UNIVERSIDAD DE EL SALVADOR

Facultad de Ingeniería y Arquitectura

Escuela de Ingeniería Eléctrica

EL LENGUAJE "ASSEMBLY" 8088 Y SUS APLICACIONES EN INGENIERIA ELECTRICA PARA EL MANEJO DE HARDWARE

TRABAJO DE GRADUACION PRESENTADO POR:

ERNESTO CHINCHILLA MENJIVAR

JOSE ROBERTO TREJO BARRERA

PARA OPTAR AL TITULO DE:

INGENIERO ELECTRICISTA

MAYO DE 1991

SAN SALVADOR, EL SALVADOR, CENTRO AMERICA

TRABAJO DEDICADO A:

Mis padres: Jose Efrain Trejo.

Martha Enma Barrera de Trejo.
(Q.D.D.G)

Mis hermanos: Jose Aristides, Efrain, Manuel, Guillermo
Y Esther.

Mis compañeros de estudio y demas amigos.

Y muy especialmente a mi amiga Julia Alicia Burgos.

Jose Roberto Trejo Barrera.

UNIVERSIDAD DE EL SALVADOR

RECTOR: DR. JOSE BENJAMIN LOPEZ GUILLEN

SECRETARIO GENERAL: DRA. GLORIA ESTELA GOMEZ DE FEREZ

FACULTAD DE INGENIERIA Y ARQUITECTURA

DECANO: ING. JOAQUIN ALBERTO VANEGAS AGUILAR

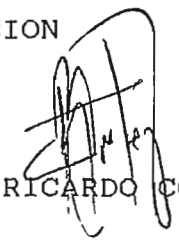
SECRETARIO: ING. MARIO ARNOLDO MOLINA ARGUETA

ESCUELA DE INGENIERIA ELECTRICA

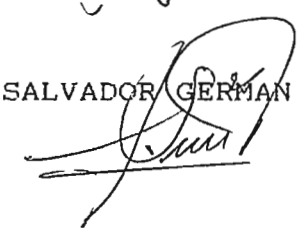
COORDINADOR: ING. JOSE RIGOBERTO MURILLO CAMPOS

UNIVERSIDAD DE EL SALVADOR

TRABAJO DE GRADUACION



COORDINADOR ING. RICARDO CORTEZ



ASESOR ING SALVADOR GERMAN

ACTA DE CONSTANCIA DE NOTA Y DEFENSA FINAL

En esta fecha, 14 de mayo 1991, en el local de la Sala de lectura de la Escuela de Ingeniería Eléctrica a las 8.30 a.m. horas, con la presencia de las siguientes autoridades de la Escuela de Ingeniería Eléctrica de la Universidad de El Salvador:

Firma

1- Ing. José Rigoberto Murillo Campos ...
Coordinador

2- Ing. Hector Jacinto Vásquez ...
Secretario

3- Ing. Ricardo Ernesto Cortéz ...
Coordinador de Investigación

Y con el Honorable Jurado de evaluación integrado por las personas siguientes:

Firma

1 Ing. Guillermo Adolfo Guillén Villeda ...

2 Ing. Salvador de Jesus German ...

3-Sr. Jaime Mauricio Guevara ...

4- ...

5- ...

6- ...

Se efectuó la defensa final reglamentaria del Trabajo de Graduación: EL LENGUAJE "ASSEMBLY" 8088 Y SUS APLICACIONES EN INGENIERIA ELECTRICA PARA EL MANEJO DE HARDWARE.

A cargo del (los) Br(es):

1- Ernesto Chinchilla Menjivar.

2- José Roberto Trejo Barrera.

Habiendo obtenido el presente trabajo una nota final, global de: 8.0 (Ocho punto Cero).

ESCUELA DE INGENIERIA ELECTRICA
FACULTAD DE INGENIERIA
Y ARQUITECTURA
Universidad de El Salvador

TRABAJO DEDICADO A:

Mis pradres: Roberto Chinchilla.

Blanca Lidia Menjivar de Chinchilla.

Mis hermanos Yohalmo y Roberto.

Mis tios, familia y mi querida Abuela.

Mis companeros de estudios y demas amigos.

Y muy especialmente a mi novia Yazyka Armida Ramos.

Ernesto Chinchilla Menjivar.

AGRADECIMIENTOS:

PRESENTAMOS NUESTROS AGRADECIMIENTOS A LAS SIGUIENTES PERSONAS POR SU DESINTERESADA COLABORACION EN EL TRANSCURSO DE NUESTRA CARRERA O EN EL DESARROLLO DE ESTE TRABAJO.

Ing. Ricardo Cortéz. Coordinador del trabajo
Ing. Salvador German. Asesor del trabajo

Ing. y Lic. Luis Alfredo Chinchilla. Gerente Ing. de sistemas N.C.R

Ing. Jaime A. Anaya Catedrático de la U.E.S

Ing. Rene Barbier. Director del I.G.N
Ing. Julio Bran. Sub Director del I.G.N

A LOS COMPAÑEROS DE TRABAJO DEL DPTO. DE DIBUJO DEL I.G.N

Arq. Ena Mendoza.
Br. Gabriela Zarate.
Ing. Reinaldo Medina.
Ing. Ranulfo A. Rivas.
Br. Uriel Muñoz.
Br. Alfredo A. Ramirez.
Br. Jhony Albino.

TABLA DE CONTENIDOS

Capítulo — Página

Capítulo Página

II. SOFTWARE

Capítulo Página

III. HARDWARE ALREDEDOR DE UN SISTEMA 286

Capítulo Página

IV. SOFTWARE DE COMUNICACIONES POR MEDIO DEL 8088

Capítulo Página

V. EL SOFTWARE EN ROM

VI APENDICE

LISTA DE TABLAS

PREFACIO

El presente trabajo se ha realizado como una respuesta a la necesidad de asentar las bases de lo que será la implementación del área de estudio de las microcomputadoras. Por su puesto, que no basta un solo trabajo para cubrir un área tan extensa. A pesar de ser un tema bastante amplio, el presente trabajo, pone al alcance del estudiante interesado, gran cantidad de conocimiento práctico.

El objetivo de este trabajo, es la recopilación y organización de información necesaria para que los futuros estudiantes de materias relacionadas con microprocesadores, sistemas digitales y microcomputadoras en general, dominen el lenguaje de ensamble, conozcan algunas de las aplicaciones de dicho lenguaje utilizando la familia de microprocesadores 8088 ampliamente utilizadas en las computadoras IMB PC y compatibles.

Los alcances que se plantearón para este trabajo fueron:

1. Modificación de programas en lenguaje de ensamble
2. Creación de rutinas en lenguaje de máquina, para la transmisi
   de información entre microcomputadoras.
3. Creación de guías de laboratorio para la enseñanza de siste
   electrónicos programables, usando el microprocesador 8088.

En lo que respecta a creación y modificación de programas en lenguge de ensamble, podemos decir, que fue algo que no solamente se logró, si no que se superó con las aplicaciones del programa diseñado en el capitulo dos, que ademas de mostrar como se aplican las técnicas profesionales de programación, resulta de gran utilidad práctica y academica, al mostrar claramente la forma en que el DOS almacena la información en los discos.

Con respecto al diseño de rutinas en lenguaje de máquina, para la transmisión de información entre computadoras, se realizarón algunas modificaciones al descubrir que ni las funciones del DOS ni del BIOS resultan la mejor opción, para este tipo de aplicaciones. En su lugar se da información del hardware que poseen muchas computadoras especialmente diseñado para comunicaciones, es decir, se estudio el UART, dispositivo que se encuentra en una tableta serie opcional especialmente diseñado para comunicaciones asincronas, y la forma de programarlo por medio de lenguajes de alto nivel, utilizando el estandar de comunicación RS-232C.

Y por último en cuanto a las guías de laboratorio debido a la falta de equipo adecuado para realizar pruebas con microprocesadores, esta parte del trabajo, se sustituyó por el diseño de un circuito de prueba con el microprocesador 80286. El cual puede ser muy útil para comprender conceptos de software y de hardware implicados en un diseño con microprocesador

RESUMEN

En resumen el presente trabajo contiene las bases en las áreas de diseño de software, comprensión del hardware de una PC y del software de comunicaciones entre PC ; ademas se muestra como funciona el debug, programa que resultó muy útil en el desarrollo de este trabajo en la depuración de programas, tambien se incluye una introduccion a lo que es la memoria ROM-BIOS

Brevemente los temas desarrollados son los siguientes:
En el capitulo 1 se da una introducción a lo que son los microprocesadores, realizando una comparación entre distintos microprocesadores en cuanto a su estructura interna, juego de registros, modos de direccionamientos de la familia 8088 de Intel.

El Capitulo 2 trata sobre en lenguaje de ensamble del microprocesador 8088, y sobre el Macro Assembler. En este capitulo se desarrollan una serie de ejemplos prácticos todos interrelacionados con un fin comun de producir al final, un programa con aplicaciones prácticas, como lo son la exploración y modificación del contenido de un disco a nivel de sectores. El conocimiento que se adquirió con este programa, hizo posible lograr aplicaciones tales como la modificación de programas en lenguaje de máquina, resultando relativamente sencillo, traducir programas cuyos mensajes estan en ingles, al español. Otra aplicación que se realizó con todo exito, fue la de recuperar archivos que ya se habian eliminado por medio de comandos del DOS.

El capitulo 3 trata sobre el diseño de un sistema utilizando el microprocesador 80286 y la familia de chips de soporte con este micro, a la vez que se describe en forma general cada función de los chips de soporte haciendo mayor enfasis a aquellas señales de comunicación con el 286; este capitulo termina con el diseño de un circuito de prueba con el 286, de manera que se pueda observar su funcionamiento o dectectar por medio de uso de leds si estan funcionando correctamente las partes mas criticas de este sistema, como por ejemplo: generación de ciclos de reloj, generación de señales de READY, etc..

El capitulo 4 contiene información sobre la teoría de comunicaciones entre microcomputadoras, asi como tambien, una buena cantidad de datos sobre posibles fallas que se puedan dar cuando dos computadoras intercambian información.

En el capitulo 5 se tratan aspectos relacionados con las memorias de la ROM-BIOS, este conocimiento resulta indispensable para programadores que quieren explotar al maximo los recursos disponibles en una PC.

Otra información de gran valor que se presenta en este trabajo, es la que se refiere al Debug. Este programa que resulto de gran ayuda en la realización de este trabajo, se explica de una manera sencilla, superando lo escueto de las explicaciones que se encuentran en los manuales del DOS.

El debug a demas de facilitar la comprensión de la estructura logica de un microprocesador, se utilizó para encontrar errores, debidos al programador, e incluso, a veces, errores producidos por el mismo compilador o por el procesador de texto utilizado.

Como en todo trabajo de investigación, en el camino se encuentran sorpresas, que a veces, modifican las prioridades inicialmente, planteadas. En ese sentido, este trabajo no fue la excepción, y rutinas de comunicación entre microcomputadoras, que originalmente, se pensaban realizar en lenguaje de ensamble utilizando funciones del DOS, se realizaron utilizando las ventajas que ofrece en lenguaje BASIC; aunque, tambien se agrega en los anexos las instrucciones equivalentes en lenguaje C.
El motivo que originó este giro, fué que se decubrieron inconsistencias en las funciones del DOS y del BIOS.

Es necesario aclarar que para comprender el programas DSI.COM que se presenta en el desarrollo de este trabajo, es necesario ir construyendolo paso a paso, pues es demasiado largo para comprenderlo de una sola vez,ademas; este programa contiene muchas rutinas de uso geneara que se utlizan en la mayoria de programas escritos en leguaje de máquina, por lo que consideramos que se será de gran ayuda para que el interesado escriba sus propios programas. En el anexo 3 se da una descripcion detallada del Macro Assembles que es el compilador utilizado en este trabajo.

En cuanto al programa DESCARGA.BAS que se presenta al final del capitulo 4, es bastante fácil de comprender. Los conceptos que se dan en este programa son muy utiles para diseñar experimentos que mejoren nuestra comprensión del hardware y del software de comunicaciones, como ejemplo se puede mensionar, el envio de señales de handshaking al puerto serie y verificar su existencia midiendo voltages en los pines respectivos. Esto es muy útil a la hora de detectar fallas de hardware.

Lo último que se desea agregar, es que esperamos que este trabajo contribuya al desarrollo del area de estudio de las computadoras personales, que día a día tienen mas aplicaciones en cualquier parte del mundo.

CAPITULO I

LOS MICROPROCESADORES DE 8 Y 16 BITS

1.1 Evolución del 8086,8088, Y el 80286

Los microprocesadores más antiguos fueron dispositivos de 4 bits. Esto indicaba que ellos solo podían transferir información de cuatro bits a la vez, así que para transferir más de cuatro bits, necesitaban ejecutar operaciones separadas, siendo por lo tanto muy lentos en velocidad de procesamiento, debido a este inconveniente es que la compañía INTEL, en 1972, introduce el microprocesador comercial de 8 bits (este transfería información de 8 bits a la vez) llamándolo 8008 ó "microprocesador de la primera generación".Diseñado con una arquitectura igual a una calculadora, el 8008 contaba con un acumulador, seis registros, un stack pointer,(registro especial para direccionar el stack), ocho registros de direcciones, e instrucciones especiales para ejecutar operaciones de entrada y salidas.
En 1973, INTEL introduce una segunda generación de la versión del 8008, llamándolo microprocesador 8080.
El 8080 fué una mejora y ampliación del 8008 contando con más rango de direcciones, mayor capacidad de entrada-salida, y una mayor velocidad en el tiempo de ejecución.
La organización interna del 8080 es mucho mejor, a pesar de que INTEL mantuvo toda la filosofía de la arquitectura del 8008.
El 8080 fué, históricamente, de facto, el estandar de los microprocesadores de la segunda generación.
En 1976 los avances tecnológicos llevaron a INTEL a producir una versión ampliada del 8080, el microprocesador 8085.
Escencialmente el 8085 agregó muchos adelantos, tales como la función de reseateado (inicializar el microprocesador), vectores de interrupción (servicio de necesidades a los periféricos), un puerto serie de I/O (para conexión de impresores y otros periféricos), y una fuente de poder de 5 volt. (el 8008 requería de dos fuentes).
En esta época en la cual INTEL lanza al mercado el microprocesador 8085 de 8 bits, se enfrenta a una fuerte competencia con compañías diseñadoras de microprocesadores, entre las cuales se destacan la Corporación ZILOG con el microprocesador 8080 mejorado, el Z80, la compañía MOTOROLA con el microprocesador 68000 , y el microprocesador 6502 con tecnología MOS (ahora COMMODORE),a pesar de ello INTEL introduce al mercado en el año de 1978 el microprocesador 8086; un microprocesador más poderoso que sus adversarios, el cual esta diseñado para trabajar con 16 bits, lo cual lo hace diez veces más veloz que el 8080.
La característica de mayor importancia de este microprocesador fue la compatibilidad de su software con el 8080 a nivel de

lenguaje de bajo nivel (ASSEMBLY); significando, que con un mínimo de cambios en los programas del 8080, podrían ser ensamblados y ejecutados en el 8086.
Para ello los registros y juego de instrucciones del 8080 aparecen como un sub-arreglo de los registros e instrucciones del 8086. Con esta compatibilidad del 8080, el 8085 de INTEL capitalizó experiencia y mayor aceptación en aplicaciones mas sofisticadas.En la misma disposición, INTEL percibió que muchos diseñadores buscaban usar un soporte barato de 8 bits en sus periféricos, con todo los sistemas de 16 bits, pero con un bus de datos de 8 bits. Por esta razón, INTEL lanza al mercado su microprocesador 8088, el cual fué utilizado ampliamente en las computadoras personales (PC) de IBM, PC XT y compatibles.
En 1982 la INTEL produce el 80186 y su compañero el 80188 de 8 bits, en el cual integraba todo el avance del procesamiento del 8086 más un soporte de 15 chips adicionales.
Esta "computadora de un chip" contenía dos canales de acceso de direccionamiento de memoria (DMA) independientes, una para periféricos de alta velocidad, como los disk drivers, y un controlador de interrupciones programable.
En términos de software se agregaron mayor cantidad de instrucciones del 8086, que eran de gran valor para las personas que diseñan traductores de lenguaje de alto nivel, o compiladores.
En este mismo año INTEL produce el 80286, los microprocesadores que controlaron el trabajo de las computadoras IBM PC e IBM AT.
El 80286 es una expansión del microprocesador 80186 y provee características necesarias para el manejo y protección de memorias y programas de sistemas operativos.
Estas características son importantes para aplicaciones de multi-usuario que comparten los sistemas de computadoras (como por ejemplo una red de área local ó LAN).

