
                ret ArgLen
_FreeDiskSpace  ENDP
```

Manejo Dinámico de Memoria

Las versiones actuales del MS-DOS pueden manejar a los sumo, 1 Mb de memoria RAM. En los PCs IBM y compatibles, la memoria ocupada por el sistema operativo abarca de la dirección 0000h hasta 09FFFFh; esta área de 640 Kb de memoria es conocida como *memoria convencional*. El rango de direcciones por encima de esta área está reservado para dispositivos periféricos, *buffers* de video, etc.

El área de memoria bajo el control del MS-DOS se divide en dos grandes secciones:

Area del sistema operativo.
Area del programa de aplicación o transiente.

El área del sistema operativo comienza a partir de la dirección 0000h y ocupa la porción más baja de la memoria. Contiene la tabla de vectores de interrupción, las tablas y *buffers* del sistema operativo, los manejadores de dispositivos instalables (*device drivers*) y la porción residente del interpretador de comandos (*COMMAND.COM*). La cantidad de memoria ocupada por el MS-DOS varía de versión a versión y depende de la cantidad de *buffers* para disco y el tamaño de los *device drivers* instalados.

El área de memoria transiente (TPA), llamada comúnmente *memory arena*, ocupa la memoria restante. Esta memoria es asignada dinámicamente en bloques llamados *arena entries*. Cada *arena entrie* posee un estructura de control especial llamada *arena header*, y todas estas estructuras forman una cadena. El MS-DOS suministra tres funciones para crear, modificar y eliminar este tipo de estructuras. Estas funciones son usadas por el propio sistema operativo cuando carga y ejecuta un programa.

Uso de las funciones de asignación dinámica de memoria.

Las funciones de asignación dinámica de memoria es empleada fundamentalmente con dos objetivos:

Ajustar la memoria ocupada por un programa al mínimo requerido, de tal forma que haya memoria suficiente para ejecutar otra aplicación.

Reservar memoria en forma dinámica a medida que esta sea requerida por un programa, y liberarla cuando ya no sea necesaria.

Ajuste de la memoria ocupada por el programa de aplicación.

Cuando el programa de aplicación comienza su ejecución, ocupa toda la memoria disponible en el área transiente, de tal manera que cualquier intento de efectuar una asignación dinámica de memoria resultaría en error, por no haber memoria disponible para ello. Esto obliga, a que los programas al ser ejecutados efectúen ajustes en la memoria reservada de tal manera que esta sea la mínima necesaria para su correcta ejecución liberando el resto para otras aplicaciones, o para asignación dinámica.

En la sección actual, se presentan 4 rutinas, que pueden ser usadas para creación, ajuste y eliminación de bloques de memoria dinámicos y para optimización del espacio de memoria empleado por un programa de aplicación.

Rutina	Descripción
_ProgMem	Libera la memoria no usada por el programa, para que esta pueda ser asignada dinámicamente. Debe ser empleada antes de usar _Malloc o _Realloc. En caso de error, se enciende el bit de acarreo.
_Malloc	Asigna dinámicamente la cantidad de párrafos de memoria especificados y devuelve en AX el segmento base del bloque de memoria. Un párrafo es igual a 16 bytes. En caso de error, el bit de acarreo se enciende y AX contiene el código del error y DX el mayor bloque de memoria disponible.

_Realloc	Modifica el tamaño de un bloque de memoria asignado dinámicamente. En caso de error, se enciende el bit de acarreo, AX contiene el código del error y DX contiene el mayor bloque de memoria disponible.
_Free	Libera un bloque de memoria reservado dinámicamente. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Rutina _ProgMem: Libera la memoria no usada por el programa, para que esta pueda ser asignada dinámicamente. Debe ser empleada antes de usar _Malloc o _Realloc. En caso de error, se enciende el bit de acarreo.

Invocación: push <segmento del PSP>
push <número de párrafos empleados por el programa>
call _ProgMem

```
                PUBLIC _ProgMem
_ProgMem        PROC NEAR
                ARG Size:WORD,PSPSeg:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push es

                mov ah,4Ah              ; ajustar el tamaño del bloque de memoria
                mov bx,Size             ; asignado al programa de aplicación
                mov es,PSPSeg
                int 21h

                pop es                  ; recuperar registros
                pop bx
                pop ax
                pop bp
                ret ArgLen
_ProgMem        ENDP
```

Rutina _Malloc: Asigna dinámicamente la cantidad de párrafos de memoria especificados y devuelve en AX el segmento base del bloque de memoria. Un párrafo es igual a 16 bytes. En caso de error, el bit de acarreo se enciende y AX contiene el código del error y DX el mayor bloque de memoria disponible.

Invocación: push <número de párrafos requeridos>
call _Malloc

```
                PUBLIC _Malloc
_Malloc         PROC NEAR
                ARG Size:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros

                mov ah,48h              ; función 48h
                mov bx,Size
                int 21h                 ; del DOS
                jnc @@exit

                mov dx,bx               ; en caso de error, DX= mayor bloque disponible

@@exit:         pop bx                  ; recuperar registros
                pop bp
                ret ArgLen
_Malloc         ENDP
```

Rutina _Realloc: Modifica el tamaño de un bloque de memoria asignado dinámicamente. En caso de error, se enciende el bit de acarreo, AX contiene el código del error y DX contiene el mayor bloque de memoria disponible.

Invocación: push <segmento del bloque de memoria>
push <nuevo tamaño requerido>
call _Realloc

```
                PUBLIC _Realloc
_Realloc        PROC NEAR
                ARG Size:WORD,MemSeg:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push es

                mov ah,4Ah              ; función 4Ah
                mov bx,Size
                mov es,MemSeg
                int 21h                 ; del DOS
                jnc @@exit

                mov dx,bx               ; en caso de error, DX= mayor bloque disponible

@@exit:         pop es                  ; recuperar registros
                pop bx
                pop bp
                ret ArgLen
_Realloc        ENDP
```

Rutina _Free: Libera un bloque de memoria reservado dinámicamente. En caso de error, se enciende el bit de acarreo y AX contiene el código del error.

Invocación: push <segmento del bloque de memoria>
call _Free

```
                PUBLIC _Free
_Free           PROC NEAR
                ARG MemSeg:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push es                 ; salvar registros

                mov ah,49h              ; función 49h
                mov es,MemSeg
                int 21h                 ; del DOS

                pop es                  ; recuperar registros
                pop bp
                ret ArgLen
_Free           ENDP
```

Impresora y Puerto Serial.

El MS-DOS soporta el uso de impresoras, plotters, modems y otros dispositivos de impresión o comunicaciones a través de *device drivers* para los puertos paralelo y serial. Los puertos paralelos reciben este nombre, por transfieren la información (8 bits) en paralelo a través de cables independientes. Los puertos serial transfieren la información *serialmente* (1 bit a la vez) a través de un única conexión física.

Los puertos paralelos se emplean generalmente para dispositivos de salida de alta velocidad, tales como impresoras, a una distancia corta. Los puertos seriales son empleados para dispositivos lentos, como modems y terminales, que requieren comunicación bidireccional con el computador y a mayores distancias (normalmente, hasta 1 kilómetro).

El interfaz más común para puertos seriales es el llamado RS-232. Este estándar especifica una conexión de 25 cables con ciertas características eléctricas, el uso de ciertas señales de control (*handshaking*) y un conector estándar DB-25.

El MS-DOS incorpora manejadores para tres adaptadores paralelos y dos seriales. Los nombres lógicos de estos adaptadores son LPT1, LPT2 y LPT3 para los puertos paralelos y COM1 y COM2 para los puertos seriales. El dispositivo estándar de impresión (PRN1) y dispositivo estándar auxiliar (AUX) están generalmente asociados a LPT1 y COM1 respectivamente.

Al igual que en el caso de manejo de teclado y video, existen tres niveles de manejo para los puertos paralelo y serial:

Funciones del MS-DOS orientadas a manejador (*handle*).

Funciones tradicionales del MS-DOS.

Funciones del BIOS.

En esta sección se presentan 8 rutinas, 5 para manejo del puerto paralelo (impresora) y 3 para el serial.

Rutina	Descripción
_ResetPrinter	Inicializa el puerto de la impresora y devuelve el estado de la misma (en AX).
_GetPrinterStatus	Devuelve en AX el estado del puerto de la impresora.
_LPutC	Envía un carácter al puerto de la impresora y devuelve en AX el estado del mismo.
_LPutCR	Envía un CR/LF a la impresora, empleando _LPutC
_LPutS	Envía una cadena de caracteres al dispositivo estándar de impresión.
_ComReset	Inicializa el puerto serial de comunicaciones. Devuelve en AH el estado del puerto y en AL el estado del modem.
_ComWrite	Escribir un carácter en el puerto serial de comunicaciones. En caso de error, se enciende el bit 7 de AX y los bits 0 al 6 contienen el código del error.
_ComRead	Lee un carácter del puerto serial de comunicaciones y lo devuelve en AX. En caso de error, se enciende el bit 7 de AX y los bits 0 al 6 contienen el código del error.

Rutina _ResetPrinter: Inicializa el puerto de la impresora y devuelve el estado de la misma (en AX).
Invocación: push <número del puerto>
call _ResetPrinter

```
                PUBLIC _ResetPrinter
_ResetPrinter   PROC NEAR
                ARG Port:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros

                mov ah,01h              ; función 01h
                mov dx,Port
                int 17h                 ; del BIOS (impresora)

                mov al,ah               ; AX= estado de la impresora
                xor ah,ah

                pop dx                  ; recuperar registros
                pop bp
                ret ArgLen
```

```
_ResetPrinter       ENDP
```

Rutina _GetPrinterStatus: Devuelve en AX el estado del puerto de la impresora.
Invocación: push <número del puerto>
call _GetPrinterStatus

```
                    PUBLIC _GetPrinterStatus
_GetPrinterStatus   PROC NEAR
                    ARG Port:WORD= ArgLen
                    push bp                 ; salvar BP
                    mov bp,sp               ; permitir acceso a los argumentos
                    push dx                 ; salvar registros

                    mov ah,02h              ; función 02h
                    mov dx,Port
                    int 17h                 ; del BIOS (impresora)

                    mov al,ah               ; AX= estado de la impresora
                    xor ah,ah

                    pop dx                  ; recuperar registros
                    pop bp
                    ret ArgLen
_GetPrinterStatus   ENDP
```

Rutina _LPutC: Envía un carácter al puerto de la impresora y devuelve en AX el estado del mismo.
Invocación: push <número del puerto>
push <carácter>
call _LPutC

```
                    PUBLIC _LPutC
_LPutC              PROC NEAR
                    ARG Char:BYTE,Port:WORD= ArgLen
                    push bp                 ; salvar BP
                    mov bp,sp               ; permitir acceso a los argumentos
                    push dx                 ; salvar registros

                    mov ah,00h              ; función 00h
                    mov al,Char
                    mov dx,Port
                    int 17h                 ; del BIOS (impresora)

                    mov al,ah               ; AX= estado de la impresora
                    xor ah,ah

                    pop dx                  ; recuperar registros
                    pop bp
                    ret ArgLen
_LPutC              ENDP
```

Rutina _LPutCR: Envía un CR/LF a la impresora, empleando _LPutC
Invocación: push <número del puerto>
call _LPutCR

```
                    PUBLIC _LPutCR
_LPutCR             PROC NEAR
                    ARG Port:WORD= ArgLen
                    push bp                 ; salvar BP
                    mov bp,sp               ; permitir acceso a los argumentos

                    push Port
                    mov ax,CR
                    push ax
```

```
                call _LPutC             ; imprimir retorno de carro

                push Port
                mov ax,LF
                push ax
                call _LPutC             ; imprimir salto de línea

                pop bp                  ; recuperar registros
                ret ArgLen
_LPutCR         ENDP
```

Rutina _LPutS: Envía una cadena de caracteres al dispositivo estándar de impresión.
Invocación: push SEG <cadena>
push OFFSET <cadena>
push <longitud cadena>
call _LPutS

```
                PUBLIC _LPutS
_LPutS          PROC NEAR
                ARG  Length:WORD,Var:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ds                 ; salvar registros

                mov ax,StdPrn           ; dispositivo estándar de salida
                push ax
                lds ax,Var
                push ds
                push ax
                push Length
                call _FWrite            ; función FWrite (ver manejo de archivos)

                pop ds                  ; recuperar registros
                pop bp
                ret ArgLen
_LPutS          ENDP
```

Rutina _ComReset: Inicializa el puerto serial de comunicaciones. Devuelve en AH el estado del puerto y en AL el estado del modem.
Invocación: push <número del puerto>
push <parámetro de inicialización>
call _ComReset

```
                PUBLIC _ComReset
_ComReset       PROC NEAR
                ARG iPar:BYTE,PortNo:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros

                mov ah,00h              ; función 00h
                mov al,iPar
                mov dx,PortNo
                int 14h                 ; del BIOS (puerto serial)

                pop dx                  ; recuperar registros
                pop bp
                ret ArgLen
_ComReset       ENDP
```

Rutina _ComWrite: Escribir un carácter en el puerto serial de comunicaciones. En caso de error, se enciende el bit 7 de AX y los bits 0 al 6 contienen el código del error.
Invocación: push <número del puerto>

```
        push <carácter>
        call _ComWrite

                PUBLIC _ComWrite
_ComWrite       PROC NEAR
                ARG Char:BYTE,PortNo:WORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros

                mov ah,01h              ; función 01h
                mov al,Char
                mov dx,PortNo
                int 14h                 ; del BIOS (puerto serial)
                mov al,ah               ; AX= estado del puerto
                xor ah,ah

                pop dx                  ; recuperar registros
                pop bp
                ret ArgLen
_ComWrite       ENDP
```

Rutina _ComRead: Lee un carácter del puerto serial de comunicaciones y lo devuelve en AX. En caso de error, se enciende el bit 7 de AX y los bits 0 al 6 contienen el código del error.

Invocación:
```
        push <número del puerto>
        call _ComWrite

                PUBLIC _ComRead
_ComRead        PROC NEAR
                ARG PortNo:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push dx                 ; salvar registros

                mov ah,02h              ; función 02h
                mov dx,PortNo
                int 14h                 ; del BIOS (puerto serial)
                test ah,80h             ; si hay error, AX= código
                jz @@exit
                mov al,ah
                xor ah,ah

@@exit:         pop dx                  ; recuperar registros
                pop bp
                ret ArgLen
_ComRead        ENDP
```

Ratón.

Los manejadores para dispositivos apuntadores (*pointing devices*), son suministrados por el fabricante y cargados en memoria por medio de la sentencia **DEVICE** en el archivo **CONFIG.SYS**. Aunque las características de *hardware* de estos dispositivos difiere significativamente, casi todos presentan el mismo interfaz de *software*: la Int 33h, protocolo usado por el manejador de ratón de Microsoft.

Empleando este interfaz, se presentan en esta sección 6 rutinas que implementan el manejo mínimo del ratón para ser usado en una aplicación.

Rutina	Descripción
_ResetMouse	Inicializa el manejador del ratón y devuelve en AX el número de botones del ratón. En caso de error, AX contiene 0.
_ShowMousePointer	Muestra el cursor del ratón.

_HideMousePointer	Esconde el cursor del ratón.
_GetMouseButtonStatus	Devuelve en AX el estado de los botones del ratón.
_GetMousePosition	Devuelve en DX:AX la posición actual del cursor del ratón.
_SetMouseXY	Establece la posición del cursor del ratón.

Rutina _ResetMouse: Inicializa el manejador del ratón y devuelve en AX el número de botones del ratón. En caso de error, AX contiene 0.
Invocación: call _ResetMouse

```
                PUBLIC _ResetMouse
_ResetMouse     PROC NEAR
                LOCALS
                push bx                 ; salvar registros

                mov ax,0000h            ; función 0000h
                int 33h                 ; del ratón
                cmp ax,0FFFFh
                jne @@exit              ; si hay error, terminar
                mov ax,bx               ; si no, AX= # de botones

@@exit:         pop bx                  ; recuperar registros
                ret
_ResetMouse     ENDP
```

Rutina _ShowMousePointer: Muestra el cursor del ratón.
Invocación: call _ShowMousePointer

```
                PUBLIC _ShowMousePointer
_ShowMousePointer PROC NEAR
                push ax                 ; salvar registros

                mov ax,0001h            ; función 0001h
                int 33h                 ; del ratón

                pop ax                  ; recuperar registros
                ret
_ShowMousePointer ENDP
```

Rutina _HideMousePointer: Esconde el cursor del ratón.
Invocación: call _HideMousePointer

```
                PUBLIC _HideMousePointer
_HideMousePointer PROC NEAR
                push ax                 ; salvar registros

                mov ax,0002h            ; función 0002h
                int 33h                 ; del ratón

                pop ax                  ; recuperar registros
                ret
_HideMousePointer ENDP
```

Rutina _GetMouseButtonStatus: Devuelve en AX el estado de los botones del ratón.
Invocación: call _GetMouseButtonStatus

```
                PUBLIC _GetMouseButtonStatus
_GetMouseButtonStatus PROC NEAR
                push bx                 ; salvar registros
                push cx
                push dx

                mov ax,0003h            ; función 0003h
                int 33h                 ; del ratón
```

```
                        mov ax,bx               ; AX= estado de los botones

                        pop dx                  ; recuperar registros
                        pop cx
                        pop bx
                        ret
_GetMouseButtonStatus   ENDP
```

Rutina _GetMousePosition: Devuelve en DX:AX la posición actual del cursor del ratón.
Invocación: call _GetMousePosition

```
                        PUBLIC _GetMousePosition
_GetMousePosition       PROC NEAR
                        push bx                 ; salvar registros
                        push cx

                        mov ax,0003h            ; función 0003h
                        int 33h                 ; del ratón
                        mov ax,cx               ; AX= posición del ratón

                        pop cx                  ; recuperar registros
                        pop  bx
                        ret
_GetMousePosition       ENDP
```

Rutina _SetMouseXY: Establece la posición del cursor del ratón.
Invocación: push <coordenada X>
push <coordenada Y>
call _SetMouseXY

```
                        PUBLIC _SetMouseXY
_SetMouseXY             PROC NEAR
                        ARG Y:WORD,X:WORD= ArgLen
                        push bp                 ; salvar BP
                        mov bp,sp               ; permitir acceso a los argumentos
                        push ax                 ; salvar registros
                        push cx
                        push dx

                        mov  ax,0004h           ; función 0004h
                        mov cx,X
                        mov dx,Y
                        int 33h                 ; del ratón

                        pop dx                  ; recuperar registros
                        pop cx
                        pop ax
                        pop bp
                        ret ArgLen
_SetMouseXY             ENDP
```

Conversión de Datos.

La información e un microcomputador, puede estar representada de diversas maneras dependiendo del tipo de manejo que de ella se quiera hacer. Las rutinas de conversión de datos, presentadas en esta sección (15 en total) permiten la transformación de un bloque de información de un representación a otra. Las cadenas manejadas son del tipo ASCIIZ. Una cadena ASCIIZ es un secuencia de cero o más elementos, terminada en un carácter nulo (código ASCII 0).

Rutina	Descripción
_UpCase	Convierte a mayúscula (en AX) el carácter dado.
_LoCase	Convierte a minúscula (en AX) el carácter dado.

_Atoi	Convierte una cadena ASCIIZ en un entero. Devuelve el entero en AX.
_Atou	Convierte una cadena ASCIIZ en un entero sin signo. Devuelve el entero en AX.
_Atox	Convierte una cadena ASCIIZ en un entero sin signo en hexadecimal. Devuelve el entero en AX.
_Atol	Convierte una cadena ASCIIZ en un entero largo. El entero largo es devuelto en DX:AX.
_Atoul	Convierte una cadena ASCIIZ en un entero largo sin signo. El entero largo es devuelto en DX:AX.
_Itoa	Convierte un entero, en una cadena ASCIIZ.
_Utoa	Convierte un entero sin signo, en una cadena ASCIIZ.
_Ltoa	Convierte un entero largo, en una cadena ASCIIZ.
_Ultoa	Convierte un entero largo sin signo, en una cadena ASCIIZ.
_Htoa	Convierte un número hexadecimal de 8 bits , en un cadena ASCIIZ.
_Wtoa	Convierte un número hexadecimal de 16 bits, en una cadena ASCIIZ.
_Str$toASCIIZ	Convierte una cadena terminada en $ a una cadena ASCIIZ.
_StrtoASCIIZ	Convierte un cadena de caracteres a una cadena ASCIIZ.

Rutinas _UpCase: Convierte a mayúscula (en AX) el carácter dado.
Invocación: push <carácter>
call _UpCase

```
                PUBLIC _UpCase
_UpCase         PROC NEAR
                ARG Chr:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos

                mov al,Chr              ; AX= carácter
                xor ah,ah

                push ax
                call _IsLower
                jnc @@1
                sub al,'a' -'A'         ; si está entre 'a' y 'z' convertir
                jmp SHORT @@ok

@@1:            cmp al,'á'              ; si es igual a 'á', convertir
                jne @@2
                mov al,'A'
                jmp SHORT @@ok

@@2:            cmp al,'é'              ; si es igual a 'é', convertir
                jne @@3
                mov al,'E'
                jmp SHORT @@ok

@@3:            cmp al,'í'              ; si es igual a 'í', convertir
                jne @@4
                mov al,'I'
                jmp SHORT @@ok

@@4:            cmp al,'ó'              ; si es igual a 'ó', convertir
                jne @@5
                mov al,'O'
                jmp SHORT @@ok

@@5:            cmp al,'ú'              ; si es igual a 'ú', convertir
```

```
                jne @@6
                mov al,'U'
                jmp SHORT @@ok

@@6:            cmp al,'ñ'              ; si es igual a 'ñ', convertir
                jne @@7
                mov al,'Ñ'
                jmp SHORT @@ok

@@7:            cmp al,'ü'              ; si es igual a 'ü', convertir
                jne @@ok
                mov al,'U'

@@ok:           pop bp                  ; recuperar registros
                ret ArgLen
_UpCase         ENDP
```

Rutina _LoCase: Convierte a minúscula (en AX) el carácter dado.
Invocación: push <carácter>
call _LoCase

```
                PUBLIC _LoCase
_LoCase         PROC NEAR
                ARG Chr:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos

                mov al,Chr              ; AX= carácter
                xor ah,ah

                push ax
                call _IsUpper
                jnc @@1
                add al,'a'-'A'          ; si está entre 'A' y 'Z', convertir
                jmp SHORT @@ok

@@1:            cmp al,'Ñ'              ; si es 'Ñ', convertir
                jne @@ok
                mov al,'ñ'

@@ok:           pop bp                  ; recuperar registros
                ret ArgLen
_LoCase         ENDP
```

Rutina _Atoi: Convierte una cadena ASCIIZ en un entero. Devuelve el entero en AX.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _Atoi

```
                PUBLIC _Atoi
_Atoi           PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push si
                push ds

                lds si,Str              ; DS:SI= cadena
                xor bx,bx               ; BX= 0

                mov al,[si]
```

```
                xor ah,ah
                push ax
                call _IsDigit           ; si el primer carácter es un dígito, convertir
                jc @@1
                cmp al,"-'              ; si no, determinar el signo
                je @@2
                cmp al,'+'
                jne @@exit
                jmp SHORT @@3

@@2:            inc bx
@@3:            inc si

@@1:            push ds
                push si
                call _Atou              ; convertir

                or bx,bx                ; si el signo era '-', calcular complemento a dos
                jz @@exit
                neg ax

@@exit:         pop ds                  ; recuperar registros
                pop si
                pop bx
                pop bp
                ret ArgLen
_Atoi           ENDP
```

Rutina _Atou: Convierte una cadena ASCIIZ en un entero sin signo. Devuelve el entero en AX.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _Atou

```
                PUBLIC _Atou
_Atou           PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push dx
                push si
                push ds

                lds si,Str              ; DS:SI= cadena
                xor cx,cx               ; CX= 0
                xor ax,ax               ; AX= 0

@@1:            mov al,[si]             ; AX= carácter
                push ax
                call _IsDigit
                jnc @@eos               ; si no es un dígito, terminar
                sub al,'0'              ; convertir el carácter a un número
                push ax                 ; almacenar en la pila
                inc cx                  ; incrementar contador
                inc si                  ; siguiente carácter
                jmp SHORT @@1

@@eos:          or cx,cx
                jz @@exit               ; si la cadena es nula, salir

                xor si,si               ; SI= 0
                mov bx,1                ; BX= 1
```

```
@@2:        pop ax                  ; recuperar carácter
            mul bx                  ; multiplica por el peso de la posición
            add si,ax               ; acumula en SI
            mov ax,10
            mul bx
            mov bx,ax               ; BX= BX*10
            loop @@2

@@exit:     mov ax,si               ; AX= entero
            pop ds                  ; recuperar registro
            pop si
            pop dx
            pop cx
            pop bx
            pop bp
            ret ArgLen
_Atou       ENDP
```

Rutina _Atox: Convierte una cadena ASCIIZ en un entero sin signo en hexadecimal. Devuelve el entero en AX.

Invocación:
```
push SEG <cadena>
push OFFSET <cadena>
call _Atox
```

```
            PUBLIC _Atox
_Atox       PROC NEAR
            ARG Str:DWORD= ArgLen
            LOCALS
            push bp                 ; recuperar BP
            mov bp,sp               ; permitir acceso a los argumentos
            push bx                 ; salvar registros
            push cx
            push dx
            push si
            push ds

            lds si,Str              ; DS:SI = cadena
            xor cx,cx               ; CX= 0
            xor ax,ax               ; AX= 0

@@1:        mov al,[si]             ; AX= carácter
            push ax
            call _IsXDigit          ; si no es un dígito hexadecimal,
            jnc @@eos               ; terminar
            sub al,'0'              ; convertir el dígito hexadecimal
            cmp al,9
            jbe @@3
            sub al,'A'-'9'-1

@@3:        push ax                 ; almacenar
            inc cx                  ; incrementar contador
            inc si                  ; siguiente carácter
            jmp SHORT @@1

@@eos:      or cx,cx                ; si la cadena es nula, salir
            jz @@exit

            xor si,si               ; SI= 0
            mov bx,1
@@2:        pop ax
            mul bx
            add si,ax               ; SI= acumulador
            mov ax,16
            mul bx
```

```
                mov bx,ax               ; BX= BX*16
                loop @@2

@@exit:         mov ax,si               ; AX= entero
                pop ds                  ; recuperar registro
                pop si
                pop dx
                pop cx
                pop bx
                pop bp
                ret ArgLen
_Atox           ENDP
```

Rutina _Atol: Convierte una cadena ASCIIZ en un entero largo. El entero largo es devuelto en DX:AX.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _Atol

```
                PUBLIC _Atol
_Atol           PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push si
                push ds

                lds si,Str              ; DS:SI= cadena
                xor bx,bx               ; BX= 0

                mov al,[si]             ; AX= carácter
                xor ah,ah
                push ax
                call _IsDigit
                jc @@1                  ; si no es dígito, identificar signo
                cmp al,'-'
                je @@2
                cmp al,'+'
                jne @@exit
                jmp SHORT @@3

@@2:            inc bx                  ; BX= BX+1
@@3:            inc si                  ; incrementar SI

@@1:            push ds
                push si
                call _Atoul             ; convertir

                or bx,bx
                jz @@exit               ; si el número es negativo,
                not ax                  ; calcular complemento a dos
                not dx
                inc ax
                adc dx,0

@@exit:         pop ds                  ; recuperar registros
                pop si
                pop bx
                pop bp
                ret ArgLen
_Atol           ENDP
```

Rutina _Atoul: Convierte una cadena ASCIIZ en un entero largo sin signo. El entero largo es devuelto en DX:AX.

Invocación:
```
push SEG <cadena>
push OFFSET <cadena>
call _Atoul
```

```
                PUBLIC _Atoul
_Atoul          PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push di
                push si
                push ds

                lds si,Str              ; DS:SI= cadena
                xor cx,cx               ; CX= 0
                xor ax,ax               ; AX= 0

@@1:            mov al,[si]             ; AX= carácter
                push ax
                call _IsDigit
                jnc @@eos               ; si no es dígito, terminar
                sub al,'0'              ; convertir
                push ax
                inc cx                  ; incrementa contador
                inc si                  ; siguiente carácter
                jmp SHORT @@1

@@eos:          or cx,cx                ; si la cadena es nula, salir
                jz @@exit

                xor si,si               ; SI:DI= acumulador
                xor di,di
                mov ax,1
                xor dx,dx
@@2:            pop bx                  ; recuperar carácter
                push ax
                push dx
                call Lmul
                add di,ax               ; acumular
                adc si,dx
                pop dx
                pop ax
                mov bx,10
                call Lmul
                loop @@2

                mov ax,di               ; DX:AX= entero largo
                mov dx,si

@@exit:         pop ds                  ; recuperar registros
                pop si
                pop di
                pop cx
                pop bx
                pop bp
                ret ArgLen
_Atoul          ENDP
```

Rutina _Itoa: Convierte un entero, en una cadena ASCIIZ.

Invocación: push <entero>
push SEG <cadena ASCIIZ>
push OFFSET <cadena ASCIIZ>
call _Itoa

```
                PUBLIC _Itoa
_Itoa           PROC NEAR
                ARG Str:DWORD,Value:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push dx
                push di
                push ds

                mov ax,Value            ; AX= entero
                lds di,Str              ; DS:SI= cadena

                test ax,8000h           ; si es negativo, incluir signo '-'
                jnz @@1
                mov BYTE PTR [di],' '
                jmp SHORT @@2

@@1:            mov BYTE PTR [di],'-'
                neg ax                  ; calcular complemento a dos

@@2:            inc di
                push ax
                push ds
                push di
                call _Utoa              ; convertir

                pop ds                  ; recuperar registros
                pop di
                pop dx
                pop ax
                pop bp
                ret ArgLen
_Itoa           ENDP
```

Rutina _Utoa: Convierte un entero sin signo, una cadena ASCIIZ.
Invocación: push <entero>
push SEG <cadena ASCIIZ>
push OFFSET <cadena ASCIIZ>
call _Utoa

```
                PUBLIC _Utoa
_Utoa           PROC NEAR
                ARG Str:DWORD,Value:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx
                push di
                push es

                mov ax,Value            ; AX= valor
                les di,Str              ; ES:DI= cadena
                cld                     ; incrementar contadores
```

```
                mov bx,10               ; base 10
                xor cx,cx               ; contador= 0
@@1:            xor dx,dx               ; residuo= 0
                div bx                  ; AX= AX/10
                add dx,'0'              ; convertir a ASCII
                push dx                 ; almacenar
                inc cx                  ; incrementa contador
                or ax,ax                ; si el cociente es igual a 0, terminar
                jnz @@1

@@2:            pop ax                  ; almacenar a cadena ASCIIZ
                stosb
                loop @@2

                xor al,al
                stosb

                pop es                  ; recuperar registros
                pop di
                pop dx
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_Utoa           ENDP
```

Rutina _Ltoa: Convierte un entero largo, en una cadena ASCIIZ.
Invocación: push <16 bits menos significativos del valor>
push <16 bits más significativos del valor>
push SEG <cadena ASCIIZ>
push OFFSET <cadena ASCIIZ>
call _Ltoa

```
                PUBLIC _Ltoa
_Ltoa           PROC NEAR
                ARG Str:DWORD,HiWord:WORD,LoWord:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push dx
                push di
                push ds

                mov ax,LoWord           ; DX:AX= valor
                mov dx,HiWord
                lds di,Str              ; DS:DI= cadena

                test dx,8000h
                jnz @@1
                mov BYTE PTR [di],' '
                jmp SHORT @@2

@@1:            mov BYTE PTR [di],'-'   ; si es negativo, incluir signo '-' en la cadena
                not dx                  ; y calcular complemento a dos del valor
                not ax
                inc ax
                adc dx,0

@@2:            inc di
                push ax                 ; convertir
                push dx
```

```
                push ds
                push di
                call _Ultoa

                pop ds                  ; recuperar registros
                pop di
                pop dx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_Ltoa           ENDP
```

Rutina _Ultoa: Convierte un entero largo sin signo, en una cadena ASCIIZ.
Invocación: push <16 bits menos significativos del valor>
push <16 bits más significativos del valor>
push SEG <cadena ASCIIZ>
push OFFSET <cadena ASCIIZ>
call _Ultoa

```
                PUBLIC _Ultoa
_Ultoa          PROC NEAR
                ARG Str:DWORD,HiWord:WORD,LoWord:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx
                push di
                push es

                xor cx,cx               ; contador= 0
                mov ax,LoWord           ; DX:AX= valor
                mov dx,HiWord
                les di,Str              ; ES:DI= cadena
                cld                     ; incrementar apuntadores