1.2 Una vista elemental al Microprocesador 80286

1.2.1 Modos de operación

El 8086 y el 8088 solo operan en un modo. Sin embargo, el 80286 tiene dos modos de operación, llamado direccionamiento REAL y direccionamiento VIRTUAL PROTEGIDO, o simplemente modo real y modo virtual, respectivamente.

1.2.2 Modo de direccionamiento real

En el modo de direccionamiento real, el 80286 esencialmente opera como un 8086.
En este modo soporta todas las instrucciones del 8086 (más algunas propias), pero corre programas de dos a cinco veces más rápido que el 8086.
Cuando la computadora es activada el 80286 se inicializa en el

modo de direccionamiento real , este es sostenido hasta que un programa e plícitamente lo activa al modo de protección o virtual.

1.2.3 Modo de protección

En este modo, el 80286 ejecuta todo lo de el modo real, además de algunas características sofisticadas para la protección de memoria .
El aspecto de mayor significado en este modo es que permite al 80286 accesar una mayor cantidad de memoria , que en el modo real, utilizando la técnica llamada de "direccionamiento virtual".

1.2.4 Direccionamiento virtual

Con el direccionamiento virtual, el 80286 cuenta con dos tipos de direccionamiento de memoria:
-Un direccionamiento de área física , y un direccionamiento de área virtual.
El direccionamiento de área física es la cantidad de memoria principal con la cual el 80286 trabaja normalmente, mientras que el espacio virtual es la cantidad de memoria aprovechable por el computador al utilizar la memoria del disco duro como una ampliación de memoria RAM virtual.
El área de direccionamiento físico puede ser arriba de 16 millones de bytes (o 16 MB) de longitud, esto es 2^{24} bytes.
En cambio el área de direccionamiento virtual puede tener de un millon de bytes de longitud (2^{30}) o un gigabyte.
La memoria virtual es usada para programas que necesitan accesar un área de memoria que no tiene un espacio físico direccionable, como por ejemplo un sistema operativo, el 80286 permite esta capacidad simplemente cambiando a la sección correspondiente de memoria virtual.
Este cambio de memoria es invisible al programador, así se pueden escribir programas que libremente accesen todo el espacio de dirección posible , con la única limitante de un gigabyte.
En general el modo protegido virtual es la mejor salida para diseñadores de sistemas operativos de sistemas multi-usuarios ya que estos emplean una cantidad enorme de memoria que solo el 80286 puede suplir.

1.2.5 Registros internos

Internamente el microprocesador sostiene toda la información en grupos de cajas de 16 bits llamados registros,siendo 14 registros en total agrupados de la siguiente manera: 12 registros de datos y direcciones, más un apuntador de dirección de programa y un registro de estado o " flags ".

Los registros de datos y direcciones se pueden dividir en tres grandes grupos de cuatro registros, llamados registro de datos, registro apuntador indexado y registro segmentado. Los registros segmentados juegan un papel importante en el camino del procesamiento de ordenación de programas en la memoria principal siendo conocido este sistema como segmentación.

1.2.6 Segmentación

Como muchos microprocesadores, para referenciar la localización de una memoria solo se requiere un número, es decir la dirección de localización .
Con el 8086,8088, y el 80286 sin embargo, cada referencia a la memoria requiere de dos términos : un número segmentado y un offset.
La razón de esta dirección segmentada es porque estos microprocesadores necesitan que la instrucciones de programa y datos estén en diferentes segmentos,como ya se explicó para el caso del 8088.

1.3 Que es el 8088

El 8088 forma la Unidad Central de Proceso (CPU) en cualquier IBM PC o compatible, este chip es miembro de la serie de procesadores 8086 de la corporación INTEL. La familia 8086 está compuesta por los procesadores 8086,8088,80186,80286, y 80386. Cada uno de estos chips no solo implementan cualidades únicas que estan a parte de sus predecesores si no que también mantienen compatibilidad con su primeras versiones. Así, un 80186 puede hacer todo lo que hace un 8086 o un 8088, además de tener su propias y únicas operaciones. Además un 80286 puede hacer todo lo que hace un 80186 y un poco más. Lo mismo sucede para los otros MICROPROCESADORES
Al igual que todos los microprocesadores existentes, la estructura lógica de la familia 8088 se puede entender describiendo los registros que componen cada uno de estos procesadores. Pero para comprender la función de algunos de estos registros es necesario que primero se estudie la forma de como el 8088 direcciona la memoria.

1.3.1 Direccionamiento de memoria del 8088

Existe un problema inherente en el diseño del 8088: Hallar la mejor forma de representar un valor de dirección de 20 bits, utilizando registros de 16 bits del 8088. El 8088 tiene 20 pines para direcciones, lo cual permite un direccionamiento de 2 a la 20 (1,048,576, o 1 Mb) localizaciones de memoria, pero los registros tienen solamente un ancho de 16 bits.

La solución es dividir una dirección absoluta de memoria en "pedazos" que pueden ser almacenados individualmente en registros de 16 bits. Así, dos registros son utilizados para representar una sola dirección: uno de estos registros almacena una dirección base (**Registro segmento**), y el otro almacena un complemento de la dirección base. (**registro complemento**). Teóricamente, tal método puede generar 2 elevado a la 32 (más de 4 billones) de direcciones, lo que requeriría 32 líneas de direccionamiento para el MICROPROCESADOR lo cual va más alla de las capacidades del 8088.
En la práctica el esquema de direccionamiento desarrollado por Intel es el siguiente: Tomando en cuenta el contenido del **registro segmento** (el registro que contiene la dirección base de la dirección absoluta) como los 16 bit más significantes de la dirección absoluta; los cuatro bits menos significantes se asumen como cero. En otras palabras, el **registro segmento** contiene los cuatro dígitos hexadecimales más significantes de la dirección, y los menos significantes no contenidos en dicho registro (los de la derecha) son cero. Sumándole el contenido del **registro complemento** a ésta dirección base calculada resulta la dirección absoluta. La figura 1.1 muestra como se determina la dirección absoluta a partir de los valores del par de registros **segmento-complemento.**

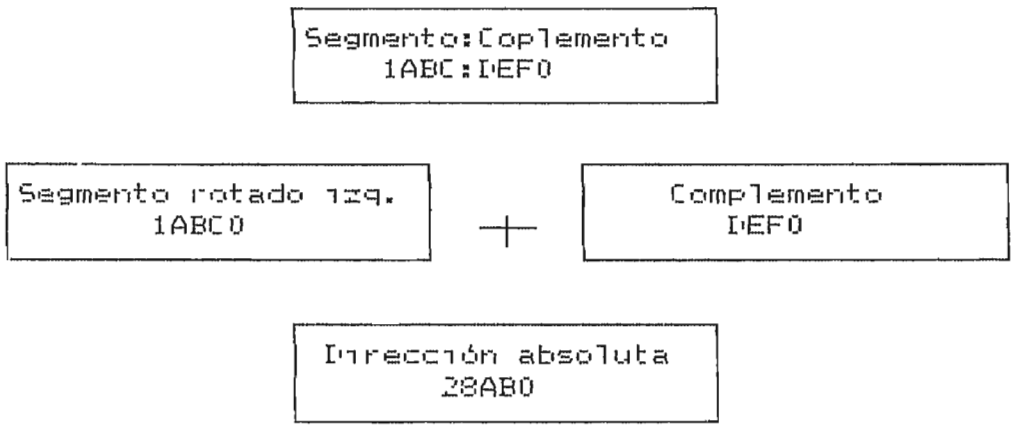


Figura 1.1 Direccionamiento de memoria

1.3.2 Registros del 8088

La familia 8088 usa 14 registros de 16 bits cada uno, los cuales pueden ser agrupados según la función que realizan, en cuatro categorías:

- Registros de propósito general
- Registros segmento (de dirección base)
- Registros de complemento

- Registros de banderas.

Los registros y su categorización se muestra en la tabla 1.1

Tabla 1.1 Conjunto de registros del 8088

Registro	categoría	Uso
AX	Propósito general	
BX	Propósito general	
CX	Propósito general	
DX	Propósito general	
CS	Segmento	Segmento de código
DS	Segmento	Segmento de datos
ES	Segmento	Segmento e tra
SS	Segmento	Segmento del stack
SP	Complemento	Puntero del stack
BP	Complemento	Puntero de base
SI	Complemento	Indice fuente
DI	Complemento	Indice Destino
IP	Complemento	Puntero a instrucción
Flags	Bandera	Estatus de banderas.

Cuando se tratan los registros individuales se está trabajando con el CPU a nivel del Hardware. Note que aunque esto es realizado por medio del lenguaje de ensamble, los lenguajes de alto nivel como BASIC, C, y Pascal, todos ellos tienen una forma conveniente de accesar los registros del 8088.
Como se mencionó anteriormente, los registros del 8088 pueden ser divididos de acuerdo a su propósito, en cuatro diferentes categorías. A continuación se examinará cada categoría y sus registros.

1.3.3 Registros de proposito general.

Como su nombre lo indica, los registros de propósito general son utilizados para propósitos tales como el almacenamiento de resultados u otras necesidades temporales. Cuando Se utilizan funciones del BIOS o del DOS, se deben de cargar estos registros con los valores que se necesitan para complementar la función. Siempre se incluirá un valor que representa a la función específica, así como otros parámetros que pueden ser necesitados. El número de parámetros varía de función a función, y será tema de muchas secciones en el presente trabajo a la hora de programar en lenguaje de ensamble. Similarmente, al retorno de una función del BIOS o el DOS los valores que pueden ser usados por un programa retornarán en los registros.

Los registros de propósito general son AX, BX, CX, y DX. Para facilitar el uso de valores de 8 y 16 bits, cada uno de estos registros puede ser direccionados como un par de registros de 8 bits. Los nombres de registros AL, AH, BL, BH, etc. son utilizados para direccionar los 8 bit más bajos o más altos (L y H significan Low y High, respectivamente). La figura 1.2 muestra ésta relación.

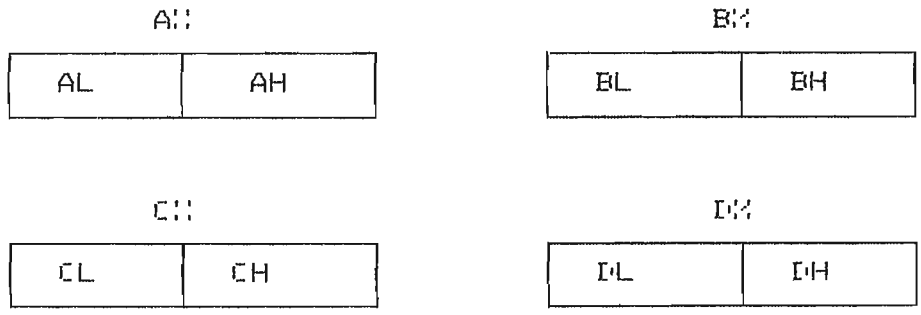


Figura 1.2 Los registros de 16 bits pueden ser direccionados como un par de registros de 8 bits.

Estos registros se utilizan grandemente en programación, siempre que se está trabajando en lenguaje de ensamble o utilizando desde lenguajes de alto nivel llamadas o funciones del DOS o del BIOS.

1.3.4 Registros de segmento

Los registros de segmento juegan un papel muy importante en el esquema de direccionamiento de memoria del 8088. Ellos almacenan valores de 16 bits que representan la dirección base de un segmento de 64 K de memoria. Estos valores representan los 16 bits más significantes de la dirección de 20 bits: los cuatro bits menos significantes se asume que son ceros. El hardware de direccionamiento de memoria del 8088 combina ésta dirección base con un valor complemento almacenado en un de los registros complemento del MICROPROCESADOR los cuales se discutirán más adelante.

Los registro segmento son:

1. El CS (code segment) Segmento de código
2. El DS (data segment) Segmento de datos
3. EL ES (e tra segment) Segmento e tra
4. El SS (stacl segment) Segmento de pila

Cada uno de estos registros de segmento especifica un segmento

distinto. Como programador, básicamente se es libre de utilizar estos registros segmento en la forma que se escoja, dentro de ciertos límites (descritos cortamente) más adelante.

Los registros de segmento estan diseñados para ser utilizados de la siguiente manera:

-CS Sostiene la dirección base del código que se está ejecutanto en ese momento.

-DS Sostiene la dirección base de los datos del programa.

-ES Suplementa al registro DS, sosteniento la dirección base de un segmento "e tra" utilizado para datos.

-SS Sostiene la dirección base para la pila (el stack) del programa. la cual es utilizada para el almacenamiento temporal de datos.

Los límites previamente mencionados en el uso de los registros segmentos incluyen la restricción en el uso de los registros CS y SS. En general para operar propiamente, el 8088 espera que el registro CS siempre apunte al segmento del programa que se está ejecutanto en ese momento y que el registro SS siempre apunte a el stack (pila).