@@1:            mov bx,10               ; DX:AX= DX:AX/10
                call Ldiv
                add bx,'0'              ; convertir residuo a ASCII
                push bx                 ; salvar
                inc cx                  ; incrementar contador
                mov bx,ax               ; si el cociente es igual a cero, terminar
                or bx,dx
                jnz @@1

@@2:            pop ax                  ; almacenar dígitos en la cadena
                stosb
                loop @@2

                xor al,al
                stosb

                pop es                  ; recuperar registros
                pop di
                pop dx
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_Ultoa          ENDP
```

Rutina _Htoa: Convierte un número hexadecimal de 8 bits , en un cadena ASCIIZ.
Invocación: push <valor>
push SEG <cadena ASCIIZ>
push OFFSET <cadena ASCIIZ>
call _Htoa

```
                PUBLIC _Htoa
_Htoa           PROC NEAR
                ARG Str:DWORD,Val:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx
                push di
                push es

                mov al,Val              ; AL= valor
                les di,Str              ; ES:DI= cadena
                cld                     ; incrementar apuntadores

                mov cx,2                ; contador= 2
                mov bl,16               ; base 16 (hexadecimal)
                xor dh,dh
@@1:            xor ah,ah               ; AL= AL/16
                div bl
                mov dl,ah               ; convertir residuo a ASCII
                add dl,'0'
                cmp dl,'9'
                jbe @@2
                add dl,'A'-'9'-1

@@2:            push dx                 ; salvar
                loop @@1

                mov cx,2
@@3:            pop ax                  ; almacenar dígitos en la cadena ASCIIZ
                stosb
                loop @@3

                xor al,al
                stosb

                pop ds                  ; recuperar registros
                pop di
                pop dx
                pop cx
                pop bx
                pop ax

                pop bp
                ret ArgLen
_Htoa           ENDP
```

Rutina _Wtoa: Convierte un número hexadecimal de 16 bits, en una cadena ASCIIZ.
Invocación: push <valor>
push SEG <cadena ASCIIZ>
push OFFSET <cadena ASCIIZ>
call _Wtoa

```
                PUBLIC _Wtoa
_Wtoa           PROC NEAR
                ARG Str:DWORD,Val:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push di
                push ds

                mov ax,Val              ; AX= valor
                lds di,Str              ; DS:SI= cadena

                xchg al,ah              ; intercambia byte alto con bajo
                push ax
                push ds
                push di
                call _Htoa              ; convierte byte más significativo

                mov al,ah
                add di,2
                push ax
                push ds
                push di
                call _Htoa              ; convierte byte menos significativo

                pop ds                  ; recuperar registros
                pop di
                pop ax
                pop bp
                ret ArgLen
_Wtoa           ENDP
```

Rutina _Str$toASCIIZ: Convierte una cadena terminada en $ a una cadena ASCIIZ.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _Str$toASCIIZ

```
                PUBLIC _Str$toASCIIZ
_Str$toASCIIZ   PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push di
                push es

                cld                     ; incrementa apuntador
                les di,Str              ; ES:DI= cadena
                mov al,'$'
@@1:            scasb                   ; localizar el carácter '$'
                jne @@1

                mov BYTE PTR [es:di-1],0    ; sustituir el carácter '$' por un código ASCII 0

                pop es                  ; recuperar registros
                pop di
                pop ax
                pop bp
                ret ArgLen
_Str$toASCIIZ   ENDP
```

Rutina _StrtoASCIIZ: Convierte un cadena de caracteres a una cadena ASCIIZ.

Invocación: push SEG <cadena>
push OFFSET <cadena>
push <longitud>
call _StrtoASCIIZ

```
                PUBLIC _StrtoASCIIZ
_StrtoASCIIZ    PROC NEAR
                ARG Len:WORD,Str:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push di
                push ds

                lds di,Str              ; DS:SI= cadena
                mov bx,Len              ; BX= longitud de la cadena

                mov BYTE PTR [di+bx],0  ; anexar un código ASCII 0 al final de la cadena

                pop ds                  ; recuperar registros
                pop di
                pop bx
                pop bp
                ret ArgLen
_StrtoASCIIZ    ENDP
```

Cadenas.

Las cadenas son secuencias de caracteres ASCII. Cuando esta secuencia va terminada en un código ASCII nulo (ASCII 0), se le llama cadena ASCIIZ. En esta sección se desarrollan 8 rutinas que permiten operaciones con cadenas ASCIIZ tales como copia, determinación de la longitud, concatenación, comparación, conversión a mayúscula o minúsculas, inversión del orden y búsqueda de un carácter en una cadena.

Rutina	Descripción
_StrCpy	Copia la cadena ASCIIZ fuente en la cadena destino.
_StrLen	Devuelve en AX la longitud de la cadena ASCIIZ dada.
_StrCat	Concatena dos cadenas ASCIIZ y el resultado lo deposita en la cadena destino.
_StrCmp	Compara dos cadenas ASCIIZ. Si son iguales el bit de cero se enciende. Si la primera cadena es alfabéticamente menor que la segunda, se enciende el bit de signo. Si la segunda cadena es alfabéticamente mayor que la segunda, los bit de cero y signo permanecen apagados.
_StrRev	Invierte el contenido de una cadena ASCIIZ.
_StrChr	Devuelve en AX la posición del carácter especificado dentro de la cadena ASCIIZ dada. Si el carácter no es encontrado, AX contiene -1.
_StrUpr	Convierte los caracteres en la cadena dada a mayúsculas.
_StrLwr	Convierte los caracteres en la cadena dada a minúsculas.

Rutina _StrCpy: Copia la cadena ASCIIZ fuente en la cadena destino.
Invocación: push SEG <cadena destino>
push OFFSET <cadena destino>
push SEG <cadena fuente>
push SEG <cadena fuente>
call _StrCpy

```
                PUBLIC _StrCpy
_StrCpy         PROC NEAR
                ARG  SrcStr:DWORD,DstStr:DWORD= ArgLen
```

```
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push di                 ; salvar registros
                push si
                push ds
                push es

                cld                     ; incrementar apuntadores
                lds si,SrcStr           ; DS:SI= cadena fuente
                les di,DstStr           ; ES:DI= cadena destino

@@loop1:  cmp BYTE PTR [si],0   ; si es fin de cadena, terminar
                je @@exit
                movsb                   ; si no, copiar
                jmp SHORT @@loop1

@@exit:         mov BYTE PTR [es:di],0  ; carácter de terminación
                pop es                  ; recuperar registros
                pop ds
                pop si
                pop di
                pop bp
                ret ArgLen
_StrCpy         ENDP
```

Rutina _StrLen: Devuelve en AX la longitud de la cadena ASCIIZ dada.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _StrLen

```
                PUBLIC _StrLen
_StrLen         PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push si                 ; salvar registros
                push ds

                xor ax,ax               ; contador= 0
                lds si,Str              ; DS:SI= cadena

@@loop1:  cmp BYTE PTR [si],0   ; si es fin de cadena, terminar
                je @@exit
                inc si                  ; si no, incrementar apuntador y contador
                inc ax
                jmp SHORT @@loop1

@@exit:         pop ds                  ; recuperar registros
                pop si
                pop bp
                ret ArgLen
_StrLen         ENDP
```

Rutina _StrCat: Concatena dos cadenas ASCIIZ y el resultado lo deposita en la cadena destino.
Invocación: push SEG <cadena destino>
push OFFSET <cadena destino>
push SEG <cadena fuente>
push SEG <cadena fuente>
call _StrCat

```
                PUBLIC _StrCat
```

```
_StrCat         PROC NEAR
                ARG SrcStr:DWORD,DstStr:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push di                 ; salvar registros
                push si
                push ds
                push es

                les si,SrcStr           ; ES:SI= cadena fuente
                lds di,DstStr           ; DS:DI= cadena destino
@@loop1: cmp BYTE PTR [di],0    ; si es fin de cadena, concatenar
                je @@1
                inc di
                jmp SHORT @@loop1

@@1:            push ds
                push di
                push es
                push si
                call _StrCpy            ; concatenar

                pop es                  ; recuperar registros
                pop ds
                pop si
                pop di
                pop bp
                ret ArgLen
_StrCat         ENDP
```

Rutina _StrCmp: Compara dos cadenas ASCIIZ. Si son iguales el bit de cero se enciende. Si la primera cadena es alfabéticamente menor que la segunda, se enciende el bit de signo. Si la segunda cadena es alfabéticamente mayor que la segunda, los bit de cero y signo permanecen apagados.

Invocación:
```
push SEG <cadena A>
push OFFSET <cadena A>
push SEG <cadena B>
push SEG <cadena B>
call _StrCmp
```

```
                PUBLIC _StrCmp
_StrCmp         PROC NEAR
                ARG SrcStr:DWORD,DstStr:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push di                 ; salvar registros
                push si
                push ds
                push es

                cld                     ; incrementar apuntadores
                lds si,SrcStr           ; DS:SI= cadena 1
                les di,DstStr           ; ES:DI= cadena 2

@@loop1: cmp BYTE PTR [si],0    ; si es final de cadena, terminar
                je @@1
                cmpsb                   ; comparar cadenas
                je @@loop1
                jmp SHORT @@2

@@1:            cmpsb

@@2:            pop es                  ; recuperar registros
```

```
                pop ds
                pop si
                pop di
                pop bp
                ret ArgLen
_StrCmp         ENDP
```

Rutina _StrRev: Invierte el contenido de una cadena ASCIIZ.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _StrRev

```
                PUBLIC _StrRev
_StrRev         PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push di
                push si
                push ds

                lds si,SrcStr           ; DS:SI= cadena
                mov di,si
@@loop1: cmp BYTE PTR [si],0    ; si es fin de cadena, terminar
                je @@loop2
                inc si
                jmp SHORT @@loop1

@@loop2: dec si                 ; invertir cadena
                cmp si,di
                jle @@exit
                mov al,[di]
                mov ah,[si]
                mov [di],ah
                mov [si],al
                inc di
                jmp SHORT @@loop2

@@exit:         pop ds                  ; recuperar registros
                pop si
                pop di
                pop ax
                pop bp
                ret ArgLen
_StrRev         ENDP
```

Rutina _StrChr: Devuelve en AX la posición del carácter especificado dentro de la cadena ASCIIZ dada. Si el carácter no es encontrado, AX contiene -1.
Invocación: push SEG <cadena>
push OFFSET <cadena>
push <carácter>
call _StrChr

```
                PUBLIC _StrChr
_StrChr         PROC NEAR
                ARG Chr:BYTE,Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push cx                 ; salvar registros
                push di
```

```
                push ds

                cld                     ; incrementar apuntadores
                mov al,Chr              ; AL= carácter
                les di,Str              ; ES:DI= cadena

@@loop1: cmp BYTE PTR [es:di],0     ; si es final de cadena, terminar
                je @@notfound
                scasb                   ; buscar el carácter
                je @@found
                inc cx                  ; incrementar el contador
                jmp SHORT @@loop1

@@found:        mov ax,cx               ; AX= posición del carácter
                jmp SHORT @@exit

@@notfound:     mov ax,-1               ; AX= -1 si el carácter no fue encontrado

@@exit:         pop es                  ; recuperar registros
                pop di
                pop cx
                pop bp
                ret ArgLen
_StrChr         ENDP
```

Rutina _StrUpr: Convierte los caracteres en la cadena dada a mayúsculas.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _StrUpr

```
                PUBLIC _StrUpr
_StrUpr         PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push di
                push es

                cld                     ; incrementar apuntadores
                xor ah,ah               ; AH= 0
                les di,Str              ; ES:DI= cadena

@@loop1: mov al,[es:di]
                or al,al
                je @@exit               ; si es fin de cadena, terminar
                push ax
                call _UpCase            ; si no, convertir a mayúscula
                stosb
                jmp SHORT @@loop1

@@exit:         pop es                  ; recuperar registros
                pop di
                pop ax
                pop bp
                ret ArgLen
_StrUpr         ENDP
```

Rutina _StrLwr: Convierte los caracteres en la cadena dada a minúsculas.
Invocación: push SEG <cadena>
push OFFSET <cadena>
call _StrLwr

```
                PUBLIC _StrLwr
_StrLwr         PROC NEAR
                ARG Str:DWORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push di
                push es

                cld                     ; incrementar apuntadores
                xor ah,ah               ; AH= 0
                les di,Str              ; ES:DI= cadena

@@loop1: mov al,[es:di]
                or al,al
                je @@exit               ; si es final de cadena, terminar
                push ax
                call _LoCase            ; si no, convertir a minúscula
                stosb
                jmp SHORT @@loop1

@@exit:         pop es                  ; recuperar registros
                pop di
                pop ax
                pop bp
                ret ArgLen
_StrLwr         ENDP
```

Misceláneos.

En esta sección se agrupan un conjunto de rutinas varias. Entre ellas tenemos, rutinas especiales para multiplicación y división de enteros largos, de acceso a la línea de comandos del DOS, manipulación de la fecha y hora del sistema, manipulación de vectores de interrupción, obtención de la versión del DOS y clasificación de caracteres.

Rutina	Descripción
_Ldiv	Divide un número de 32 bits (DX:AX) entre otro de 16 bits (BX) produciendo un cociente de 32 bits (DX:AX) y un residuo de 16 bits (BX)
_Lmul	Multiplica un número de 32 bits en DX:AX por uno de 16 bits en BX y produce un resultado de 32 bits en DX:AX
_ArgCount	Devuelve en AX el número de parámetros disponibles en la línea de comandos. Considera el espacio en blanco como separador de comandos.
_GetArg	Copia el comando especificado en la variable dada. Si la operación fue exitosa, devuelve en AX la longitud del parámetro. En caso de error devuelve -1 en AX.
_GetDate	Devuelve en un estructura SDate, la fecha del sistema. SDate tiene la siguiente estructura: SDate STRUC Day DB ? Month DB ? Year DW ? DayOfWeek DB ? ENDS
_SetDate	Establece la fecha del sistema dada en un estructura SDate. Si la operación es exitosa AX contiene 00h. En caso de error, AX contiene FFh.

_GetTime	Devuelve en un estructura STime, la hora del sistema. STime presenta la siguiente estructura: STime STRUC Hour DB ? Minutes DB ? Seconds DB ? Hundredths DB ? ENDS
_SetTime	Establece la hora del sistema dada en un estructura STime. Si la operación es exitosa AX contiene 00h. En caso de error AX contiene FFh.
_GetVect	Devuelve en DX:AX el contenido actual del vector de interrupción especificado.
_SetVect	Establece una nueva rutina manejadora de interrupciones, para el vector especificado.
_IsAlNum	Determina si el carácter dado es alfanumérico. Si lo es, el bit de acarreo se enciende.
_IsDigit	Determina si el carácter dado es un dígito decimal. Si lo es, el bit de acarreo se enciende.
_IsXDigit	Determina si el carácter dado es un dígito hexadecimal.
_IsAlpha	Determina si el carácter dado es alfabético.
_IsLower	Determina si el carácter dado es alfabético en minúscula.
_IsUpper	Determina si el carácter dado es alfabético en mayúscula.
_DOSver	Devuelve en AX, la versión del DOS.

Rutina Ldiv: Divide un número de 32 bits (DX:AX) entre otro de 16 bits (BX) produciendo un cociente de 32 bits (DX:AX) y un residuo de 16 bits (BX)

Invocación:
```
mov dx,<parte alta del dividendo>
mov ax,<parte baja del dividendo>
mov bc,<divisor>
call LDiv
```

```
Ldiv    PROC NEAR
        push cx                 ; salvar CX
        mov cx,bx               ; CX= BX
        mov bx,ax               ; BX= AX
        mov ax,dx               ; AX= DX
        xor dx,dx               ; DX= 0
        div cx                  ; AX= DX:AX/CX
        xchg ax,bx      ; AX= BX, BX= AX
        div cx                  ; AX= DX:AX/CX
        xchg dx,bx      ; DX= BX, BX= DX
        pop cx                  ; recuperar CX
        ret
Ldiv    ENDP
```

Rutina Lmul: Multiplica un número de 32 bits en DX:AX por uno de 16 bits en BX y produce un resultado de 32 bits en DX:AX

Invocación:
```
mov dx,<parte alta del primer multiplicando>
mov ax,<parte baja del primer multiplicando>
mov bx,<segundo multiplicando>
call Lmul
```

```
Lmul    PROC NEAR
        push cx                 ; salvar registros
        push si

        mov cx,dx               ; CX= DX
        mul bx                  ; DX:AX= AX*BX
        mov si,dx               ; SI= DX
```

```
                xchg ax,cx          ; AX= CX, CX= AX
                mul bx              ; DX:AX= AX*BX
                mov dx,si           ; DX= SI+AX
                add dx,ax
                mov ax,cx           ; AX= CX

                pop si              ; recuperar registros
                pop cx
                ret
Lmul            ENDP
```

Rutina _ArgCount: Devuelve en AX el número de parámetros disponibles en la línea de comandos. Considera el espacio en blanco como separador de comandos.

Invocación: push <segmento PSP>
call _ArgCount

```
                PUBLIC _ArgCount
_ArgCountPROC NEAR
                ARG PSPseg:WORD= ArgLen
                LOCALS
                push bp             ; salvar BP
                mov bp,sp           ; permitir acceso a los argumentos
                push cx             ; salvar registros
                push si
                push ds

                mov ds,PSPseg       , DS= segmento del PSP
                mov cl,[ds:80h]     ; CX= longitud de la línea de comandos
                xor ch,ch
                xor ax,ax
                or cx,cx            ; si es cero, terminar
                jz @@exit
                mov si,81h

@@1:            cmp BYTE PTR [si],20h   ; si es carácter separador
                je @@2              ; incrementar contador de argumentos
                cmp BYTE PTR [si],0Dh   ; si es fin de cadena, terminar
                je @@exit
                jmp SHORT @@3

@@2:            inc si              ; incrementar apuntador
                loop @@1

@@3:            inc ax
@@4:            cmp BYTE PTR [si],20h
                je @@2
                inc si
                loop @@4

@@exit:         pop ds              ; recuperar registros
                pop di
                pop cx
                pop bp
                ret ArgLen
_ArgCountENDP
```

Rutina _GetArg: Copia el comando especificado en la variable dada. Si la operación fue exitosa, devuelve en AX la longitud del parámetro. En caso de error devuelve -1 en AX.

Invocación: push <segmento PSP>
push <índice parámetro>
push SEG <variable>
push OFFSET <variable>
call _GetArg

```
                PUBLIC _GetArg
_GetArg         PROC NEAR
                ARG Buffer:DWORD,No:WORD,PSPseg:WORD= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push bx                 ; salvar registros
                push cx
                push di
                push si
                push ds
                push es

                cld                     ; incrementar apuntadores

                mov ds,PSPseg           ; DS= segmento del PSP
                mov cl,[ds:80h]         ; CX= longitud de la línea de comandos
                xor ch,ch
                or cx,cx
                jz @@notfound           ; si es cero, terminar
                mov bx,No       ; BX= # del argumento buscado
                mov si,81h

@@1:            cmp BYTE PTR [si],20h   ; comparar con carácter separador
                jne @@2

@@4:            inc si                  ; incrementar apuntador
                loop @@1
                jmp SHORT @@notfound

@3:             inc si                  ; incrementar apuntador
                cmp BYTE PTR [si],20h   ; comparar con carácter separador
                je @@4
                loop @@3
                jmp SHORT @@notfound

@@2:            cmp BYTE PTR [si],0Dh   ; si es fin de cadena, terminar
                je @@notfound
                dec bx
                jnz @@3
                xor ax,ax
                les di,Buffer

@@loop1: inc ax                 ; copiar el argumento, a la variable dada
                movsb
                cmp BYTE PTR [si],20h
                je @@exit
                cmp BYTE PTR [si],0Dh
                je @@exit
                jmp SHORT @@loop1

@@notfound:     mov ax,-1               ; AX= -1 si el argumento no fue encontrado

@@exit:         pop es                  ; recuperar registros
                pop ds
                pop si
                pop di
                pop cx
                pop bx
                pop bp
                ret ArgLen
_GetArg         ENDP
```

Rutina _GetDate: Devuelve en un estructura SDate, la fecha del sistema. SDate tiene la siguiente estructura:

```
SDate       STRUC
Day         DB ?
Month       DB ?
Year        DW ?
DayOfWeek   DB ?
            ENDS
```

Invocación:
```
push SEG <estructura SDate>
push OFFSET <estructura SDate>
call _GetDate
```

```
            PUBLIC _GetDate
_GetDate    PROC NEAR
            ARG Date:DWORD= ArgLen
            push bp                 ; salvar BP
            mov bp,sp               ; permitir acceso a los argumentos
            push ax                 ; salvar registros
            push bx
            push cx
            push dx
            push ds

            mov ah,2Ah              ; función 2Ah
            int 21h                 ; del DOS

            lds bx,Date             ; copiar información
            mov [bx].DayOfWeek,al   ; en la estructura SDate
            mov [bx].Day,dl
            mov [bx].Month,dh
            mov [bx].Year,cx

            pop ds                  ; recuperar registros
            pop dx
            pop cx
            pop bx
            pop ax
            pop bp
            ret ArgLen
_GetDate    ENDP
```

Rutina _SetDate: Establece la fecha del sistema dada en un estructura SDate. Si la operación es exitosa AX contiene 00h. En caso de error, AX contiene FFh. SDate tiene la siguiente estructura:

```
SDate       STRUC
Day         DB ?
Month       DB ?
Year        DW ?
DayOfWeek   DB ?
            ENDS
```

Invocación:
```
push SEG <estructura SDate>
push OFFSET <estructura SDate>
call _SetDate
```

```
            PUBLIC _SetDate
_SetDate    PROC NEAR
            ARG Date:DWORD= ArgLen
            push bp                 ; salvar BP
            mov bp,sp               ; permitir acceso a los argumentos
            push bx                 ; salvar registros
            push cx
            push dx
            push ds
```

```
                lds bx,Date             ; recuperar datos de la estructura SDate
                mov dl,[bx].Day
                mov dh,[bx].Month
                mov cx,[bx].Year

                mov ah,2Bh              ; función 2Bh
                int 21h                 ; del DOS

                xor ah,ah

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bx
                pop bp
                ret ArgLen
_SetDate        ENDP
```

Rutina _GetTime: Devuelve en un estructura STime, la hora del sistema. STime presenta la siguiente estructura:

```
STime       STRUC
Hour        DB ?
Minutes DB ?
Seconds     DB ?
Hundredths  DB ?
            ENDS
```

Invocación:
```
push SEG <estructura STime>
push OFFSET <estructura STime>
call _GetTime
```

```
                PUBLIC _GetTime
_GetTime        PROC NEAR
                ARG Time:DWORD= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push bx
                push cx
                push dx
                push ds

                mov ah,2Ch              ; función 2Ch
                int 21h                 ; del DOS

                lds bx,Time             ; vaciar información
                mov [bx].Hour,ch        ; en la estructura STime
                mov [bx].Minutes,cl
                mov [bx].Seconds,dh
                mov [bx].Hundredths,dl

                pop ds                  ; recuperar registros
                pop dx
                pop cx
                pop bx
                pop ax
                pop bp
                ret ArgLen
_GetTime        ENDP
```

Rutina _SetTime: Establece la hora del sistema dada en un estructura STime. Si la operación es exitosa AX contiene 00h. En caso de error AX contiene FFh. STime presenta la siguiente estructura:

```
STime       STRUC
Hour        DB ?
```

```
Minutes DB ?
Seconds     DB ?
Hundredths  DB ?
            ENDS
```

Invocación:
```
push SEG <estructura STime>
push OFFSET <estructura STime>
call _SetTime
```

```
            PUBLIC _SetTime
_SetTime    PROC NEAR
            ARG Time:DWORD= ArgLen
            push bp                 ; salvar BP
            mov bp,sp               ; permitir acceso a los argumentos
            push bx                 ; salvar registros
            push cx
            push dx
            push ds

            lds bx,Time             ; recuperar datos de la estructura STime
            mov ch,[bx].Hour
            mov cl,[bx].Minutes
            mov dh,[bx].Seconds
            mov dl,[bx].Hundredths

            mov ah,2Dh              ; función 2Dh
            int 21h                 ; del DOS

            xor ah,ah

            pop ds                  ; recuperar registros
            pop dx
            pop cx
            pop bx
            pop bp
            ret ArgLen
_SetTime    ENDP
```

Rutina _GetVect: Devuelve en DX:AX el contenido actual del vector de interrupción especificado.

Invocación:
```
push <número de interrupción>
call _GetVect
```

```
            PUBLIC _GetVect
_GetVect    PROC NEAR
            ARG IntNo:BYTE= ArgLen
            push bp                 ; salvar BP
            mov bp,sp               ; permitir acceso a los argumentos
            push bx                 ; salvar registros
            push es

            mov ah,35h              ; función 35h
            mov al,IntNo
            int 21h                 ; del DOS

            mov dx,es               ; DX:AX= contenido del vector de interrupción
            mov ax,bx

            pop es                  ; recuperar registros
            pop bx
            pop bp
            ret ArgLen
_GetVect    ENDP
```

Rutina _SetVect: Establece una nueva rutina manejadora de interrupciones, para el vector especificado.

Invocación: push <número de interrupción>
push SEG <rutina manejadora>
push OFFSET <rutina manejadora>
call _SetVect

```
                PUBLIC _SetVect
_SetVect        PROC NEAR
                ARG IntHandler:DWORD,IntNo:BYTE= ArgLen
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
                push dx
                push ds

                mov ah,25h              ; función 25h
                mov al,IntNo
                lds dx,IntHandler
                int 21h                 ; del DOS

                pop ds                  ; recuperar registros
                pop dx
                pop ax
                pop bp
                ret ArgLen
_SetVect        ENDP
```

Rutina _IsAlNum: Determina si el carácter dado es alfanumérico. Si lo es, el bit de acarreo se enciende.
Invocación: push <carácter>
call _IsAlNum

```
                PUBLIC _IsAlNum
_IsAlNum        PROC NEAR
                ARG Char:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros

                mov al,Char             ; AL= carácter

                push ax                 ; si es un dígito, es alfanumérico
                call _IsDigit
                jc @@exit

                push ax                 ; si es un carácter alfabético
                call _IsAlpha           ; es alfanumérico

@@exit:         pop ax                  ; recuperar registros
                pop bp
                ret ArgLen
_IsAlNum        ENDP
```

Rutina _IsDigit: Determina si el carácter dado es un dígito decimal. Si lo es, el bit de acarreo se enciende.
Invocación: push <carácter>
call _IsDigit

```
                PUBLIC _IsDigit
_IsDigit        PROC NEAR
                ARG Char:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros
```

```
                mov al,Char             ; AL= carácter
                cmp al,'0'
                jb @@noDigit            ; si es menor que 0, no es un dígito
                cmp al,'9'
                ja @@noDigit            ; si es mayor que 9, no es un dígito
                stc                     ; si está entre 0 y 9, es un dígito
                jmp SHORT @@exit

@@noDigit:      clc

@@exit:         pop ax                  ; recuperar registros
                pop bp
                ret ArgLen
_IsDigit        ENDP
```

Rutina _IsXDigit: Determina si el carácter dado es un dígito hexadecimal.
Invocación: push <carácter>
call _IsXDigit

```
                PUBLIC _IsXDigit
_IsXDigit       PROC NEAR
                ARG Char:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
                mov bp,sp               ; permitir acceso a los argumentos
                push ax                 ; salvar registros

                mov al,Char             ; AX= carácter
                xor ah,ah

                push ax                 ; si es un dígito decimal,
                call _IsDigit           ; es un dígito hexadecimal
                jc @@isXdigit

                cmp al,'A'              ; si está entre 'A' y 'F', es un dígito hexadecimal
                jb @@noXdigit
                cmp al,'F'
                jbe @@isXdigit

                cmp al,'a'              ; si está entre 'a' y 'f', es un dígito hexadecimal
                jb @@noXdigit
                cmp al,'f'
                ja @@noXdigit

@@isXdigit:     stc
                jmp SHORT @@exit

@@noXdigit:     clc

@@exit:         pop ax                  ; recuperar registros
                pop bp
                ret ArgLen
_IsXDigit       ENDP
```

Rutina _IsAlpha: Determina si el carácter dado es alfabético.
Invocación: push <carácter>
call _IsAlpha

```
                PUBLIC _IsAlpha
_IsAlpha        PROC NEAR
                ARG Char:BYTE= ArgLen
                LOCALS
                push bp                 ; salvar BP
```

```
                    mov bp,sp               ; permitir acceso a los argumentos
                    push ax                 ; salvar registros

                    mov al,Char             ; AL= carácter

                    push ax                 ; si es un carácter en minúscula
                    call _IsLower
                    jc @@exit

                    push ax                 ; o mayúscula,
                    call _IsUpper           ; es un carácter alfabético

@@exit:             pop ax                  ; recuperar registros
                    pop bp
                    ret ArgLen
_IsAlpha            ENDP
```

Rutina _IsLower: Determina si el carácter dado es alfabético en minúscula.
Invocación: push <carácter>
call _IsLower

```
                    PUBLIC _IsLower
_IsLower            PROC NEAR
                    ARG Char:BYTE= ArgLen
                    LOCALS
                    push bp                 ; salvar BP
                    mov bp,sp               ; permitir acceso a los argumentos
                    push ax                 ; salvar registros

                    mov al,Char             ; AL= carácter
                    cmp al,'a'              ; si está entre 'a' y 'z',
                    jb @@noLower            ; es un carácter en minúscula
                    cmp al,'z'
                    ja @@noLower

                    stc
                    jmp SHORT @@exit

@@noLower:          clc

@@exit:             pop ax                  ; recuperar registros
                    pop bp
                    ret ArgLen
_IsLower            ENDP
```

Rutina _IsUpper: Determina si el carácter dado es alfabético en mayúscula.
Invocación: push <carácter>
call _IsUpper

```
                    PUBLIC _IsUpper
_IsUpper            PROC NEAR
                    ARG Char:BYTE= ArgLen
                    LOCALS
                    push bp                 ; salvar BP
                    mov bp,sp               ; permitir acceso a los argumentos
                    push ax                 ; salvar registros

                    mov al,Char             ; AL= carácter
                    cmp  al,'A'             ; si está entre 'A' y 'Z'
                    jb @@noUpper            ; es un carácter en mayúsculas
                    cmp al,'Z'
                    ja @@noUpper

                    stc
```

```
                jmp SHORT @@exit

@@noUpper:      clc

@@exit:         pop ax                  ; recuperar registros
                pop bp
                ret ArgLen
_IsUpper        ENDP
```

Rutina _DOSver: Devuelve en AX, la versión del DOS.
Invocación: call _DOSver

```
                PUBLIC _DOSver
_DOSver         PROC NEAR
                push bx                 ; salvar registros
                push cx

                mov ah,30h              ; función 30h
                int 21h                 ; del DOS

                pop cx                  ; recuperar registros
                pop bx
                ret
_DOSver         ENDP
```

Programas Residentes.

El MS-DOS es un sistema operativo que no soporta la ejecución de múltiples tareas a la vez. Sin embargo, Microsoft desde las primeras versiones, desarrolló un método que simulaba la ejecución multitarea en MS-DOS. Este método fue usado en programas de utilidad tales como PRINT.COM y MODE.COM. Estos programas se cargan en memoria y permanecen allí. Ambos son conocidos como programas residentes (TSR). El programa se ejecuta, pero permanece en memoria al terminar.

Generalidades de un programa residente.

La ejecución multitarea bajo MS-DOS no es posible a través de los medios convencionales de programación, por varios motivos:

La mayoría de las rutinas del MS-DOS no son reentrantes.

El microprocesador 8086, para el cual escrito originalmente el MS-DOS, no proporciona mecanismo de protección entre tareas.

Los 640K de memoria convencional establecen un límite importante al número de aplicaciones que puedan coexistir en la memoria.

Sin embargo, haciendo uso de la técnica para escritura de programas residentes, es posible conseguir que uno o más programas coexistan en memoria.

Estos programas son cargados en el área de memoria transiente. La porción transiente del programa termina su ejecución conservando el espacio de memoria destinado a la porción residente del mismo, la cual permanecerá en la memoria.

Otras aplicaciones convencional podrán ser luego cargada en memoria y ejecutada. EL TSR permanecerá inactivo hasta la ocurrencia de un evento (generalmente una interrupción) que lo ejecute. Este evento puede ser causado por una combinación especial de teclas o por un dispositivo externo. A una combinación especial de teclas, empleada para la activación de un programa TSR se le llama *hotkey*.

Una vez que el programa residente es activado, este toma control del procesador y del sistema. Este entonces efectuará las verificaciones pertinentes para determinar si es factible su ejecución en ese momento sin poner en peligro la estabilidad del sistema (deben estar inactivos el DOS, disco y video). Si la ocasión es propicia para la ejecución del TSR, este deberá salvar el estado actual del sistema y restaurarlo al finalizar su ejecución para luego regresar el control al programa de aplicación actual.

Existen dos tipos básicos de programas residentes: los pasivos y los activos. Los pasivos son ejecutados desde un estado conocido del sistema y son por lo tanto más fáciles y simples de escribir. Generalmente son invocados desde otra aplicación por medio de una interrupción por software. Un TSR activo es más complejo, ya que el estado del sistema al momento de su activación es desconocido. Estos programas son activados normalmente por eventos asincrónicos asociados a un dispositivo externo tales como el teclado o el puerto serial.

El código de un programa residente consta de dos partes fundamentales: porción transiente y porción residente. La porción transiente instala al programa residente en memoria, modificando los vectores de interrupción que sea necesario y finalizando su ejecución. La porción residente es la parte que permanece en memoria y contiene el código necesario para la tarea encomendada al programa TSR.

Reglas básicas para la escritura de programas TSR.

1. La porción transiente instalará el programa TSR en memoria y modificará los vectores de interrupción necesarios, salvando su antiguo contenido para que estos puedas ser restaurados en el caso de que la porción residente sea desinstalada.

2. El programa instalador debe finalizar con la función 31h del DOS, la cual es el único método aprobado por Microsoft para este fin. Esta función permite indicar la cantidad e memoria requerida por la porción residente y devuelve un código de retorno a la aplicación invocadora.

3. Ninguna de las funciones del DOS o del BIOS son reentrantes por lo que el programa residente deberá asegurarse de que no están siendo usadas por ninguna otra tarea, si ella pretende hacerlo. Para conseguir estos, es necesario interceptar los siguientes vectores de interrupción:

Int 13: Interceptando esta interrupción es posible determinar si existe actividad en las funciones relacionadas con disco. EL programa TSR no debe ser activado mientras se esté ejecutando a rutina de servicio de disco.

Int 10: La interrupción 10h lleva a cabo todas las labores del BIOS relacionadas con video. La actividad en esta función indica que no existen las condiciones apropiadas para la ejecución del programa residente.

Int 16h: Por medio de esta interrupción podemos determinar si existe actividad del BIOS relacionada con el teclado. El programa TSR no deberá ser activado si tal actividad está presente.

Int 28h: Esta interrupción es ejecutada por el MS-DOS cada vez que se encuentra a la espera de comandos. Cuando el DOS está en esta situación, el programa TSR puede ser activado.

La última verificación necesaria para determinar la conveniencia de activación de un programa está relacionada con la bandera **InDos**. Esta es una bandera del DOS que es incrementada cada vez que es ejecutada la **Int 21h** y decrementada cada vez que termina. Indica por lo tanto, si se está ejecutando una función del DOS. Un valor cero en dicha bandera señala un estado seguro para la activación del programa TSR.

4. Un programa TSR generalmente es activado de manera asincrónica, por medio de un secuencia de teclas. Para conseguir esto es necesario interceptar las funciones siguientes:

Int 09: Esta interrupción se genera cada vez que es presionada alguna tecla. Nuestra rutina de servicio tras determinar si el estado del sistema es conveniente, activara al programa TSR. Si las

condiciones del sistema no son adecuadas, deberá activarse una bandera que señale que el programa TSR está a la espera de poder ser activado.

Int 1Ch: La interrupción 1Ch es ejecutada 18.2 veces por segundo. En ella podemos incluir código que determine si el programa residente está a la espera de ser activado y si las condiciones del sistema son ahora apropiadas para ello.

Int 28h: Si el programa residente se encuentra a la espera por ser activado, conviene que la rutina de servicio para la interrupción 28h verifique esta situación y determine si las condiciones del sistema son apropiadas y en caso afirmativo, que ejecute al programa residente.

5. Una vez que se ha determinado que el programa residente puede ser activado, es necesario que este efectúe ciertos arreglos antes de comenzar:

Ctrl-C, Ctrl-Break y **Error Crítico**: Es conveniente que instale sus propios manejadores para las interrupciones Ctrl-C, Ctrl-Break y Error Crítico. Estas deben ser restauradas al terminar el programa residente.

Pila: Deberá salvar la pila del DOS y activará su propia pila. Nada de esto es hecho en forma automática. Al finalizar la ejecución del TSR, la pila del DOS deberá ser restaurada.

DTA y **PSP**: El programa residente debe salvar el DTA y el PSP de la aplicación actual y activar los suyos. El DTA y PSP originales deben ser restaurados antes de finalizar la ejecución del TSR.

TSRBusy: Es necesario que el programa TSR mantenga un bandera que indica que está en actividad a fin de evitar la ejecución recursiva del mismo. Esta bandera será verificada por posteriores ejecuciones del programa TSR para determinar si las condicione son apropiadas para su activación.

6. La porción transiente al instalar un programa TSR, debe verificar la existencia previa de una copia del mismo en memoria. Esto evita la reinstalación del programa residente y proporciona un mecanismo para la desinstalación del mismo. Un método comúnmente usado para este fin es el de emplear un vector de interrupción normalmente libre, para señalar la existencia del programa residente.

La desinstalación de un programa residente involucra dos pasos: restaurar todos los vectores de interrupción alterados por el TSR, para el cual es necesario que este verifique que los mismo no han sido alterados posteriormente por otra aplicación y la liberación de la memoria reservada para el TSR. La liberación de la memoria se consigue mediante el empleo de la función 49h de la Int 21h aplicada al bloque de variables de ambiente y al PSP del programa TSR.

7. Un programa residente no podrá hacer uso de las funciones para manejo dinámico de memoria, ni podrá ejecutar otros programas.

Template para escritura de Programas Residentes.

En esta sección se presenta un *template* general que permite la escritura de programas residentes, empleando técnicas que hacen posible la utilización en el mismo, del los recursos del BIOS y del DOS. El mismo suministra además un mecanismo para detección de una copia previa del programa TSR.

Las variables **Signature**, **HotKey** y **ScanCode** deben ser suministradas por el programador, así como la rutina **TSR**.

En este caso, la rutina presentada salva en un archivo llamado *SCREEN.DAT* el contenido de la memoria de video en modo texto, para un adaptador a color. La combinación de activación es **Alt-F12**.

; TSRGEN Programa genérico para creación de rutinas residentes. El programador debe suministrar la rutina **TSR**, con el código adecuado
; para la labor deseada. Pueden ser usadas todas las funciones del BIOS y todas las DOS con excepción de los servicios 00h al 12h.

```
KbPort      EQU 60h          ; puerto del teclado
MaxWait     EQU 6            ; máximo tiempo de espera por activación
```

```
                JUMPS
                ASSUME CS:PROGRAM,DS:PROGRAM,ES:PROGRAM,SS:PROGRAM
                PROGRAM         SEGMENT WORD PUBLIC 'Program'
                ORG 100h

Start:          jmp Install

iSS             DW 0                    ; segmento de pila del programa interrumpido
iSP             DW 0                    ; apuntador de pila del programa interrumpido
iPSP            DW 0                    ; segmento del PSP del programa interrumpido

iDTA            LABEL DWORD             ; DTA del programa interrumpido
iDTAOFS         DW 0
iDTASEG         DW 0

                                        ; el Signature es una cadena de 8 caracteres que
                                        ; permitirá detectar la presencia del programa
                                        ; residente en la memoria.
                                        ; KeyMask y ScanCode establecen la combinación
                                        ; de teclas que activarán la rutina residente. En este
                                        ; caso, la combinación es Alt - F12
Signature       DB 'EXAMPLE '           ; cadena identificadora (8 caracteres)
KeyMask         DW 8h                   ; máscara de Estado del HotKey (Alt)
ScanCode        DB 58h                  ; código de búsqueda del HotKey (F12)

TsrBusy         DB 0                    ; bandera de actividad del TSR
DiskBusy        DB 0                    ; bandera de actividad de disco.
VideoBusyDB 0                   ; bandera de actividad de video.
KeybBusy        DB 0                    ; bandera de actividad del teclado.

Waiting         DB 0                    ; bandera de TSR en espera
WaitCounter     DB 0                    ; ticks de espera

InDos           LABEL DWORD             ; bandera InDos
InDosOFSDW 0
InDosSEGDW 0

OldInt09        LABEL DWORD             ; antiguo vector 09h
OldInt09OFS     DW 0
OldInt09SEG     DW 0

OldInt10        LABEL DWORD             ; antiguo vector 10h
OldInt10OFS     DW 0
OldInt10SEG     DW 0

OldInt13        LABEL DWORD             ; antiguo vector 13h
OldInt13OFS     DW 0
OldInt13SEG     DW 0

OldInt16        LABEL DWORD             ; antiguo vector 16h
OldInt16OFS     DW 0
OldInt16SEG     DW 0

OldInt1B        LABEL DWORD             ; antiguo vector 1Bh
OldInt1BOFS     DW 0
OldInt1BSEG     DW 0

OldInt1C        LABEL DWORD             ; antiguo vector 1Ch
OldInt1COFS     DW 0
OldInt1CSEG     DW 0

OldInt23        LABEL DWORD             ; antiguo vector 23h
OldInt23OFS     DW 0
OldInt23SEG     DW 0
```

```
OldInt24        LABEL DWORD             ; antiguo vector 24h
OldInt24OFS     DW 0
OldInt24SEG     DW 0

OldInt28        LABEL DWORD             ; antiguo vector 28h
OldInt28OFS     DW 0
OldInt28SEG     DW 0

Error           DB "TSR previamente instalado",13,10
ErrorLength     = $-Error

                                        ; la int09 intercepta el vector 09h (interrupción del
                                        ; teclado) y permite determinar si ha sido presionada
                                        ; la combinación de tecla que activará la rutina
                                        ; residente
int09           PROC FAR
                LOCALS
                sti                     ; habilita interrupciones
                push ax                 ; salvar ax
                in al,KbPort            ; leer puerto del teclado

                pushf                   ; invocar vector antiguo
                call cs:OldInt09

                cli                     ; deshabilitar interrupciones
                cmp cs:TsrBusy,0        ; verificar si el TSR está activo
                jne @@exit              ; si es así, terminar

                cmp al,128              ; si no, verificar si fue activado
                jae @@exit              ; el HotKey
                cmp cs:ScanCode,128
                jae @@1
                cmp al,cs:ScanCode
                jne @@exit

@@1:            push ax                 ; salvar AX
                push ds                 ; salvar DS
                mov  ax,40h             ; verificar estado del teclado
                mov ds,ax
                mov ax,ds:[17h]
                and ax,cs:KeyMask
                cmp ax,cs:KeyMask
                pop ds                  ; recuperar DS
                pop ax                  ; recuperar AX
                jne @@exit

                cmp cs:DiskBusy,0       ; determinar si puede ser activado el TSR
                jne @@2

                cmp cs:VideoBusy,0
                jne @@2

                cmp cs:KeybBusy,0
                jne @@2

                push bx                 ; salvar BX
                push ds                 ; salvar DS
                lds bx,cs:InDos         ; verificar el estado de INDOS
                cmp BYTE PTR [bx],0
                pop ds                  ; recuperar DS
                pop bx                  ; recuperar BX
                jne @@2

                call StartTSR           ; invocar al procedimiento TSR
```

```
                jmp SHORT @@exit

@@2:            mov cs:WaitCounter,0    ; no puede ser activado el TSR
                mov cs:Waiting,1

@@exit:         pop ax                  ; recuperar AX
                iret
int09           ENDP

                                        ; int13 intercepta el vector 13h (servicio de disco)
                                        ; y permite señalar la existencia de actividad
                                        ; relacionada con manejo de disco
int13           PROC FAR
                LOCALS
                inc cs:DiskBusy         ; incrementa la bandera de actividad de disco

                pushf                   ; invoca el antiguo vector
                call cs:OldInt13
                pushf                   ; almacena el resultado de la función

                dec cs:DiskBusy         ; decrementa la bandera de actividad de disco
                jnz @@exit              ; si es una invocación recursiva, terminar

                push bx                 ; salvar BX
                push ds                 ; salvar DS
                lds bx,cs:InDos         ; verificar INDOS
                cmp BYTE PTR [bx],0
                pop ds                  ; recuperar BX
                pop bx                  ; recuperar DS
                jne @@exit

                call Active             ; determinar si el TSR espera por activación

@@exit:         popf                    ; recupera resultado del vector antiguo
                ret 2                   ; retornar
int13           ENDP

                                        ; int1C intercepta el vector 1C (interrupción del reloj)
                                        ; que se produce 18.2 veces por segundo. Permite
                                        ; monitorear periódicamente si las condiciones están
                                        ; dadas, para la activación de la rutina residente.
int1C           PROC FAR
                inc cs:WaitCounter      ; incrementa el contador de ticks
                jmp cs:OldInt1C         ; invoca al antiguo vector
int1C           ENDP

                                        ; rutina de manejo de errores críticos.
int24           PROC FAR
                mov ax,3                ; error crítico
                iret
int24           ENDP

                                        ; int28 intercepta el vector 28h (estado iddle del
                                        ; DOS). Señala si el DOS está en espera de
                                        ; comandos y determina si las condiciones están
                                        ; dadas para la activación del TSR.
int28           PROC FAR
                LOCALS
                pushf                   ; invoca al antiguo vector
                call cs:OldInt28

                cmp cs:DiskBusy,0       ; determinar si el TSR puede ser
                jne @@exit              ; activado

                cmp cs:VideoBusy,0
```

```
                jne @@exit

                cmp cs:KeybBusy,0
                jne @@exit

                call Active             ; determinar si el TSR espera por activación

@@exit:         iret
int28           ENDP

                                        ; rutina manejadora de Ctrl-Break y Ctrl-C
DummyHandler    PROC FAR
                iret                    ; para las interrupciones 1Bh y 23h
DummyHandler    ENDP

                                        ; int10 intercepta la rutina de servicio de video.
                                        ; Señala la existencia de actividad en la misma.
int10           PROC FAR
                inc cs:VideoBusy        ; incrementa la bandera de actividad de video

                pushf                   ; invoca al antiguo vector
                call cs:OldInt10

                dec cs:VideoBusy        ; decrementa al antiguo vector

                iret
int10           ENDP

                                        ; intercepta la rutina de servicio de teclado del BIOS,
                                        ; señalando la presencia de actividad en la misma.
int16           PROC FAR
                inc cs:KeybBusy         ; incrementa la bandera de actividad del teclado

                pushf                   ; invocar al antiguo vector
                call cs:OldInt16

                pushf
                dec cs:KeybBusy         ; decrementa la bandera de actividad del
                popf                    ; teclado

                ret 2
int16           ENDP

                                        ; determina si el programa residente está a la
                                        ; espera por ser activado.
Active          PROC NEAR
                LOCALS
                cmp cs:Waiting,0        ; verificar si el TSR está en espera
                je @@exit

                mov cs:Waiting,0        ; apagar la bandera de espera

                cmp cs:WaitCounter,MaxWait
                jbe StartTSR

@@exit:         ret
Active          ENDP

                                        ; efectúa los preparativos necesarios, antes y
                                        ; después de la ejecución de la rutina residente
                                        ; suministrada por el programador
StartTSR        PROC NEAR
                LOCALS
                mov cs:TsrBusy,1        ; encender la bandera de actividad del TSR
```

```
                cli                     ; deshabilitar interrupciones
                push ax                 ; salvar AX y DS en la pila actual
                push ds
                mov cs:iSS,ss           ; salvar SS y SP actuales
                mov cs:iSP,sp

                mov ax,cs               ; establecer DS y SS del TSR
                mov ds,ax
                mov ss,ax
                mov sp,OFFSET StackEND-2

                sti                     ; reestablecer interrupciones

                push bx                 ; salvar registros
                push cx
                push dx
                push bp
                push si
                push di
                push es

                mov cx,64
                mov si,iSP
                mov ds,iSS
@@1:            push WORD PTR [si]
                add si,2
                loop @@1
                mov ds,ax

                mov ah,1                ; si el hotkey fue almacenado en el buffer
                int 16h                 ; removerlo
                jz @@2
                xor ah,ah
                int 16h

@@2:            mov ax,3523h            ; salvar antiguo vector 23h
                int 21h
                mov OldInt23OFS,bx
                mov OldInt23SEG,es

                mov ax,351Bh            ; salvar antiguo vector 1Bh
                int 21h
                mov OldInt1BOFS,bx
                mov OldInt1BSEG,es

                mov ax,3524h            ; salvar antiguo vector 24h
                int 21h
                mov OldInt24OFS,bx
                mov OldInt24SEG,es

                mov ax,2523h            ; establecer nuevo vector 23h
                mov dx,OFFSET DummyHandler
                int 21h

                mov ax,251Bh            ; establecer nuevo vector 1Bh
                int 21h

                mov ax,2524h            ; establecer nuevo vector 24h
                mov dx,OFFSET int24
                int 21h

                mov ah,51h              ; Salvar la dirección del PSP
                int 21h                 ; del programa interrumpido
                mov iPSP,bx
```

```
                mov ah,2Fh              ; Salvar la dirección del DTA
                int 21h                 ; del programa interrumpido
                mov iDTAOFS,bx
                mov iDTASEG,es

                mov ah,50h              ; establecer el PSP del TSR
                mov bx,cs
                int 21h

                mov ah,1Ah              ; establecer el DTA del TSR
                mov dx,80h
                int 21h

                mov ax,cs               ; establecer ES
                mov es,ax

                call TSR                ; invocar a la rutina principal del programa residente

                mov ah,1Ah              ; restaurar el DTA del programa
                lds dx,cs:iDTA          ; interrumpido
                int 21h

                mov ah,50h              ; restaurar el PSP del programa
                mov bx,cs:iPSP          ; interrumpido
                int 21h

                mov ax,2523h            ; restaurar el vector 23h
                lds dx,cs:OldInt23
                int 21h

                mov ax,251Bh            ; restaurar el vector 1Bh
                lds dx,cs:OldInt1B
                int 21h

                mov ax,2524h            ; restaurar el vector 24h
                lds dx,cs:OldInt24
                int 21h

                mov cx,64               ; restaurar la pila del DOS
                mov ds,cs:iSS
                mov si,cs:iSP
                add si,128
@@3:            sub si,2
                pop WORD PTR [si]
                loop @@3

                pop es                  ; recuperar registros
                pop di
                pop si
                pop bp
                pop dx
                pop cx
                pop bx

                cli                     ; suprimir interrupciones
                mov ss,cs:iSS           ; activar la pila
                mov sp,cs:iSP           ; del programa interrumpido

                pop ds                  ; recuperar registros
                pop ax

                mov cs:TsrBusy,0
                sti                     ; activar interrupciones

                ret
```

```
StartTSR        ENDP

                                        ; ******************************************************
                                        ; esta es la rutina residente suministrada por el
                                        ; programador. En este caso, la rutina almacena
                                        ; el contenido de la memoria de video en modo
                                        ; texto y para un adaptador a color, en un archivo
                                        ; en disco llamado SCREEN.DAT.

FileName        DB "SCREEN.DAT",0
ScreenBuffer    DW 80 DUP (?)
MensajeOk       DB " Operación completada exitosamente, presione cualquier tecla"
                DB 20 DUP (" ')
MensajeError    DB " Error en la Operación, presione cualquier tecla"
                DB 32 DUP (' ')

TSR             PROC NEAR
                LOCALS
                push ds                 ; salvar la línea superior
                mov si,0B800h           ; de la pantalla
                mov ds,si
                xor si,si
                mov di,OFFSET ScreenBuffer
                mov cx,80
                REP movsw
                pop ds

                mov ah,3                ; salvar posición original
                xor bh,bh               ; del cursor
                int 10h
                push dx

                mov ah,2                ; ubicar el cursor en la esquina
                xor bh,bh               ; superior izquierda de la pantalla
                xor dx,dx
                int 10h

                mov ah,3Ch              ; crear archivo
                xor cx,cx               ; normal
                mov dx,OFFSET FileName
                int 21h
                jc @@error

                push ds                 ; copiar pantalla en archivo
                mov bx,ax
                mov ah,40h
                mov cx,80*25*2
                mov dx,0B800h
                mov ds,dx
                xor dx,dx
                int 21h
                pop ds

                mov ah,3Eh              ; cerrar archivo
                int 21h

                mov ah,40h              ; mostrar mensaje Ok
                mov bx,1
                mov cx,80
                mov dx,OFFSET MensajeOk
                int 21h
                jmp SHORT @@exit

@@error:        mov ah,40h              ; mostrar mensaje Error
                mov bx,1
```

```
                mov cx,80
                mov dx,OFFSET MensajeError
                int 21h

@@exit:         xor ah,ah               ; esperar a que el usuario
                int 16h                 ; presione alguna tecla

                push es                 ; recuperar la línea superior
                mov di,0B800h           ; de la pantalla
                mov es,di
                xor di,di
                mov si,OFFSET ScreenBuffer
                mov cx,80
                REP movsw
                pop es

                mov ah,2                ; reestablecer la posición original
                xor bh,bh               ; del cursor
                pop dx
                int 10h

                ret
TSR             ENDP

StackFrame      DW 128 DUP (0)          ; pila del TSR
StackENDEQU THIS NEAR

                                        ; esta es la porción transiente del programa,
                                        ; encargada de la instalación de la porción
                                        ; residente del mismo, en memoria.
Install         PROC NEAR
                LOCALS

                mov ax,3560h            ; determinar si existe una copia previa
                int 21h                 ; en memoria
                mov di,bx
                mov si,OFFSET Signature
                mov cx,8
                cld
                REPE cmpsb              ; comparar la cadena identificadora
                jne @@continue          ; si son diferentes, continuar

                mov ah,40h              ; si no, mostrar mensaje de error
                mov bx,1
                mov cx,ErrorLength
                mov dx,OFFSET Error
                int 21h
                mov ax,4C00h            ; y terminar
                int 21h

@@continue:     mov ax,2560h            ; establecer nuevo vector 60h
                mov dx,OFFSET Signature
                int 21h

                mov ah,34h              ; determinar dirección de InDos
                int 21h
                mov InDosOFS,bx
                mov InDosSEG,es

                mov ax,3509h            ; salvar antiguo vector 09h
                int 21h
                mov OldInt09OFS,bx
                mov OldInt09SEG,es

                mov ax,3510h            ; salvar antiguo vector 10h
```

```
                int 21h
                mov OldInt10OFS,bx
                mov OldInt10SEG,es

                mov ax,3513h            ; salvar antiguo vector 13h
                int 21h
                mov OldInt13OFS,bx
                mov OldInt13SEG,es

                mov ax,3516h            ; salvar antiguo vector 16h
                int 21h
                mov OldInt16OFS,bx
                mov OldInt16SEG,es

                mov ax,351Ch            ; salvar antiguo vector 1Ch
                int 21h
                mov OldInt1COFS,bx
                mov OldInt1CSEG,es

                mov ax,3528h            ; salvar antiguo vector 28h
                int 21h
                mov OldInt28OFS,bx
                mov OldInt28SEG,es

                mov ax,2509h            ; establecer nuevo vector 09
                mov dx,OFFSET int09
                int 21h

                mov ax,2510h            ; establecer nuevo vector 10
                mov dx,OFFSET int10
                int 21h

                mov ax,2513h            ; establecer nuevo vector 13
                mov dx,OFFSET int13
                int 21h

                mov ax,2516h            ; establecer nuevo vector 16
                mov dx,OFFSET int16
                int 21h

                mov ax,251Ch            ; establecer nuevo vector 1C
                mov dx,OFFSET int1C
                int 21h

                mov ax,2528h            ; establecer nuevo vector 28
                mov dx,OFFSET int28
                int 21h

                mov ax,OFFSET Install
                xor dx,dx
                mov bx,16
                div bx
                inc ax
                mov dx,ax               ; DX= memoria requerida (párrafos)

                mov ax,3100h            ; código de retorno 0
                int 21h                 ; terminar  y dejar residente
Install         ENDP

PROGRAM         ENDS

                END Start
```

Interfaz con lenguajes de Alto Nivel.

Mientras que muchos programadores prefieren escribir aplicaciones completamente en lenguaje ensamblador, otros reservan para este último sólo aquellas labores de muy bajo nivel o que exigen un muy alto rendimiento. Existen incluso otros, que prefieren escribir sus programas principalmente en ensamblador, aprovechando ocasionalmente las ventajas que proporcionan las librerías y construcciones de lenguajes de alto nivel.

Esta sección explica como usar lenguaje ensamblador para rutinas que vayan a ser mezcladas con programas escritos en lenguajes de alto nivel.

Interfaz con PASCAL.

El Turbo Assembler proporciona facilidades extensas y poderosas para la adición de rutinas en lenguaje ensamblador a programas escritos en Turbo PASCAL.

¿ Por qué usar rutinas en lenguaje ensamblador en programas escritos en PASCAL ?. Aunque la mayoría de las aplicaciones serán escritas sólo en PASCAL, existen ocasiones en las cuales este no provea alguna facilidad específica o se requiera de una gran velocidad de ejecución, sólo alcanzable con lenguaje ensamblador.

Mapa de memoria de un programa en PASCAL.

Antes de escribir rutina en lenguaje ensamblador que trabaje con un programa en PASCAL, es conveniente conocer como es organizada por el compilador, la información en memoria. Existe un segmento global de datos, que permite un rápido acceso a las variables globales y a las constantes. Sin embargo cada *UNIT* tiene su propio segmento de código. El bloque de memoria dinámica o *heap* puede crecer hasta ocupar toda la memoria disponible. Las direcciones en Turbo PASCAL siempre son FAR (32 bits) de tal manera que puedan ser referenciados objetos en cualquier lugar de la memoria.

PSP.

EL prefijo de segmento de programa (PSP) es un área de 256 bytes creada por el sistema operativo cuando es ejecutado un programa. Contiene entre otras, información sobre la línea de comandos, cantidad de memoria disponible y variables de ambiente del DOS.

El PASCAL proporciona una variable global llamada *PrefixSeg*, que contiene el segmento del PSP y permite acceso al mismo desde un programa en PASCAL.

Segmento de Código.

Todos los programas en PASCAL tienen al menos dos segmentos de código: uno para el programa principal y otro para la librería estándar (*Run Time Library*). Adicionalmente cada *UNIT* ocupa un segmento de código independiente. Desde Turbo Assembler, el segmento de código tiene el nombre **CODE** o **CSEG**.

Segmento de Datos.

El segmento de datos globales está situado a continuación del segmento de la unidad del sistema. Contiene hasta 64K de datos inicializados y no inicializados: constantes y variables globales. El segmento de datos es direccionado por medio del registros **DS**. EL segmento de datos recibe el nombre de **DATA** o **DSEG**.

La Pila.

Al ejecutarse el programa, los registros SS y SP apuntan al primer byte después del segmento de pila. El tamaño de la pila puede alcanzar los 64K. Su tamaño por defecto es de 16K.

Mapa de memoria de un programa en Turbo PASCAL

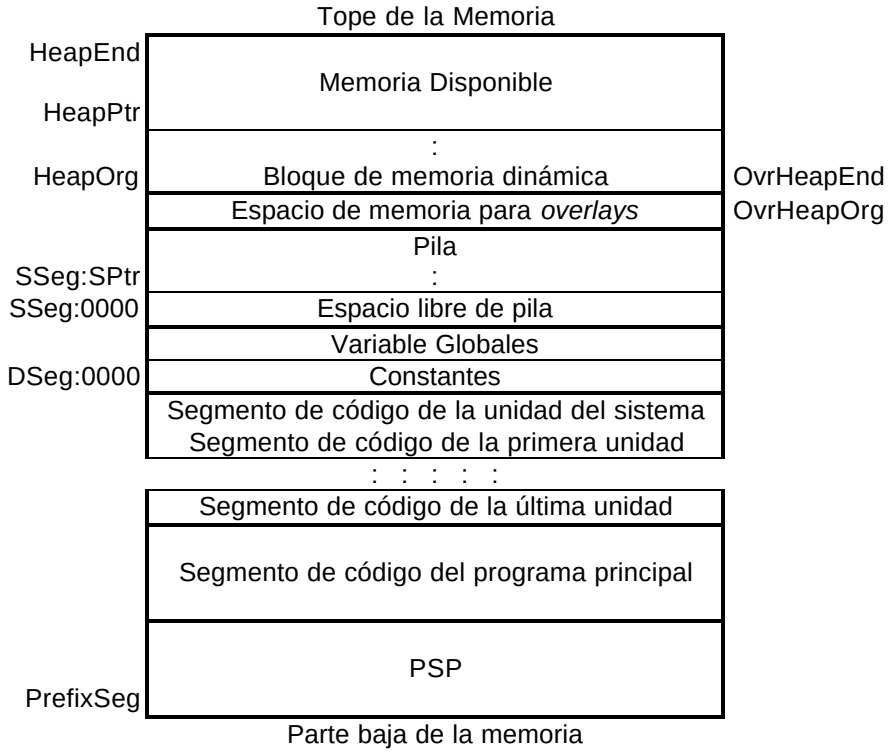


Figura 8-1

Espacio de memoria para *overlay*.

Este espacio de memoria es usado para contener código *overlay* en el caso de que esta técnica sea empleada. El tamaño de dicho *buffer* dependerá del módulo *overlay* más largo en el programa; si el programa no emplea *overlays*, el tamaño es cero.

Bloque de memoria dinámica: *Heap*.

El bloque de memoria dinámica, o *heap* está situado en el tope del mapa de memoria. Contiene las variables dinámicas, reservadas a través de la invocación a las funciones **New** y **GetMem**. Este bloque ocupa toda o parte de la memoria libre. El tamaño actual depende de los valores mínimo y máximo asignados.

La dirección de la parte baja del *heap* está almacenada en la variable *HeapOrg* y el tope en *HeapPtr*.

Uso de los registros del microprocesador.

El Turbo PASCAL impone un mínimo de restricciones para el uso de los registros del microprocesador: deben ser preservados los contenidos de los registros SS, DS y BP. Además, la rutina antes de terminar debe ajustar el contenido del apuntador de pila, para eliminar lo argumentos y variables locales.

¿ NEAR o FAR ?.

Por el hecho de que el Turbo PASCAL emplea varios segmentos de código, en los programas existe una mezcla de rutinas con atributos NEAR y FAR. Las rutinas NEAR sólo pueden ser invocadas dentro del segmento de código actual mientras que las FAR pueden serlo desde cualquier parte de la memoria.

Las rutinas declaradas en la sección **INTERFACE** de una *UNIT* deben tener atributo FAR, ya que podrán ser invocadas desde cualquier segmento de código. Las declaradas en la sección **IMPLEMENTATION** pueden tener atributo NEAR, al igual que las ubicadas en el programa principal.

El atributo de los procedimientos y funciones puede ser establecido mediante el empleo de las directivas **$F+** (FAR), **NEAR** y **FAR**.

La directiva $L y el atributo *external*.

La clave para el uso de rutinas en lenguaje ensamblador desde un programa en PASCAL está en la directiva **$L** y el atributo **external**. La directiva {**$L** MYFILE.OBJ} hace que el Turbo PASCAL enlace el archivo MYFILE.OBJ con el resto del programa en PASCAL.

Para que las rutinas en lenguaje ensamblador puedan ser invocadas desde PASCAL, deben ser definidas como símbolos públicos (directiva PUBLIC) y poseer el atributo **external** en la declaración dentro del programa en PASCAL. La sintaxis de **external** es como sigue:

```
PROCEDURE AsmProc (a:Integer; b:Real); external;
```

La directiva PUBLIC.

Sólo los símbolos que sean declarados como públicos, dentro del módulo en lenguaje ensamblador, podrán ser *vistos* desde el programa en PASCAL. Cada símbolo declarado público tendrá su correspondiente declaración como función o procedimiento en el programa en PASCAL.

La directiva EXTRN.

El módulo en lenguaje ensamblador puede tener acceso a los procedimientos, funciones, variables globales y constantes declaradas en el programa en PASCAL. Esto es posible mediante el empleo de la directiva **EXTRN** la cual permite el acceso en ensamblador de símbolos declarados públicos en otros módulos.

Pase de parámetros.

El Turbo PASCAL efectúa el pase de parámetros a través de la pila. Siempre son evaluados en insertados en la pila, en el orden en que aparecen declarados, de izquierda a derecha.

Parámetros por valor.

Un parámetros por valor es aquel que no puede ser modificado por la rutina invocada. El método para el pase del parámetro dependerá del tipo de dato.

Escalar: Los tipos escalares son BOOLEAN, CHAR, SHORTINT, BYTE, WORD, LONGINT, subrangos y enumerados. Todos ellos son pasados a través de la pila y ocuparan tanto espacio en la pila como longitud tenga el dato.

Real: Los parámetros reales son pasados a través de la pila en 3 palabras de datos (6 bytes).

Apuntador: Los apuntadores son pasados a través de la pila en dos palabras de datos (FAR). La primera contiene el segmento y ocupa la posición de memoria más alta. La segunda contiene el desplazamiento.

Cadenas: El Turbo PASCAL introduce en la pila un apuntador FAR, a la zona de memoria donde se encuentre la cadena.

Registros y **Arreglos**: Si su longitud es de 1, 2 o 4 bytes, son copiados en la pila. En el caso contrario, es introducido un apuntador FAR.

Conjuntos: Al igual que con las cadenas, se introduce un apuntador FAR en la pila.

Parámetros por Referencia.

Para los parámetros por referencia (VAR), el Turbo PASCAL introduce en la pila un apuntador FAR.

***Limpieza* de la Pila**.

El Turbo PASCAL espera que la rutina invocada elimine de la pila los argumentos introducidos antes de su invocación. Esto exige que el programa en ensamblador emplee para su terminación la instrucción **RET *N*** donde ***N*** es el número de bytes ocupados por los argumentos en la pila.

La directiva ARG.

Para el acceso a los parámetros en la pila, el Turbo Assembler proporciona un método fácil y segura mediante el empleo de la directiva ARG. Esta determina los desplazamiento relativos a BP de cada argumento en la pila. Puede además calcular la cantidad de bytes ocupados para ser usado en la instrucción **RET**. Además, los símbolos definidos mediante la directiva ARG son locales. El ejemplo siguiente ilustra el empleo de la directiva ARG:

```
CODE        SEGMENT
            ASSUME CS:CODE
MyProc      PROC FAR                ; procedure MyProc (i,j: integer);external;
            PUBLIC MyProc
            ARG j:WORD, i:WORD = RetBytes
            push bp                 ; salvar BP
            mov bp,sp               ; permitir acceso a los argumentos
            mov ax,i                ; acceso a los argumento i      AX= i
            sub ax,j                ; acceso al argumento j         AX= i-j
            :
            :
            ret RetBytes
```

La directiva ARG genera automáticamente los símbolos locales i y j. La línea:

ARG j:WORD, i:WORD = RetBytes

iguala el símbolo **i** a **[WORD PTR BP+6]** ,el símbolo **j** a **[WORD PTR BP+8]** y **RetBytes** a 4. El desplazamiento asignado a cada símbolo toma en cuenta el atributo del procedimiento.

Al usar la directiva ARG es conveniente recordar que los parámetros deben ser listados en el orden contrario al de su respectiva declaración dentro del programa en PASCAL.

En una función que devuelva un cadena, la opción **RETURN** de la directiva **ARG** permite definir un símbolo que señale al *buffer* temporal previsto para el almacenamiento del resultado. Por ejemplo:

ARG j:WORD, i:WORD = RetBytes RETURNS Result: DWORD

La directiva MODEL.

La directiva MODEL establece el empleo de directivas simplificadas y permite la especificación del modelo de memoria y opciones del lenguaje. Para un programa en PASCAL el modelo de memoria será usualmente *LARGE*. El ejemplo siguiente ilustra el uso de la directiva MODEL.