1.3.5 Registros de complemento

Como su nombre lo implica, los registros de complemento generalmente son utilizados como el complemento de una parte de memoria direccionada. Los cinco registros complemento son:

-SP (stack pointer) puntero del stack

-BP (base pointer) puntero base

-SI (source index) índice fuente
-DI (destination inde) índice destino

-IP (instruction poiter) puntero de instrucciones

A acausa de que los registros en este grupo difieren en sus usos comunes, frecuentemente son divididos en dos clases separadas: Registros de puntero y registros índice.

1.3.6 Registro de puntero.

Los registros de puntero (SP y BP) suministran un forma conveniente de accesar valores actuales del segmento del Stack.

El SP siempre apunta al tope del stack y es actualizado automáticamente por varias instrucciones del lenguaje de ensamble. el otro registro de puntero, BP, típicamente es utilizado como un puntero base (o referencia) para otras operaciones indexadas. Por ejemplo, algunos programadores utilizan el registro BP para apuntar a una posición fija dentro del Stack. Este puntero luego es utilizado como referencia para recuperar variables que fueron colocadas en el stack antes de que una subrutina fuera llamada.

El puntero de instrucciones IP sostiene el complemento de la dirección de la siguiente instrucción a ser ejecutada por el MICROPROCESADOR. Cuando el registro IP y el registro segmento de código CS se combinan apuntan a la dirección absoluta de la instrucción. el par de registros (CS:IP) siempre es utilizado de ésta manera.

1.3.7 Registros indices

Los registros índice, SI y DI, son registros de complemento especializados. Típicamente, SI y DI son utilizados en unión con los registros segmento DS y ES. En operaciones con strings, por ejemplo, se puede utilizar DS:SI para apuntar a la dirección del string fuente y ES:DI para apuntar al string destino. En operaciones que no tienen que ver con strings, los programadores generalmente utilizan los registros SI y DI para lo que su nombre implica- un índice(complemento) para un dato fuente o para un dato destino.

1.3.8 Registros de banderas.

Los registros de banderas utilizan 9 de 16 bits como banderas que reflejan el estado del MICROPROCESADOR o como control de operaciones. Estas banderas son divididas en dos categorías:
Banderas de estado y banderas de control
Las banderas de estado son:

- CF (Carry flag) bandera de acarreo
- PF (parity flag) bandera de paridad
- AF (Auxiliarity carry flag) Bandera auxiliar de acarreo
- ZF (Zero flag) Indica si la operación fue cero
- OF (Overflow flag) Bandera de rebalse
- SF (Sign flag) Bandera de signo

Estas banderas reportan el estado de la última instrucción ejecutada. Si la ultima instrucción genera un valor de cero, la bandera de cero es puesta en uno. Las bandera de estado son puestas y borradas automáticamente, pero los programadores también pueden manipularlas. Muchas de las rutinas del DOS y del BIOS utilizan la bandera de carry como señal de error.

Las banderas de control son.

- DF (direction flag)
- TF (Trap flag)
- IF (interrupt flag)

La bandera DF controla ciertos aspectos de las instrucciones del 8088 para copiar rangos de memoria, y la bandera TF pone al MICROPROCESADOR en modo de "un solo paso", la bandera IF habilita y deshabilita las interrupciones.

1.4 Los procesadores de proposito general 8086 y 8088

1.4.1 Arquitectura

El 8086 y el 8088 son microprocesadores de 16 bits de propósito general de Intel. Al igual que otros muchos fabricantes/diseñadores de chips microprocesadores de ésta nueva generación, Intel ofrece varias versiones de sus productos. El 8086 es un microprocesador de 16 bits, tanto en lo que se refiere a su estructura como a sus conexiones externas, mientras que el 8088 es un procesador de 8 bits que internamente es idéntico al procesador de 16 bits 8086 (ver fig 1.3 y 1.4). En este capítulo se comentará las características más importantes de estos dos procesadores y cómo el 8088 de bus restringido se acopla en el esquema general del resto de microprocesadores de Intel.
Aunque Intel fue la primera en diseñar y fabricar los chips 8086 y 8088, hay otros fabricantes que producen su propia versión idéntica de estos chips. Estos fabricantes reciben el nombre de segundas fuentes . Actualmente, Intel está adelantado a estos fabricantes en el desarrollo de estas versiones idénticas. La razón es que algunos consumidores de chips, no diseñan ni construyen nada que utilize un chip del que solo exista un proveedor. El 8086 y 8088 están siendo fabricados por gran número de empresas (la mayoría Japonesas), incluyendo a Fujitsu. Intel, de hecho, garantiza a Fujitsu una licencia a nivel mundial para fabricar y vender los chips 8086, 8088 y 8089. La licencia no es exclusiva y no implica ningún intercambio de dinero entre las dos compañías.El resto de fabricantes de estos chips (incluyendo compañias como SIEMENS, AMD, y NEC) operan sin ninguna licencia especial, pero con el beneplácito de Intel.
El 8086 y el 8088 son prácticamente idénticos excepto por el tamaño de su bus de datos externo. Intel trata ésta igualdad interna y desigualdad externa, dividendo cada procesador 8086 y 8088 en dos sub-procesadores . O sea, cada uno consta de una unidad de ejecución (EU:Execution Unit) y una unidad interfaz de bus (BIU:Bus Interface Unit). La unidad de ejecución está encargada de realizar todas las operaciones mientras que la unidad de interfaz de bus está encargada de acceder datos e instrucciones del mundo exterior. Las unidades de ejecución son

idénticas en ambos procesadores, pero las unidades de interfaz de bus son diferentes en varias cuestiones que se tratarán más adelante. Esta aproximación es un ejemplo claro de diseño modular. Esto es, el todo (el procesador) se divide en partes (los dos sub-procesadores), y cada parte o módulo forma una unidad de trabajo encargada de ciertas subtareas. Esto es un principio fundamental en la teoría moderna de diseño del hardware y software, así como también lo es en otros temas que no tienen nada que ver con computadoras. En este caso, la ventaja de ésta aproximación modular es el ahorro de esfuerzo necesario para producir el chip 8088. Sólo una mitad del 8086 (el BIU) no debe diseñarse para producir el 8088.

1.4.2 ¿ Porque el 8088 ?

Como se ha comentado hasta ahora, el 8088 tiene un bus de datos a término reducido (8 bits). La razón para crear el 8088 con este bus de datos reducido es la de proveer la continuidad entre el 8086 y los antiguos procesadores de 8 bits de Intel, el 8080 y el 8085. Esta continuidad es especialmente importante para aquellos que han desarrollado investigaciones en estos tipos de productos. Teniendo el mismo tamaño de buses (así como requerimientos similares de control y tiempo), el 8088, que es internamente un procesador de 16 bits, puede reemplazar a esos primeros procesadores de 8 bits en un sistema ya existente. Esto mejorará el rendimiento del sistema a un costo muy bajo. Por ejemplo, hay varias placas de CPU 8088 disponibles en la actualidad para el bus S-100. Si ya se tiene un sistema con un bus S-100, el coste de pasarse a uno de 16 bits (internamente) se limita justamente a una nueva placa de CPU (de 300 a $500) y un nuevo software que, incidentalmente, está llegando a ser más caro que el hardware.
Otra ventaja del 8088 es que los nuevos sistemas, pequeños y baratos aunque muy potentes , pueden diseñarse basándose en este chip. La buena razón precio/rendimiento de los sistemas basados en el 8088, hará que sea éste el elegido por todos aquellos que deseen diseñar computadoras fáciles de vender.
El primer procesador de 8 bits barato fue el 8080, que llegó a ser uno de los favoritos de los diseñadores, por encima de otros como el 6502 y el 6800 que salieron al mercado casi al mismo tiempo y eran incluso más baratos. Se produjo una gran cantidad de hardware y software para sistemas basados en el 8080. sin embargo, el 8080 tenía ciertas desventajas, como el requerir tres tensiones de alimentación y dos señales de reloj diferentes. Más tarde se diseño el 8085 para soslayar estos problemas. El 8085 requería únicamente una tensión de alimentación y producía sus propias señales de reloj (hay terminales que permiten adaptar el cristal oscilador). Además, tenía una velocidad más alta que el 8080. Debido a estas mejoras, el 8085 no es directamente compatible, desde el punto de vista hardware, con el 8080, pero sí es, en un 99 por 100 de los casos, compatible desde el punto

de vista software(La diferencia radica en dos instrucciones, la RIM (Read Interrupt Mask , leer máscara de interrupción) y la SIM (Set Interrup Mask , Activar máscara de interrupción) , incluidas en el juego de instrucciones del 8085, pero no en el 8080. Además de sus funciones de lectura y activación de las máscaras de interrupción, estas instrucciones también pueden usarse para recibir y enviar datos bit a bit, en serie, por una línea especial de datos del 8085).

1.4.3 Similitudes del 8088 y del 8085

El 8088 tiene muchas señales en común con el 8085, particularmente las asociadas con la forma en que los datos y las direcciones están multiplexadas , aunque el 8088 no produce sus propias señales de reloj tal como lo hace el 8085. El 8088 y el 8085 siguen el mismo esquema de compartir los terminales correspondientes a los 8 bits más bajos del bus de direcciones con los 8 bits del bus de datos, de manera que bastan 16 terminales para direcciones y datos, en vez de los 24 previsibles.
El 8085 y el 8088 pueden, de hecho, dirigir directamente los mismos chips controladores de periféricos. Las investigaciones hardware para sistemas basados en el 8080 o el 8085 son, en su mayoría, aplicables a sistemas basados en el 8088.
Por supuesto que habrá problemas con la velocidad del sistema antiguo, pero siempre se puede bajar la velocidad del 8088 instalándole el cristal oscilador apropiado. Dado que el juego de instrucciones del 8086 y 8088 tiene una capacidad mayor que el 8080 y 8085, la compatibilidad del software está lejos de ser directa. Los registros del 8008/8086 y la mayoría de instrucciones del 8088/8085 pueden ser traducidas directamente a instrucciones del 8086/8088. Esto sin embargo, es mejor hacerlo en lenguajes fuente que en lenguaje máquina, ya que éste es enteramente diferente de uno a otro. Intel ofrece un programa traductor de fuente 8080/8085 a fuente 8086/8088. Este programa debe realizar la traducción en varios pasos (vease la fig. 1.5). entre ellos están:
1) Traducir el nombre de los registros: los registros de 8 bits A,B,C,D,E,H Y L, pasan a ser AL, CH, CL, DH, DL, BH, Y BL, respectivamente.
2) Traducir los indicadores: por ejemplo, S, Z, AC, P y CY pasan a ser SF, ZF, AF, PF y CF, respectivamente .
3) Traducir las instrucciones: por ejemplo, MOV, MVI, LXI, LDA y STA pasan a ser, todas ellas, variaciones de las instrucciones MOV.
4) Traducir los operandos: por ejemplo, un operando de 16 bits de la instrucción LXI del 8080/8085 pasa a ser una constante: un operando de 8 bits de la instrucción LDA, o uno de 16 bits de la instrucción LHLD del 8080/8085 pasan a ser una variable.
5) traducir las pseudoinstrucciones del ensamblador. La entrada

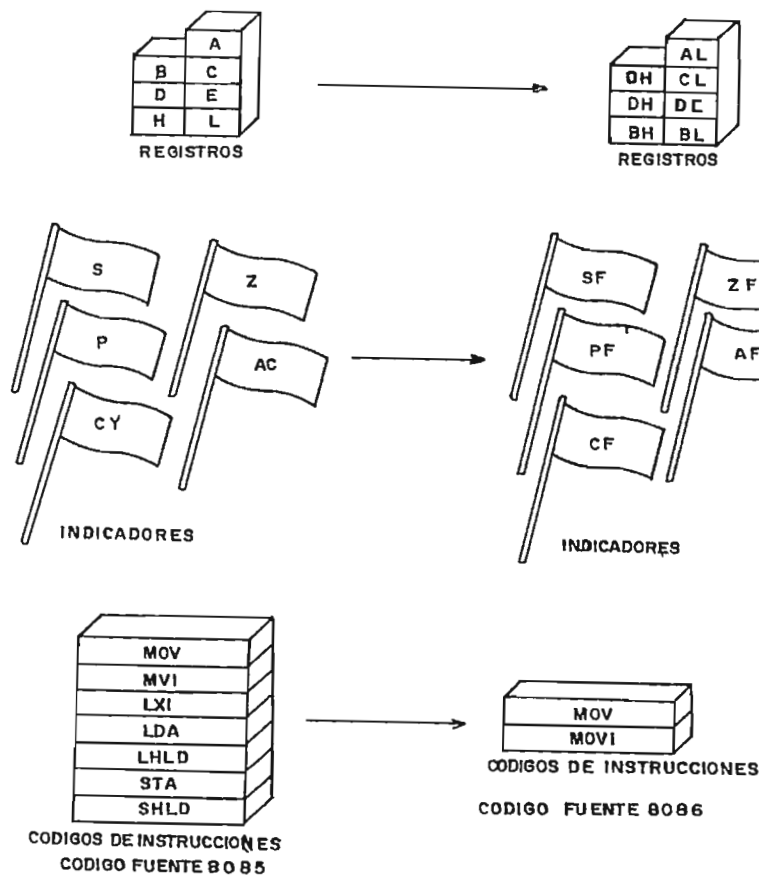


FIGURA I 5 TRADUCCION DEL CODIGO FUENTE 8085 A 8086

El programa traductor de Intel debe ser un código fuente compatible con el ensamblador Intel del 8085: la salida que dará el programa será un código fuente compatible con el ensamblador Intel del 8086. Normalmente, los códigos fuentes del 8085 y 8086 son compatibles con el ensamblador Intel. Sin embargo, existen fuentes que no lo son, como el ensamblador de Microsoft para el 8086, que usa nombres diferentes en algunas instrucciones y la gestión de los tipos de datos es diferente a la de Intel. Afortunadamente, Microsoft está creando un programa traductor para estas fuentes que pronto estará disponible.

1.5 Principales caracteristicas de la CPU del 8086/8088 :visión general

En esta sección se verán las principales características de los chips de CPU 8086 y 8088. En las siguientes secciones se estudiarán mas a fondo tópicos que veremos de forma general en esta sección, tales como las formas de direccionamiento, el juego de registros y el juego de instrucciones.

1.5.1 Capacidad de direccionamiento

Tanto el 8086 como el 8088 tienen un bus de direccionamiento de 20 bits de amplitud, lo que les provee la capacidad de direccionar un megabyte de memoria. Esto es porque:

2^{20} = 1.048.576 = 1 megabyte

Sin embargo, el registro de direccionamiento del 8086/8088, tiene únicamente una amplitud de 16 bits. Esto es equivalente a 64k bytes . Este procesador usa un método llamado segmentación ya antes descrito, para permitir el direccionamiento a todo el megabyte de memoria.
Más tarde, en este mismo capítulo, se verá como el 8086/8088 realiza la segmentación. Zilog tiene dos versiones del Z8000, una con segmentación y otra sin ella. La versión sin segmentación tiene únicamente 16 bits para direccionar, mientras que la versión con segmentación permite acceder a memoria con 24 bits de direccionamiento (16 megabytes), pero requiere el uso de un chip especial llamado Unidad de Gestión de Memoria (MMU:Memory Management Unit). El Motorola MC 68000 tiene 32 bits en sus registros de direccionamiento , pero solo envía 24 bits al mundo exterior . Obviamente, está diseñado para futuras expansiones, y al igual que el Z8000, el 68000 está diseñado para trabajar con una unidad de gestión de memoria. El 8086/8088 no necesita de ningun chip adicional para realizar la segmentación, y otras versiones (el iAPX 286) se han diseñado con un sofisticado esquema de gestión de memoria incorporado en conjunción con un direccionamiento de 24 bits. El 8088/8086 tiene

otra memoria separada, llamada espacio de E/S (vease la fig. 1.6), que puede considerarse como una memoria extra para direccionar en la cual se encuentran los dispositivos de E/S (direcciones cableadas). En la Motorola MC68000 no existe espacio de E/S: todos los dispositivos aparecen en la CPU localizados en la memoria. En el 8086/8088 (y en el Z8000), los dispositivos pueden conectarse a la memoria principal o al espacio de E/S. Asimismo, la memoria puede conectarse a la memoria principal o al espacio de E/S. El espacio de E/S del 8086/8088 usa un direccionamiento de 16 bits (permite direccionar 64k bytes). Los anteriores procesadores de ocho bits de Intel tenían únicamente un direccionamiento de 8 bits para el espacio de E/S, lo que permitia direccionar 256 bytes de dicho espacio. Esto es, realmente, una cantidad muy pequeña comparada con 64k. El direccionamiento actual de 16 bits, posibilita a Intel el poner muchas cosas en el espacio de E/S, incluyendo entre ellas, la memoria asociada para el IOP-8089, asi como el controlador de dispositivos de E/S. Se puede pensar que al poner toda la E/S en el espacio de E/S (incluyendo el IOP-8089 y todos los controladores de dispositivos que se comunican con él) se elimina gran cantidad del tráfico entre la CPU y la memoria; pero no es cierto, ya que tanto la memoria principal como el espacio de E/S usan las mismas lineas de direcciones y lineas de datos. Solo el uso de señales de control separa la memoria y el espacio de E/S. A pesar de ello es posible emplazar los controladores en buses separados, los cuales se conectarían al bus del procesador principal, con lo cual se reduciría el tráfico. Esto requeriría usar un chip de interface de bus.

1.5.2 Juego de registros

El juego de registro del 8086 y del 8088 son exactamente iguales, con 14 registros internos de 16 bits. Cada registro tiene su propia personalidad, aunque varios comparten tareas comunes. El juego de registro del 8086/8088 es una ampliación del juego de registro del 8080, que a su vez es una ampliación del juego de registro del 8008. En la figura 1.8 se puede comparar el juego de registro del 8008, 8080 y 8086.

1.5.3 Modalidades de direccionamiento

El 8086/8088 tienen 25 modalidades de direccionamiento diferentes; una modalidad de direccionamiento es un conjunto de reglas que especifican la localización (posición) de un dato usado durante la ejecución de una instrucción. En la modalidad más sencilla, un dato se localiza en un registro determinado; en la modalidad más compleja, se suma el contenido de dos registros en una cantidad de 8 ó 16 bits, que se encuentra en el programa.El resultado de la suma indica la dirección del dato. Se desarrollarán las modalidades de direccionamiento, más

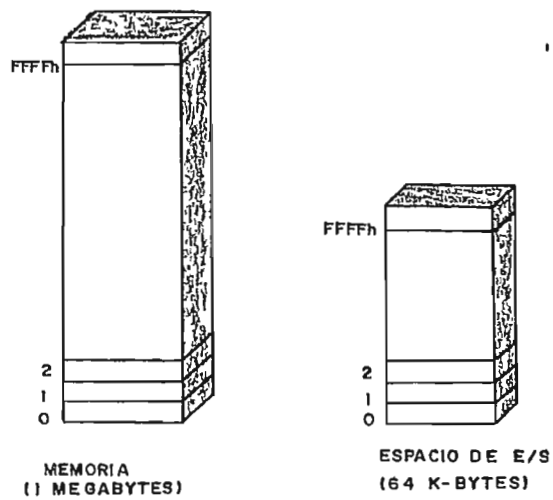


FIGURA 1.6 MEMORIA Y ESPACIO E/S

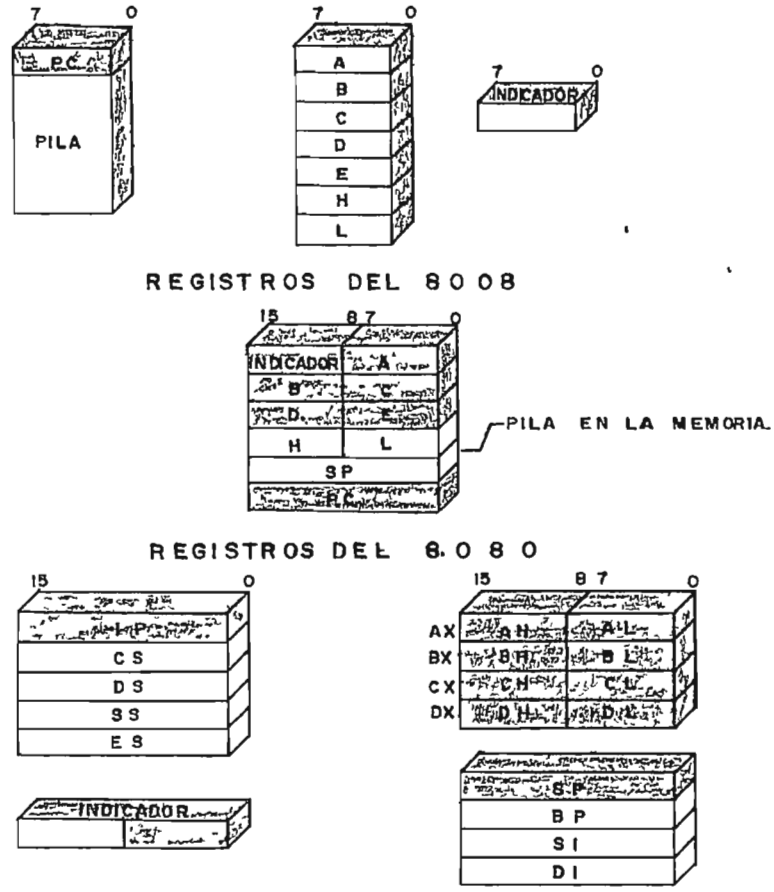


FIGURA 17 REGISTROS DEL 8008,8080,8086,8088.

adelante en ésta misma sección.

1.5.4 Señales de reloj

Al igual que los más recientes microprocesadores, el 8086/8088 requiere una única señal de reloj. Al contrario que el 8085, estos procesadores no generan su propia señal de reloj, dependiendo del generador de reloj 8284, que usa un cristal oscilador para determinar la frecuencia de señal. Intercambiando éste cristal, se puede seleccionar diferentes velocidades de operación. Intel tiene una versión de 5 MHz y otra de 8 MHz para el 8086 y el 8088. Estas versiones representan las velocidades más altas, recomendadas para estos chips. No se recomienda bajar de 2 MHz para ninguno de los dos. Dependiendo de su chip particular, una velocidad más baja o más alta puede ser correcta o puede causar error. La decisión de la velocidad a adoptar solo puede ser suyo.
Para un rendimiento óptimo, el 8086/8088 requiere una señal de reloj que se mantenga en nivel alto una tercera parte del tiempo total del ciclo. Esto significa que el reloj está activo una tercera parte del tiempo y desactivado las dos terceras partes del tiempo (vease la figura 1.8).

1.5.5 Requisitos de potencia

Tal como es estandar, el 8086/8088 requiere una alimentación de 5 voltios. Cualquier Intel de la serie 8000 relacionado con un sistema 8086/8088 tiene el mismo tipo de requerimiento.

1.5.6 Encapsulado

Los chips microprocesadores de 16 bits generalmente vienen encapsulados con 40, 48 ó 64 terminales. Tanto el 8086 como el 8088 vienen encapsulados con 40 terminales. Tanto Zilog como National Semiconductor tienen versiones de sus productos con 40 y 48 terminales: el MC68000 de Motorola viene encapsulado con 64 terminales. Sin embargo, en determinadas configuraciones, el 8086 de Intel puede producir más señales que el MC68000 a pesar de sus 64 terminales. El secreto para poderlo lograr está en la técnica de multiplexado en el tiempo y la de codificación: un recordatorio de los conceptos básicos es:
Multiplexado en el tiempo significa usar el mismo conjunto de líneas, pero en períodos de tiempo distintos, para enviar conjuntos de señales diferentes. Codificación significa convertir un conjunto de estados posibles en números, y enviar estos números por unas pocas líneas en vez de usar una línea para cada estado diferente. (Por ejemplo, ocho estados pueden ser codificados con números binarios del 0 al 7, los cuales pueden enviarse con tres únicas líneas.)

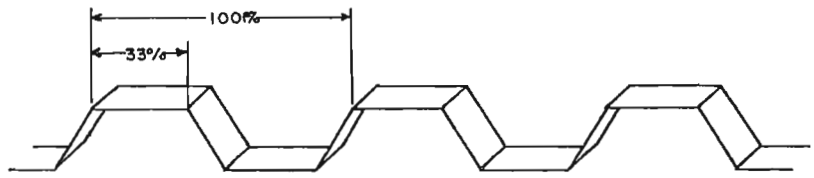


FIGURA I 8 SENALES DE RELOJ EN EL 8086/8088

1.5.7 Multiproceso y procesamiento en paralelo

El 8086/8088 tiene unas características especiales, que permiten coordinar sus actividades con otros procesadores en un conte to de Multiproceso y de Procesamiento en paralelo. El procesamiento (tratamiento) en paralelo es un sistema en el cual dos o más procesadores trabajan en tándem en la misma porción de un programa. Multiproceso es un sistema en el cual dos más procesadores trabajan en diferentes programas, pero comparten la misma memoria. El NDP 8087 utiliza el procesamiento en paralelo y el IOP 8089 utiliza el multiproceso.

1.5.8 Juego de instrucciones (generalidades)

El juego de instrucciones del 8086/8088 de Intel es comparable al de los otros líderes del mercado: El MC68000 de Motorola y el Z8000 de Zilog. Los tres fabricantes han adoptado un amplio juego de instrucciones haciendo que sus máquinas de 16 bits sean mucho más potentes que sus equivalentes de 8 bits. Han añadido operaciones como la multiplicación y la división, y han incrementado la eficacia de las antiguas operaciones como la de saltar a una dirección calculada en el programa. Es importante realizar que los procesadores de Zilog y Motorola son, realmente, en muchos aspectos, maquinas de 32 bits encapsuladas en chips de 16 bits. El 8086 de 16 bits y el 8088 de 8 bits pueden compararse de forma favorable, tanto en velocidad como en flexibilidad, con el Motorola o Zilog, en cuanto a las operaciones estándar de 8 y 16 bits. Sin embargo, por motivos como ser más lento en operaciones de multiplicar y dividir y la laguna de las operaciones directas con bits, el 8086, generalmente, se encuentra detrás en pruebas de velocidad con respecto al MC68000 y el Z8000. De todas formas, esto ocurre únicamente cuando en las pruebas no se incluyen los procesadores periféricos, el procesador de E/S 8089 y el procesador de datos numéricos 8087. Con estos periféricos el Intel 8086 forman una combinación mucho más potente que cualquiera de los otros dos trabajando solos. Además, el 186, siguiente versión de 8086, se ha diseñado para aumentar las velocidades de las instrucciones del 8086.

1.5.9 Arquitectura tubular (Pipeline)

Dado que el bus de datos del 8086 es el doble de ancho que del 8088, se podría esperar que el 8086 fuera el doble de rápido que el 8088. Esto no es verdad, por varias razones. Una razón es que muchas aplicaciones necesitan transferir datos en 8 bits. Otra es que el procesador tiene que hacer bastantes más cosas que transferir datos. La razón más importante tiene que ver con algunas características del diseño que ya se han descrito anteriormente, a saber, el procesador interno dual y la cola de instrucciones de estructura tubular (Pipeline).

Intel diseñó el 8086/8088 para realizar al mismo tiempo las principales funciones internos de transferencias de datos y búsqueda de instrucciones. Para conseguirla, tanto el 8086 como el 8088 constan de dos procesadores interconectados en la misma pieza de silicio (tal como se muestra en la figura 1.9). Una unidad está encargada de buscar instrucciones y la otra de ejecutarla. Además, la unidad encargada de buscar instrucciones utiliza un método llamado de estructura tubular (pipeline) o pon cola para almacenar nuevas instrucciones hasta que se necesiten. Al procesador principal se le llama unidad de ejecución (EU: Execution Unit). Está encargado de codificar y ejecutar todas las instrucciones. La EU es idéntica en ambos chips, el 8086 y el 8088. Al otro procesador se le llama la Unidad de Interfaz de Bus (BIU: Bus Interface Unit). La BIU está encargada de localizar las instrucciones y de transferir todos los datos entre los registros y el mundo exterior. La BIU del 8088 es más compleja, ya que debe transferir datos entre el bus de datos interno de 16 bits y el bus de datos externo de 8 bits.
Cuando la BIU localiza en memoria un byte de código máquina, lo coloca en una línea de espera especial llamada cola de instrucciones . En el 8086 esta cola tiene una longitud de 6 bytes y el código máquina se almacena en memoria de 2 en 2 bytes; en el 8088 la cola de instrucciones tiene sólo 4 bytes de longitud y el código máquina se guarda de byte en byte. La división del trabajo entre la EU y la BIU ahorra un tiempo considerable y ayuda a que el rendimiento del 8088 de 8 bits sea más comparable al 8086 de 16 bits.
La situación es muy similar a la de un jefe que tiene una secretaria para coger el teléfono y para la correspondencia. La secretaria trabaja al mismo tiempo que el jefe hace otras cosas, liberándole de muchos detalles del mundo exterior. las llamadas que se acumulan para el jefe se tratan de una en una, y en el orden que prefiere el jefe.

1.6 Juego de registros

El 8086/8088 contiene 14 registros de 16 bits. Algunos pertenecen a la **EU** y otros a la **BIU**. Los de la EU se suele usar para direccionamiento.
Como puede verse en la fig 1.10 la EU tiene los siguientes registros:

- cuatro registros generales de 16 bits (AX,BX,CX,DX), que pueden subdividirse (y ser direccionados separadamente) en ocho registros de 8 bits (AH,AL,BH,BL,CH,CL,DH,y DL). En este caso la X representa extendido (16 bits); la H, alto y la L, bajo. A representa acumulador; B representa base; C, contador, y D, datos.