```
            .MODEL Large,PASCAL
            .CODE
MyProc      PROC FAR i:WORD, j:WORD RETURNS Result:DWORD
            PUBLIC MyProc
            mov ax,i
            sub ax,j
            :
            :
            ret
```

Nótese que los argumentos no tienen que ser especificados en orden inverso. Al usar el atributo PASCAL en la directiva MODEL, se establecen automáticamente las convenciones para pase de parámetros, los nombres de los segmentos y se incluyen las instrucciones **PUSH BP** y **MOV BP,SP** al inicio del procedimiento y **POP BP** y **RET *N*** al final del mismo.

Resultado de una Función en Turbo PASCAL.

Las funciones en Turbo PASCAL tienen diferentes mecanismo para devolver resultados, dependiendo del tipo de dato:

Escalar: Los resultados escalares son devueltos en los registros del microprocesador. Si el resultado es de un byte, es devuelto en AL, si es de dos en AX y si es de 4 en DX:AX. donde DX contiene la palabra más significativa.

Real: Los valores reales son devueltos en tres registros: La palabra más significativa en DX; la intermedia en BX y la menos significativa en AX.

Cadenas: El resultado es devuelto en un espacio de memoria temporal reservado por el programa en PASCAL antes de la invocación a la función. El mismo coloca en la pila un apuntador FAR a dicha zona de memoria, antes del primer parámetro.

Apuntador: Los apuntadores son devueltos en los registros DX:AX (segmento:desplazamiento).

Variables Locales.

Las rutinas en lenguaje ensamblador pueden reservar espacio en memoria para sus variables locales tanto estáticas como volátiles.

Variables Estáticas: El programa puede reservar espacio para variables estáticas en el segmento de datos globales (**DATA** o **DSEG**) mediante el empleo de las directivas para reserva de memoria: DB, DW, etc. Por ejemplo:

```
DATA            SEGMENT PUBLIC
MyInt           DB ?
MyByte          DW ?
:
DATA            ENDS
```

Existen dos importantes restricciones para la declaración de variables estáticas: estas deben ser privadas y no pueden ser inicializadas.

Variables Volátiles: Las variables locales son reservadas en la pila, durante la ejecución de la rutina. Este espacio debe ser liberado y BP ajustado antes de la finalización de la misma. Por ejemplo:

```
CODE            SEGMENT
        ASSUME CS:CODE
MyProc          PROC FAR                        ; PROCEDURE MyProc (i:Integer);
        PUBLIC MyProc
        ARG i:WORD= RetBytes
        LOCAL a:WORD, b:WORD= LocalSpace
        push bp                 ; salvar BP
        mov bp,sp               ; permitir acceso a los argumentos
        sub sp,LocalSpace       ; reservar memoria para variables locales volátiles
        mov ax,i                ; AX= i
        mov a,ax                ; a= i
                :
```

La sentencia:

LOCAL a:WORD, b:WORD= LocalSpace

iguala los símbolos **a** a **[BP-2]**, **b** a **[BP-4]** y **LocalSpace** a **4**

Ejemplo: Rutina general de conversión a hexadecimal.

A continuación se presenta una rutina para conversión de un número a una cadena con dígitos hexadecimales. El listado siguiente corresponde al programa en lenguaje ensamblador.

```
                .MODEL Large,PASCAL
                .CODE
HexStr          PROC FAR num:WORD, byteCount:BYTE RETURNS Result:DWORD
                PUBLIC HexStr
                les di,Result           ; ES:DI= dirección del buffer
                mov dx,ds               ; salvar DS
                lds si,num              ; DS:SI= dirección del número
                mov al,byteCount        ; AX= longitud del dato
                xor ah,ah
                mov cx,ax               ; inicializa contador
                add si,ax               ; comienza por el byte más significativo
                dec si
                shl ax,1                ; determinar cuantos dígitos resultan
                cld
                stosb                   ; inicializa cadena resultante
HexLoop:        std
                lodsb                   ; obtener el byte siguiente
                mov ah,al               ; salvarlo
                shr al,1                ; obtener el nibble más significativo
                shr al,1
                shr al,1
                shr al,1
                add al,90h          ; conversión a hexadecimal
                daa
                adc al,40h
                daa                     ; conversión a ASCII
                cld
                stosb                   ; almacenar
                mov al,ah               ; repetir conversión para el nibble menos significativo
                and al,0Fh
                add al,90h
                daa
                adc al,40h
                daa
                stosb
                loop HexLoop            ; seguir hasta terminar
                mov ds,dx               ; restaurar DS
                ret
HexStr          ENDP
CODE            ENDS
                END
```

A continuación, el programa en PASCAL que usa a la rutina anterior.

```
PROGRAM HexTest
VAR
        num: Word;

FUNCTION HexStr (var num; byteCount; Byte): String; Far; External;

{$L HEXSTR.OBJ}

BEGIN
        num:= $FACE;
        WriteLn ('La cadena hexadecimal convertida es: ',HexStr(num,sizeof(num)));
END.
```

Interfaz con lenguaje C y C++.

Tradicionalmente el C++ y el lenguaje ensamblador han sido mezclados escribiendo módulos separados completamente en C++ o completamente en ensamblador, compilando los módulo en C++ y ensamblando los módulos en ensamblador y finalmente enlazando los módulos objeto. Esta es una solución muy recomendable, pero tiene un inconveniente: el programador de ensamblador debe atender a las reglas de interfaz con C++.

Reglas para mezclar programas en C++ y lenguaje ensamblador.

Para enlazar programas en C++ con programas en lenguaje ensamblador es necesario observar dos cosas:

1. El módulo en lenguaje ensamblador de emplear un esquema de segmentos compatible con C++.

2. Los módulos en C++ y ensamblador deben compartir variables y funciones en una forma aceptable para el compilador de C++.

Enlace de módulos en C++ con módulos en lenguaje ensamblador.

El enlace es un concepto importante en C++. El compilador y el enlazador deben trabajar en conjunto para asegurar que las invocaciones a funciones contienen los argumentos correctos. Un proceso llamado *name-mangling* proporciona la información necesaria referente a los tipos de argumentos. Este proceso modifica el nombre de la función para indicar los argumentos que ella toma.

Cuando el programa es escrito completamente en C++, la modificación de los nombres de las funciones ocurre automáticamente y de una manera transparente al programador. Sin embargo, al escribir rutinas en ensamblador que van a ser enlazadas con programas en C++, es necesario asegurarse de que las funciones contengan los nombres modificados.

Por ejemplo, los fragmento de código siguientes:

```
void test ()
{
}

void test (int)
{
}

void test (int,int)
{
}

void test (float,double)
{
}
```

corresponden con definiciones a nivel de ensamblador, de la siguiente manera:

```
@test$qv  PROC NEAR
                push bp
                mov bp,sp
                pop bp
@test$qv  ENDP

@test$qi   PROC NEAR
                push bp
                mov bp,sp
                pop bp
@test$qi   ENDP
```

```
@test$qii  PROC NEAR
                    push bp
                    mov bp,sp
                    pop bp
@test$qii  ENDP

@test$qfd PROC NEAR
                    push bp
                    mov bp,sp
                    pop bp
@test$qfd ENDP
```

Uso de la directiva Extern "C" para simplificar el enlace.

Existen sin embargo, una manera de emplear nombres no modificados en las rutinas en ensamblador. Esto es conveniente además, porque protege a las rutinas en ensamblador de posibles alteraciones en el algoritmo de modificación de nombres empleado por futuras versiones de los compiladores de C++. El Borland C++ permite el uso de nombres de funciones estándar C en los programas C++. Por ejemplo:

```
extern "C" {
        int add (int *a,int b);
}
```

hace que la función **add** adopte el estilo de nombres para C. La definición de la rutina en ensamblador es como sigue:

```
                    PUBLIC _add
_add                PROC
```

La declaración de una función en ensamblador, dentro de un bloque **extern "C"** elimina el trabajo de determinar el nombre modificado que espera el C++.

Modelos de Memoria y Segmentos.

Para que una rutina en ensamblador pueda ser usada desde un programa en C++, debe usar el mismo modelo de memoria y segmento de código compatible con el programa en C++. Para que una variable definida en un módulo en ensamblador pueda ser accesada desde un código en C++, el código en ensamblador debe seguir la convención de nombre del segmento de datos impuesta por el C++.

El manejo de modelos de memoria y segmentos puede ser algo complejo de implementar en ensamblador. Afortunadamente el Turbo Assembler proporciona un conjunto de directivas simplificadas, que facilitan esta labor.

Directivas simplificadas y C++.

La directiva **.MODEL** le indica al ensamblador, que los segmentos creados por las directivas simplificadas deben ser compatibles con el modelo de memoria seleccionado y controla el atributo por defecto de los procedimientos definidos por la directiva **PROC**. Los modelos de memoria definidos con la directiva **.MODEL** son directamente compatibles con los manejados con el BORLAND C++, excepto para el modelo HUGE. En este caso, se indica el modelo **TCHUGE** desde el módulo en ensamblador. Cuando se emplee el modelo LARGE, es necesario especificar el modificador **FARSTACK** en la directiva **.MODEL**, para señalar que el segmento de pila no forme parte de DGROUP.

Las directivas simplificadas **.CODE**, **.DATA**, **.DATA?**, **.FARDATA** y **FARDATA?** generan segmentos compatibles con C++. Las directivas para segmentos de variables no inicializadas no deben ser usadas cuando se emplee el modelo HUGE.

Por ejemplo:

```
                    .MODEL Small                ; modelo Small
                    .DATA                       ; segmento de datos
                    EXTRN _Repetitions:WORD     ; definida externamente
                    PUBLIC _StartingValue       : variable para otros módulos
_StartingValue      DW 0
                    .DATA?                      ; segmento de datos no inicializados
RunningTotal        DW ?
                    .CODE                       ; segmento de código
                    PUBLIC _DoTotal     ; función
                    mov cx,[_Repetitions] ; # de cuentas por hacer
                    mov ax,{_StartingValue]
                    mov [RunningTotal],ax       ; establecer valor inicial
TotalLoop: inc {RunningTotal]               ; RunningTotal++
                    loop TotalLoop
                    mov ax,[RunningTotal]       ; devuelve valor total
                    ret
_DoTotal            ENDP
                    END
```

La rutina *_DoTotal* esta lista para ser invocada desde un programa en C++ (modelo Small) por medio de la siguiente sentencia:

```
DoTotal();
```

El programa completo en C, que emplea a la rutina *_DoTotal* es:

```
extern "C" {
        int DoTotal (void);
}
extern int StartingValue;
int Repetitions;
main ()
{
        int i;
        Repetitions= 10;
        StartingValue= 2;
        printf ("%d\n",DoTotal());
}
```

Símbolos públicos y externos.

El código en ensamblador puede invocar funciones y referenciar variables en C++, así como el código en C++ puede invocar rutinas y referenciar variables en ensamblador. Una vez que se han definido loa segmentos de una manera compatible con el C++, sólo es necesario seguir unas cuantas reglas para intercambiar información entre C++ y ensamblador.

Carácter de subrayado y Lenguaje C.

Si se programa en C en lugar de C++, todas las etiquetas externas deberán ir precedidas del carácter de subrayado (_). EL compilador de C automáticamente le añade el carácter de subrayado a todos los nombre de variables y funciones externas, de tal manera que los identificadores públicos en el programa en ensamblador deben ir precedidos de dicho carácter. Por ejemplo, este código en C:

```
external int ToggleFlag();
int Flag;
main ()
{
        ToggleFlag();
}
```

enlaza apropiadamente con el siguiente programa en ensamblador:

```
                .MODEL Small
                .DATA
                EXTRN _Flag:WORD
                .CODE
                PUBLIC _ToggleFlag
_ToggleFlag     PROC
                cmp [_Flag],0           ; si la bandera está apagada,
                jz SetFlag              ; encenderla
                mov [_Flag],0           ; si no, apagarla
                jmp SHORT EndToggleFlag
SetFlag:        mov [_Flag],1
EndToggleFlag:  ret
_ToggleFlag     ENDP
                END
```

Mayúsculas y minúsculas

El Turbo Assembler normalmente no diferencia entre mayúsculas y minúsculas al evaluar los símbolos. Ya que el C++ si lo hace, es necesario conseguir que el ensamblador trabaje de esta misma manera. Esto se consigue por medio de las banderas **/ml** y **/mx**. El primero activa la diferenciación entre mayúsculas y minúsculas para todos los símbolos y la segunda sólo para los símbolos públicos (**PUBLIC**), externos (**EXTRN**), globales (**GLOBAL**) y comunes (**COMM**).

Pase de parámetros.

El Borland C++ pasa los parámetros a las funciones a través de la pila. Antes de invocar a la función, introduce los argumentos en la pila comenzando por el último (el más a la derecha). La invocación siguiente:

```
Test (i,j,1);
```

produce el siguiente código ensamblador al ser compilado:

```
mov ax,1
push ax
push WORD PTR DGROUP:_j
push WORD PTR DGROUP:_i
call NEAR PTR _Test
add sp,6
```

Al terminar la función, los parámetros que fueron introducidos en la pila se encuentran allí aun, pero ya no son útiles. En consecuencia, el C++ ajusta inmediatamente el apuntador de pila para eliminarlos. En el ejemplo anterior, se pasan tres parámetros de 2 bytes cada uno lo que suma 6 bytes. El C++ le suma 6 bytes al apuntador de pila y descarta los argumentos. El punto importante, es saber que la rutina invocadora es quien se encarga de limpiar la pila.

Las rutinas en ensamblador pueden tener acceso a los argumentos usándola directiva **ARG**. La función *Test* del ejemplo anterior podría ser escrita de la siguiente manera:

```
                .MODEL Small
                .CODE
                PUBLIC _Test
_Test           PROC
                ARG a:WORD,b:WORD,c:WORD
                push bp
                mov bp,sp
                mov ax,a            ; AX= a
                add ax,b            ; AX= a+b
                sub ax,c            ; AX= a+b-c
                pop bp
                ret
_Test           ENDP
                END
```

Para declarar variables locales automáticas, se emplea la directiva **LOCAL**. Por ejemplo:

```
LOCAL LocalArray:BYTE:100, LocalCount:WORD= AUTO_SIZE
```

define dos variables locales automáticas: un arreglo de 100 elementos de un byte de longitud (*LocalArray*) y una variable entera (*LocalCount*). *AUTO_SIZE* es el total de bytes requeridos. Este monto debe ser substraído del apuntador de pila (SP) para reservar el espacio de memoria para las variables locales.

Uso de registros.

Las funciones en ensamblador que sean invocadas desde un programa en C, deben preservar el contenido de los registros BP, SP, CS, DS y SS. Los registros SI y DI deben ser preservados dependiendo de si están o no habilitadas las variables tipo registro.

Retorno de valores.

Una función en ensamblador puede devolver valores al igual que una función en C. Si el dato devuelto corresponde a un tipo que ocupe 16 bits o menos, el mismo queda almacenado en AX. Lo mismo ocurre con apuntadores del tipo NEAR. Si el dato es de 32 bits o es un apuntador FAR, el mismo es devuelto en los registros DX:AX.

Ejemplo: Rutina que determina el número de líneas y caracteres en una cadena.

La siguiente rutina determina el número de líneas y caracteres en una cadena dada. El número de líneas es devuelto como función. EL número de caracteres es pasado por referencia.

```
; Prototipo: extern unsigned int LineCount (char *near StringToCount,unsigned int near *CharacterCountPtr);

; Datos de entrada:
; char near *StringToCount: apuntador a la cadena
; unsigned int near *CharacterCountPtr: apuntador a la variable donde se almacenará el conteo de caracteres

NewLine         EQU 0Ah                         ; salto de línea
                .MODEL Small
                .CODE
                PUBLIC _LineCount
_LineCount      PROC
                ARG String:DWORD,Count:WORD
                push bp                         ; salvar BP
                mov  bp,sp                      ; permitir acceso a los argumentos
                push si                         ; salvar registro SI
                mov si,String                   ; SI apunta a la cadena
                xor cx,cx                       ; contador de caracteres= 0
                mov dx,cx                       ; contador de líneas= 0
LineCountLoop:  lodsb                           ; lee el siguiente carácter
                and al,al                       ; si es cero, finalizó la cadena
                jz EndLineCount
                inc cx                          ; si no, contar otro carácter
                cmp al,NewLine                  ; si no es un salto de línea,
                jne LineCountLoop               ; verificar el siguiente carácter
                inc dx                          ; si si los es, contar una línea
                jmp LineCountLoop
EndLineCount:   inc dx                          ; cuenta la línea inicial
                mov bx,Count                    ; BX hace referencia a Count
                mov [bx],cx                     ; salva el resultado en la variable contadora
                mov ax,dx                       ; retorna el número de líneas como función
                pop si                          ; recuperar registro SI
                pop bp
                ret
_LineCount      ENDP
                END
```

EL siguiente código en C, es una invocación simple a la función *LineCount*:

```
char *TestString= "Line 1\nline 2\nline 3";
extern "C" {
        unsigned int LineCount (char *StringToCount,unsigned int *CharacterCountPtr);}

main ()
{
        unsigned int LCount;
        unsigned int CCount;