- cuatro registros punteros y de índice (SP,BP,SI y DI), los cuales no pueden subdividirse. SP es el puntero de pilas, BP es

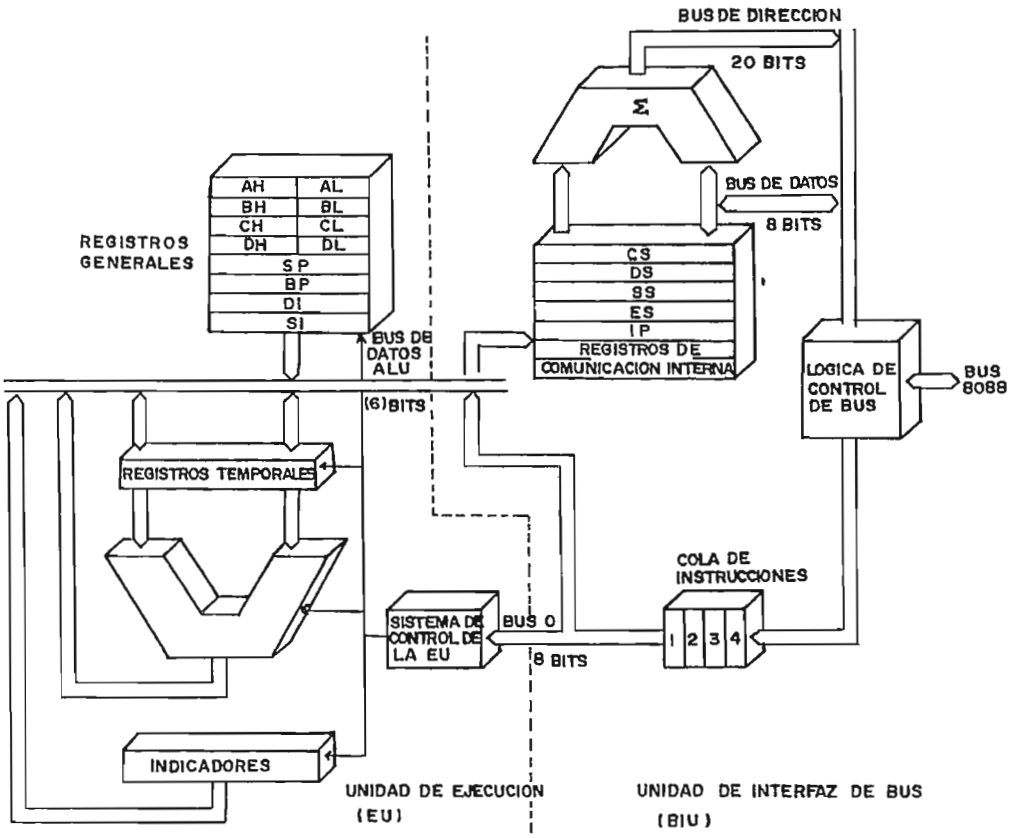


FIGURA 1 9 LA EU Y BIU EN EL 8086/8088

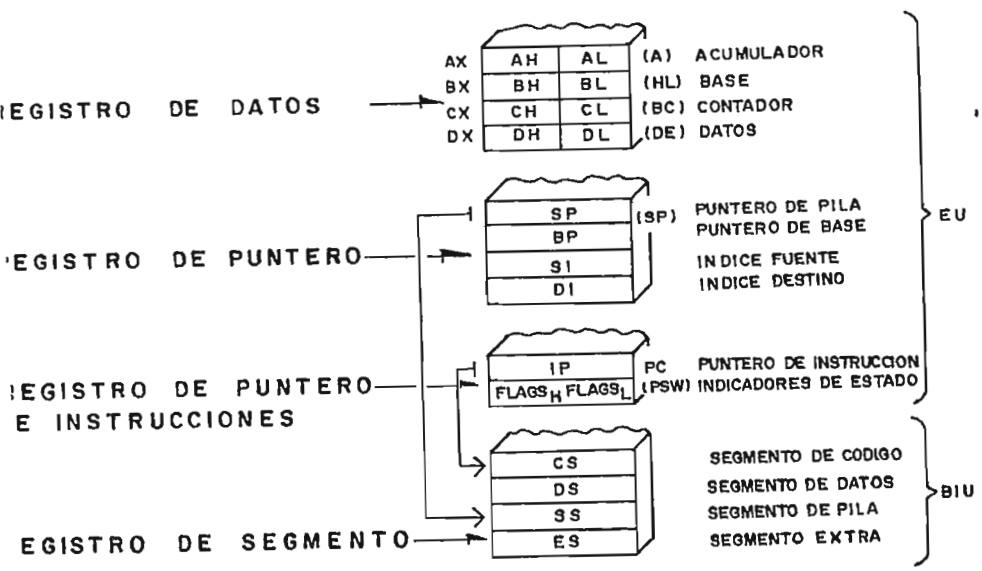


FIGURA 1.10 JUEGO DE REGISTROS DEL 8086/8088

el puntero base SI y DI indican a los registros índices y fuente respectivamente.

- Un registro de indicadores de 16 bits, que contiene varios bits de estado para el procesador. Estos incluyen : indicador de cero (ZF), indicador de signo (SF), indicador de paridad (PF), indicador de acarreo (CR), indicador auxiliar(AF), indicador de dirección(DF), indicador de interrupción (IF), indicador de desbordamiento (OF), indicador de desvío (TF) (ver fig 1.11)
La BIU tiene los restantes registros:

- Cuatro registros segmento (CS,DS,SS, y ES). Sus códigos representan a los registros segmento de código, datos, pila y extra, respectivamente.

- Un puntero de instrucciones (IP).
Intel no intenta conseguir un diseño simétrico para el juego de registros de sus microprocesadores, tal como lo hacen otros fabricantes de micros, minis y macro-computadoras. Cada registro tiene su propia personalidad. Esto era cierto para el 8008, el 8080 y el 8085, y es también cierto para el 8086/8088. Sin embargo, los registros del 8086/8088 son más potentes que los del 8080. Como se ve, cada registro tiene su nombre propio. Esto contrasta con otros muchos procesadores en los cuales es normal llamar a todo el juego de registros como R0, R1,..., R7, o D0,D1,...,D7.
Mientras muchas instrucciones como ADD, SUB,AND, y OR parecen tratar de forma similar a los registros AX,BX,CX, y DX, hay ciertos tipos de registros que producen un código máquina radicalmente diferente(normalmente más corto). Por ejemplo, al sumar datos inmediatos la mayoría de registros requiere 2 bytes de instrucciones más 1 ó 2 bytes para almacenar los datos. Sin embargo, sumar un dato inmediato al acumulador (AX) requiere únicamente 1 byte, más los necesarios para almacenar los datos. Por ello es más eficiente usar AX para cálculos en que intervengan constantes.
Hay también otras instrucciones que usan los registros. Por ejemplo, la instrucción LOOP usa el registro contador (CX) para almacenar la cuenta del número de iteraciones del bucle. La instrucción de ajuste decimal (DAA) únicamente trabaja con AL, que es la parte más baja del acumulador. Las instrucciones de multiplicar y dividir usan el registro acumulador (AX) y el de datos (DX).
"El registro AX es el acumulador de 16 bits. Usándolo a veces se provoca que el ensamblador produzca un lenguaje máquina codificado en muy pocos bytes. Su parte más baja, AL, corresponde al acumulador de 8 bits del 8080.
El registro CX se usa a menudo para almacenar datos, para contar cosas como lazos (para la instrucción LOOP), la iteración de movimientos de cadenas, desplazamientos y rotaciones (justamente CL para estos dos últimos tipos de instrucciones). El registro CX corresponde al registro par BC del 8080.

ZF = INDICADOR DE CERO

SF = " " " SIGNO

PF = " " " PARIDAD

CF = " " " ACARREO

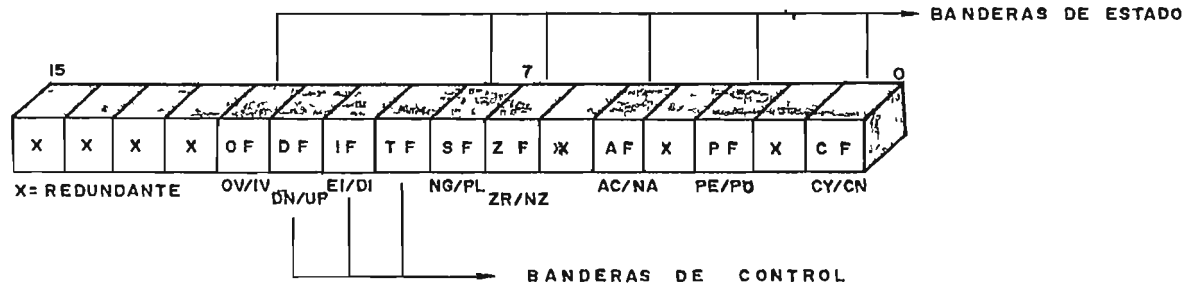
AF = " " " AUXILIAR

DF = " " " DIRECCION

IF = " " " INTERRUPCIONES

OF = " " " DESBORDAMIENTO

TF = " " " DESVIO (USADO CON EL DEBUG)



EL DESPLIEGUE DE BANDERAS EN LA PANTALLA EL DEBUG

OV/NV = OVERFLOW (YES/NO)

DN/UP = DIRECCION (DECREMENTO/INCREMENTO)

EI/DI = INTERRUPCION (ACTIVADA/DESACTIVADA)

NG/PL = SIGNO (NEGATIVO/POSITIVO)

ZR/NZ = CERO (YES/NO)

AC/NA = CARRY AUXILIAR (YES/NO)

PE/PO = PARIDAD (PAR/IMPAR)

CY/NC = CARRY (YES/NO)

FIGURA I II

REGISTRO DE INDICADORES EN EL 8086/8088

El registro DX se utiliza para almacenar datos de 16 bits. Puede pensarse que es una extensión del registro AX para multiplicaciones y divisiones con 16 bits. Es una especie de extensión del registro DE del 8080. Sin embargo, el registro DX del 8086 no puede utilizarse para direccionamiento indirecto tal y como lo hace el registro DE del 8080 con las instrucciones LDAX y STAX. Los registros SI y DI del 8086 son los encargados de realizar el direccionamiento indirecto. Esto es motivo de algunos problemas al traducir códigos del 8080 al 8086.
El registro BX (base de propósito general) se utiliza como registro base para los direccionamientos. El BX es una versión más potente del registro par HL del 8080. Los registros de propósito específicos como los de índice fuente (SI), índice destino (DI) y el puntero base (BP) se utilizan para ayudar en la localización de datos.
Los registros puntero de instrucciones (IP) y el puntero de pilas (SP) se encargan del control del flujo propio del programa. En realidad la mayoría de procesadores tiene registros de este tipo. Los registros segmento en el BIU tienen funciones especiales que se explicarán más adelante en este mismo capítulo en la sección de segmentación.
Es interesante comparar esta arquitectura con las del MC68000 y el Zilog Z8000. Estos dos procesadores tienen sistemas de numeración uniformes para sus registros. El Motorola MC68000 tiene registros de direcciones y registros de datos. Tales registros tiene 32 bits, y las instrucciones del Motorola los utilizan para guardar datos de 8, 16 y 32 bits. Por otro lado el Zilog Z8000 tiene también un bus de datos de 16 bits, pero sus registros son de 16 bits. Para guardar datos de 32 bits el Z8000 debe agrupar sus registros por pares, construyendo un registro de 32 bits a partir de dos de 16. Puede incluso formar registros de 64 bits agrupando cuatro registros de 16 bits. Esto es similar a la forma en la que los procesadores Intel 8080,8085, 8086 y 8088 forman un registro de 16 bits a partir de dos de 8 bits. En general el Z8000 y el 68000 tiene más registros y casi todos sus registros de datos admiten un uso general.

1.6.1 Modos de direccionamiento

El 8086/8088 tiene 25 modos diferentes de direccionamiento (véase tabla 1.2) o reglas para localizar un operando de una instrucción. Dichos modos son algo complicados, pero pueden verse como casos especiales de los casos tipos: referencia a registros y referencias a memorias. En el primer caso, el operando está localizado en un registro específico. Sin embargo, deben sumarse cuatro cantidades para obtener la dirección de un operando en memoria. Dichas cantidades son: 1) dirección de segmento, 2) dirección de base, 3) una cantidad índice y 4) un desplazamiento. Véase la figura 1.12 que muestra un caso general de direccionamiento a memoria.

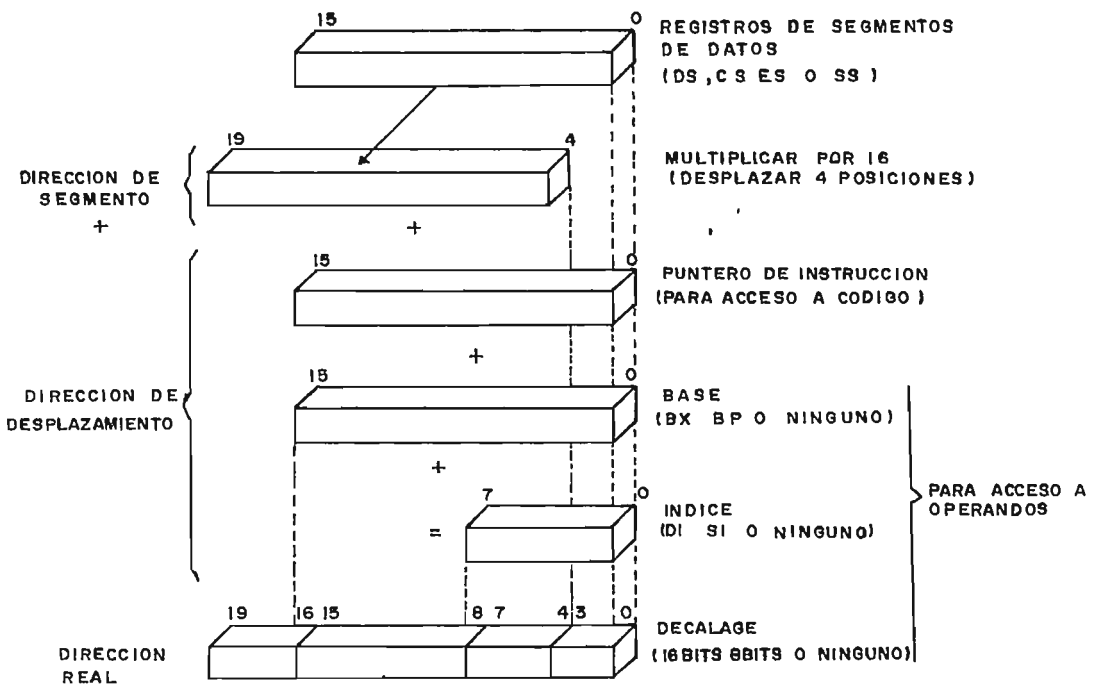


FIGURA I 12 GRAFICO DE MODOS DE DIRECCIONAMIENTO EN EL 8086

La dirección del segmento se almacena en el registro de segmentación (DS, ES, SS O CS); en la próxima sección se explicará la forma en que se hace esto. Por ahora, basta con saber que el contenido del registro de segmentación se multiplica por 16 (desplazado cuatro dígitos binarios a la derecha), antes de utilizarse para obtener la dirección real. El registro de segmentación siempre se usa para referenciar a memoria.

La base se almacena en el registro base (BX o BP). El Indice se almacena en el registro índice (SI O DI). Cualquiera de estas cantidades, las dos, o ninguna, pueden utilizarse para calcular la dirección real. El programador puede usar tanto la base como el índice de diferentes formas para gestionar ciertas cosas, tal como matrices de dos dimensiones, o estructuras internas a otra estructura, esquemas que se utilizan en las prácticas modernas de programación.

La base y el índice son variables dinámicas, ya que están almacenadas en registros de la CPU de propósito general. Es decir, pueden modificarse fácilmente mientras se ejecuta un programa. Los nombres de estas cantidades dan a entender que la base se modificará menos que el índice; pero esto queda realmente a criterio del programador.

Además del segmento, base e índice se usan unos decalages (desplazamientos) de 16 bits, o 0 bits (inexistente). Este desplazamiento es una cantidad estática. Se fija a tiempo de ensamblaje (paso de código fuente a código máquina) y no puede cambiarse durante la ejecución del programa (no debe cambiarse).

El desplazamiento se utiliza para cosas como compilar datos, organizar la memoria y reubicar más rápida y fácilmente datos. Por ejemplo, imagínese que se desea acceder a un byte variable que se encuentra siempre en la quinta posición de una tabla de 40 bytes, y suponga que dicha tabla se carga de memoria en distintos momentos, cada vez en una posición distinta (dependiendo de lo que haya en memoria en ese momento); para acceder a esta variable usted deseará utilizar un modo de direccionamiento con una base y un desplazamiento de 8 bits. La base será igual a la dirección de comienzo en curso de la tabla y el desplazamiento de 8 bits siempre será 5. Como la base se guarda en un registro base (normalmente el BX), se puede cambiar fácilmente cuando cambie la posición de la tabla. Además, como el 5 es una constante, se puede almacenar como desplazamiento estático.

En el lenguaje ensamblador, ciertos caracteres separadores colocados alrededor del operando, como paréntesis o corchetes, indican el modo de direccionamiento. Todo esto se traduce al lenguaje de máquina de formas más o menos complejas, dependiendo de la instrucción y del modo de direccionamiento.

TABLA 1.2 : MODOS DE DIRECCIONAMIENTO DEL 8086/8088

mm	aaa	Parte de desplazamiento de la dirección
00	000	(BX) + (SI)
00	001	(BX) + (DI)
00	010	(BP) + (SI)
00	011	(BP) + (DI)
00	100	(SI)
00	101	(DI)
00	110	Dirección Directa
00	111	(BX)
01	000	(BX) + (SI) + número 8 bits
01	001	(BX) + (DI) + número 8 bits
01	010	(BP) + (SI) + número 8 bits
01	011	(BP) + (DI) + número 8 bits
01	100	(SI) + número 8 bits
01	101	(DI) + número 8 bits
01	110	(BP) + número 8 bits
01	111	(BX) + número 8 bits
10	000	(BX) + (SI) + número 16 bits
10	001	(BX) + (DI) + número 16 bits
10	010	(BP) + (SI) + número 16 bits
10	011	(BP) + (DI) + número 16 bits
10	100	(SI) + número 16 bits
10	101	(DI) + número 16 bits
10	110	(BP) + número 16 bits
10	111	(BX) + número 16 bits
11	000	registro AX (palabra) o AL (byte)
11	001	registro CX (palabra) o CL (byte)
11	010	registro DX (palabra) o DL (byte)
11	011	registro BX (palabra) p BL (byte)
11	100	registro SP (palabra) o AH (byte)
11	101	registro BP (palabra) o CH (byte)
11	110	registro SI (palabra) o DH (byte)
11	111	registro DI (palabra) o BH (byte)

Nota: mm significa los 2 primeros bits del byte de direccionamiento y aaa los 3 últimos bits de dicho byte.

El siguiente ejemplo da una visión esquemática de la situación general, en este caso la instrucción es INC (incremental) y el único operando está en memoria. Se accede a él con un modo de direccionamiento que usa la base, el índice y un desplazamiento de 8 bits. En este caso, la dirección del segmento está en el segmento de datos, la base se encuentra en el registro base (BX), el índice está en el registro DI, y el desplazamiento es 6.
El lenguaje ensamblador, la instrucción aparecería así:

```
INC     6[BX][DI]       ; Incrementa
                        ; Contenido
                        ; de BX + DI + 6
```

Obsérvese que el comentario se refiere únicamente a la dirección relativa en el segmento. Hay dos razones para ello. La primera es que no hay mucho espacio para poner comentarios, y la segunda es que el programador únicamente debe pensar en términos relativos al segmento, ya que los diversos segmentos se determinan por medio de los registros de segmentación.
Considere ahora esta referencia en términos totalmente numéricos. Suponga que BX contiene 4.000 h (h = hexadecimal); DI contiene 20 h y DS contiene 3.000 h. El desplazamiento se determina de la siguiente forma:

```
desplazamiento  =  desplazamiento + (BX) + (DI) = 6 + 4000  +  20
                =  4.026 h
```

Como se ha visto en la sección de segmentación, el contenido del registro de segmentación (en este caso, registro de segmento de datos) se multiplica por 16 (decimal) antes de añadirse el resto. Dado que 16 en decimal es 10 en hexadecimal, esto produce un desplazamiento de un dígito hexadecimal (4 bits) a la izquierda. Combinando esto con la dirección relativa en el segmento, nos da:

```
Dirección real = 10 * (DS) + despl. = 3.000 * 10 + despl.
                                    = 30.000 + 4.026
                                    = 34.026h
```

Así 34.026h es la dirección real de memoria donde se encontraría está operando, mientras 4.026 es la dirección en el segmento, que es lo que interesa al programador. En este caso no interviene IP porque lo que se accesa es un dato.
Como se ha visto, este esquema de modo de direccionamiento, con su dúo de cantidades estáticas y dinámicas, se conjuga bien con la técnicas modernas de programación modular o la programación tipo caja-dentro-de-caja. El registro BX es una constante dentro de la caja exterior (módulo principal del programa), el registro DI es constante en la caja interior (último modelo de un programa), y el desplazamiento apunta a una dirección particular en la caja interior. Esto facilita el acceso a estructuras de datos de tipo caja-dentro-de-caja, tal como matrices de registros, o registro de matrices, o cualquier otra estructura que un lenguaje moderno como Pascal puede soportar o demandar.
(Ver Pascal Primer, por David Fo y Mitch Waite, Howard Sams.)
Estos modos de direccionamiento producen algunos inconvenientes en el 80P6/8088. La CPU gasta tiempo calculando una dirección compuesta de varias cantidades. Principalmente, esto se debe al hecho de que el cálculo de direcciones en el 8086/8088 esta programado en microcódigo. En la versión del 0086, la "AP" 186.

estos cálculos son cableados en la máquina y, por lo tanto, se emplea mucho menos tiempo el realizarlos.

1.6.2 Estructura de memoria de segmentación

Como se ha mencionado anteriormente, el 8086/8088 usa un esquema ingenioso llamado segmentación, para acceder correctamente a un megabyte completo de memoria, con referencias de direcciones de sólo 16 bits.
Considere cómo funciona. Cualquier dirección en el 8086/8088 tiene dos partes, cada una de las cuales es una cantidad de 16 bits.
Una parte es el desplazamiento (offset) y la otra la dirección de segmento.
El desplazamiento de 16 bits se compone de varias partes: un desplazamiento (un numero fijo), una base (almacenada en el registro base) y un índice (almacenado en el registro índice). La dirección del segmento se almacena en uno de los cuatro registros segmento (CS, DS, ES, SS). El procesador usa estas dos cantidades de 16 bits para calcular la dirección real de 20 bits, según la siguiente fórmula:

Dirección real = 16 * (dirección del segmento) + desplazamiento

Tal como se vió antes, dado que 16 en decimal es 10 en hexadecimal, multiplicar por 16 decimal es lo mismo que correr una posición a la izquierda un número hexadecimal. Por ejemplo, si la dirección del segmento es 500h y la del desplazamiento es 2341h, la dirección real será 5.2340h.
Considere con más detalle la dirección del segmento que está almacenada en uno de los cuatro registros de 16 bits llamados registros de segmentación.
Los registros de segmentación reciben los nombres siguientes: de códigos, datos, pila y extra (CS, DS, SS, ES, respectivamente).
Las instrucciones se gestionan usando el segmento código, las operaciones con pilas usan el segmento de pila, y para los datos se utilizan el segmento de datos y el extra.
Por ejemplo, supongamos que el procesador ejecuta un programa y el puntero de instrucciones (IP) contiene 234h y el registro de código contiene 800h. La siguiente fórmula indica que el siguiente byte de instrucción se encuentra en la dirección

8.234 = 800 * 10 + 234 (hexadecimal)

de memoria.
La dirección de segmento únicamente indica dónde se inicia un segmento. Pero, lo que no hace de ninguna forma es dividir el segmento en párrafos.
Hay un byte especial de prefijo que permite ignorar algunas de estas asignaciones, pero no todas. En particular, si la CPU usa la instrucción puntero para ayudarse a apuntar a memoria (os

decir, para localizar parte de una instrucción), entonces el registro de segmento de código siempre se usa. Si la CPU usa el puntero de pila para ayudarle a apuntar a memoria (tanto para introducir como para extraer de la pila), entonces se usará siempre el registro pila. El caso del destino en operación con cadenas es el único en el cual hay una restricción. La combinación del índice destino (DI) y el segmento extra (ES) se usa siempre para calcular la dirección del destino en cualquier operación con cadena. El byte prefijo puede usarse, sin embargo, para obligar a utilizar alguno de los cuatro registros segmento en el cálculo de la dirección puente de una dirección de cadenas. El valor por defecto (o sea, el valor sin byte prefijo) es el registro de datos.

Los 24 modos de referenciar la dirección (usados para el acceso de datos) pueden aceptar un byte prefijado e ignorar sus asignaciones por defecto al segmento, usando cualquier registro segmento. La asignación por defecto es o el segmento de datos (DS) o el de pila (SS) y, de hecho, se usa siempre el segmento de datos a menos que el modo de direccionamiento use el puntero base (BP), en cuyo caso se usa el registro de pila. Intel aconseja utilizar el puntero base para acceder a datos en la pila del sistema y normalmente con llamadas a subrutinas. Por ejemplo, antes de llamar a una subrutina, se pueden introducir varias, por ejemplo n,palabras de datos de 16 bits en la pila para que las use la subrutina en sus cálculos. Al principio de la subrutina se copiará el contenido del puntero de la pila en el puntero base, y en la subrutina usará el BP (con un desplazamiento) para acceder a cualquiera de estas palabras de datos. Véase la figura 1.14 para tener una idea gráfica de este funcionamiento. Se puede también devolver datos al programa principal desde la subrutina de forma similar. El uso de estos diferentes segmentos significa que hay áreas separadas para el programa, la pila, y los datos. Cada área de trabajo tiene una tamaño máximo de 64K bytes y mínimo de cero. Dado que hay cuatro registros de segmentación, uno de programa (CS), uno de pila (SS) y dos de datos -segmento de datos (DS) y segmento extra (ES)- el área de trabajo puede llegar hasta 4 64K = 256 K en un momento dado. Se supone que las distintas áreas no se superponen. Sin embargo, aunque no es necesario, es posible y aconsejable colocar los cuatro segmentos en la misma área. En este caso, programa, pila y datos residirían en la misma área de trabajo de 64 K . El programa puede determinar la posición de estos segmentos, cargando el apropiado registro de segmentación de 16 bits con la dirección del segmento apropiado. Esto se realiza normalmente al inicio del programa, pero se puede fácilmente hacer en cualquier momento mientras el programa se ejecuta, cambiando dinámicamente la dirección de estos segmentos.

Cambiando el desplazamiento, el programador puede acceder a cualquier punto del segmento. Piense en el segmento como una ampliación de memoria que forma un área de trabajo (ver fig 1.13). El desplazamiento es la única parte de la dirección que aparece normalmente en los programas en lenguaje ensamblador

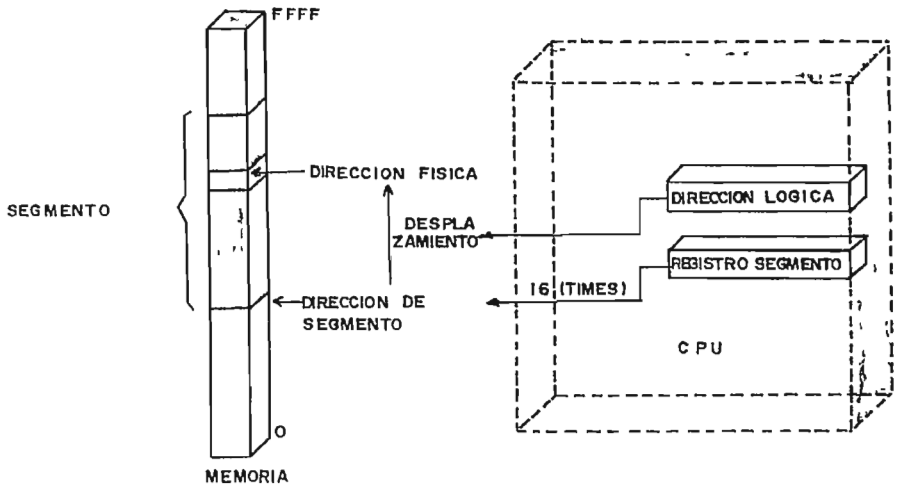


FIGURA I 13 UN SEGMENTO

durante las referencias a direcciones del área de trabajo. En la sección de modos de direccionamiento se vió como se calculaba dicho desplazamiento usando tres cantidades (base, índice y desplazamiento).
Se ha visto que la dirección del segmento puede ser cualquier número múltiplo de 16. Esto es particularmente sencillo en aritmética hexadecimal. Por ejemplo, en notación hexadecimal las direcciones iniciales de un segmento son 0h, 10h, 20h, 30h hasta FFF0h. Obsérvese que éstas direcciones ocurren en intervalos espaciados razonables y recorren toda la memoria hasta FFF0h, cubriendo 1 megabyte fácil y completamente con una perspectiva de 64K. Obsérvese que el contenido del registro segmento está desplazado un dígito hexadecimal o cuatro binarios a la izquierda, antes de sumarle el desplazamiento.La fig 1.14 muestra una visión gráfica de esto.
Hay cierta redundancia asociada con este esquema, pero tiene también sus ventajas. Una de las principales ventajas es que si se usa correctamente, fomenta una correcta programación. En particular, fomenta el desarrollo de pequeños módulos de código reubicable. Código reubicable es aquel que puede ejecutarse en cualquier posición de la memoria. Cada módulo puede diseñarse como si empezara en la posición cero. Cuando se ejecute, podrá cargarse con el inicio de cualquier posición que sea múltiplo de 16. Entonces, la dirección del segmento correspondiente se puede cargar en el segmento de código vía un salto intersegmento especial o llamada que comienza a ejecutar el código. Una vez este código se ha localizado, es muy fácil reubicarlo. De hecho el código máquina puede reubicarse dinámicamente trasladando el código (Multiplicándolo por 16) y modificando el contenido del segmento código en consonancia.
Esto es muy util en entornos de multiprogramación, donde diferentes trabajos o partes de trabajos tienen que cargarse y descargarse en diferentes posiciones de memoria ya que otros trabajos están donde estaban ellos. La mayoría de los nuevos sistemas operativos requieren esta habilidad para reubicar dinámica rápida secciones de código de tamaño pequeño y medio.