        LCount= LineCount (TestString,&CCount);
        printf ("Líneas: %d\nCaracteres: %d\n",LCount,CCount);
}
```

AAA
Ajuste ASCII después de la Adición.

O	D	I	T	S	Z	A	P	C
?				?	?	*	?	*

Efectúa un ajuste ASCII del resultado (en AL) de una operación de suma entre dos dígitos BCD no empaquetados.

AAD
Ajuste ASCII antes de la División.

O	D	I	T	S	Z	A	P	C
?				?	?	*	?	*

Prepara dos dígitos BCD no empaquetados, para una operación de división que dará como resultado un número BCD no empaquetado.

AAM
Ajuste ASCII después de la Multiplicación.

O	D	I	T	S	Z	A	P	C
?				?	?	*	?	*

Efectúa un ajuste ASCII del resultado (en AX) de la operación de multiplicación entre dos dígitos BCD no empaquetados.

AAS
Ajuste ASCII después de la Resta.

O	D	I	T	S	Z	A	P	C
?				?	?	*	?	*

Efectúa un ajuste ASCII sobre el resultado (en AL) de una operación de substracción entre dos dígitos BCD no empaquetados.

ADC *r/m8,r/m/inm8*
r/m16,r/m/inm16

Suma con Acarreo.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Efectúa la suma entre dos operandos y el bit de acarreo y el resultado es almacenado en el primer operando. No está permitido que ambos operandos hagan referencia a posiciones de memoria.

ADD *r/m8,r/m/inm8*
r/m16,r/m/inm16

Suma.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Efectúa la suma entre dos operandos y el resultado es almacenado en el primer operando.

AND *r/m8,r/m/inm8*
r/m16,r/m/inm16

Operación Lógica AND.

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Efectúa la operación lógica AND entre dos operandos y el resultado es almacenado en el primer operando. No está permitido que ambos operandos hagan referencia a posiciones de memoria.

CALL *rel16*
r/m16
ptr16:16
m16:16

Invoca una Subrutina.

O	D	I	T	S	Z	A	P	C

Ejecuta la subrutina indicada.

CBW

Convierte un Entero Corto en un Entero Largo.

O	D	I	T	S	Z	A	P	C

Convierte el contenido de AL (entero corto con signo) a un entero largo con signo en AX, extendiendo el bit más significativo de AL.

CLC

Apaga el Bit de Acarreo.

O	D	I	T	S	Z	A	P	C
								0

Pone a cero el bit de acarreo del registro de estado.

CLD

Coloca a cero el Bit de Dirección.

O	D	I	T	S	Z	A	P	C
	0							

Borra el bit de dirección del registro de estado.

CLI

Apaga el Bit de Habilitación de Interrupciones.

O	D	I	T	S	Z	A	P	C
		0						

Pone a cero el bit de habilitación de interrupciones enmascarables, del registro de estado.

CMC

Invierte el Estado del Bit de Acarreo.

O	D	I	T	S	Z	A	P	C
								*

Invierte el estado actual del bit de acarreo del registro de estado.

CMP *r/m8,r/m/inm8*
r/m16,r/m/inm16

Operación Lógica AND.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Resta el operando2 del operando1, altera las banderas del registro de estado de acuerdo con el resultado de la operación y desecha el resultado.
No está permitido que ambos operandos hagan referencia a posiciones de memoria.

DAA

Ajuste Decimal Después de una Suma.

O	D	I	T	S	Z	A	P	C
?				*	*	*	*	*

Efectúa el ajuste decimal del resultado (en AL) de la suma de dos dígitos BCD empaquetados.

DEC *r/m8*
r/m16

Decrementa en 1.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

Resta 1 del operando indicado.

DIV *r/m8*
r/m16

Efectúa una Operación de División sin signo.

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Efectúa una operación de división sin signo. El dividendo es implícito; sólo el divisor es dado como operando. El tipo de divisor determina el uso de los registros:

Tamaño	*Dividendo*	*Divisor*	*Cociente*	*Residuo*
Byte	AX	r/m8	AL	AH
Palabra	DX:AX	r/m16	AX	DX

HLT

Detiene la Ejecución del Microprocesador.

O	D	I	T	S	Z	A	P	C

Detiene la ejecución del microprocesador y lo coloca en estado HALT.

IDIV *r/m8*
r/m16

Efectúa una Operación de División con signo.

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Efectúa una operación de división con signo. El dividendo es implícito; sólo el divisor es dado como operando. El tipo de divisor determina el uso de los registros:

Tamaño	*Dividendo*	*Divisor*	*Cociente*	*Residuo*
Byte	AX	r/m8	AL	AH
Palabra	DX:AX	r/m16	AX	DX

IMUL *r/m8*
r/m16

Efectúa una Operación de Multiplicación con signo.

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

Efectúa una operación de multiplicación con signo. Uno de los multiplicandos es implícito; el otro es dado como operando. Los registros empleados dependen del operando dado.

Tamaño	*Mult. 1*	*Mult.2*	*Resultado*

Byte	AL	r/m8	AX
Palabra	AX	r/m16	DX:AX

IN *AC,inm8*
AC,DX

Transfiere un dato del Puerto especificado al Operando.

O	D	I	T	S	Z	A	P	C

Transfiere un dato del puerto indicado al operando especificado.

INC *m/r8*
m/r16

Incrementa en 1.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

Incrementa en uno el contenido del operando especificado.

INT *inm8*
INTO

Invoca una Rutina de Servicio de Interrupción.

O	D	I	T	S	Z	A	P	C
		0	0					

Genera una llamada a una rutina de servicio de interrupción. El operando, representa el índice dentro de la tabla de vectores de interrupción, de donde esta obtiene la dirección de la rutina a ejecutarse.
La instrucción INTO difiere de INT en que el operando es 4 de manera implícita y en que la interrupción es ejecutada sólo si el bit de desbordamiento (O) está activado.

IRET

Retorno de Interrupción.

O	D	I	T	S	Z	A	P	C
*	*	*	*	*	*	*	*	*

Extrae de la pila el contador de programa, el segmento CS y el registro de estado, reasumiendo la ejecución del programa interrumpido.

Jcc *rel8*

Salto Condicional.

O	D	I	T	S	Z	A	P	C

Verifica el estado de la bandera indicada y efectúa el salto si la condición es verdadera.
Las condiciones posible son:

Con Signo	***Sin Signo***	***Condición***

JE o JZ	JE o JZ	igual
JNE o JNZ	JNE o JNZ	diferente
JG o JNLE	JA o JNBE	mayor que
JGE o JNL	JAE o JNB	mayor o igual
JL o JNGE	JB o JNAE	menor que
JLE o JNG	JBE o JNA	menor o igual.

JMP *rel8*
rel16
r/m16
ptr16:16
m16:16

Salto incondicional.

O D I T S Z A P C

Efectúa un salto incondicional al punto especificado.

LAHF

Carga el Registro de Estado en el Registro AH.

O D I T S Z A P C

Transfiere el byte menos significativo del registro de estado en el registro AH.

LEA *r16,m*

Carga la Dirección Efectiva.

O D I T S Z A P C

Calcula la dirección efectiva y la almacena en el registro especificado.

LOCK

Coloca en Estado Acertado la Señal LOCK.

O D I T S Z A P C

Ocasiona que la señal LOCK del microprocesador esté en su estado de aserción durante la ejecución de la próxima instrucción. Esto garantiza que el microprocesador tendrá el control de los recursos del sistema durante la ejecución de dicha instrucción.

LODSB
LODSW

Carga una Cadena.

O	D	I	T	S	Z	A	P	C

Carga el registro acumulador (AL o AX) con el contenido de la posición de memoria señalada por el registro índice fuente (SI). Luego SI es avanzado a la siguiente posición. Si el bit de dirección está en cero, el registro SI es incrementado y si es uno, el registro SI es decrementado. El incremento o decremento será de uno para operaciones con bytes (postfijo B) y de dos para operaciones con palabras (postfijo W).
Puede ir precedida del prefijo REP para procesamiento de bloques de memoria.

LOOP *rel8*
LOOPcond *rel8*

Control de Lazo.

O	D	I	T	S	Z	A	P	C

Decrementa el contenido del registro contador (CX) si alterar el registro de estado. Si el contenido de CX es diferente de cero, se efectúa un salto a la etiqueta especificada. En el caso de LOOPcond, el salto se efectúa si la condición especificada resulta verdadera.

MOV *r/m8,r/m/inm8*
r/m16,r/m/inm16
Sreg,r/m16
r/m16,Sreg

Transfiere Datos.

O	D	I	T	S	Z	A	P	C

Copia el contenido del segundo operando en el primer operando.
No está permitido que ambos operandos hagan referencia a posiciones de memoria.

MOVSB
MOVSW

Copia una Cadena.

O	D	I	T	S	Z	A	P	C

Copia el contenido de la posición de memoria apuntada por DS:SI en la posición señalada por ES:DI. No se permite contrarrestación de segmento en el destino. Luego SI y DI son avanzado automáticamente a la siguiente posición. Si el bit de dirección está en cero, los registros SI y DI serán incrementados y si es uno, serán decrementados. El incremento o decremento será de uno para operaciones con bytes (postfijo B) y de dos para operaciones con palabras (postfijo W)
Puede ir precedida del prefijo REP para procesamiento de bloques de memoria.

MUL *r/m8*
r/m16

Efectúa una Operación de Multiplicación sin signo.

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

Efectúa una operación de multiplicación sin signo. Uno de los multiplicandos es implícito; el otro es dado como operando. Los registros empleados dependen del operando dado.

Tamaño	*Mult. 1*	*Mult.2*	*Resultado*
Byte	AL	r/m8	AX
Palabra	AX	r/m16	DX:AX

NEG *r/m8*
r/m16

Complemento a Dos.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Reemplaza el contenido actual del operando por su complemento a dos. El bit de acarreo será puesto a 1 a menos que el operando sea cero, en cuyo caso el bit de acarreo queda en cero.

NOP

No Operación

O	D	I	T	S	Z	A	P	C

No lleva a cabo ninguna operación. Sólo afecta al contador de programa.

NOT *r/m8*
r/m16

Complemento a Uno.

O	D	I	T	S	Z	A	P	C

Invierte el contenido del operando; cada uno pasa a cero y viceversa.

OR *r/m8,r/m/inm8*
r/m16,r/m/inm16

Operación Lógica OR.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Calcula la operación lógica OR entre dos operandos y coloca el resultado en el primero.
No está permitido que ambos operandos hagan referencia a posiciones de memoria.

OUT *inm8,AL*
inm16,AX
DX,AC

Escribe un Dato en un Puerto.

O	D	I	T	S	Z	A	P	C

Transfiere un dato de 8 o 16 bits al puerto especificado.

POP *r/m16/Sreg*

Extrae un Dato de la Pila.

O	D	I	T	S	Z	A	P	C

Reemplaza el contenido previo del operando con el dato en el tope de la pila. Luego el apuntador de pila es incrementado en 2.

POPA (286+)

Extrae de la Pila todos los Registros Generales.

O	D	I	T	S	Z	A	P	C

Extrae de la pila el contenido de los 8 registros generales de 16 bits.

POPF

Extrae de la Pila el Registro de Estado.

O	D	I	T	S	Z	A	P	C
*	*	*	*	*	*	*	*	*

Extrae el valor en el tope de la pila y lo deposita en el registro de estado.

PUSH *r/m16/Sreg*
inm16 (286+)

Introduce un Dato en la Pila.

O	D	I	T	S	Z	A	P	C

Decrementa el apuntador de pila en dos y coloca el contenido del operando en el nuevo tope de la pila.

PUSHA (286+)

Introduce los Registros Generales en la Pila.

O	D	I	T	S	Z	A	P	C

Almacena el contenido de los 8 registros generales de 16 bits en la pila y actualiza el apuntador de pila.

PUSHF

Introduce el Registro de Estado en la Pila.

O	D	I	T	S	Z	A	P	C

Decrementa el apuntador de pila en dos y deposita el contenido del registro de estado en el nuevo tope de la pila.

RCL *r/m8,1*
r/m8,CL
r/m16,1
r/m16,CL
RCR
RCL
ROR

Rotación.

O	D	I	T	S	Z	A	P	C
*								*

Rota el contenido del operando dado. Las operaciones de rotación a la izquierda copian cada bit en la posición inmediata superior excepto por el más significativo que es copiado en la posición menos significativa. Las de rotación a la derecha hacen lo contrario: los bits son desplazados hacia abajo y el menos significativo es copiado en la posición más significativa.

En las instrucciones RCL y RCR, el bit de acarreo forma parte del valor rotado. RCR desplaza el contenido del bit de acarreo al bit menos significativo y el más significativo al bit de acarreo; RCR copia el bit de acarreo en la posición más significativa y el bit menos significativo en el bit de acarreo.

La rotación es repetida tantas veces como sea indicado por el segundo operando.

Las instrucciones RCL, RCR, RCL y ROR presentan las mismas combinaciones posibles de operandos.

REP *instrucción*
REPE
REPZ
REPNE
REPNZ

Repite la Instrucción de Cadena Asociada.

O	D	I	T	S	Z	A	P	C
					*			

Ejecuta la instrucción de cadena asociada, tantas veces como lo indique el contenido del registro CX. Las instrucciones REPE y REPZ pueden terminar la auto ejecución si el bit de cero está en cero. REPNE y REPNZ finalizarán la auto ejecución si el bit de cero está en uno.

RET *inm16*

Retorno de una Subrutina.

O	D	I	T	S	Z	A	P	C

Transfiere el control a la dirección de retorno ubicada en la pila. El parámetro numérico opcional, indica el número de bytes que deben ser liberados en la pila luego que es extraída la dirección de retorno.

SAHF

Almacena AH en el Registro de Estado.

O	D	I	T	S	Z	A	P	C
				*	*	*	*	*

Almacena el contenido del registro AH en el registro de estado.

SAL *r/m8,1*
r/m8,CL
r/m16,1
r/m16,CL

SAR

SHL

SHR

Desplazamiento.

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	*

La instrucción SAL (o su sinónimo SHL) desplazan los bits del operando hacia arriba. El bits más significativo es colocado en el bit de acarreo y el menos significativo es puesto a cero.
SAR y SHR desplazan los bits del operando hacia abajo. El bit menos significativo es colocado en el bit de acarreo y el más significativo permanece inalterado.
El desplazamiento es repetido tantas veces como sea indicado por el segundo operando.
Las instrucciones SAL, SAR, SHL y SHR presentan las mismas combinaciones posibles de operandos.

SCASB

SCASW

Compara Cadenas.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Resta el contenido de la posición de memoria apuntada por ES:DI del contenido del registro acumulador empleado. El resultado es desechado pero las banderas del registro de estado son actualizadas de acuerdo con el resultado de la operación. No es posible la contrarrestación de segmentos en el operando destino. Luego DI es avanzado automáticamente a la siguiente posición. Si el bit de dirección está en cero, el registro DI será incrementado y si es uno, será decrementado. El incremento o decremento será de uno para operaciones con bytes (postfijo B) y de dos para operaciones con palabras (postfijo W)
Puede ir precedida del prefijo REP para procesamiento de bloques de memoria.

STC

Enciende el Bit de Acarreo.

O	D	I	T	S	Z	A	P	C
								1

Coloca el bit de acarreo en uno.

STD

Enciende el Bit de Dirección.

O	D	I	T	S	Z	A	P	C
	1							

Coloca el bit de dirección en uno.

STI

Habilita las Interrupciones Enmascarables.

O	D	I	T	S	Z	A	P	C
		1						

Coloca el bit de habilitación de interrupciones en uno.

STOSB
STOSW

Almacena Cadenas.

O	D	I	T	S	Z	A	P	C

Transfiere el contenido del acumulador empleado a la posición de memoria apuntada por ES:DI No es posible la contrarrestación de segmentos en el operando destino. Luego DI es avanzado automáticamente a la siguiente posición. Si el bit de dirección está en cero, el registro DI será incrementado y si es uno, será decrementado. El incremento o decremento será de uno para operaciones con bytes (postfijo B) y de dos para operaciones con palabras (postfijo W)
Puede ir precedida del prefijo REP para procesamiento de bloques de memoria.

SUB *r/m8,r/m/inm8*
r/m16,r/m/inm16

Resta.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Substrae el contenido del segundo operando del primer operando y el resultado es colocado en el primer operando. Las banderas del registro de estado son actualizadas.
No está permitido que ambos operandos hagan referencia a posiciones de memoria.

TEST *r/m8,r/m/inm8*
r/m16,r/m/inm16

Comparación Lógica.

O	D	I	T	S	Z	A	P	C
0				*	*	*	*	0

Calcula la operación lógica AND entre los dos operandos y actualiza los bits del registro de estado de acuerdo con el resultado. El resultado es desechado.
No está permitido que ambos operandos hagan referencia a posiciones de memoria.

WAIT

Espera hasta que la Señal BUSY este inactiva.

O	D	I	T	S	Z	A	P	C

Suspende la ejecución del microprocesador hasta que la señal BUSY este inactiva. La señal BUSY es manejada por el coprocesador matemático.

XCHG*r/m8,r/m8*
r/m16,r/m16

Intercambia el Contenido de los Operandos.

O	D	I	T	S	Z	A	P	C

Intercambia el contenido de los dos operandos.
No está permitido que ambos operandos hagan referencia a posiciones de memoria.

XOR *r/m8,r/m/inm8*
r/m16,r/m/inm16

Operación Lógica OR exclusiva.

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Calcula la operación lógica or exclusiva entre los operandos dados y deposita el resultado en el primer operando. Es actualizado el registro de estado.
No está permitido que ambos operandos hagan referencia a posiciones de memoria.

Apéndice B
Turbo Assembler: Operadores

( ) **(** *expresión* **)**
Ideal, MASM

Altera el orden de precedencia de las operaciones, asignando máxima prioridad de evaluación.

***** *expresion1* ***** *expresion2*
Ideal, MASM

Multiplica dos expresiones enteras.

+(binario) *expresion1* **+** *expresion2*
Ideal, MASM

Suma dos expresiones.

+ (unario) **+** *expresión*
Ideal, MASM

Indica que una expresión es positiva.

- (binario) *expresion1* **-** *expresion2*
Ideal, MASM

Resta dos expresiones.

- (unario) **-** *expresión*
Ideal, MASM

Indica que una expresión es negativa.

. *estructura***.***campo*
Ideal, MASM

Selecciona un campo de una estructura.

/ *expresion1* **/** *expresion2*

Ideal, MASM

Divide dos expresiones enteras.

: *segmento o grupo* **:** *expresión*

Ideal, MASM

Genera una contrarrestación de segmento o grupo.

? *Dx* **?**

Ideal, MASM

Inicializa un variable con un valor indeterminado.

[ ] *expresion1* **[** *expresión***]**
[*expresion1***][***expresion2***]**

Ideal, MASM

En modo MASM indica direccionamiento indirecto a memoria. En modo Ideal especifica referencia a memoria.

AND *expresion1* **AND** *expresion2*

Ideal, MASM

Efectúa la Operación Lógica AND entre dos operandos.

BYTE **BYTE** *expresión*

Ideal

Obliga a que una expresión sea del tamaño de un byte.

BYTE PTR **BYTE PTR** *expresión*

Ideal, MASM

Obliga a que una expresión sea del tamaño de un byte.

CODEPTR **CODEPTR** *expresión*

Ideal, MASM

Devuelve el atributo de direccionamiento por defecto.

DATAPTR **DATAPTR** *expresión*

Ideal

Obliga a la expresión a que tenga el atributo de direccionamiento correspondiente al modelo de memoria empleado.

DUP

cantidad **DUP** *(expresión[,expresión]...)*

Ideal, MASM

Repite la asignación de variable tantas veces como lo indique cantidad.

DWORD

DWORD *expresión*

Ideal

Obliga a la expresión a ser doble palabra.

DWORD PTR

DWORD PTR *expresión*

Ideal, MASM

Obliga a la expresión a ser doble palabra.

EQ

expresion1 **EQ** *expresion2*

Ideal, MASM

Devuelve verdad si la dos expresiones son iguales.

FAR

FAR *expresión*

Ideal

Obliga a una expresión a ser un apuntador FAR.

FAR PTR

FAR PTR *expresión*

Ideal, MASM

Obliga a una expresión a ser un apuntador FAR.

FWORD

FWORD *expresión*

Ideal

Obliga a una expresión a ser un apuntador de 32 bits.

FWORD PTR

FWORD PTR *expresión*

Ideal, MASM

Obliga a una expresión a ser un apuntador FAR.

GE

expresion1 **GE** *expresion2*

Ideal, MASM

Devuelve verdad si la primera expresión es mayor o igual que la segunda.

GT *expresion1* **GT** *expresion2*

Ideal, MASM

Devuelve verdad si la primera expresión es mayor que la segunda.

HIGH **HIGH** *expresión*

Ideal, MASM

Devuelve los 8 bits más significativos de la expresión.

HIGH *tipo* **HIGH** *expresión*

Ideal

Devuelve la parte alta de la expresión.

LARGE **LARGE** *expresión*

Ideal, MASM

Establece el desplazamiento de la expresión en 32 bits.

LE *expresion1* **LE** *expresion2*

Ideal, MASM

Devuelve verdad si la primera expresión es menor o igual que la segunda.

LENGTH **LENGTH** *nombre*

Ideal, MASM

Devuelve el número de elementos de datos asignados bajo el nombre especificado.

LOW **LOW** *expresion2*

Ideal, MASM

Devuelve los 8 bits menos significativos de la expresión.

LOW *tipo* **LOW** *expresión*

Ideal

Devuelve la parte baja de la expresión.

LT *expresion1* **LT** *expresion2*

Ideal, MASM

Devuelve verdad si la primera expresión es menor que la segunda.

MASK **MASK** *campo de registro*
MASK *campo*

Ideal, MASM

Devuelve la máscara de bits para el campo del registro o para el registro completo.

MOD *expresion1* **MOD** *expresion2*

Ideal, MASM

Calcula el residuo de la división entera entre las expresiones.

NE *expresion1* **NE** *expresion2*

Ideal, MASM

Devuelve verdad si la dos expresiones no son iguales.

NEAR **NEAR** *expresion1*

Ideal

Obliga a la expresión a ser un apuntador NEAR.

NEAR PTR **NEAR PTR** *expresión*

Ideal, MASM

Obliga a la expresión a ser un apuntador NEAR.

NOT **NOT** *expresión*

Ideal, MASM

Invierte la expresión.

OFFSET **OFFSET** *expresión*

Ideal, MASM

Devuelve el desplazamiento de la expresión dentro del segmento o grupo al cual pertenece.

OR *expresion1* **OR** *expresion2*

Ideal, MASM

Efectúa una operación lógica OR entre las dos expresiones.

PROC **PROC** *expresión*

Ideal

Obliga a una expresión a ser un apuntador a código.

PROC PTR

PROC PTR *expresión*

Ideal, MASM

Obliga a una expresión a ser un apuntador a código.

PTR

tipo **PTR** *expresión*

Ideal, MASM

Obliga a una expresión a tener el tamaño especificado por tipo.

PWORD

PWORD *expresión*

Ideal

Obliga a una expresión a ser un apuntador de 32 bits.

PWORD PTR

PWORD PTR *expresión*

Ideal, MASM

Obliga a una expresión a ser un apuntador de 32 bits.

QWORD

QWORD *expresión*

Ideal

Obliga a una expresión a ser cuádruple palabra.

QWORD PTR

QWORD PTR *expresión*

Ideal, MASM

Obliga a una expresión a ser cuádruple palabra.

SEG

SEG *expresión*

Ideal, MASM

Devuelve el segmento de la expresión que hace referencia a una posición de memoria.

SHL

expresión **SHL** *cuenta*

Ideal, MASM

Desplaza al valor de la expresión hacia la izquierda tantas veces como lo indique cuenta. Una cuenta negativa produce un desplazamiento hacia la derecha.

SHORT

SHORT *expresión*

Ideal, MASM

Obliga a la expresión a ser un apuntador corto (entre -128 y +127 bytes a partir de la posición actual).

SHR

expresión **SHR** *cuenta*

Ideal, MASM

Desplaza al valor de la expresión hacia la derecha tantas veces como lo indique cuenta. Una cuenta negativa produce un desplazamiento hacia la izquierda.

SIZE

SIZE *nombre*

Ideal, MASM

Devuelve el tamaño de la variable asociada al nombre especificado.

SMALL

SMALL *expresión*

Ideal, MASM

Establece el desplazamiento de la expresión e 16 bits.

SYMTYPE

SYMTYPE

Ideal

Devuelve información que describe la expresión.

TBYTE

TBYTE *expresión*

Ideal

Obliga a un expresión a ser de 10 bytes de longitud.

TBYTE PTR

TBYTE PTR *expresión*

Ideal, MASM

Obliga a un expresión a ser de 10 bytes de longitud.

THIS

THIS *tipo*

Ideal, MASM

Crea un operando cuya dirección es el segmento y desplazamiento actual.

.TYPE

.TYPE *expresión*

MASM

Devuelve una descripción de la expresión.

TYPE

TYPE *nombre1 nombre2*

Ideal

Aplica el tipo de un variable o miembro de estructura a otra variable o miembro de estructura.

TYPE

TYPE *expresión*

MASM

Devuelve un número que indica el tipo y tamaño de la expresión.

UNKNOWN

UNKNOWN *expresión*

Ideal

Remueve la información de tipo de una expresión de dirección.

WIDTH

WIDTH *campo de estructura*
WIDTH *estructura*

Ideal, MASM

Devuelve el número de bits de un campo de una estructura o de la estructura completa.

WORD

WORD *expresión*

Ideal

Obliga a un expresión a ser del tamaño de una palabra.

WORD PTR

WORD PTR *expresión*

Ideal, MASM

Obliga a un expresión a ser del tamaño de una palabra.

XOR

expresión1 **XOR** *expresión2*

Ideal, MASM

Calcula el OR exclusivo bit a bit entre las dos expresiones dadas.

&

&*nombre*

Ideal, MASM

Substituye el valor actual del parámetro nombre.

< >

<*texto***>**

Ideal, MASM

Procesa el texto literalmente.

! **!***carácter*

Ideal, MASM

Procesa el carácter dado literalmente.

% **%***texto*

Ideal, MASM

Evalúa a texto como una expresión y reemplaza a texto por el valor obtenido.

;; **;;***comentario*

Ideal, MASM

Suprime el almacenamiento del comentario en la definición del macro.

Apéndice C
Turbo Assembler: Directivas

.186
MASM

Habilita el ensamblaje de instrucciones del 80186.

.286
MASM

Habilita el ensamblaje de instrucciones del 80286 en modo real y del coprocesador matemático 80287.

.286C
MASM

Habilita el ensamblaje de instrucciones del 80286 en modo real y del coprocesador matemático 80287.

.286P
MASM

Habilita el ensamblaje de todas las instrucciones del 80286 (incluyendo modo protegido) y del coprocesador matemático 80287.

.287
MASM

Habilita el ensamblaje de instrucciones del coprocesador matemático 80287.

.386
MASM

Habilita el ensamblaje de instrucciones del 80386 en modo real y del coprocesador matemático 80387.

.386C
MASM

Habilita el ensamblaje de instrucciones del 80386 en modo real y del coprocesador matemático 80387.

.386P

MASM

Habilita el ensamblaje de todas las instrucciones del 80386 (incluyendo modo protegido) y del coprocesador matemático 80387.

.387

MASM

Habilita el ensamblaje de instrucciones del coprocesador matemático 80387.

.486

MASM

Habilita el ensamblaje de instrucciones para el i486 en modo real.

.486C

MASM

Habilita el ensamblaje de instrucciones para el i486 en modo real.

.486P

MASM

Habilita el ensamblaje de instrucciones para el i486 en modo protegido.

.8086

MASM

Habilita el ensamblaje de instrucciones para el 8086 solamente. Este es el modo por defecto.

.8087

MASM

Habilita el ensamblaje de instrucciones del coprocesador matemático 8087.

: *:nombre*

Ideal, MASM

Define una etiqueta con atributo NEAR.

= *nombre = expresión*

Ideal,MASM

Define o redefine una asignación numérica.

ALIGN

ALIGN *límite*

Ideal, MASM

Sitúa el apuntador de sentencias en la posición de memoria múltiplo de dos de acuerdo con *límite*, más cercana a la actual.

.ALPHA

MASM

Establece que el orden de los segmentos sea alfabético.

ARG

ARG *argumento[,argumento]...[= símbolo] [***RETURNS** *argumento [,argumento]]*

Ideal, MASM

Establece los argumentos de un procedimiento en la pila. A cada argumento se le asigna un desplazamiento positivo con respecto a BP, asumiendo que la dirección de retorno y el propio registro BP han sido previamente almacenados en la pila. Cada argumento tiene la siguiente sintaxis:

*argumento [***[***cantidad***]***] [:[tamaño][tipo] [:cantidad 2]]*

ASSUME

ASSUME *registro de segmento: nombre de segmento [,...]*
ASSUME *registro de segmento***:NOTHING**
ASSUME NOTHING

Ideal, MASM

Define el registro de segmento que será empleado para calcular la dirección efectiva de todos los símbolos declarados dentro de un segmento en particular. La palabra **NOTHING** cancela la asociación entre un registro de segmento y un segmento.

%BIN

%BIN *tamaño*

Ideal,MASM

Establece el número de columnas del campo de código objeto en el archivo de listado.

CATSTR

nombre **CATSTR** *cadena [,cadena]...*

Ideal, MASM51

Concatena varias cadenas en una sola.

.CODE

.CODE *[nombre]*

MASM

Define el inicio del segmento de código cuando es empleada la directiva **.MODEL**.

CODESEG

CODESEG *[nombre]*

Ideal,MASM

Define el inicio del segmento de código cuando es empleada la directiva **.MODEL**.

COMM

COMM *definición [,definición]...*

Ideal, MASM

Define una variable comunal. Cada definición tiene la siguiente sintaxis:

[atributo][lenguaje] nombre del símbolo [cantidad] :tipo [cantidad 2]

El *atributo* puede ser **NEAR** o **FAR**. Si el atributo es **NEAR**, el tamaño del arreglo viene definido por *cantidad*. Si el atributo es **FAR** *cantidad2* define el número de elementos de tamaño *cantidad*. El *lenguaje* define la convención de invocación de **C**, **PASCAL**, **BASIC**, **FORTRAN**, **NOLANGUAGE** o **PROLOG**. El *tipo* puede ser: **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE** o un nombre de estructura.

COMMENT

COMMENT *delimitador*
[texto]
delimitador

MASM

Inicia un comentario de más de una línea.

%COND

Ideal, MASM

Muestra en el listado todas las sentencias incluidas en bloques condicionales.

CONST

Ideal, MASM

Define el inicio del segmento de datos constantes.

.CREF

MASM

Establece que a partir de su aparición se recolecte información para referencia cruzada, de todos los símbolos encontrados.

%CREF

Ideal,MASM

Establece que a partir de su aparición se recolecte información para referencia cruzada, de todos los símbolos encontrados.

%CREFALL

Ideal,MASM

Establece que todos los símbolos en el archivo fuente, aparezcan en el listado de referencia cruzada. Este es el estado por defecto.

%CREFREF
Ideal,MASM

Deshabilita la aparición de símbolos no referenciados, en el listado de referencia cruzada.

%CREFUREF
Ideal,MASM

Muestra en el listado de referencia cruzada, sólo los símbolos no referenciados.

%CTLS
Ideal, MASM

Establece que las directivas de control de listado (tales como **%LIST**, **%INCL**, etc.) sean mostradas en el archivo de listado.

.DATA
MASM

Define el inicio del segmento de datos inicializados. Previamente debe ser usada la directiva **.MODEL** para especificar el modelo de memoria. El segmento es puesto en un grupo llamado **DGROUP** junto con los segmentos definidos por las directivas **.STACK**, **.CONST** y **.DATA?**.

DATASEG
Ideal

Ideal define el inicio del segmento de datos inicializados. Previamente debe ser usada la directiva **.MODEL** para especificar el modelo de memoria. El segmento es puesto en un grupo llamado **DGROUP** junto con los segmentos definidos por las directivas **.STACK**, **.CONST** y **.DATA?**.

.DATA?
MASM

Establece el inicio del segmento de datos no inicializados. Previamente debe usarse la directiva **.MODEL** para especificar el modelo de memoria. El segmento es puesto en un grupo llamado **DGROUP** junto con los segmentos definidos por las directivas **.STACK**, **.CONST** y **.DATA**.

DB *[nombre]* **DB** *expresión [,expresión]...*
Ideal, MASM

Reserva e inicializa un espacio de memoria de un byte de longitud. La expresión puede ser una constante, un signo de interrogación, una cadena de caracteres o una expresión de **DUP**licación.

DD *[nombre]* **DD** *[tipo* **PTR***] expresión [,expresión]...*
Ideal, MASM

Reserva e inicializa un espacio de memoria de cuatro bytes (doble palabra) de longitud. El *tipo* puede ser cualquiera de los siguientes: **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**,

TBYTE, **SHORT**, **NEAR**, **FAR** o un nombre de estructura. La *expresión* puede ser una constante, un número de 32 bits en punto flotante, un signo de interrogación, un apuntador o una expresión de **DUP**licación.

%DEPTH

%DEPTH *ancho*

Ideal,MASM

Establece el ancho en columnas del campo profundidad en el archivo de listado.

DF

[nombre] **DF** *[tipo* **PTR***] expresión [,expresión]...*

Ideal, MASM

Reserva e inicializa un espacio de memoria de seis bytes (apuntador de 48 bits) de longitud. El *tipo* puede ser cualquiera de los siguientes: **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, **SHORT**, **NEAR**, **FAR** o un nombre de estructura. La *expresión* puede ser una constante, un signo de interrogación, un apuntador o una expresión de **DUP**licación.

DISPLAY

DISPLAY *"texto"*

Ideal, MASM

Muestra el "*texto*" por pantalla.

DOSSEG

Ideal, MASM

Establece que el orden de los segmentos sea de acuerdo al DOS.

DP

[nombre] **DP** *[tipo* **PTR***] expresión [,expresión]...*

Ideal, MASM

Reserva e inicializa un espacio de memoria de seis bytes (apuntador de 48 bits) de longitud. El *tipo* puede ser cualquiera de los siguientes: **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, **SHORT**, **NEAR**, **FAR** o un nombre de estructura. La *expresión* puede ser una constante, un signo de interrogación, un apuntador o una expresión de **DUP**licación.

DQ

[nombre] **DQ** *expresión [,expresión]...*

Ideal, MASM

Reserva e inicializa un espacio de memoria de ocho bytes (cuádruple palabra) de longitud. La expresión puede ser una constante, un número en BCD empaquetado, un signo de interrogación, un número en punto flotante de 80 bits o una expresión de **DUP**licación.

DT

[nombre] **DT** *expresión [,expresión]...*

Ideal, MASM

Reserva e inicializa un espacio de memoria de diez bytes de longitud. La expresión puede ser una constante, un número en BCD empaquetado, un signo de interrogación, un número en punto flotante de 80 bits o una expresión de **DUP**licación.

DW

[nombre] **DW** *[tipo* **PTR***] expresión [,expresión]...*

Ideal, MASM

Reserva e inicializa un espacio de memoria de dos bytes (una palabra) de longitud. El *tipo* puede ser cualquiera de los siguientes: **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, **SHORT**, **NEAR**, **FAR** o un nombre de estructura. La *expresión* puede ser una constante, un signo de interrogación, un apuntador o una expresión de **DUP**licación.

ELSE

IF condición
    sentencias 1
[ELSE
    sentencias 2]
ENDIF

Ideal, MASM

Inicia un bloque alterno de compilación condicional.

ELSEIF

IF condición1
    sentencias 1
[ELSEIF *condición2*
    sentencias 2]
ENDIF

Ideal, MASM

Inicia un bloque anidado de compilación condicional. Otras opciones para **ELSEIF** son: **ELSEIF1**, **ELSEIF2**, **ELSEIFB**, **ELSEIFDEF**, **ELSEIFDIF**, **ELSEIFDIFI**, **ELSEIFE**, **ELSEIFIDN**, **ELSEIFIDNI**, **ELSEIFNB** y **ELSEIFNDEF**.

EMUL

Ideal, MASM

Ocasiona que toda las instrucciones para el coprocesador matemático sean generadas como instrucciones emuladas, en lugar de las reales.

END

END *[dirección de inicio]*

Ideal,MASM

Marca el final del archivo fuente y opcionalmente indica el punto de entrada al programa.

ENDIF

IF condición
    sentencias 1
ENDIF

Ideal, MASM

Marca el final de un bloque de compilación condicional.

ENDM

Ideal, MASM

Marca del final de un bloque de repetición o de un macro.