1.6.3 Juego de instrucciones del 8088/8086

En esta sección se hará una breve clasificación de las diferentes instrucciones del 8086/8088 por grupos y se finalizará comparando estas con las de las anteriores computadoras de 8 bits, y las de los otros procesadores de 16 bits. Se verá el aumento de potencia de los procesadores de 16 bits respecto a su predecesores de 8 bits, no sólo en el tamaño de los datos que pueden gestionar, sino también en sus otras características, como las nuevas operaciones de multiplicar y dividir, y la nuevas instrucciones para gestión de entornos de multi-proceso y de proceso en paralelo.
En general, las instrucciones del procesador de 16 bits se pueden clasificar en los siguientes grupos:**1)transferencia de dato, 2)**

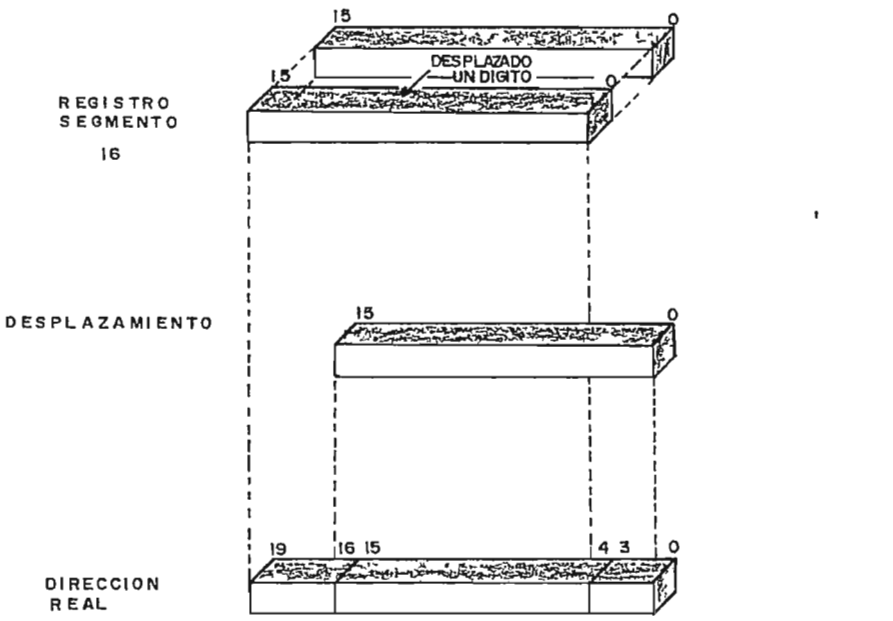


FIGURA 14 CALCULO DE LA DIRECCION.

aritmética entera binaria, 3) operaciones lógicas, 4) desplazamientos y rotaciones, 5) gestión de bits, 6) aritmética codificada en binario, 7) gestión de cadenas, 8) control del programa, 9) control del sistema.
Estas categorías son algo arbitrarias, pero pueden aplicarse de forma general a los tres tipos de procesadores de 16 bits actuales: el INTEL 8086/8088, el Zilog Z8000, y el Motorola MC68000.
Para las instrucciones del 8086/8088 se usarán los nemotécnicos de Microsoft dado que son probablemente los más populares, las cuales se aplicarán paso a paso en el capítulo 2.
Con estos nemotécticos muchas instrucciones indican (como parte de su código de operación) si los operandos son en bytes o palabras o si la fuente son datos inmediatos. Una B extra en el nemotécnico de la operación indica modo byte (datos de 8 bits), mientras que su ausencia indica modo de palabra (datos de 16 bits). Una I extra en el nemotécnico de la operación indica datos inmediatos en la fuente. Cuando la fuente es un dato inmediato de 8 bits, entonces aparecerán la B y la I (en este orden)en el nemotécnico de la operación. El uso de la B y la I como parte del **código de operación** no deberá de confundirse con los modos de direccionamiento (descritos en la sección previa) y la forma en que éstos afectan al operando.

1.6.4 Instrucciones de transferencia de datos

Las instrucciones de trasnferencia de datos son las encargadas de mover datos de un sitio a otro de la computadora como pueden ser la memoria, el espacio de E/S y los registros de la CPU. Las instrucciones más comunes en los programas escritos en lenguaje ensamblador son típicamente éstas de transferencia de datos, entre ellas se pueden mencionar:

MOV destino, fuente	Trasnfiere una palabra de la fuente al punto destino
MOVB destino, fuente	Transfiere un byte de la fuente al punto de destino
MOVI destino, dato	Transfiere el dato (inmediato) de la fuente al punto de destino
MOVBI destino, dato	Transfiere el dato(inmediato) al byte de destino
XCHG destino, fuente	Intercambia los contenidos de las palabras fuente y destino
XCHGB destino, fuente	Intercambia los contenidos de los bytes fuente y destino
PUSH fuente	Introduce la fuente en la pila
POP destino	Extrae un elemento de la fuente y lo lleva al punto de destino
IN fuente	Lleva a **AX** el contenido de la fuen-

	te (palabra)
INB Fuente	Lleva a AL el contenido de la fuente (byte)
IN	Lleva a AX la posición DX (palabra)
INB	Lleva a AL la posición DX (byte)
OUT destino	Lleva a AX al punto de destino (palabra)
OUTB destino	Lleva a AL al punto de destino (byte)
OUT	lleva a AX a la posición DX (palabra)
OUTB	Lleva a AL a la posición DX (byte)
XLAT	Traduce (utilizando una tabla)
LEA registro,fuente	Carga la dirección efectiva
LDS registro, fuente	carga DS y registro
LES registro,fuente	Carga ES y registro

La instrucción MOVE permite transferir datos de 8 bits (con el nemotécnico **MOVB**) o de 16 bits (**MOV**):1) de registro a registro, 2) de registro a memoria, 3) de memoria a registro, 4) un dato inmediato a registro y 5) un dato inmediato a memoria. En los casos 2),3),y 5) la referencia a memoria puede hacerse en cualquiera de los 24 modos de direccionamiento de la CPU. Los casos 1),2), y 3) utilizan nemotécnico **MOV** (para 16 bits) o **MOVB** (datos de 8 bits); y los casos 4),y 5) utilizan el nemotécnico **MOVI** (datos de 16 bits) o **MOBVI** (datos de de 8 bits).
Ejemplos de la instrucción **MOV**:

```
MOV  B[],CAT : equivalente a la LHLD
               CAT del 8080
MOVI B[],CAT : equivalente a la LXI H,
               CAT del 8080
MOVB AL,CAT : equivalente a la LDA CAT,
              del 8080
```

1.6.5 Aritmética binaria

Las operaciones en aritmética binaria entera permiten a la CPU realizar cálculos con números enteros positivos y negativos, estos últimos en complemento a dos. dichas instrucciones incluyen:

NEG destino	Hace negativo el número que encuentra en la posición de destino

NEGB destino	Hace negativo el byte de destino
ADD destino, fuente	Suma fuente y destino (palabras)
ADDB destino,fuente	Suma fuente y destino (bytes)
ADDI destino, dato	Suma el dato inmediato a destino (palabras)
ADDBI destino, dato	Suma el dato inmediato a destino (byte)
ADC destino, fuente	Suma la fuente + acarreo a destino (palabras)
ADCB destino, fuente	Suma la fuente + acarreo a destino (bytes)

1.6.6 Operaciones lógicas

Las operaciones lógicas se utilizán para poner a 1, borrar (poner a 0) y cambiar o eliminar bits individuales de la computadora. La ventaja principal de trabajar en lenguaje ensamblador es que se consigue un mejor control de la computadora, y estas operaciones pueden ser una gran ayuda en este aspecto.
Posibles aplicaciones pueden ser el eliminar bits de estado para la E/S y para el control del sistema, o poner a 1 cierto bits de control. Otro uso específico de estas instrucciones puede ser el desarrollo de un ensamblador. En este caso, los bits de control controlan la CPU .Entre las operaciones lógicas se pueden mencionar :

NOT destino	Niega (hace la función NOT) el destino (palabra)
NOTB destino	Niega (hace la función NOT) el destino (byte)
AND destino, fuente	Producto lógico (AND) de destino y fuente (palabra)
ANDB destino, fuente	Producto lógico de destino y fuente (byte)
ANDI destino dato	Producto lógico de destino y el dato inmediato (palabra)
ANDBI destino,dato	Producto lógico de destino y dato (byte)
OR destino, fuente	Suma lógica (OR) de destino y fuente (palabra)
ORB destino, fuente	Suma lógica (OR) de destino y fuente (byte)
ORI destino, dato	Suma lógica (OR) de destino y dato inmediato (palabra)
ORBI destino, dato	Suma lógica (OR) de destino y dato inmediato (byte)
XOR destino, fuente	XOR de destino y fuente (palabra)

XORB destino, fuente	XOR de destino y fuente (byte)
XORI destino, dato	XOR de destino y dato inmediato (palabra)
XORBI destino, dato.	XOR de destino y dato inmediato (byte)

1.6.7 Desplazamientos y rotaciones

Las operaciones de desplazamiento y rotación permiten mover bits a la izquierda o a la derecha en celdas de memoria (en memoria central o en registros de la CPU) de la computadora. No hay definiciones aceptadas universalmente para nombrar la gran cantidad de variaciones posibles de estas operaciones, pero los nemotécnicos del 8086/8088 constituyen una buena nomenclatura de estas instrucciones.
Básicamente, una rotación mueve (a izquierda o derecha) los bits de la palabra o byte de una forma circular, como si la palabra o byte se hubiera curvado hasta conectar el bit de más a la derecha con el de más a la izquierda; mientras que un desplazamiento mueve los bits de una forma lineal.En cualquiera de os casos, Intel ofrece dos variantes.
Para la rotación, hay una rotación pura, y otra que incluye el bit de acarreo. Para los desplazamientos, existe el desplazamiento lógico y el desplazamiento aritmético. Entre las instrucciones se mencionan las siguientes:

SHL destino	desplazamiento lógico a la izquierda (una posición)
SHL destino, CL	despalzamiento lógico a la izquierda (CL posiciones)
SHLB destino	desplazamiento lógico a la izquierda de un byte (una posición)
RORB destino,	rotación de un byte a la derecha (una posición)
RCRB destino	rotación a la derecha de un byte con acarreo incluido (una posición)

Ejemplos:

```
RORB  AL : equivalente a RRC del 8080
RCRB  AL : equivalente a RAR del 8080
```

1.6.8 Tratamientos de bits

No hay verdaderas operaciones de bits en el 8086/8088. Una operación de bits es aquella en la que se modofica (se pone a 1, se borra o se complementa) un bit explícitamente nombrado. Dichas operaciones se pueden simular con las instrucciones lógicas o con macros (instrucciones en lenguaje ensamblador definidas por el usuario) escribiendo instrucciones particulares de tratamiento de bits.

1.6.9 Aritmética decimal codificada en binario

Las operaciones de aritmética decimal codificada en binario incluyen:

AAA Ajuste ASCII para la suma **DAA** Ajuste decimal para la suma
AAS Ajuste ASCII para la resta **DAS** Ajuste decimal para la resta
AAM Ajuste ASCII para la mult. **AAD** Ajuste decimal para la div.

Estas operaciones se usan junto a las de suma, resta, multiplicación y división de números binarios, ADD, ADC, SUB, SBB, MUL, IMUL, e IDIV, para producir resultados en código decimal codificado en binario (BDC) empaquetado o desempaquetado.

1.6.10 Gestión de cadenas

Recuerde que una cadena es una secuencia de bytes o palabras. El 8086/8088 trabaja tanto en cadenas de bytes como de palabras. Técnicamente, una cadena es un tipo de dato que se guarda en un tipo de almacenamiento llamado bloque. Sin embargo, la distinción entre tipo de datos y tipos de almacenamiento es a veces confusa, y especialmente en este caso. Por ejemplo se puede hablar igualmente de mover una cadena (movimiento de cadena), como de mover el contenido de un bloque de memoria (transferencia de bloques).
Las operaciones de cadenas del 8086/8088 incluyen:

REP Repetir
MOVB Mover los caracteres de una cadena (bytes)
MOVW Mmover los caracteres de una cadena (palabras)
CMPB Comparar caracteres de una cadena (bytes)
CMPW Comparar caracteres de una cadena (palabras)
SCAB Buscar caracteres en una cadena
SCAW Buscar palabras en una cadena
LODB Introducir caracteres en una cadena
LODW Introducir palabras en una cadena
STOB Guardar caracteres en una cadena
STOW Guardar palabras de una cadena
CLD Borrar el indicador de dirección
STD Poner a 1 el indicador de dirección

Estas instrucciones de tratamiento de cadenas resultan útiles en las transferencias rápidas de bloques de memoria, en la búsqueda en tablas o en textos y en la codificación de datos.