ENDP

ENDP *nombre de segmento* **ó** *nombre de estructura*
nombre de segmento **ó** *nombre de estructura* **ENDP**

Ideal, MASM

Marca el final del segmento, estructura o unión actual.

EQU

nombre **EQU** *expresión*

Ideal, MASM

Define a la etiqueta *nombre* como una cadena, alias o número resultante de la evaluación de la expresión.

.ERR

MASM

Obliga a que se genere un error en la compilación.

ERR

Ideal,MASM

Obliga a que se genere un error en la compilación.

.ERR1

MASM

Obliga a que se genere un error en el primer pase de la compilación.

.ERR2

MASM

Obliga a que se genere un error en el segundo pase de la compilación, si la compilación multi-pase ha sido habilitada.

.ERRB

.ERRB *argumento*

MASM

Obliga a la generación de un error en la compilación, si el argumento está en blanco.

.ERRDEF

.ERRDEF *símbolo*

MASM

Obliga a la generación de un error en la compilación, si el símbolo está definido.

.ERRDIF

.ERRDIF *argumento1,argumento2*

MASM

Obliga a la generación de un error en la compilación, si los argumentos son diferentes. La comparación diferencia mayúsculas de minúsculas.

.ERRDIFI

.ERRDIFI *argumento1,argumento2*

MASM

Obliga a la generación de un error en la compilación, si los argumentos son diferentes.

.ERRE

.ERRE *expresión*

MASM

Obliga a la generación de un error en la compilación, si la expresión es falsa (0).

.ERRIDN

.ERRIDN *argumento1,argumento2*

MASM

Obliga a la generación de un error en la compilación, si los argumentos son idénticos. La comparación diferencia mayúsculas de minúsculas.

.ERRIDNI

.ERRIDNI *argumento1,argumento2*

MASM

Obliga a la generación de un error en la compilación, si los argumentos son idénticos.

ERRIF

ERRIF *expresión*

Ideal,MASM

Obliga a la generación de un error en la compilación, si la expresión es verdadera (diferente de cero).

ERRIF1

Ideal,MASM

Obliga a la generación de un error en el primer pase de compilación.

ERRIF2

Ideal,MASM

Obliga a la generación de un error en el segundo pase de la compilación, si la compilación multi-pase ha sido habilitada.

ERRIFB

ERRIFB *argumento*

Ideal,MASM

Obliga a la generación de un error en la compilación, si el argumento está en blanco.

ERRIFDEF

ERRIFDEF *símbolo*

Ideal,MASM

Obliga a la generación de un error en la compilación, si el símbolo ha sido definido.

ERRIFDIF

ERRIFDIF *argumento1,argumento2*

Ideal,MASM

Obliga a la generación de un error en la compilación, si los argumentos son diferentes. La comparación diferencia mayúsculas de minúsculas.

ERRIFDIFI

ERRIFDIFI *argumento1,argumento2*

Ideal,MASM

Obliga a la generación de un error en la compilación, si los argumentos son diferentes.

ERRIFE

ERRIFE *expresión*

Ideal,MASM

Obliga a la generación de un error en la compilación, si la expresión es falsa (0).

ERRIFIDN

ERRIFIDN *argumento1,argumento2*

Ideal,MASM

Obliga a la generación de un error en la compilación, si los argumentos son idénticos. La comparación diferencia mayúsculas de minúsculas.

ERRIFIDNI

ERRIFIDNI *argumento1,argumento2*

Ideal,MASM

Obliga a la generación de un error en la compilación, si los argumentos son idénticos.

ERRIFNB

ERRIFNB *argumento*

Ideal,MASM

Obliga a la generación de un error en la compilación, si el argumento no está en blanco.

ERRIFNDEF

ERRIFNDEF *símbolo*

Ideal,MASM

Obliga a la generación de un error en la compilación, si el símbolo no ha sido definido.

.ERRNB

.ERRNB *argumento*

MASM

Obliga a la generación de un error en la compilación, si el argumento no está en blanco.

.ERRNDEF

.ERRNDEF *símbolo*

MASM

Obliga a la generación de un error en la compilación, si el símbolo no está definido.

.ERRNZ

.ERRNZ *expresión*

MASM

Obliga a la generación de un error en la compilación, si la expresión es verdadera (diferente de cero).

EVEN

Ideal,MASM

Ubica al apuntador de sentencias en la posición par más cercana a la actual.

EVENDATA

Ideal, MASM

Ubica al apuntador de sentencias en la posición par más cercana a la actual, en el segmento de datos.

EXITM

Ideal, MASM

Cancela la expansión de un macro o bloque de repetición y transfiere el control a la sentencia inmediata siguiente al macro o bloque de repetición.

EXTRN

EXTRN *definición [,definición]...*

Ideal,MASM

Indica que un símbolo determinado está definido en otro módulo fuente. La *definición* describe al símbolo y tiene el siguiente formato:

[lenguaje] nombre [cantidad] :tipo [:cantidad2]

El *lenguaje* define la convención de invocación de **C**, **PASCAL**, **BASIC**, **FORTRAN**, **NOLANGUAGE** o **PROLOG**. El *nombre* establece el identificador del símbolo y puede ir precedido de *cantidad* que permite definir un arreglo. El tipo puede ser: **NEAR**, **FAR**, **PROC**, **BYTE**, **WORD**, **DWORD**, **DATAPTR**, **CODEPTR**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, **ABS** o un nombre de estructura. La *cantidad2* define el número de símbolo externos definidos con las características anteriores.

.FARDATA

.FARDATA *[nombre de segmento]*

MASM

Define el inicio de un segmento FAR de datos inicializados.

FARDATA

FARDATA *[nombre de segmento]*

Ideal,MASM

Define el inicio de un segmento FAR de datos inicializados.

.FARDATA?

.FARDATA? *[nombre de segmento]*

MASM

Define el inicio de un segmento FAR de datos no inicializados.

GLOBAL

GLOBAL *definición [,definición]...*

Ideal,MASM

Actúa como una combinación de **EXTRN** y **PUBLIC** para definir un símbolo global. La *definición* tiene la siguiente sintaxis:

*[lenguaje] nombre [**[**cantidad**]**] : tipo [:cantidad2]*

El *lenguaje* define la convención de invocación de **C**, **PASCAL**, **BASIC**, **FORTRAN**, **NOLANGUAGE** o **PROLOG**. La *cantidad* define un arreglo de elementos. El *tipo* puede ser cualquiera de los siguientes: **NEAR**, **FAR**, **PROC**, **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, **ABS** o un nombre de estructura. La *cantidad2* establece el número de símbolos definidos.

GROUP

GROUP *grupo segmento [,segmento]*
grupo **GROUP** *segmento [,segmento]*

Ideal,MASM

Asocia un nombre de grupo a uno o más segmentos, de tal manera que todos lo símbolos definidos en dichos segmentos tengan su desplazamiento calculado con respecto al inicio del grupo.

IDEAL

Ideal,MASM

Activa el modo **Ideal**.

IF

IF *expresión*
    sentencias1
[ELSE
    sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si la *expresión* es verdadera, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IF1

IF1
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si se está ejecutando el pase uno de ensamblaje, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IF2

IF2
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si está habilitado el ensamblaje multi-pase, y se está ejecutando el pase dos, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IFB

IFB *argumento*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *argumento* está en blanco, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IFDEF

IFDEF *símbolo*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *símbolo* ha sido definido, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IFDIF

IFDIF *argumento1,argumento2*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *argumento1* es diferente del *argumento2*, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*. La comparación diferencia las letras mayúsculas de las minúsculas.

IFDIFI

IFDIFI *argumento1,argumento2*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *argumento1* es diferente del *argumento2*, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IFE

IFE *expresión*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si la *expresión* es falsa, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IFIDN

IFIDN *argumento1,argumento2*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *argumento1* es idéntico al *argumento2*, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*. La comparación diferencia las letras mayúsculas de las minúsculas.

IFIDNI

IFIDNI *argumento1,argumento2*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *arguemtno1* es igual al *argumento2*, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IFNB

IFNB *argumento*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *argumento* no está en blanco, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

IFNDEF

IFNDEF *símbolo*
sentencias1
[ELSE
sentencias2]
ENDIF

Ideal,MASM

Inicia un bloque condicional. Si el *símbolo* no está definido, serán ensambladas las *sentencias1.* En caso contrario son ensambladas las *sentencias2*.

%INCL

Ideal,MASM

Habilita el listado de los archivos de inclusión. Este es el estado por defecto.

INCLUDE

INCLUDE *archivo*
INCLUDE *"archivo"*

Ideal,MASM

Incluye el código fuente del archivo especificado a partir de la posición actual en el módulo que está siendo ensamblado.

INCLUDELIB

INCLUDELIB *archivo*
INCLUDELIB *"archivo"*

Ideal,MASM

Ocasiona la inclusión de la librería identificada por *archivo* en el proceso de enlace. Si no se especifica la extensión, se asume LIB.

INSTR

nombre **INSTR** *[inicio,] cadena1, cadena2*

Ideal,MASM

Asigna a *nombre* la posición de la primera aparición de *cadena1* en *cadena2*. La búsqueda comienza a partir de la posición indicada por *inicio* (o uno si inicio no es especificada). Si la *cadena1* no está presente en la *cadena2, nombre* es puesto a cero.

IRP

IRP *parámetro, arg1[,arg2]...*
sentencias
ENDM

Ideal,MASM

Repite un bloque de sentencias, con sustitución de cadena. Las sentencias son ensambladas, una por cada argumento presente. Cada vez que se ensambla un bloque, el próximo argumento es sustituido en cada aparición del *parámetro.*

IRPC

IRPC *parámetro, cadena*
sentencias
ENDM

Ideal,MASM

Repite un bloque de sentencias, con substitución de caracteres. Las *sentencias* son ensambladas tantas veces como caracteres haya en la cadena. Cada vez que un bloque es ensamblado, el siguiente argumento es sustituido en cada aparición del *parámetro* en las *sentencias*.

JUMPS

Ideal,MASM

Ocasiona que el ensamblador sustituya una instrucción de salto condicional por una con la condición complementaria y una instrucción **jmp**, si la dirección destino es muy lejana para un desplazamiento **SHORT**.

LABEL

nombre **LABEL** *tipo*
LABEL *nombre tipo*

Ideal,MASM

Define un símbolo especificado por *nombre* con el *tipo* indicado. El *tipo* puede ser: **NEAR**, **FAR**, **PROC**, **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE** o un nombre de estructura.

.LALL

MASM

Habilita el listado de las expansiones de los macros.

.LFCOND

MASM

Muestra en el listado, todas las sentencias en bloques condicionales.

%LINUM

%LINUM *tamaño*

Ideal,MASM

Establece el ancho del campo de números de línea. El valor por defecto es de cuatro columnas.

%LIST

Ideal,MASM

Muestra los archivos fuente en el listado. Este es el estado por defecto.

LOCAL

En macros:
LOCAL *símbolo [,símbolo]...*

En procedimientos:
LOCAL *elemento [,elemento]... [= símbolo]*

Ideal,MASM

Define variables locales para macros y procedimientos. Dentro de un macro, **LOCAL** define símbolos que serán reemplazados por nombres únicos en cada expansión del macro. **LOCAL** debe aparecer antes que cualquier otra sentencia.
Dentro de un procedimiento, **LOCAL** define accesos a localidades en la pila con desplazamientos negativos relativos al registro BP. Si la lista de argumento es terminada con el signo = y un *símbolo*, este será igualado al número de bytes que ocupa la lista de argumentos. Cada *elemento* tiene la siguiente sintaxis:

nombre [[cantidad]] [:[tamaño] [:tipo] [:cantidad2]]

La *cantidad* define el número de elementos del arreglo, por defecto igual a 1.El *tipo* puede ser: **BYTE**, **WORD**, **DATAPTR**, **CODEPTR**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, **NEAR**, **FAR**, **PROC** o un nombre de estructura. Si no se especifica el tipo, se asume **WORD**. La *cantidad2* establece el número de símbolos con las características anteriores.

LOCALS

LOCALS *[prefijo]*

Ideal,MASM

Habilita los símbolos locales, cuyos nombres comenzarán con los caracteres @@ o con los especificados en el *prefijo*.

MACRO

MACRO *nombre [parámetro [,parámetro]...]*
nombre **MACRO** *[parámetro [,parámetro]...]*

Ideal,MASM

Define un macro que será expandido cada vez que aparezca su *nombre*. Los *parámetros* son símbolos que pueden ser usados en el cuerpo del macro y que serán sustituidos por los valores actuales de los argumentos en cada expansión del macro.

%MACS

Ideal,MASM

Habilita el listado de las expansiones de las expansiones de los macros.

MASM

Ideal,MASM

Activa el modo MASM. Este es el modo por defecto.

MASM51

Ideal,MASM

Activa el modo MASM 5.1.

MODEL

MODEL *[modificador de modelo] modelo de memoria [nombre del módulo] [,[modificador de lenguaje] lenguaje] [,modificador de modelo]*

Ideal,MASM

Establece el modelo de memoria para las directivas de segmentación simplificadas. El *modificador de modelo* puede ser indicado al principio o al final de la sentencia y puede ser **NEARSTACK** o **FARSTACK**. Los *modelos de memoria* son: **TINY**, **SMALL**, **MEDIUM**, **COMPACT**, **LARGE**, **HUGE** y **TCHUGE**. El *nombre del módulo* es usado en los modelos grandes para especificar el nombre del segmento de código. El *modificador de lenguaje* puede ser **WINDOWS**, **ODDNEAR**, **ODDFAR** o **NORMAL** y especifica la generación del código de entrada y salida para MSWindows. El *lenguaje* especifica el lenguaje que invoca al procedimiento en este módulo y puede ser: **C**, **PASCAL**, **BASIC**, **FORTRAN**, **PROLOG** o **NOLANGUAGE**. El ensamblador generará automáticamente el código de entrada y salida apropiado al usarse las directivas **PROC** y **ENDP**. El *lenguaje* también instruye al ensamblador acerca de la convención de nombres a usar para símbolos externos y públicos, y el orden en que serán introducidos los argumentos en la pila al invocar un procedimiento. Adicionalmente se generará la versión apropiada de la instrucción **RET** para la remoción de los argumentos de la pila.

.MODEL

.MODEL *[modificador de modelo] modelo de memoria [nombre del módulo] [,[modificador de lenguaje] lenguaje] [,modificador de modelo]*

MASM

Establece el modelo de memoria para las directivas de segmentación simplificadas. El *modificador de modelo* puede ser indicado al principio o al final de la sentencia y puede ser **NEARSTACK** o **FARSTACK**. Los *modelos de memoria* son: **TINY**, **SMALL**, **MEDIUM**, **COMPACT**, **LARGE**, **HUGE** y **TCHUGE**. El *nombre del módulo* es usado en los modelos grandes para especificar el nombre del segmento de código. El *modificador de lenguaje* puede ser **WINDOWS**, **ODDNEAR**, **ODDFAR** o **NORMAL** y especifica la generación del código de entrada y salida para MSWindows. El *lenguaje* especifica el lenguaje que invoca al procedimiento en este módulo y puede ser: **C**, **PASCAL**, **BASIC**, **FORTRAN**, **PROLOG** o **NOLANGUAGE**. El ensamblador generará automáticamente el código de entrada y salida apropiado al usarse las directivas **PROC** y **ENDP**. El *lenguaje* también instruye al ensamblador acerca de la convención de nombres a usar para símbolos externos y públicos, y el orden en que serán introducidos los argumentos en la pila al invocar un procedimiento. Adicionalmente se generará la versión apropiada de la instrucción **RET** para la remoción de los argumentos de la pila.

MULTERRS

Ideal,MASM

Permite que puedan se reportados varios errores en un misma sentencia fuente.

NAME

NAME *nombre del módulo*

Ideal,MASM

Establece el nombre del módulo objeto. Puede ser usada en modo MASM pero sólo opera en modo Ideal.

%NEWPAGE

Ideal,MASM

Inicia una nueva página en el archivo de listado.

%NOCONDS

Ideal,MASM

Deshabilita la inclusión en el archivo de listado, de las sentencias localizadas en bloques condicionales.

%NOCREF

%NOCREF *[símbolo,...]*

Ideal,MASM

Deshabilita la recolección de información para referencia cruzada.

%NOCTLS

Ideal,MASM

Deshabilita la inclusión de las directivas de control de listado en el archivo de listado.

NOEMUL

Ideal,MASM

Obliga a que todas las instrucciones del coprocesador matemático sean generadas como instrucciones reales, en lugar de instrucciones emuladas.

%NOINCL

Ideal,MASM

Deshabilita el listado de las sentencias fuentes en archivos de inclusión.

NOJUMPS

Ideal,MASM

Deshabilita el ajuste de saltos condicionales establecido con la directiva **JUMPS**. Este es el estado por defecto.

%NOLIST

Ideal,MASM

Deshabilita el listado de sentencias fuentes.

NOLOCALS

Ideal,MASM

Deshabilita los símbolos locales establecidos con la directiva **LOCALS**. Este es el modo por defecto.

%NOMACS

Ideal,MASM

Especifica que sean listados solamente las expansiones de macros que produzcan código. Este es el modo por defecto.

NOMASM51

Ideal,MASM

Deshabilita el modo MASM 5.1. Este es el modo por defecto.

NOMULTERRS

Ideal,MASM

Permite el reporte de un sólo error por sentencia. Este es el modo por defecto.

NOSMART

Ideal,MASM

Deshabilita las optimizaciones que producen código diferente al MASM.

%NOSYMS

Ideal,MASM

Deshabilita la inclusión de la tabla de símbolos en el archivo de listado.

%NOTRUNC

Ideal,MASM

Impide el truncado de los campos cuyo contenido excede la longitud del campo correspondiente.

NOWARNS

NOWARNS *[clase]*

Ideal,MASM

Deshabilita los mensaje del tipo **warning**.

ORG

ORG *expresión*

Ideal,MASM

Color el apuntador de sentencias en el desplazamiento indicado por *expresión* dentro del segmento actual.

%OUT

%OUT *texto*

Ideal,MASM

Despliega el *texto* por pantalla.

P186

Ideal,MASM

Habilita el ensamblaje de instrucciones del 80186.

P286N

Ideal,MASM

Habilita el ensamblaje de instrucciones del 80286 en modo real y del coprocesador matemático 80287.

P286

Ideal,MASM

Habilita el ensamblaje de todas las instrucciones del 80286 (incluyendo modo protegido) y del coprocesador matemático 80287.

P286N

Ideal,MASM

Habilita el ensamblaje de instrucciones del 80286 en modo real y del coprocesador matemático 80287.

P286P

Ideal,MASM

Habilita el ensamblaje de todas las instrucciones del 80286 (incluyendo modo protegido) y del coprocesador matemático 80287.

P287

Ideal,MASM

Habilita el ensamblaje de instrucciones del coprocesador matemático 80287.

P386

Ideal,MASM

Habilita el ensamblaje de instrucciones del 80386 en modo real y del coprocesador matemático 80387.

P386N

Ideal,MASM

Habilita el ensamblaje de instrucciones del 80386 en modo real y del coprocesador matemático 80387.

P386P

Ideal,MASM

Habilita el ensamblaje de todas las instrucciones del 80386 (incluyendo modo protegido) y del coprocesador matemático 80387.

P387

Ideal,MASM

Habilita el ensamblaje de instrucciones del coprocesador matemático 80387.

P486

Ideal,MASM

Habilita el ensamblaje de instrucciones para el i486 en modo real.

P486N

Ideal,MASM

Habilita el ensamblaje de instrucciones para el i486 en modo real.

P8086

Ideal,MASM

Habilita el ensamblaje de instrucciones para el 8086 solamente. Este es el modo por defecto.

P8087

Ideal,MASM

Habilita el ensamblaje de instrucciones del coprocesador matemático 8087.

PAGE

PAGE *[filas] [,columnas]*

MASM

Establece el ancho y longitud de las páginas de listado y comienza un página nueva.

%PAGESIZE

%PAGESIZE *[filas] [;columnas]*

Ideal, MASM

Establece el ancho y longitud de las páginas de listado y comienza un página nueva.

%PCNT

%PCNT *ancho*

Ideal, MASM

Establece el ancho, en columnas del campo segmento:desplazamiento del archivo de listado. El valor por defecto es 4 para segmento de 16 bits y 8 para segmentos de 32 bits.

PNO87

Ideal, MASM

Impide el ensamblaje de instrucciones para el coprocesador matemático.

%POPLCTL

Ideal, MASM

Restablece las condiciones de listado tal y como estaban antes de la última ejecución de la directiva **%PUSHLCTL**.

PROC

PROC *[modificador de lenguaje] [lenguaje] nombre [atributo]*
[USES items,] [argumento [,argumento]...]
[RETURNS argumento [,argumento]...]
nombre **PROC** *[modificador de lenguaje][lenguaje] [atributo]*
[USES items,] [argumento [,argumento]...]
[RETURNS argumento [,argumento]...]

Ideal,MASM

Define el inicio de un procedimiento. El *modificador de lenguaje* puede ser **WINDOWS** o **NOWINDOWS**, para especificar la generación de código de entrada y salida para MSWindows. El *lenguaje* indica el lenguaje que invocará a dicho procedimiento y puede ser: **C**, **PASCAL**, **BASIC**, **FORTRAN**, **NOLANGUAGE** o **PROLOG**. Esto determina las convenciones de símbolos, el orden de los argumentos en la pila y si los argumentos son dejados o no en la pila cuando el procedimiento finaliza. El *atributo* puede ser **NEAR** o **FAR** y establece el tipo de instrucción **RET** a ser usada. Los *items* son una lista de registros y/o variables a ser salvados en la pila y recuperados antes de finalizar el procedimiento Los *argumento* describen los parámetros con que será invocado el procedimiento. Cada *argumento* tiene el formato siguiente:

*nombre [[cantidad]] [[:atributo] [**PTR**] tipo] [:cantidad2]*

El *nombre* es el identificador del argumento. La *cantidad* establece el tamaño del arreglo, por defecto igual a 1.El *atributo* puede ser **NEAR** o **FAR** para indicar que el argumento es un apuntador del nombre especificado. El *tipo* define el tipo del argumento y puede ser: **BYTE**, **WORD**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE** o un nombre de estructura. Si no se especifica, se asume **WORD**. La *cantidad2* establece el número de elementos de este tipo.

RETURNS introduce uno o más argumentos en la pila, que no serán extraídos cuando finalice el procedimiento.

PUBLIC

PUBLIC *[lenguaje] símbolo [,[lenguaje] símbolo]...*

Ideal, MASM

Establece que un símbolo pueda ser referido desde otro módulo fuente. Si se especifica el *lenguaje* (**C**, **PASCAL**, **FORTRAN**, **ASSEMBLER** o **PROLOG**), se le aplican al símbolo las convenciones correspondientes al mismo.

PUBLICDLL

PUBLICDLL *[lenguaje] símbolo [,[lenguaje] símbolo]...*

Ideal, MASM

Establece que un símbolo pueda ser referido como punto de entrada de enlace dinámico desde otro módulo. El *símbolo* (nombre de programa o procedimiento, nombre de variable o constante definida con **EQU**) podrá ser referido por otros módulos bajo OS/2. Si se especifica el *lenguaje* (**C**, **PASCAL**, **FORTRAN**, **ASSEMBLER** o **PROLOG**), se le aplican al símbolo las convenciones correspondientes al mismo.

PURGE

PURGE *nombre de macro [,nombre de macro]...*

Ideal, MASM

Elimina la definición del macro especificado.

%PUSHLCTL

Ideal, MASM

Salva el estado actual del control de listado en una pila de 16 niveles.

QUIRKS

Ideal, MASM

Permite que el Turbo Assembler emule errores de compilación existentes en Macro Assembler.

.RADIX

.RADIX *base numérica*

MASM

Establece la base numérica por defecto para constantes enteras.

RADIX

RADIX *base numérica*

Ideal, MASM

Establece la base numérica por defecto para constantes enteras.

RECORD

nombre **RECORD** *campo [,campo]...*
RECORD *nombre campo [,campo]...*

Ideal, MASM

Define una estructura que contiene *campos* de bits. Cada *campo* define un grupo de bits y tiene el siguiente formato:

nombre del campo: ancho [= expresión]

donde el *nombre del campo* es el nombre del campo de la estructura, y *ancho* establece el número de bits para dicho campo (1..16). Si el número total de bits en la estructura es menor o igual a 8, la estructura ocupará un byte. Si es mayor a 8 pero menor que 16, ocupara dos bytes y en cualquier otro caso, ocupará cuatro bytes.

REPT

REPT *expresión*
sentencias
ENDM

Ideal,MASM

Repite un bloque de *sentencias* tantas veces como lo indique *expresión*.

RETCODE

Ideal,MASM

Genera un instrucción **RET** con atributo **NEAR** o **FAR**, dependiendo del modelo de memoria establecido. Los modelos **TINY**, **SMALL** y **COMPACT** genera un retorno **NEAR**. Los modelos **MEDIUM**, **LARGE** y **HUGE** generan un retorno **FAR**.

RETF

Ideal,MASM

Genera una instrucción de retorno (**RET**) con atributo **FAR**.

RETN

Ideal,MASM

Genera una instrucción de retorno (**RET**) con atributo **NEAR**.

.SALL

Ideal,MASM

Suprime el listado de las todas las sentencias en expansiones de macros.

SEGMENT

SEGMENT *nombre [alineación] [combinación] [uso] ['clase']*
nombre **SEGMENT** *[alineación] [combinación] [uso] ['clase']*

Ideal,MASM

Define un segmento. Si previamente se ha definido otro segmento con el mismo nombre, este es considerado como continuación del anterior. La *alineación* establece el punto de arranque del segmento y puede ser: **BYTE**, **WORD**, **DWORD**, **PARA** (por defecto) o **PAGE**. La *combinación* especifica la manera como serán combinados segmentos con el mismo nombre en definidos en módulos diferentes y puede ser: **AT** *expresión* (localiza un segmento en una localidad absoluta indicada por *expresión*), **COMMON** (ubica al segmento, junto con todos los demás que tengan el mismo nombre, en la misma dirección de memoria), **MEMORY** (concatena todos los segmentos con el mismo nombre para formar un único segmento físico), **PRIVATE** (no combina al segmento con ningún otro; este es el estado por defecto), **PUBLIC** (igual que **MEMORY**), **STACK** (concatena todos los segmentos con el mismo nombre e inicializa el registro **SS** al comienzo del segmento y **SP** con la longitud del segmento) y **VIRTUAL** (define un tipo de segmento especial, que será tratado como un área común y agregado a otro segmento al momento del enlace). El *uso* define la longitud de palabra para el segmento, si está habilitada la generación de código **386** y puede ser: **USE16** o **USE32**. La *'clase'* el orden de los segmentos al momento del enlace: los segmentos con el mismo nombre son cargados juntos en memoria independientemente del orden en el cual hayan sido declarados en el archivo fuente.

.SEQ

MASM

Establece que el ordenamiento de los segmentos sea secuencial. Este es el modo por defecto.

.SFCOND

MASM

Impide la aparición en el listado, de sentencias asociadas a bloques condicionales falsos.

SIZESTR

nombre **SIZESTR** *cadena*

Ideal,MASM51

Asigna a *nombre* el número de caracteres presentes en la *cadena*. Una *cadena* vacía tiene una longitud de cero.

SMART

Ideal,MASM

Habilita las optimizaciones de código.

.STACK

.STACK *[tamaño]*

MASM

Define el comienzo del segmento de pila reservando el número de bytes especificado por *tamaño*. Si no se indica *tamaño*, se reservan 1024 bytes.

STACK

STACK *[tamaño]*

Ideal,MASM

Define el comienzo del segmento de pila reservando el número de bytes especificado por *tamaño*. Si no se indica *tamaño*, se reservan 1024 bytes.

STRUC

nombre **STRUC**
campos
[nombre] ENDS

STRUC *nombre*
campos
ENDS [nombres]

Ideal,MASM

Define una estructura. Cada campo usa las directivas normales para asignación de espacio de memoria (**DB**, **DW**, etc.) para definir su tamaño. Los *campos* pueden o no tener nombre. Los nombres de los campos deben ser únicos, si se trabaja en modo MASM.

SUBSTR

nombre **SUBSTR** *cadena, posición [,tamaño]*

Ideal,MASM51

Define una cadena llamada *nombre* con los caracteres de *cadena* a partir de *posición* y con la longitud especificada por *tamaño*. Si no se especifica *tamaño* se toman todos los caracteres hasta el final de la cadena.

SUBTTL

SUBTTL *texto*

MASM

Establece el subtítulo para el archivo de listado.

%SUBTTL

%SUBTTL *texto*

Ideal,MASM

Establece el subtítulo para el archivo de listado.

%SYMS

Ideal, MASM

Habilita la inclusión de la tabla de símbolos en el archivo de listado. Este es el modo por defecto.

%TABSIZE

%TABSIZE *ancho*

Ideal, MASM

Establece el ancho en columnas del tabulador, en el archivo de listado. El valor por defecto es 8.

%TEXT

%TEXT *ancho*

Ideal, MASM

Establece el ancho del campo de sentencias fuente en el archivo de listado.

.TFCOND

Ideal, MASM

Cambia el control de listado de sentencias sometidas a compilación condicional, al estado contrario al actual.

TITLE

TITLE *texto*

MASM

Establece el título del archivo de listado.

%TITLE

%TITLE *texto*

Ideal, MASM

Establece el título del archivo de listado.

%TRUNC

Ideal, MASM

Trunca las líneas de listados que sean demasiado largas.

UDATASEG

Ideal, MASM

Define el inicio de un segmento para variables no inicializadas.

UFARDATA

Ideal, MASM

Define el inicio de un segmento **FAR** para variables no inicializadas.

UNION

UNION *nombre*
campos
ENDS [nombre]

nombre **UNION**
campos
[nombre] ENDS

Ideal, MASM

Define una unión. Una unión es como una estructura, pero difiere en el hecho de que todos sus miembros están ubicados a partir del comienzo de la unión, de tal manera que todos los campos estén solapados, permitiendo la referencia a misma zona de memoria con diferentes nombres y tipos de datos. La longitud de la unión es igual al tamaño del campo más extenso. Cada *campo* usa las directivas convencionales para asignación de memoria (**DB**, **DW**, etc.).

USES

USES *item [,item]...*

Ideal, MASM

Establece los registros o variables simples que serán salvadas al comienzo de la ejecución de un procedimiento y recuperadas justo antes de que termine. Esta directiva debe ser usada antes de cualquier instrucción que genere código en el procedimiento.

WARN

WARN *[clase]*

Ideal, MASM

Habilita los mensaje de atención del tipo indicado por *clase*. Si no se especifica *clase* son habilitados todos los mensaje de atención. La *clase* puede ser: **ALN**, **ASS**, **BRK**, **ICG**, **LCO**, **OPI**, **OPP**, **OPS**, **OVF**, **PDC**, **PRO**, **PQK**, o **TPI**.

.XALL

MASM

Ocasiona que sean listadas sólo las expansiones de macros que produzcan código.

.XCREF

MASM

Deshabilita la generación de información para referencia cruzada.

.XLIST

MASM

Deshabilita el listado de las sentencias fuentes subsecuentes.

Apéndice D
Turbo Assembler: Opciones.

Opción	Acción
/a,/s	Ordenamiento de los segmentos alfabéticamente.
/c	Generar referencia cruzada en el listado.
/dSIM[=VAL]	Define el símbolo SIM = 0 o = al valor VAL
/e,/r	Habilita la generación de instrucciones de punto flotante reales o emuladas.
/h,/?	Muestra la pantalla de ayuda.
/iCAMINO	Especifica el camino para los archivos de inclusión. Estos archivos son especificados mediante la directiva INCLUDE.
/jCMD	Define un DIRECTIVA que será ensamblada antes de la primera línea del código fuente.
/kh#	Establece el máximo número de símbolos que puede contener el programa fuente.
/l,/la	Genera archivo de listado:l= Normal, la= expandido
/ml,/mx,/mu	Especifica al ensamblador que sean diferenciadas las letras minúsculas de las mayúsculas en los símbolos: ml= todos los símbolos, mx= sólo los globales, mu= ninguno.
/mv#	Establece la longitud máxima de los símbolos.
/m#	Permite la realización de múltiples pases en el proceso de ensamblaje, para la resolución de referencias hacia adelante.
/n	Suprime la generación de la tabla de símbolos en el archivo de listado.
/o,/op	Genera código para Overlay.
/p	Le indica al ensamblador, que señale la presencia de código que pueda presentar problemas en modo protegido.
/q	Suprime del archivo objeto, la información no requerida para el enlace.
/t	Suprime el mensaje de ensamblaje exitoso.
/w0, /w1, /w2	Establece el nivel de mensajes de advertencia: w0= ninguno, w1,w2= encendido.
/w-xxx, /w+xxx	Controla la generación de mensajes de advertencia permitiendo la habilitación o deshabilitación de casos específicos.
/x	Indica al ensamblador que incluye las sentencias condicionalmente falsas, en el archivo de listado.
/z	Indica al ensamblador que muestre junto con el error, la línea del archivo fuente donde se generó.
/zi, /zd	Permite la inclusión en el archivo objeto, de información para depuración: zi= toda, zd= sólo los números de línea

Apéndice E
Turbo Link: Opciones.

Opción	Acción
/3	Habilita el procesamiento de 32 bits.
/c	Diferencia entre mayúsculas y minúsculas al evaluar los símbolos.
/C	Diferencia entre mayúsculas y minúsculas para la sección de EXPORTS e IMPORTS.
/d	Muestra un mensaje de advertencia si encuentra símbolos duplicados en librerías.
/e	Ignora el diccionario extendido.
/i	Inicializa los segmentos.
/l	Incluye los números de línea.
/L	Especifica el paso o camino de las librerías.
/m	Crea un archivo mapa con los símbolos públicos.
/n	No usa las librerías por defecto.
/o	Activa overlays.
/P	Segmentos de código empaquetado.
/s	Crea mapas detallados de los segmentos.
/t	Genera archivo COM.
/Td	Crea un ejecutable para DOS.
/Tdc	Crea un ejecutable COM.
/Tde	Crea un ejecutable EXE.
/Tw	Crea un ejecutable para Windows (EXE o DLL).
/Twe	Crea una aplicación para Windows (EXE).
/Twd	Crea un DLL para Windows.
/v	Incluye toda la información simbólica para depuración.
/x	No crea el archivo mapa.
/ye	Usa memoria expandida.
/yx	Usa memoria extendida.

Apéndice F
Funciones del DOS

Int 20h
Termina el Proceso Actual.

Finaliza la ejecución del programa actual. Es uno de los métodos disponibles para que el programa en ejecución le transfiera el control al sistema operativo.

Parámetros de Entrada: CS = segmento del PSP

Datos de Salida: ninguno.

Observaciones: A partir de la versión 2.0 del MSDOS es preferible usar la Int 21h Función 4Ch.

Int 21h
Función 01h
Termina el Proceso Actual.

Finaliza la ejecución del programa actual. Es uno de los métodos disponibles para que el programa en ejecución le transfiera el control al sistema operativo.

Parámetros de Entrada: AH = 00h
CS = segmento del PSP.

Datos de Salida: ninguno.

Observaciones: A partir de la versión 2.0 del MSDOS es preferible usar la Int 21h Función 4Ch.

Int 21h
Función 01h
Lee un Carácter con Eco.

Lee un carácter del dispositivo estándar de entrada y lo envía al dispositivo estándar de salida. Si el carácter no está disponible, espera hasta que los esté. La entrada puede ser redireccionada.

Parámetros de Entrada: AH = 01h

Datos de Salida: AL = código ASCII del carácter leído.

Observaciones: Para la lectura de códigos ASCII extendido, la función debe ser ejecutada dos veces. En la primera invocación, un código ASCII 00h señala la presencia de un carácter extendido.

Int 21h
Función 02h
Escribe un Carácter.

Envía un carácter al dispositivo estándar de salida. La salida puede ser redireccionada.

Parámetros de Entrada: AH = 02h
DL = código ASCII del carácter a escribir

Datos de Salida: ninguno

Int 21h
Función 03h
Lee un Carácter del Dispositivo Auxiliar.

Lee un carácter del dispositivo estándar auxiliar. Por defecto este es el primer puerto serial (COM1).

Parámetros de Entrada: AH = 03h

Datos de Salida: AL = carácter leído.

Observaciones: Para la lectura de códigos ASCII extendido, la función debe ser ejecutada dos veces. En la primera invocación, un código ASCII 00h señala la presencia de un carácter extendido.

Int 21h
Función 04h
Escribe un Carácter en el Dispositivo Auxiliar.

Envía un carácter al dispositivo estándar auxiliar. Por defecto este es el primer puerto serial (COM1).

Parámetros de Entrada: AH = 04h
DL = carácter a escribir

Datos de Salida: ninguno

Int 21h
Función 05h
Escribe un Carácter en la Impresora.

Envía un carácter al dispositivo estándar de listado. Por dispositivo por defecto es el primer puerto paralelo (LPT1).

Parámetros de Entrada: AH = 05h
DL = carácter a escribir

Datos de Salida: ninguno

Int 21h
Función 06h
Manejo Directo de la Consola.

Permite leer o escribir del dispositivo estándar de entrada o del de salida, cualquier carácter existente, sin interferencia del sistema operativo.

Parámetros de Entrada: AH = 06h
DL = función:

00h-FEh	escritura
FFh	lectura

Datos de Salida: Si fue invocado con DL= 00h-0Fh
ninguno

Si DL= FFh y había un carácter disponible:
ZF = apagado
AL = código ASCII

en caso contrario:
ZF = encendido

Observaciones: Para la lectura de códigos ASCII extendido, la función debe ser ejecutada dos veces. En la primera invocación, un código ASCII 00h señala la presencia de un carácter extendido.

Int 21h
Función 07h
Lee Directamente un Carácter sin Eco.

Lee un carácter del dispositivo estándar de entrada sin que este sea filtrado por el sistema operativo. Sin no existe un carácter disponible, espera hasta que lo haya. La entrada puede ser redireccionada.

Parámetros de Entrada: AH = 07h

Datos de Salida: AL = carácter leído

Observaciones: Para la lectura de códigos ASCII extendido, la función debe ser ejecutada dos veces. En la primera invocación, un código ASCII 00h señala la presencia de un carácter extendido.

Int 21h
Función 08h
Lee un Carácter sin Eco.

Lee un carácter del dispositivo estándar de entrada. Sin no existe un carácter disponible, espera hasta que lo haya. La entrada puede ser redireccionada.

Parámetros de Entrada: AH = 08h

Datos de Salida: AL = carácter leído

Observaciones: Para la lectura de códigos ASCII extendido, la función debe ser ejecutada dos veces. En la primera invocación, un código ASCII 00h señala la presencia de un carácter extendido.

Int 21h
Función 09h
Escribe una Cadena de Caracteres.

Envía una cadena de caracteres al dispositivo estándar de salida. La salida puede ser redireccionada.

Parámetros de Entrada: AH = 09h
DS:DX = segmento:desplazamiento de la cadena de caracteres

Parámetros de Salida:ninguno

Observaciones: La cadena debe finalizar con el carácter $ (24h), el cual no es transmitido.

Int 21h
Función 0Ah
Lee una Cadena de Caracteres.

Lee una cadena de caracteres del dispositivo estándar de entrada, hasta e incluyendo el retorno de carro (0Dh), y la coloca en la localidad de memoria especificada. Lo caracteres son mostrados a través del dispositivo estándar de salida. La entrada puede ser redireccionada.

Parámetros de Entrada: AH = 0Ah
DS:DX = segmento:desplazamiento del *buffer* de memoria

Datos de Salida: ninguno (El *buffer* dado contiene la cadena leída).

Observaciones: El *buffer* de memoria empleado por esta función presenta el formato siguiente:

Byte	***Contenido***
0	Máximo # de caracteres a leer.
1	# de caracteres leídos, excluyendo el retorno de carro
2..	cadena leída del dispositivo estándar de entrada

Si el *buffer* es llenado con una cantidad de caracteres igual al máximo indicado menos uno, serán ignoradas las entradas siguientes y se producirá un sonido indicativo hasta que se detecte un carácter de retorno de carro.

Todos los comandos de edición están disponibles.

Int 21h
Función 0Bh
Estado del Dispositivo Estándar de Entrada.

Determina si existe un carácter disponible en el dispositivo estándar de entrada. La entrada puede ser redireccionada.

Parámetros de Entrada: AH = 0Bh

Datos de Salida: AL = estado del dispositivo:
00h = no hay carácter disponible
FFh = hay al menos un carácter disponible

Int 21h
Función 0Ch
Vacía el *buffer* de Lectura e Invoca a Función de Lectura.

Limpia el *buffer* del dispositivo estándar de entrada y luego invoca a alguna de las funciones de lectura. La entrada puede ser redireccionada.

Parámetros de Entrada: AH = 0Ch
AL = función de lectura a invocar (01h, 06h, 07h, 08h o 0Ah)

Si AL= 0Ah
DS:DX = segmento:desplazamiento del *buffer* de entrada

Datos de Salida: Si fue invocada con AL= 01h, 06h, 07h o 08h
AL = carácter leído

Si fue invocada con AL= 0Ah
ninguno.

Int 21h
Función 0Dh
Actualiza la Información en Disco.

Limpia todos los *buffer* de archivo internos. Toda la información temporalmente almacenada en *buffers* internos es físicamente escrita en disco.

Parámetros de Entrada: AH = 0Dh

Datos de Salida: ninguno

Int 21h
Función 0Eh
Selecciona una Unidad de Disco.

Establece la unidad de disco por defecto y devuelve la cantidad total de unidades lógicas instaladas.

Parámetros de Entrada: AH = 1Eh
DL = unidad de disco (0= A, 1= B, etc.)

Datos de Salida: Al = # de unidades lógicas en el sistema

Int 21h
Función 0Fh
Abre un Archivo (FCB).

Abre un archivo, haciéndolo disponible para futuras operaciones de lectura y escritura.

Parámetros de Entrada:
AH = 0Fh
DS:DX = segmento:desplazamiento del FCB

Datos de Salida:
Si la operación fue exitosa:
AL = 00h
(El FCB contiene información relacionada con la operación efectuada).

en caso contrario:
AL = FFh

Observaciones:
El FCB (*file control block*) presenta la siguiente estructura:

Byte	***Descripción***
00h	Unidad de disco (0= actual,1= A, 2= B, etc.)
01h-08h	Nombre del archivo
09h-0Bh	Extensión del archivo.
0Ch-0Dh	# del bloque actual
0Eh-0Fh	Tamaño del registro: (80h por defecto)
10h-13h	Tamaño del archivo
14h-15h	Fecha del archivo
16h-17h	Hora del archivo
18h-1Fh	Reservado
20h-	# del registro actual
21h-24h	# del registro aleatorio

A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 3Dh (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

Int 21h
Función 10h
Cierra un Archivo (FCB)

Cierra un archivo, limpia todos los *buffers* internos asociados con él y actualiza el directorio de disco.

Parámetros de Entrada:
AH = 10h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
Si la operación fue exitosa:
AL = 00h

en caso contrario:
AL = FFh

Observaciones:
A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 3Eh (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

Int 21h
Función 11h
Encuentra el Primer Archivo (FCB).

Busca en el directorio actual de la unidad de disco especificada, por un archivo que concuerde con las indicaciones.

Parámetros de Entrada:
AH = 11h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
Si la operación fue exitosa:
AL = 00h
(El FCB contiene información relacionada con la operación efectuada).

en caso contrario:
AL = FFh

Observaciones:
A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 4Eh (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

Int 21h
Función 12h
Encuentra el Siguiente Archivo (FCB).

Tras una previa invocación a la Int 21h Función 11h, encuentra el siguiente archivo que cumpla con las especificaciones.

Parámetros de Entrada:
AH = 12h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
Si la operación fue exitosa:
AL = 00h
(El FCB contiene información relacionada con la operación efectuada).

en caso contrario:
AL = FFh

Observaciones:
A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 4Fh (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

Int 21h
Función 13h
Borra uno o más Archivo.

Borra todos los archivos, en el directorio actual, que cumplan con las especificaciones.

Parámetros de Entrada:
AH = 13h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
Si la operación fue exitosa:
AL = 00h

en caso contrario:
AL = FFh

Observaciones: A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 41h (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

Int 21h
Función 14h
Lectura Secuencial de Datos (FCB).

Lee el próximo bloque de datos de un archivo y actualiza el apuntador del mismo.

Parámetros de Entrada:
AH = 14h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
AL = estado de la operación:
- 00h = operación exitosa
- 01h = fin de archivo
- 02h = final de segmento
- 03h = lectura parcial

Observaciones: A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 3Fh (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

La información es depositada en memoria en el área de transferencia de disco actual, establecido por la invocación más reciente a la Int 21h Función 1Ah.

Int 21h
Función 15h
Escritura Secuencial de Datos (FCB).

Escribe el siguiente bloque de información en un archivo y actualiza el apuntador del mismo.

Parámetros de Entrada:
AH = 15h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
AL = estado de la operación:
- 00h = operación exitosa
- 01h = disco lleno
- 02h = final de segmento

Observaciones: A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 40h (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

La información escrita es obtenida del área de transferencia de disco actual, establecido por la invocación más reciente a la Int 21h Función 1Ah.

Int 21h
Función 16h
Crea un Archivo (FCB).

Crea un archivo nuevo en el directorio actual, o trunca a cero un archivo existente con el mismo nombre. El archivo queda disponible para posteriores operaciones.

Parámetros de Entrada:
AH = 16h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
Si la operación fue exitosa:
AL = 00h

en caso contrario:
AL = FFh

Observaciones:
A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 3Ch (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

Int 21h
Función 17h
Renombra un Archivo (FCB).

Altera el nombre del archivo especificado, en el directorio actual.

Parámetros de Entrada:
AH = 17h
DS:DX = segmento:desplazamiento del FCB especial

Datos de Salida:
Si la operación fue exitosa:
AL = 00h

en caso contrario:
AL = FFh

Observaciones:
El FCB especial presenta la siguiente estructura:

Byte	***Descripción***
00h	Unidad de disco
01h-08h	Nombre original del archivo
09h-0Bh	Extensión original del archivo
11h-18h	Nuevo nombre del archivo
19h-1Bh	Nueva extensión del archivo

A partir de la versión 2.0 del MSDOS, es preferible el uso de la Int 21h, Función 56h (*handle*) la cual permite el uso de estructuras jerárquicas de directorios.

Int 21h
Función 19h
Determina la Unidad de Disco por Defecto.

Devuelve el código de la unidad de disco por defecto.

Parámetros de Entrada:
AH = 19h

Datos de Salida: AL = código de la unidad de disco (0= A, 1= B, etc.)

Int 21h
Función 1Ah
Establece el Area de Transferencia de Disco (DTA).

Especifica el área de transferencia de disco (DTA).

Parámetros de Entrada: AH = 1Ah
DS:DX = segmento:desplazamiento del área de transferencia de disco

Datos de Salida: ninguno

Observaciones: Si esta función no es invocada nunca, el área de transferencia de disco (DTA) por defecto se ubica a partir de la posición 80h en el PSP y tiene una longitud de 128 bytes.

Int 21h
Función 1Ch
Obtiene Información de la Unidad de Disco Actual.

Devuelve la información básica indispensable para determinar la capacidad del disco en la unidad de disco actual, contenida en la tabla de asignaciones de archivos (FAT).

Parámetros de Entrada: AH = 1Ch

Datos de Salida: Si la operación fue exitosa:
AL = sectores por pista
DS:BX = segmento:desplazamiento del código identificador de tipo
CX = tamaño en bytes de un sector físico
DX = # de pistas en el disco actual

en caso contrario:
AL = FFh

Observaciones: El código identificador de tipo tiene el siguiente significado:

Valor	***Significado***
F0h	3.5", doble cara, 18 sectores (1.44 Mb) o 3.5", doble cara, 36 sectores, (2.88 Mb) o 5.25", doble cara, 15 sectores, (1.2 Mb)
F8h	disco duro
F9h	5.25", doble densidad, 15 sectores, (720 Kb) o 3.5", doble densidad, 9 sectores, (1.2 Mb)
FAh	5.25", una cara, 8 sectores, (320 Kb)
FBh	3.5", doble cara, 8 sectores, (640 Kb)
FCh	5.25", simple cara, 9 sectores, 40 cilindros, (180 Kb)
FDh	5.25", doble densidad, 9 sectores, 40 pistas, (360Kb)
FEh	5.25", simple densidad, 8 sectores, (160 Kb)
FFh	5.25", doble densidad, 8 sectores, (320 Kb)

Int 21h
Función 21h
Lectura Aleatoria.

Lee el registro seleccionado de una archivo a memoria.

Parámetros de Entrada:
AH = 21h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
AL = estado de la operación:
- 00h = operación exitosa
- 01h = fin de archivo
- 02h = fin del segmento
- 03h = lectura parcial

Observaciones: La información es depositada en memoria en el área de transferencia de disco actual, establecido por la invocación más reciente a la Int 21h Función 1Ah.

Int 21h
Función 22h
Escritura Aleatoria.

Escribe el contenido de una porción de memoria en el registro especificado en un archivo.

Parámetros de Entrada:
AH = 22h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
AL = estado de la operación:
- 00h = operación exitosa
- 01h = disco lleno
- 02h = fin del segmento

Observaciones: La información escrita es obtenida del área de transferencia de disco actual, establecido por la invocación más reciente a la Int 21h Función 1Ah.

Int 21h
Función 23h
Obtiene el Tamaño de un Archivo (FCB).

Devuelve el tamaño del archivo especificado.

Parámetros de Entrada:
AH = 23h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida:
Si la operación fue exitosa:
AL = 00h
(El FCB contiene información relacionada con la operación efectuada).

en caso contrario:
AL = FFh

Int 21h
Función 24h
Establece el Número del Registro Relativo (FCB).

Hace que el número de registro relativo indicado en un FCB corresponda la posición actual del apuntador de archivo.

Parámetros de Entrada: AH = 24h
DS:DX = segmento:desplazamiento del FCB (ver Int 21h Función 0Fh)

Datos de Salida: ninguno
(El FCB contiene información relacionada con la operación efectuada).

Int 21h
Función 25h
Altera el Contenido de un Vector de Interrupción.

Altera, en forma segura, el contenido del vector de interrupción especificado.

Parámetros de Entrada: AH = 25h
AL = # de la interrupción
DS:DX = segmento:desplazamiento de la nueva rutina de servicio.

Datos de Salida: ninguno

Int 21h
Función 2Ah
Obtiene la Fecha del Sistema.

Obtiene el día de la semana, día del mes, mes y año del sistema.

Parámetros de Entrada: AH = 2Ah

Datos de Salida: CX = año (1980 hasta 2099)
DH = mes (1 al 12)
DL = día (1 al 31)
AL = día de la semana (0= domingo, 1= lunes, etc.)

Int 21h
Función 2Bh
Establece la Fecha del Sistema.

Inicializa el manejador del reloj del sistema con la fecha especificada.

Parámetros de Entrada: AH = 2Bh
CX = año (1989 hasta 2099)
DH = mes (1 al 12)
DL = día (1 al 31)

Datos de Salida: AL = estado de la operación
00h = operación exitosa
FFh = fecha no válida

Int 21h
Función 2Ch
Obtiene la Hora del Sistema.

Obtiene la hora del sistema en horas, minutos, segundos y centésimas de segundo.

Parámetros de Entrada: AH = 2Ch

Datos de Salida:
CH = hora (0 a 23)
CL = minutos (0 a 59)
DH = segundos (0 a 59)
DL = centésimas de segundo (0 a 99)

Int 21h
Función 2Dh
Establece la Hora del Sistema.

Inicializa el manejador del reloj del sistema con la hora especificada.

Parámetros de Entrada:
AH = 0Dh
CH = hora (0 a 23)
CL = minutos (0 a 59)
DH = segundos (0 a 59)
DL = centésimas de segundo (0 a 99)

Datos de Salida:
AL = estado de la operación
00h = operación exitosa
FFh = fecha no válida

Int 21h
Función 2Fh
Obtiene la Dirección del DTA Activo.

Devuelve la dirección el área de transferencia de disco activa.

Parámetros de Entrada: AH = 2Fh

Datos de Salida: ES:BX = segmento:desplazamiento del DTA.

Observaciones: El área de transferencia de disco (DTA) por defecto se ubica a partir de la posición 80h en el PSP y tiene una longitud de 128 bytes.

Int 21h
Función 30h
Obtiene la Versión del MS-DOS.

Devuelve la versión del DOS bajo el cual se está ejecutando el programa.

Parámetros de Entrada: AH = 30h

AL = 00h

Datos de Salida: Si esla versión 1.00
AL = 00h

Si es cualquier versión posterior a la 1.00
AL = parte entera de la versión
AH = parte fraccionaria de la versión
BH = serial del fabricante original del equipo
BL:CX = serial del fabricante (opcional)

Int 21h
Función 31h
Terminar y dejar Residente.

Termina la ejecución del programa actual, pasando un código de retorno al proceso invocador, pero reserva parte o toda la memoria ocupada por el mismo, de tal manera que no pueda ser sobre escrita por otra aplicación.

Parámetros de Entrada: AH = 31h
AL = código de retorno
DX = cantidad de memoria a reservar (en párrafos)

Datos de Salida: ninguno

Observaciones: Esta función debe ser usada en preferencia a la Int 27h, ya que permite el paso de un código de retorno y la reserva de cantidades de memoria mayores.

Int 21h
Función 35h
Obtiene el Contenido de un Vector de Interrupción.

Devuelve la dirección de la rutina de servicio asociada al vector de interrupción especificado.

Parámetros de Entrada: AH = 35h
AL = # del vector de interrupción

Datos de Salida: ES:BX = segmento:desplazamiento de la rutina de servicio.

Int 21h
Función 36h
Obtiene Información sobre la Asignación de Espacio en Disco.

Devuelve información sobre la unidad de disco especificada, que permite determinar la capacidad y espacio libre en la misma.

Parámetros de Entrada: AH = 36h
AL = código de la unidad (0= actual, 1= A, 2= B, etc.)

Datos de Salida: Si la operación fue exitosa:
AX = # de sectores por pista

BX = # de pistas disponibles
CX = # de bytes por sector
DX = # de pistas por disco

en caso de error:
AX = FFFFh

Int 21h
Función 39h
Crea un Directorio.

Crea un directorio usando la unida de disco y paso especificados.

Parámetros de Entrada: AH = 39h
DS:DX = segmento:desplazamiento del nombre del directorio (ASCIIZ)

Datos de Salida: Si la operación es exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error
03h = paso no encontrado
05h = acceso denegado

Int 21h
Función 3Ah
Borra un Directorio.

Remueve un directorio empleando la unidad y paso especificados.

Parámetros de Entrada: AH = 3Ah
DS:DX = segmento:desplazamiento del nombre del directorio (ASCIIZ)

Datos de Salida: Si la operación es exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error
03h = paso no encontrado
05h = acceso denegado
06h = directorio actual
10h = directorio actual

Int 21h
Función 3Bh
Establece el Directorio Actual.

Establece el directorio actual, empleando la unidad y paso especificados.

Parámetros de Entrada: AH = 3Bh

DS:DX = segmento:desplazamiento del nombre del directorio (ASCIIZ)

Datos de Salida: Si la operación es exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error
03h = paso no encontrado

Int 21h
Función 3Ch
Crea un Archivo (*handle*).

Dada una cadena ASCIIZ, crea un archivo en la unidad y paso especificados o actuales. Si el archivo especificado ya existe, es truncado a cero bytes. En cualquier caso, devuelve un manejador (*handle*) que permite la manipulación del archivo.

Parámetros de Entrada:
AH = 3Ch
CX = atributo del archivo
DS:DX = segmento:desplazamiento del nombre del archivo (ASCIIZ)

Datos de Salida: Si la operación fue exitosa:
CF = apagado
AX = manejador (*handle*)

en caso de error:
CF = encendido
AX = código de error:
03h = paso no encontrado
04h = manejador no disponible
05h = acceso denegado

Observaciones: El atributo del archivo tiene la siguiente estructura:

Bit(s)	***Significado***
0	Sólo lectura
1	Escondido
2	sistema
3	Etiqueta de disco
4	Reservado (0)
5	Archivo
6-15	Reservado (0)

Int 21h
Función 3Dh
Abre un Archivo (*handle*).

Dada un cadena ASCIIZ, abre el archivo especificado en la unidad y pasos especificados o actuales. Devuelve un manejador que puede ser usado para operaciones posteriores con el archivo.

Parámetros de Entrada:
AH = 3Dh
AL = modo de acceso
DS:DX = segmento:desplazamiento de la cadena ASCIIZ

Datos de Salida: Si la operación fue exitosa:
CF = apagado
AX = manejador (*handle*)

en caso de error:
CF = encendido
AX = código de error:
01h = función inválida
02h = archivo no encontrado
03h = paso no encontrado
04h = manejador no disponible
05h = acceso denegado
0Ch = modo de acceso inválido

Observaciones: El modo de acceso presenta la estructura siguiente:

Bit(s)	***Significado***
0-2	000 = sólo lectura
	001 = sólo escritura
	010 = lectura/escritura
3	Reservado (0)
4-6	000 = modo de compatibilidad
	001 = denegada toda operación
	010 = no escribir
	011 = no lectura
	100 = permitida cualquier operación
7	0 = el proceso ejecutado hereda el manejador
	1 = el proceso ejecutado no hereda el manejador

Int 21h
Función 3Eh
Cierra un Archivo (*handle*).

Dado un manejador obtenido de una operación exitosa previa de apertura o creación de archivo, limpia todos los *buffers* internos asociados, cierra el archivo y libera el manejador. Si el archivo fue modificado, se actualizan la fecha y hora del archivo.

Parámetros de Entrada: AH = 3Eh
BX = manejador (*handle*)

Datos de Salida: Si la operación fue exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error:
06h = manejador inválido

Int 21h
Función 3Fh
Lee un Archivo un Dispositivo (*handle*).

Dado un manejador obtenido de una operación exitosa previa de apertura o creación de archivo, transfiere el número de bytes indicados desde el archivo hasta la localidad de memoria especificada y actualiza el apuntador de archivo.

Parámetros de Entrada:
AH = 3Fh
BX = manejador (*handle*).
CX = # de bytes a leer
DS:DX = segmento:desplazamiento del *buffer* de memoria

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
AX = # de bytes transferidos

en caso de error:
CF = encendido
AX = código de error:
05h = acceso denegado
06h = manejador inválido

Int 21h
Función 40h
Escribe en un Archivo o Dispositivo (*handle*).

Dado un manejador obtenido de una operación exitosa previa de apertura o creación de archivo, escribe el número de bytes indicados de la localidad de memoria especificada al archivo y actualiza el apuntador de archivo.

Parámetros de Entrada:
AH = 40h
BX = manejador (*handle*).
CX = # de bytes a escribir
DS:DX = segmento:desplazamiento del *buffer* de memoria

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
AX = # de bytes transferidos

en caso de error:
CF = encendido
AX = código de error:
05h = acceso denegado
06h = manejador inválido

Int 21h
Función 41h
Borra un Archivo.

Borra un archivo en la unidad y directorio especificados o actuales.

Parámetros de Entrada:
AH = 41h
DS:DX = segmento:desplazamiento del nombre del archivo (ASCIIZ)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error:

02h = archivo no encontrado
05h = acceso denegado

Int 21h
Función 42h
Establece la Posición del Apuntador de Archivo (*handle*).

Establece la posición del apuntador de archivo relativo al inicio, final o posición actual.

Parámetros de Entrada:

AH = 42h
AL = método:
- 00h = relativo al inicio del archivo
- 01h = relativo a la posición actual del apuntador
- 02h = relativo al final del archivo

BX = manejador (*handle*).
CX = 16 bits más significativos del desplazamiento
DX = 16 bits menos significativos del desplazamiento

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
DX = 16 bits más significativos de la posición actual del apuntador
AX = 16 bits menos significativos de la posición actual del apuntador

en caso de error:
CF = encendido
AX = código de error:
- 01h = función inválida
- 06h = manejador inválido

Observaciones: El método 02h puede ser usado para determinar la longitud de un archivo invocando a la función con un desplazamiento 0 y examinando la posición del apuntador devuelta.

Int 21h
Función 43h
Determina o Establece el Atributo de un Archivo .

Obtiene o altera el atributo de un archivo o directorio.

Parámetros de Entrada:

AX = 43h
AL = modo
- 00h = obtener atributo
- 01h = alterar atributo

CX = atributo del archivo (si AL= 01h) (Ver Int 21h Función 3Ch)
DS:DX = segmento:desplazamiento del nombre del archivo (ASCIIZ)
BX = desplazamiento en X del punto activo
CX = desplazamiento en Y del punto activo
ES:DX = segmento:desplazamiento del buffer que contiene la imagen

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
CX = atributo del archivo (Ver Int 21h Función 3Ch)

en caso de error:
CF = encendido
AX = código de error:

01h = función inválida
02h = archivo no encontrado
03h = paso no encontrado
05h = acceso denegado

Int 21h
Función 45h
Duplica un Manejador (*handle*).

Dado un manejador de archivo o dispositivo, devuelve otro manejador que hace referencia al mismo archivo o dispositivo.

Parámetros de Entrada:
AH = 45h
BX = manejador (*handle*)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
AX = nuevo manejador (*handle*)

en caso de error:
CF = encendido
AX = código de error:
04h = manejador no disponible
06h = manejador inválido

Int 21h
Función 46h
Redirecciona un Manejador (*handle*).

Dados dos manejadores válidos, hace que el segundo manejador hagan referencia al mismo dispositivo o archivo que el primero.

Parámetros de Entrada:
AH = 46h
BX = manejador (*handle*)
CX = manejador a ser redireccionado (*handle*)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error:
04h = manejador no disponible
06h = manejador inválido

Int 21h
Función 47h
Obtiene el Directorio Actual.

Devuelve una cadena ASCIIZ que describe el paso y el nombre del directorio actual.

Parámetros de Entrada:
AH = 47h

DL = código de la unidad (0= actual, 1= A, 2= B, etc.)
DS:SI = segmento:desplazamiento del *buffer* de 64 bytes

Datos de Salida: Si la operación fue exitosa:
CF = apagado
(el *buffer* contiene el paso y nombre del directorio actual)

en caso de error:
CF = encendido
AX = código de error:
0Fh = unidad de disco inválida

Int 21h
Función 48h
Reserva un Bloque de Memoria.

Reserva un bloque de memoria y devuelve un apuntador al inicio del mismo.

Parámetros de Entrada: AH = 48h
BX = tamaño del bloque en párrafos

Datos de Salida: Si la operación fue exitosa:
CF = apagado
AX = segmento base del bloque de memoria

en caso de error:
CF = encendido
AX = código de error:
07h = bloque de control de memoria destruido
08h = memoria insuficiente
BX = tamaño del bloque más grande disponible

Int 21h
Función 49h
Libera un Bloque de Memoria.

Libera un bloque de memoria, asignado dinámicamente.

Parámetros de Entrada: AH = 49h
ES = segmento base del bloque de memoria

Datos de Salida: Si la operación fue exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error:
07h = bloque de control de memoria destruido
09h = dirección del bloque de memoria inválida

Int 21h
Función 4Ah
Modifica el Tamaño de un Bloque de Memoria.

Acorta o alarga dinámicamente, el tamaño de un bloque de memoria.

Parámetros de Entrada:
AH = 4Ah
BX = nuevo tamaño del bloque de memoria
ES = segmento base del bloque de memoria

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error:
CF = encendido
AX = código de error:
- 07h = bloque de control de memoria destruido
- 08h = memoria insuficiente
- 09h = dirección del bloque de memoria inválida

BX = máximo bloque de memoria disponible

Int 21h
Función 4Ch
Termina el Proceso Actual y Devuelve un Código.

Termina el proceso actual, pasando un código de retorno al programa invocador.

Parámetros de Entrada:
AH = 4Ch
AL = código de retorno

Datos de Salida:
ninguno

Observaciones:
Este el método más recomendable para la finalización de procesos ya que permite la devolución de un código al programa invocador.

Int 21h
Función 4Dh
Obtiene el Código de Retorno.

Permite que el programa invocador obtenga el código de retorno devuelto por el programa invocado al finalizar, por medio de la Int 21h Función 4Ch.

Parámetros de Entrada:
AH = 4Dh

Datos de Salida:
Ah = causa de la terminación del proceso:
- 00h = normal
- 01h = Ctrl C
- 02h = error crítico
- 03h = programa residente

Al = código de retorno

Int 21h
Función 4Eh
Encuentra el Primer Archivo.

Dada una especificación de archivo en la forma de una cadena ASCIIZ, busca en la unidad y directorio especificados o actuales, el primer archivo que las cumpla.

Parámetros de Entrada:

AH = 4Eh
CX = atributo (Ver Int 21h Función 3Ch)
DS:DX = segmento:desplazamiento de la cadena ASCIIZ

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
y los resultados de la búsqueda quedan en el DTA activo con la siguiente estructura:

Byte(s)	***Descripción***
00h-14h	Reservado (0)
15h	Atributo del archivo encontrado
16h-17h	Hora del archivo
	bits 00h-04h = incrementos de 2 segundos (0 al 29)
	bits 05h-0Ah = minutos (0 al 59)
	bits 0Bh-0Fh = horas (0 al 23)
18h-19h	Fecha del archivo
	bits 00h-04h = día (1 al 31)
	bits 05h-08h = mes (1 al 12)
	bits 09h-0Fh = año (relativo a 1980)
1Ah-1Dh	tamaño del archivo
1Eh-2Ah	cadena ASCIIZ con el nombre y extensión del archivo

en caso de error:
CF = encendido
AX = código de error:
02h = archivo no encontrado
03h = paso inválido
12h = no más archivos

Int 21h
Función 4Fh
Encuentra el Siguiente Archivo.

Asumiendo un operación exitosa previa de la Int 21h Función 4Eh, encuentra el siguiente archivo que cumpla con las especificaciones dadas.

Parámetros de Entrada:

AH = 4Fh

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
y los resultados de la búsqueda quedan en el DTA activo.

en caso de error.
CF = encendido
AX = código de error:
12h = no más archivos

Int 21h
Función 56h
Renombra un Archivo.

Renombra un archivo y/o lo mueve a otro directorio en la misma unidad.

Parámetros de Entrada:
AH = 56h
DS:DX = segmento:desplazamiento del nombre y paso original (ASCIIZ)
ES:DI = segmento:desplazamiento del nuevo nombre y paso (ASCIIZ)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error.
CF = encendido
AX = código de error:
- 02h = archivo no encontrado
- 03h = paso no encontrado
- 05h = acceso denegado
- 11h = unidades diferentes

Int 21h
Función 57h
Obtiene o Establece la Fecha y Hora de un Archivo (*handle*).

Obtiene o modifica la fecha y hora de un archivo.

Parámetros de Entrada:
AH = 57h
AL = operación:
- 00h = modificar fecha y hora
- 01h = obtener fecha y hora

BX = manejador (*handle*)

Si AL= 00h
CX = hora
- bits 00h-04h = incrementos de 2 segundos (0 al 29)
- bits 05h-0Ah = minutos (0 al 59)
- bits 0Bh-0Fh = horas (0 al 23)

DX = fecha
- bits 00h-04h = día (1 al 31)
- bits 05h-08h = mes (1 al 12)
- bits 09h-0Fh = año (relativo a 1980)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
CX = hora
DX = fecha

en caso de error.
CF = encendido
AX = código de error:
- 00h = función inválida
- 06h = manejador inválido

Int 21h
Función 5Ah
Crea un Archivo Temporal (*handle*).

Crea un archivo con nombre único en la unidad y directorio especificados o actuales, y devuelve un manejador que podrá ser usado para subsecuentes accesos al archivo. El nombre del archivo es retornado en el *buffer* especificado.

Parámetros de Entrada:
AH = 5Ah
CX = atributo (Ver Int 21h Función 3Ch)
DS:DX = segmento:desplazamiento del paso (ASCIIZ)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
AX = manejador (*handle*)
DS:DX = segmento:desplazamiento del nombre y paso del archivo

en caso de error.
CF = encendido
AX = código de error:
- 03h = paso no encontrado
- 04h = manejador no disponible
- 05h = acceso denegado

Int 21h
Función 5Bh
Crea un Archivo Nuevo (*handle*).

Dada un cadena ASCIIZ, crea un archivo en la unidad y directorio especificados o actuales, y devuelve una manejador que podrá ser usado para subsecuentes operaciones con el archivo. Si existe previamente un archivo con el mismo nombre, la función falla.

Parámetros de Entrada:
AH = 5Bh
CX = atributo (Ver Int 21h Función 3Ch)
DS:DX = segmento:desplazamiento de la cadena ASCIIZ

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
AX = manejador (*handle*)

en caso de error.
CF = encendido
AX = código de error:
- 03h = paso no encontrado
- 04h = manejador no disponible
- 05h = acceso denegado
- 50h = el archivo ya existe

Int 21h
Función 62h
Obtiene la Dirección del PSP.

Obtiene la dirección del segmento del PSP del programa en ejecución.

Parámetros de Entrada: AH = 62h

Datos de Salida: BX = segmento del PSP.

Int 21h
Función 68h
Vacía los *buffers* de Archivo (*handle*).

Obliga a que todos los *buffers* asociados con un archivo sean físicamente escritor en disco. Si el manejador se refiere a un archivo y el mismo ha sido modificado, serán actualizados la fecha y hora del mismo.

Parámetros de Entrada:
AH = 68h
BX = manejador (*handle*)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error.
CF = encendido
AX = código de error:
06h = manejador inválido

Int 25h
Lectura Absoluta de Disco.

Transfiere el contenido de uno o más sectores del disco al *buffer* especificado, accesando en forma directa los sectores lógicos del mismo. El retornar la función, se encuentra en la pila el valor del registro de estado.

Parámetros de Entrada:
AL = unidad de disco (0= A, 1= B, etc.)
CX = # de sectores a leer
DX = primer sector a ser leído
DS:BX = segmento:desplazamiento del *buffer* de memoria

Para acceso a particiones mayores de 32 Mb (MSDOS 4.00 en adelante)
AL = unidad de disco (0= A, 1= B, etc.)
CX = -1
DS:BX = segmento:desplazamiento del bloque de parámetros

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error.
CF = encendido
AX = código de error:
01h = comando erróneo
02h = marca de dirección errónea
03h = disco protegido contra escritura
04h = sector no encontrado
08h = error de DMA
10h = error de CRC
20h = falla en la controladora de disco
40h = falló la operación de búsqueda
80h = el dispositivo no responde

Observaciones: Todos los registros de segmento son destruidos.

Cuando la función retorna se encuentra en la pila el valor del registro de estado al ser invocada la interrupción.

El bloque de parámetros, empleado para tener acceso a particiones mayores de 32 Mb tiene la siguiente estructura:

Bytes	***Descripción***
00h-03h	# del sector (32 bits)
04h-05h	# de sectores a ser transferidos
06h-07h	Desplazamiento del *buffer* de memoria
08h-09h	Segmento del *buffer* de memoria

Int 26h
Escritura Absoluta a Disco.

Transfiere el contenido del *buffer* especificado a uno o más sectores de disco, accesando en forma directa los sectores lógicos del mismo. El retornar la función, se encuentra en la pila el valor del registro de estado.

Parámetros de Entrada:
AL = unidad de disco (0= A, 1= B, etc.)
CX = # de sectores a leer
DX = primer sector a ser leído
DS:BX = segmento:desplazamiento del *buffer* de memoria

Para acceso a particiones mayores de 32 Mb (MSDOS 4.00 en adelante)
AL = unidad de disco (0= A, 1= B, etc.)
CX = -1
DS:BX = segmento:desplazamiento del bloque de parámetros

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error.
CF = encendido
AX = código de error:
- 01h = comando erróneo
- 02h = marca de dirección errónea
- 03h = disco protegido contra escritura
- 04h = sector no encontrado
- 08h = error de DMA
- 10h = error de CRC
- 20h = falla en la controladora de disco
- 40h = falló la operación de búsqueda
- 80h = el dispositivo no responde

Observaciones: Todos los registros de segmento son destruidos.

Cuando la función retorna se encuentra en la pila el valor del registro de estado al ser invocada la interrupción.

El bloque de parámetros, empleado para tener acceso a particiones mayores de 32 Mb tiene la siguiente estructura:

Bytes	***Descripción***
00h-03h	# del sector (32 bits)
04h-05h	# de sectores a ser transferidos
06h-07h	Desplazamiento del *buffer* de memoria
08h-09h	Segmento del *buffer* de memoria

Int 27h
Terminar y dejar Residente.

Termina la ejecución del programa actual, pero reserva parte o toda la memoria ocupada por el mismo, de tal manera que no pueda ser sobre escrita por otra aplicación.

Parámetros de Entrada: DX = desplazamiento del último byte mas 1 (relativo al PSP) de programa.

CS = dirección del segmento del PSP

Datos de Salida: ninguno

Observaciones: Es preferible el uso de la Int 21h Función 31h, ya que este permite la devolución de un código de retorno al programa invocador.

Int 10h
Función 00h
Selecciona el Modo de Video activo.

Selecciona el modo de video activo. Además selecciona la controladora de video activa, si existe más de una.

Parámetros de Entrada: AH = 00h
AL = modo de video

Datos de Salida: ninguno.

Observaciones: Los modos de video aplicables se muestran en la siguiente tabla:

Modo	Resolución	Colores	Texto/ Gráfico	MDA	CGA	PCjr	EGA	MCGA	VGA
00H	40x25	16	Texto		X	X	X	X	X
01H	40x25	16	Texto		X	X	X	X	X
02H	80x25	16	Texto		X	X	X	X	X
03H	80x25	16	Texto		X	X	X	X	X
04H	320x200	4	Gráfico		X	X	X	X	X
05H	320x200	4	Gráfico		X	X	X	X	X
06H	640x200	2	Gráfico		X	X	X	X	X
07H	80x25	2	Texto	X		X	X		X
08H	160x200	16	Gráfico			X			
09H	320x200	16	Gráfico			X			
0AH	640x200	4	Gráfico			X			
0BH	Reservado								
0CH	Reservado								
0DH	320x200	16	Gráfico				X		X
0EH	640x200	16	Gráfico				X		X
0FH	640x350	2	Gráfico				X		X
10H	640x350	4	Gráfico				X		X
10H	640x350	16	Gráfico				128Kb		X
11H	640x480	2	Gráfico					X	X
12H	640x480	16	Gráfico						X
13H	320x200	256	Gráfico					X	X

Tabla G-1

Int 10h
Función 01h
Selecciona el Tipo de Cursor

Establece el tamaño del cursor en modo texto.

Parámetros de Entrada: AH = 01h
CH = línea de comienzo
CL + línea final

Datos de Salida: ninguno.

Observaciones: El valor por defecto establecido en el ROM BIOS es:

Modo	*Inicio*	*Final*
07h	11	12
00h-03h	6	7

El cursor puede ser apagado, invocando la función con CH=20h.

Int 10h
Función 02h
Selecciona la posición del cursor.

Selecciona la posición del cursor en modo texto.

Parámetros de Entrada: AH = 02h
BH = página de video
DH = fila
DL = columna

Datos de Salida: ninguno

Int 10h
Función 03h
Obtiene la posición del cursor.

Obtiene la posición actual del cursor en modo texto.

Parámetros de Entrada: AH = 03h
BH = página de video

Datos de Salida: CH = línea de comienzo del cursor
CL = línea final
DH = fila
DL = columna

Int 10h
Función 05h
Selecciona la página de video activa.

Selecciona la página de video activa.

Parámetros de Entrada: AH = 05h
AL = página de video

Datos de Salida: ninguno

Observaciones El número de páginas está determinado por el modo de video:

Modo	*Páginas*
00h-01h	0-7
02h-03h	0-3 (CGA)
02h-03h	0-7 (EGA,MCGA,VGA)
07h	0-7 (EGA,VGA)
0Dh	0-7 (EGA,VGA)
0Eh	0-3 (EGA,VGA)
0Fh	0-1 (EGA,VGA)
10h	0-1 (EGA,VGA)

Int 10h
Función 06h
Desplaza la Pantalla hacia arriba.

Inicializa la ventana especificada con espacios en blanco en el color especificado o desplaza hacia arriba el contenido de una ventana, el número de líneas indicado.

Parámetros de Entrada:
AH = 06h
AL = # de líneas a desplazar
BH = atributo a usarse en el área *blanqueada*
CH = coordenada Y de la esquina superior izquierda
CL = coordenada X de la esquina superior izquierda
DH = coordenada Y de la esquina inferior derecha
DL = coordenada X de la esquina inferior derecha

Datos de Salida: ninguno

Observaciones: Si AL contiene un valor distinto de cero, la ventana es desplazada hacia arriba.

Int 10h
Función 07h
Desplaza la Pantalla hacia abajo.

Inicializa la ventana especificada con espacios en blanco en el color especificado o desplaza hacia abajo el contenido de una ventana, el número de líneas indicado.

Parámetros de Entrada:
AH = 06h
AL = # de líneas a desplazar
BH = atributo a usarse en el área *blanqueada*
CH = coordenada Y de la esquina superior izquierda
CL = coordenada X de la esquina superior izquierda
DH = coordenada Y de la esquina inferior derecha
DL = coordenada X de la esquina inferior derecha

Datos de Salida: ninguno

Int 10h
Función 08h
Lee el Carácter y Atributo en la posición del cursor.

Obtiene el código ASCII y el atributo del carácter en la posición actual del cursor en la página de video especificada.

Parámetros de Entrada:
AH = 08h
BH = página de video

Datos de Salida:
AH = atributo
AL = código ASCII

Int 10h
Función 09h
Escribe un Carácter y su Atributo en la posición actual del cursor.

Escribe un carácter ASCII y su atributo en la posición actual del cursor en la página de video especificada. La posición del cursor no es actualizada.

Parámetros de Entrada:
AH = 09h
AL = carácter
BH = página de video
BL = atributo
CX = # de caracteres a escribir

Datos de Salida: ninguno

Observaciones: La posición del cursor no es actualizada.

Int 10h
Función 0Ah
Escribe un Carácter en la posición actual del cursor.

Escribe un carácter en la posición actual del cursor en la página de video especificada.

Parámetros de Entrada:
AH = 0Ah
AL = carácter
BH = página de video
CX = # de caracteres a escribir

Parámetros de Salida: ninguno

Observaciones: La posición del cursor no es actualizada.

Int 10h
Función 0Ch
Enciende un pixel

Enciende un pixel en las coordenadas especificadas.

Parámetros de Entrada: AH = 0Ch

AL = color
BH = página de video
CX = coordenada X
DX = coordenada Y

Datos de Salida: ninguno

Observaciones: El rango válido de coordenadas depende del modo de video. Ver Tabla G-1.

Si el bit 7 de Al está encendido, el nuevo pixel resultará de la operación OR exclusiva con el valor previo del pixel.

Int 10h
Función 0Dh
Lee el estado del Pixel en las coordenadas especificadas.

Lee el estado del pixel en las coordenadas especificadas.

Parámetros de Entrada:
AH = 0Ch
AL = color
BH = página de video
CX = coordenada X
DX = coordenada Y

Datos de Salida: ninguno

Observaciones: El rango válido de coordenadas depende del modo de video. Ver Tabla G-1.

Int 10h
Función 0Eh
Escribe un Carácter en modo teletipo.

Escribe un carácter ASCII en la posición actual del cursor. La posición del cursor es actualizada.

Parámetros de Entrada:
AH = 0Eh
AL = carácter
BH = página de video
BL = color del fondo (modo gráfico)

Datos de Salida: ninguno

Int 10h
Función 0Fh
Obtiene el Modo de Video activo.

Obtiene el modo de video activo en la controladora activa.

Parámetros de Entrada: AH = 0Fh

Datos de Salida:
AH = # de columnas
AL = modo de video (Ver Tabla G-1)
BH = página de video activa

Int 10h
Función 13h
Escribe un Cadena en modo teletipo.

Transfiere un cadena de caracteres a la memoria de video, a partir de la posición especificada.

Parámetros de Entrada:

AH = 13h
AL = modo de escritura
BH = página de video
BL = atributo (modos 0 y 1)
CX = longitud de la cadena
DH = coordenada Y (fila)
DL = coordenada X (columna)
ES:BP = segmento:desplazamiento de la cadena

Datos de Salida: ninguno

Observaciones: Los modo de escritura son:

Modo	***Descripción***
0	La cadena contiene sólo caracteres ASCII. El atributo se obtiene de BL. El cursor no es actualizado.
1	La cadena contiene sólo caracteres ASCII. El atributo se obtiene de BL. El cursor es actualizado.
2	La cadena contiene caracteres ASCII y atributos intercalados. La posición del cursor no es actualizada.
3	La cadena contiene caracteres ASCII y atributos intercalados. La posición del cursor es actualizada.

Int 11h
Obtiene la Configuración del Equipo.

Obtiene el código del ROM BIOS, que describe la configuración del equipo.

Parámetros de Entrada: ninguno

Datos de Salida: AX = lista del equipo

Bit	***Significado***
0	= 1 si hay disco(s) flexible(s) instalado(s)
1	= 1 si está instalado el coprocesador matemático
2-3	= RAM en la tarjeta madre
	00 = 16Kb
	01 = 32Kb
	10 = 48Kb
	11 = 64Kb
4-5	= modo inicial de video
	00 = reservado
	01 = 40x25 color
	10 = 80x25 color
	11 = 80x25 monocromático
6-7	= # de unidades de disco flexible
	00 = 1
	01 = 2
	10 = 3

11 = 4
8 = reservado
9-11 = # de puertos RS232 instalados
12 = 1 si está instalado el adaptador para juegos
13 = 1 si hay instalado un modem interno
14-15 = # de puertos paralelos instalados.

Observaciones: El valor retornado por los bits 2-3 es usado sólo por el ROM BIOS de los PCs originales.

Int 12h
Obtiene el Tamaño de la Memoria Convencional.

Devuelve la cantidad de memoria convencional disponible para programas de aplicación.

Parámetros de Entrada: ninguno

Datos de Salida: AX = tamaño de la memoria (en Kb).

Int 13h
Función 00h
Inicializa la Controladora de Discos.

Inicializa la controladora de discos, recalibra las unidades de discos conectadas y la prepara para transferencia de información.

Parámetros de Entrada:
AH = 00h
DL = unidad de disco (A=00h)
bit 7 = 0 para disco flexible
= 1 para disco duro

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
AH = 00hh

en caso de error:
CF = encendido
AH = estado de la controladora (ver Int 13h Función 01h)

Int 13h
Función 01h
Obtiene el Estado de la Controlador de Disco.

Devuelve el estado de la controladora de disco, después de la operación más reciente.

Parámetros de Entrada:
AH = 01h
DL = unidad de disco (A=00h)
bit 7 = 0 para disco flexible
= 1 para disco duro

Datos de Salida:
AH = 00h
AL = estado de la controladora

Valor	*Significado*
00h	Sin error.
01h	Comando inválido.
02h	No se encontró marca de dirección.
03h	Disco protegido contra escritura (Disco Flexible).
04h	Sector no encontrado.
05h	Inicialización fallida (Disco Duro).
06h	Disco flexible removido.
07h	Tabla de parámetros corrompida (Disco Duro).
08h	DMA desbordado (Disco Flexible)
09h	Violación del límite de 64Kb del DMA.
0Ah	Sector dañado (Disco Duro).
0Bh	Pista dañada (Disco Duro).
0Ch	Disco sin formatear (Disco Flexible).
0Dh	Número de sectores inválido (Disco Duro).
0Eh	Detectada marca de control de dirección (Disco Duro).
0Fh	Nivel de arbitraje del DMA fuera de rango (Disco Duro).
10h	Error irrecuperable de CRC o ECC.
11h	Error corregido de ECC (Disco Duro).
20h	Falla en la controladora.
40h	Error en la búsqueda.
80h	Expiró el tiempo de respuesta.
AAh	Unidad no preparada (Disco Duro)
BBh	Error indefinido (Disco Duro)
CCh	Falla de escritura (Disco Duro)
E0h	Error en el registro de estado (Disco Duro).
FFh	Operación fallida.

Int 13h
Función 02h
Leer Sector.

Transfiere el contenido de uno o más sectores a memoria.

Parámetros de Entrada:

AH = 02h
AL = # de sectores
CH = cilindro
CL = sector
DH = cabezal
DL = unidad de disco (A=00h)
bit 7 = 0 para disco flexible
= 1 para disco duro
ES:BX = segmento:desplazamiento de la zona de memoria

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
AH = 00hh
AL = # de sectores transferidos

en caso de error:
CF = encendido
AH = estado de la controladora (ver Int 13h Función 01h)

Observaciones:

En discos duros, los 2 bits más significativos del # de cilindro (10 bits) son colocados en los 2 bits más significativos del registro CL.

En disco flexibles, es posible que se presente un error si la función es invocada estando apagado el motor de la unidad, ya que el ROM BIOS no considera este aspecto de manera automática. El programa que haga uso de esta función en discos flexibles, deberá reinicializar el sistema y repetirla tres veces como mínimo antes de asumir la causa del problema.

Int 13h
Función 03h
Escribir Sector.

Transfiere uno o más sectores de memoria a disco.

Parámetros de Entrada:

AH = 03h
AL = # de sectores
CH = cilindro
CL = sector
DH = cabezal
DL = unidad de disco (A=00h)
bit 7 = 0 para disco flexible
= 1 para disco duro
ES:BX = segmento:desplazamiento de la zona de memoria

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
AH = 00hh
AL = # de sectores transferidos

en caso de error:
CF = encendido
AH = estado de la controladora (ver Int 13h Función 01h)

Observaciones:

En discos duros, los 2 bits más significativos del # de cilindro (10 bits) son colocados en los 2 bits más significativos del registro CL.

En disco flexibles, es posible que se presente un error si la función es invocada estando apagado el motor de la unidad, ya que el ROM BIOS no considera este aspecto de manera automática. El programa que haga uso de esta función en discos flexibles, deberá reinicializar el sistema y repetirla tres veces como mínimo antes de asumir la causa del problema.

Int 13h
Función 04h
Verificar Sector.

Verifica el campo de dirección de uno o más sectores.

Parámetros de Entrada:

AH = 04h
AL = # de sectores
CH = cilindro
CL = sector
DH = cabezal
DL = unidad de disco (A=00h)
bit 7 = 0 para disco flexible

= 1 para disco duro
ES:BX = segmento:desplazamiento de la zona de memoria

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
AH = 00hh
AL = # de sectores verificados

en caso de error:
CF = encendido
AH = estado de la controladora (ver Int 13h Función 01h)

Observaciones:

En discos duros, los 2 bits más significativos del # de cilindro (10 bits) son colocados en los 2 bits más significativos del registro CL.

En disco flexibles, es posible que se presente un error si la función es invocada estando apagado el motor de la unidad, ya que el ROM BIOS no considera este aspecto de manera automática. El programa que haga uso de esta función en discos flexibles, deberá reinicializar el sistema y repetirla tres veces como mínimo antes de asumir la causa del problema.

Int 13h
Función 08h
Obtiene los Parámetros de la Unidad de Disco.

Devuelve los parámetros de la unidad de disco especificada.

Parámetros de Entrada:

AH = 08h
DL = unidad de disco (A=00h)
bit 7 = 0 para disco flexible
= 1 para disco duro

Datos de Salida:

Si la operación fue exitosa:
CF = apagado
BL = tipo de unidad de disco
01h = 360Kb, 40 pistas, 5,25"
02h = 1,2Mb, 80 pistas, 5,25"
03h = 720Kb, 80 pistas, 3,5"
04h = 1,44Mb, 80 pistas, 3,5"
CH = 8 bits menos significativos del # máximo de cilindros.
CL = bit 6-7 bits más significativos del # máximo de cilindros
= bit 0-5 número máximo de sectores
DH = máximo número de cabezales
DL = número de unidades de disco.
ES:DI = segmento:desplazamiento de la tabla de parámetros de disco

en caso de error:
CF = encendido
AH = estado de la controladora (ver Int 13h Función 01h)

Int 14h
Función 00h
Inicializa el Puerto Serial.

Inicializa el puerto serial especificado con la rata de baudios, paridad, longitud de palabra y número de bits de parada indicados.

Parámetros de Entrada:

AH = 00h
AL = parámetro de inicialización
DX = # del puerto (0= COM1, 1= COM2, etc.)

Datos de Salida:

AH = estado del puerto

Bit	***Significado***
0	datos listos.
1	error de desbordamiento
2	error de paridad
3	error de enmarcaje
4	ruptura de transmisión
5	registro de transmisión vacío
6	registro de desplazamiento de transmisión vacío
7	expiro el tiempo

AL = estado del modem

Bit	***Significado***
0	cambio en el estado de *CLEAR TO SEND*.
1	cambio en el estado de *DATA SET READY*
2	indicador de flanco
3	cambio en la línea de detección de señal
4	listo para enviar.
5	datos preparados.
6	Indicador de portadora
7	detección de señal de recepción

Observaciones:

El parámetro de inicialización se define de la siguiente manera:

7 6 5	**4 3**	**2**	**1 0**
Rata de Transmisión	***Paridad***	***Bits de Parada***	***Longitud de Palabra***
000= 110	X0= ninguna	0= 1 bit	10= 7 bits
001= 150	01= impar	1= 2 bits	11= 8 bits
010= 300	11= par		
011= 600			
100= 1200			
101= 2400			
110= 4800			
111= 9600			

Int 14h
Función 01h
Escribe un Carácter en el Puerto Serial.

Escribe un carácter en el puerto serial especificado y devuelve el estado del puerto.

Parámetros de Entrada:

AH = 01h
AL = carácter
DX = # del puerto (0= COM1, 1= COM2, etc.)

Datos de Salida:

Si la operación fue exitosa:

AH = bit 7 apagado
AH = bits 0-6: estado del puerto (ver Int 14h Función 00h)
AL = carácter

en caso de error:
AH = bit 7 encendido
AL = carácter

Int 14h
Función 02h
Lee un Carácter del Puerto Serial.

Lee un carácter del puerto serial especificado y devuelve el estado del mismo.

Parámetros de Entrada:
AH = 02h
DX = # del puerto (0= COM1, 1= COM2, etc.)

Datos de Salida:
Si la operación fue exitosa:
AH = bit 7 apagado
AH = bits 0-6: estado del puerto (ver Int 14h Función 00h)
AL = carácter

en caso de error:
AH = bit 7 encendido

Int 14h
Función 03h
Obtiene el Estado del Puerto Serial.

Devuelve el estado del puerto serial especificado.

Parámetros de Entrada:
AH = 03h
DX = # del puerto (0= COM1, 1= COM2, etc.)

Datos de Salida:
AH = estado del puerto (ver Int 14h Función 00h)
Al = estado del modem (ver Int 14h Función 00h)

Int 15h
Función 84h
Lee el Adaptador para Juegos.

Devuelve el estado de los interruptores y valores de los potenciómetros del adaptador para juegos.

Parámetros de Entrada:
AH = 84h
DX = SubFunción:
00h = leer el estado de los interruptores (botones)
01h = leer los valores de los potenciómetros

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

Si DX= 00h

AL = estado de los interruptores (bits 4-7)

Si DX= 01h
AX = valor X del adaptador A
BX = valor Y del adaptador A
CX = valor X del adaptador B
DX = valor Y del adaptador B

en caso de error:
CF = encendido

Int 15h
Función 86h
Retardo.

Suspende la ejecución del programa por el tiempo especificado.

Parámetros de Entrada:
AH = 86h
CX:DX = tiempo de retardo (en microsegundos)

Datos de Salida:
Si la operación fue exitosa:
CF = apagado

en caso de error:
CF = encendido

Observaciones:
La duración del retardo es siempre un múltiplo entero de 976 microsegundos.

Int 16h
Función 00h
Lee un Carácter del Teclado

Lee un carácter del teclado y devuelve su código de búsqueda.

Parámetros de Entrada:
AH = 00h

Datos de Salida:
AH = código de búsqueda
AL = código ASCII del carácter

Int 16h
Función 01h
Obtiene el Estado del Teclado.

Determina si existe algún carácter en el buffer del teclado por ser leído y lo lee en caso de estar disponible.

Parámetros de Entrada:
AH = 01h

Datos de Salida:
Si existe algún carácter:
ZF = apagado
AH = código de búsqueda
AL = carácter

en caso contrario:
ZF = encendido

Observaciones: El carácter devuelto por esta función no es removido del buffer del teclado, por lo que el mismo será devuelto por una posterior invocación a la Int 16h Función 00h.

Int 16h
Función 02h
Obtiene el Estado de las Banderas del Teclado.

Devuelve las banderas del ROM BIOS que describen el estado de varias teclas especiales.

Parámetros de Entrada: AH = 02h

Datos de Salida: AL = banderas

Bit	*Significado*
0	la tecla Shift derecha está presionada
1	la tecla Shift izquierda está presionada
2	la tecla Ctrl está presionada
3	la tecla Alt está presionada
4	Scroll Lock está encendido
5	Num Lock está encendido
6	Caps Lock está encendido
7	Insert está encendido

Observaciones: Las banderas del ROM BIOS están almacenadas en la posición de memoria 0000:0417h.

Int 16h
Función 05h
Introduce un Carácter en el Buffer del Teclado.

Coloca el código ASCII y el código de búsqueda de un carácter en el buffer del teclado.

Parámetros de Entrada:
AH = 05h
CH = código de búsqueda
CL = carácter

Datos de Salida:
Si la operación fue exitosa:
CF = apagado
AL = 00h

en caso de error:
CF = encendido
AL = 01h

Int 16h
Función 10h
Lee un Carácter de un Teclado Enhanced.

Lee un carácter del teclado y devuelve su código de búsqueda.

Parámetros de Entrada: AH = 10h

Datos de Salida: AH = código de búsqueda
AL = código ASCII del carácter

Observaciones: Use esta función para teclados *enhanced* en lugar de Int 16h Función 00h. Permite obtener el código de búsqueda de las teclas F11, F12 y las teclas adicionales de cursor.

Int 16h
Función 11h
Obtiene el Estado de un Teclado Enhanced.

Determina si existe algún carácter en el buffer del teclado por ser leído y lo lee en caso de estar disponible.

Parámetros de Entrada: AH = 01h

Datos de Salida: Si existe algún carácter:
ZF = apagado
AH = código de búsqueda
AL = carácter

en caso contrario:
ZF = encendido

Observaciones: El carácter devuelto por esta función no es removido del buffer del teclado, por lo que el mismo será devuelto por una posterior invocación a la Int 16h Función 00h.

Use esta función para teclados *enhanced* en lugar de Int 16h Función 01h. Permite obtener el código de búsqueda de las teclas F11, F12 y las teclas adicionales de cursor.

Int 16h
Función 12h
Obtiene el Estado de las Banderas de un Teclado Enhanced.

Devuelve las banderas del ROM BIOS que describen el estado de varias teclas especiales.

Parámetros de Entrada: AH = 02h

Datos de Salida: AL = banderas

Bit	***Significado***
0	la tecla Shift derecha está presionada
1	la tecla Shift izquierda está presionada
2	alguna de las teclas Ctrl está presionada
3	alguna de las teclas Alt está presionada
4	Scroll Lock está encendido
5	Num Lock está encendido
6	Caps Lock está encendido
7	Insert está encendido
8	la tecla Ctrl izquierda está presionada
9	la tecla Alt izquierda está presionada
10	la tecla Ctrl derecha está presionada

11	la tecla Alt derecha está presionada
12	la tecla Scroll está presionada
13	la tecla NumLock está presionada
14	la tecla Caps Lock está presionada
15	la tecla SysRq está presionada

Observaciones: Use esta función para teclados *enhanced* en lugar de Int 16h Función 02h.

Int 17h
Función 00h
Escribe un Carácter en la Impresora.

Envía un carácter al puerto paralelo especificado y devuelve el estado del mismo.

Parámetros de Entrada:
AH = 00h
AL = carácter
DX = # de la impresora (0= LPT1, 1= LPT2,etc.)

Datos de Salida: AH = estado del puerto

Bit	***Significado***
0	expiró el tiempo de respuesta
1-2	no se usan
3	error de E/S
4	impresora seleccionada
5	sin papel
6	reconocimiento positivo
7	impresor no ocupado

Int 17h
Función 01h
Inicializa el Puerto de la Impresora.

Inicializa el puerto paralelo especificado y devuelve su estado.

Parámetros de Entrada:
Ah = 01h
DX = # de la impresora (0= LPT1, 1= LPT2,etc.)

Datos de Salida: AH = estado del puerto (ver Int 17h Función 00h)

Int 17h
Función 02h
Obtiene el Estado del Puerto de la Impresora.

Devuelve el estado actual del puerto paralelo especificado.

Parámetros de Entrada:
Ah = 02h
DX = # de la impresora (0= LPT1, 1= LPT2,etc.)

Datos de Salida: AH = estado del puerto (ver Int 17h Función 00h)

Int 1Ah
Función 00h
Lee el Conteo del Reloj interno.

Devuelve el número de *ticks* del reloj interno.

Parámetros de Entrada: AH = 00h

Datos de Salida: CX:DX = conteo (16 bits más significativos en CX)

Observaciones: El valor devuelto es el acumulado desde la media noche anterior. Un *tick* se produce 18,2 veces por segundo. Cuando el contador alcanza el número 1.573.040 es puesto a cero nuevamente.

Int 1Ah
Función 01h
Establece el Conteo del Reloj interno.

Almacena un valor de 32 bits en el contador del reloj interno.

Parámetros de Entrada: AH = 01h
CX:DX = conteo (16 bits más significativos en CX)

Datos de Salida: ninguno

Int 33h
Función 00h
Inicializa el Ratón y obtiene su Estado.

Inicializa el manejador del ratón y devuelve el estado del mismo. Si el cursor del ratón estaba es visible, el mismo es removido de la pantalla.

Parámetros de Entrada: AX = 0000h

Datos de Salida: Si el ratón está disponible:
AX = FFFFh
BX = # de botones en el ratón

en caso contrario:
AX = 0000h

Int 33h
Función 01h
Muestra el Cursor del Ratón.

Muestra el cursor del ratón y cancela cualquier área de exclusión previamente definida.

Parámetros de Entrada: AX = 0001h

Datos de Salida: ninguno

Int 33h
Función 02h
Esconde el Cursor del Ratón.

Remueve el cursor del ratón de la pantalla.

Parámetros de Entrada: AX = 0002h

Datos de Salida: ninguno

Int 33h
Función 03h
Obtiene la Posición del Ratón y el Estado de los Botones.

Devuelve la posición actual del ratón y el estado de los botones.

Parámetros de Entrada: AX = 0003h

Datos de Salida: BX = estado de los botones

Bits(s)	***Significado***
0	botón izquierdo presionado
1	botón derecho presionado
2	botón central presionado
3-15	reservado

CX = coordenada X
DX = coordenada Y

Int 33h
Función 04h
Establece la Posición del Ratón.

Establece la posición del cursor del ratón. El cursor será mostrado en la nueva posición a menos que haya sido removido con Int 33h Función 02h o que la nueva posición esté dentro de el área de exclusión.

Parámetros de Entrada: AX = 0004h
CX = coordenada X
DX = coordenada Y

Datos de Salida: ninguno

Int 33h
Función 05h
Obtiene la Información de Presión de los Botones.

Devuelve el estado de actual de todos los botones y el número de veces que ha sido presionado el botón especificado desde la última invocación a esta función para ese botón.

Parámetros de Entrada: AX = 0005h
BX = identificador del botón:
0 = botón izquierdo
1 = botón derecho

2 = botón central

Datos de Salida:
AX = estado de los botones (ver Int 33h Función 03h)
BX = # de veces que ha sido presionado el botón
CX = coordenada X de la última vez que fue presionado
DX = coordenada Y de la última vez que fue presionado

Int 33h
Función 06h
Obtiene la Información de Depresión de los Botones.

Devuelve el estado de actual de todos los botones y el número de veces que ha sido depresionado el botón especificado desde la última invocación a esta función para ese botón.

Parámetros de Entrada:
AX = 0006h
BX = identificador del botón:
0 = botón izquierdo
1 = botón derecho
2 = botón central

Datos de Salida:
AX = estado de los botones (ver Int 33h Función 03h)
BX = # de veces que ha sido depresionado el botón
CX = coordenada X de la última vez que fue depresionado
DX = coordenada Y de la última vez que fue depresionado

Int 33h
Función 07h
Selecciona los Límites Horizontales para el Cursor del Ratón.

Limita el área de movimiento asignando los valores horizontales máximo y mínimo para el cursor del ratón.

Parámetros de Entrada:
AX = 0007h
CX = limite horizontal inferior (X)
DX = límite horizontal superior (X)

Datos de Salida: ninguno

Int 33h
Función 08h
Selecciona los Límites Verticales para el Cursor del Ratón.

Limita el área de movimiento asignando los valores verticales máximo y mínimo para el cursor del ratón.

Parámetros de Entrada:
AX = 0008h
CX = limite vertical inferior (Y)
DX = límite vertical superior (Y)

Datos de Salida: ninguno

Int 33h
Función 09h
Selecciona la Forma del Cursor del ratón en Modo Gráfico.

Define la forma, color y punto activo del cursor del ratón en modo gráfico.

Parámetros de Entrada:
AX = 0009h
BX = desplazamiento en X del punto activo
CX = desplazamiento en Y del punto activo
ES:DX = segmento:desplazamiento del buffer que contiene la imagen

Datos de Salida: ninguno

Observaciones: El buffer de la imagen es una área de 64 bytes. Los primeros 32 bytes contienen una máscara de bits que será mostrado por pantalla como el resultado de una operación AND con el contenido de la misma. Los segundos 32 bytes, se mostrarán por pantalla tras una operación OR exclusiva con el contenido de la pantalla.

El desplazamiento del punto activo es relativo a la esquina superior izquierda de la imagen.

Int 33h
Función 0Ah
Establece el Tipo de Cursor para el ratón en Modo Texto.

Define el tipo y atributo del cursor del ratón en modo texto.

Parámetros de Entrada:
AX = 000Ah
BX = tipo de cursor:
0 = cursor por *software*
1 = cursor por *hardware*
CX = máscara AND (si BX=0) o
= línea inicial (si BX= 1)
DX = máscara XOR (si BX= 0) o
= línea final (si BX= 1)

Datos de Salida: ninguno

Int 33h
Función 10h
Establece el Area de Exclusión para el Cursor del Ratón.

Define el área de exclusión para el cursor del ratón. Cuando el cursor entra en esa área, desaparece.

Parámetros de Entrada:
AX = 0010h
CX = coordenada X de la esquina superior izquierda
DX = coordenada Y de la esquina superior izquierda
SI = coordenada X de la esquina inferior derecha
DI = coordenada Y de la esquina inferior derecha

Datos de Salida: ninguno

Apéndice H
Lectura de Referencia.

Adam Osborne, 1980. *An Introduction to Microcomputers*, *Volume 1: Basic Concepts, Second Edition*. OSBORNE/McGraw-Hill.

Miguel Angel Rodríguez Roselló, 1988. *8088-8086/8087 Programación Ensamblador en entorno MS DOS*. Ediciones ANAYA Multimedia, S.A.

José Mª Angulo Usátegui, 1982. *Microprocesadores, Arquitectura, programación y desarrollo de sistemas Segunda Edición* Paraninfo.

Enrique Mandado, 1981. *Sistemas Electrónicos Digitales, Cuarta Edición*. Marcombo Boixareu Editores.

José María Angulo, 1983. *Técnicas de Informática Hoy, Tomo 3: Microprocesadores, Fundamento, diseño y aplicación en la industria y en los microcomputadores.* Paraninfo.

Russel Rector - George Alexy, 1980. *The 8086 Book*. OSBORNE/McGraw-Hill.

Nabajyoti Barkakati, 1989. *The Waite Group's Microsoft Macro Assembler Bible*. Howard W. Sams & Company.

1990. *Turbo Assembler 2.0 User's Guide*. Borland International Inc.

1990. *Turbo Assembler 2.0 Reference Guide.* Borland International Inc.

Tom Swan, 1989. *Mastering Turbo Assembler*. Hayden Books.

Ray Duncan, 1988. *Advanced MSDOS Programming, Second Edition*. Microsoft Press.

Allen L. Wyatt, 1990. *Using Assembly Language, Second Edition*. QUE Corporation.

John Argenmeyer, Kevin Jaeger, Raj Kumar Bapna, Nabajyoti Barkakati, Rajagopalan Dhesikan, Walter Dixon, Andrew Dumke, Jon Fleig, Michael Goldman, 1989. *The Waite Group's MS-DOS Developer's Guide, Second Edition*. Howard W. Sams & Company.

1989. *DOS and BIOS Quick Reference*. QUE Corporation.

1990. *Lenguaje Ensamblador, Manual de bolsillo*. Addison-Wesley Iberoamericana.

Carl Townsend, 1989. *Advanced MS-DOS Expert Techniques for Programmers.* Howard W. Sams & Company.

Peter Norton - John Socha, 1988. *Guía del programador en Ensamblador para IBM PC, XT, AT y compatibles*. ANAYA Multimedia.
1989. *System BIOS for IBM PC/XT/AT Computers and Compatibles*. Phoenix Technologies Ltd.

Michael J. Young, 1988. *MS DOS Advanced Programming*. SYBEX.

Tom Swan, 1989. *Mastering Turbo Pascal 5.5, Third Edition*. Hayden Books.

1991. *Microsoft MS-DOS Programmer's Reference*. Microsoft Press.

Michael Tischer, 1991. *Turbo PASCAL 6 System Programming*. Abacus.

Nabajyoti Barkakati, 1989. *The Waite Group's Turbo C Bible*. Howard W. Sams & Company.

Donald Hearn / M. Pauline Baker, 1988. *Gráficas por Computadora*. Prentice-Hall Hispanoamericana, S.A.

Jan-Feb 1992 Vol 7 No 1. *Microsoft Systems Journal: Strategies and Techniques for Writing State-of-the-Art TSRs that Exploit MS-DOS 5*. Microsoft.

#186 March 1992. *Dr. Dobb's Journal: The UCR standard assembly language library.* American Business Press

Indice Alfabético

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X