1.6.11 Control del programa

El término control del programa se refiere a la tarea de controlar el flujo de instrucciones de un programa. Este control se consigue a través del puntero de instrucciones (IP) y el

registro de segmento de código (CS).Por ejemplo, se puede saltar a una instrucción dada sencillamente cargando en el registro IP([ ] en el CS) la dirección del objetivo .
Algunas operaciones de control de programa del 8086/8088 incluyen las siguientes:

JMP objetivo	salto directo en el mismo segmento
JMP objetivo segmento	salto directo a un nuevo segmento
JMPS destino	salto corto
JMPI destino	salto indirecto en el mismo segmento
JMPL destivo	salto indirecto largo (nuevo segmento)
JE objetivo	salto si es igual
JZ objetivo	salto si es cero
JNE objetivo	salto si no es igual

ejemplo

```
JMPI  BX : equivalente a PCHL del 8080
JMP  FUN1: salta a la función #1 en este
           mismo segmento
```

1.6.12 Control de sistema

Se incluyen aquí una serie de instrucciones bajo el nombre de control de sistema que incluyen instrucciones de interrupción software, la instrucción de parada (HLT) y espera (WAIT) e instrucción de gestión de los indicadores.
Las operaciones de control del sistema del 8086/8088 incluyen las siguientes:

INT	Interrupción
INTO	Interrupción si hay capacidad excedida
IRET	Vuelta de interrupción
CLI	Borrar el indicador de interrupción
STI	Activar el indicador de interrupción
HLT	Parada
WAIT	Espera
LOCK	Bloques
ESC	Escape

CONCLUSIONES DEL CAPITULO I

En este capitulo se han explorado diversas facetas de los chips microprocesadores 8086 y 8088 de Intel. Se han visto como se relacionan entre ellos, y con sus predecesores de 8 bits, el 8080 y el 8085 de Intel. Se han comparado también con sus rivales de 16 bits, el Motorola MC68000 y el Zilog Z8000. Se han descrito su estructura interna, incluyendo la arquitectura pipeline, sus procesadores duales en un diseño de chip simple y su juego de registros de 16 bits.
Finalmente se ha mencionado su juego de instrucciones más común que incluye operaciones de 8 y 16 bits, aritméticas y lógicas y operaciones en aritmética decimal codificada en binario; un juego completo de instrucciones de tratamiento de cadenas; y algunas instrucciones especiales (LOCK y ESCAPE) para el multiproceso y los procesos paralelos.

BIBLIOGRAFIA

1 Len J. Scanlon : 8086/8088/80286 ASSEMBLY LANGUAJE :Brady New York. 1986. Cap 3.

2 Robert Erskine : Programación del 8088/8086 : Anaya 1986. Cap 1. Cap 2 y Cap 3.

3 Robert L. Tokheim : Fundamentos de los Microprocesadores Schaum. Mcgraw-Hill. 1986

CAPITULO II

SOFTWARE

Introducción

La palabra Software significa un conjunto de programas y documentos asociados a un computador. Este conjunto de programas se pueden escribir desde distintos niveles de programacion. Cada uno de estos niveles genera ciertos tipos de lenguajes con sus propias ventajas y desventajas que dependeran del tipo de problema a resolver. Es así como los lenguajes de programación se pueden clasificar en tres categorías, una de las cuales es llamada lenguaje de ensamble que será el tema de estudio del presente capítulo, cuyo objetivo es que además de conocer las instrucciones utilizadas en el lenguaje de ensamble del microprocesador 8088, se conozcan algunas técnicos de programación que se necesitan para una programación clara y efectiva. El capítulo se finalizará con la creación de un programa que permitirá explorar y modificar el contenido de los sectores de un diskette. Como una ayuda para la comprensión de los conceptos fundamentales sobre microprocesadores, tales como registros, localizaciones de memoria, interupciones, etc. se explica como utilizar un programa del DOS llamado Debug. En la segundo parte de este capítulo se muestra como hacer uso del Macro Assembler, que permite compilar programas creados en lenguaje de ensamble. A continuación se discute un poco sobre los otros niveles de programación antes de iniciar con el lenguaje de ensamble.

2.1 LENGUAJES DE ALTO NIVEL.

Son lenguajes orientados a los usuarios y no a las máquinas. Su principal ventaja es que la codificación de un algorítmo resulta relativamente sencilla, pero poseen la desventaja de ser hasta cierto punto limitados por la razón de que cada uno de ellos han sido orientados hacia la solución ya sea de problemas específicos o la practica de ciertas técnicas de programación. Entre los principales lenguajes de programación de alto nivel estan los siguientes:

FORTRAN: Que significa traductor de formulas. Fué el primer lenguaje de programación de alto nivel. Este lenguaje esta orientado hacia la solución de problemas científicos: pero, en la realidad, aceptuando su capacidad para operar con números complejos, cualquier tarea matemática se puede desarrollar tambien con lenguajes tales como el BASIC, el Pascal, etc

BASIC: Este es un lenguaje de propositos generales, su principal ventaja es que esta diseñado para la programación estructurada la cual si bien es cierto que requiere un poco de mas esfuerzo de parte del programador, compensa con creces a la hora de depurar o de darle mantenimiento a un programa.

QBASIC: Este lenguaje es de propositos generales, cuya capacidad de resolución de problemas es bastante subestimada debido a la ignorancia de muchas personas al creer que lo complejo es necesariamente siempre mas eficiente. La principal desventaja de este lenguaje es que es el bastante "lento", pero para aplicaciones en donde no se manejarán archivos con cientos de datos, o programas que impliquen miles de iteraciones, como lo son los programas profesionales de juegos o paquetes comerciales, la ventaja que representa su sintaxis e tremadamente fácil es digna de considerarse. Si un estudiante domina este lenguaje y tiene en su poder una computadora de bolsillo puede facilitarse enormemente la solución de gran cantidad de problemas que se plantean en el transcurso de cualquier carrera universitaria, y ya no digamos en Ingeniería.

COBOL: Este es un lenguaje orientado hacia la solución de problemas comerciales, y su principal ventana es la fácil y eficiente manipulación de archivos que se logra con el.

C : Este es un lenguaje muy versatil ya que a pesar de ser un lenguaje de alto nivel de gran poder, tambien se considera de relativo bajo nivel, por la fácilidad que presenta para accesar los recursos internos de una computadora. Su principal desventaja es que presenta un sintaxis y una lógica poco convensional en el sentido de que no es tan fácil de comprender el propósito de un programa con solo mirar su listado fuente. Existen sistemas operativos que han sido escritos casi en su totalidad en este lenguaje, lo que indica una alta eficiencia a la hora de compilar un programa.

La cara opuesta a los lenguajes de alto nivel lo constituyen el lenguaje de maquina el cual se considera a continuacion.

2.2 LENGUAJE DE MAQUINA

Este es el unico tipo de lenguaje que una computadora es capaz de leer y entender directamente. Es decir que los lenguajes de alto nivel para poder ser ejecutados primero deben de ser traducidos al lenguaje de maquina, proceso que se conoce con el nombre general de compilación. Además de los compiladores existen programas llamados interpretes, que a diferencia de un compilador la traducción a lenguaje de máquina no la realiza sobre todas las instrucciones del programa de una sola vez, si no que la traducción se hace cada vez que se ejecuta una instrucción, el GWBASIC es un ejemplo típico de un interprete.

El lenguaje de máquina es propio de cada clase de máquina, cosa que no sucede con los lenguajes de alto nivel: es decir que el hecho de que un programa corra en una computadora determinada no garantiza que corra en otra cuya arquitecutura puede ser distinta. Si a eso se le añade que no es nada fácil la escritura de programas en lenguajes de máquina, resulta obvio el porque no se debe escribir programas en lenguaje de máquina a menos que sea absolutamente necesario. Las rutinas escritas en lenguaje de maquina se justifican unicamente en los casos en donde la velocidad de procesamieto debe de ser muy rapida como por ejemplo cuado se esta obteniendo información a partir de señales electricas.

Como una alternativa a los lenguajes de alto nivel y al lenguaje maquina esta el nivel intermedio de los lenguajes de programación el cual se conoce como lenguaje de ensamble y lograr su dominio, en lo que respecta al microprecesador 8088, constituye uno de los propositos principales de este tabajo.

2.3 LENGUAJE DE ENSAMBLE

El lenguaje de ensamble por si solo es muy rapido, pero si a la hora de programar se utilizan tecnicas efectivas de programación la velocidad de los los programas se duplicara o se triplicara, asi como tambien se podran escribir programas mas legibles.

2.3.1 PORQUE EL LENGUAJE DE ENSAMBLE 8088

Una razón, quizas la mas obvia, es que los programas en lenguaje de ensamble son el corazón de cualquier computadora IBM PC o compatible. En relación a todos los otros lenguajes de programación, el lenguaje de ensamble es el mas bajo comun denominador. Le toca ejecutar lo que los lenguajes de alto nivel indican. Asi aprender el lenguaje de ensamble significa comprender la operación del Microprocesador 8088 que esta dentro de la computadora.

Una vez que se comprenden la operación del Microprocesador 8088, muchos elementos que se ven en otros programas y lenguajes de alto nivel tendrán mayor significado. Por ejemplo, ya se habrá notado que el entero mas grande que se puede tener en BASIC es 32767. De dónde vino este número?, es un número impar para un límite superior. Pero como se verá mas adelante, el número 32767 esta directamente relacionado a la forma como la computadora IBM PC almacena los numeros.

Además, tambien se puede estar interesado en la velocidad o en el tamaño de los programas. Como una regla, los programas en lenguaje de ensamble son mucho mas rápidos que aquellos escritos en cualquier otro lenguaje, tipicamente son dos o tres veces mas rápidos que sus equivalentes en C o en Pascal, o 15 veces mas que

los escritos en BASIC. Los programas en lenguaje de ensamble son tambien mas pequeños. Al final de este capitulo se construirá un programa de ejemplo muy útil que ocuperá alrededor de un kilobyte. Un progama similar en C o en Pascal llevaría 10 veces ese tamaño. Por estas razones junto con otras que se verán mas adelante, la Lotus Development Corporation escribio el 1-2-3 completamente en lenguaje de ensamble.

El lenguaje de ensamble tambien suministra un completo acceso a la computadora. Un número de programas tales como SideKick, Prokey, y SuperKey, permanecen en la memoria despues que se corren. Tales programas cambian la forma en que la máquina trabaja, y ellos tienen caracteristicas disponibles solo para programas escritos en lenguaje de ensamble.

2.4 UTILIZACION DEL DEBUG Y ESTRUCTURA LOGICA DEL 8088

EQUIPO REQUERIDO

El equipo requerido para poder correr los ejemplos en este trabajo serán: Una computadora IBM PC o compatible, con al menos 128 K de memoria y una unidad de disco. Tambien se necesitará una versión 2.00 o posterior de PC-DOS (o MS-DOS), y en la segunda parte de este trabajo se necesitará el Microsoft Macro Assembler.

2.4.1 LENGUAJE DE MAQUINA, DEBUG Y ARITMETICA

Antes de comenzar con el lenguaje de ensamble del 8088 primero es necesario recordar que las computadoras cuentan con numero binarios.

Estamos interesados en números binarios porque ellos son la forma en la cual los números son procesados dentro del Microprocesador 8088. Pero mientras la computadora tiene éxito con los numeros binarios, estas cadenas de unos y ceros pueden ser largas y complicadas de escribirlos. La solucion? números he adecimales. Esta es una forma mas compacta de escribir números binarios. Se Asume que el lector esta familiarizado con la aritmetica binaria, he adecimal, y con la forma en que estos sistemas se relacionan entre si.

2.4.2 NUMEROS HEXADECIMALES

Los números he adecimales son mas faciles de manipular que los numeros binarios- Al menos en términos de longitud- Se comenzará con números he adecimales (he a por mas corto), y utilizaremos el DEBUG.COM, un programa que se encuentra junto con el sistema operativo MS-DOS. Utilizará el Debug aquí y tambien mas adelante para entrar y correr programas en lenguaje de máquina una instrucción a la vez. Como en BASIC, el Debug provee un agradable

medio interactivo, pero a diferencia de BASIC, no conoce de números decimales. Para el Debug, el número 10 es un número hexadecimal- no 10. Y puesto que el Debug solo "habla" en Hexadecimal, es necesario saber acerca de los números hexadecimales, pero primero, se hará una corta introducción al Debug mismo.

2.4.3 DEBUG

Por que este programa se llama Debug? Bugs, (en español Bichos), en el lenguaje de las computadoras, significa errores en un programa. Un programa que trabaja correctamenete no tiene Bichos, mientras que un programa que no trabaja correctamente tiene por lo menos un bicho. Utilizando el Debug para correr un programa una instrucción a la vez, y observando como el programa trabaja, se pueden hallar los errores y corregirlos. Esto se conoce como **depurcación (debugging)** de un programa, de aqui su nombre Debug.

NOTA: De aqui en adelante, El texto que se deberá de escribir a la hora de dar los comandos en la computadora aparecerá en negritas para distinguirlo de la respuesta que dará la computadora.
Al escribir el texto y presionar ENTER se obervara una respuesta similar a la que aparece en este trabajo. No siempre se observará la misma respuesta, ya que no todas la computadoras tienen la misma cantidad de memoria
Incluso, se usarán letras mayusculas en todos los ejemplos esto es solo para evitar la confusión entre la letra l (ele) y el número 1 (uno). Si se desea se pueden escribir los ejemplos en letras minusculas.

Ahora, se comenzará con el debug, escribiendo este nombre despues del indicador del DOS (el cual sera A en los ejemplos) Pero muy bien puede ser C

```
A DEBUG
-
```

EL guión que se ve es la respuesta al comando. Es el indicador del Debug, así como A es el indicador del DOS. Esto significa que el Debug esta esperando por un comando.

Para dejar el Debug y retornar al DOS, solo se escribe **Q** (por **Quit, dejar**) en el indicador del Debug y se presiona ENTER.

2.4.4 HEXARITMETICA

Se utiliza el comando llamado**H. (Hexaritmetica)**,y como su nombre lo sugiere se sumaran y restaran dos números he adecimales. considera como trabaja H comensando con 2+3. Se Sabe que 2+3 = 5 para números decimales. Sera cierto para números He adecimales? Para asegurarse, se escribirá en respuesta al guión del Debug lo siguiente:

```
-H 3 2
0005   0001
```

El debug imprime ambos la suma (0005) y la diferencia (0001) de 3 y 2. el comando H siempre cálcula la suma y la diferencia de dos numeros.

Que sucederia si se escribe el siguiente comando?

```
 H 2 3
0005   FFFF
```

La respuesta de 2 - 3 fue FFFF en lugar de -1, lo que sucede es que los números negativos se representan por medio de su complemento A dos

BITS, BYTES, WORD, Y NOTACION BINARIA

Los digitos he adecimales, tales como 4Ch (la h indica que los numeros estan en he adecimal), se pueden escribir como 0100 1100b (la b significa que este número esta en binario) y que esta compuesto de 8 digitos, los cuales se separan en grupos de cuatros para mayor legibilidad. Cada uno de estos digitos binarios se conoce como un **bit**, asi un número como 0100 1100 o 4Ch tiene ocho bits de longitud.

Muy a menudo, es conveniente enumerar cada uno de los bits en una cadena larga, con el bit 0 se identifica el que esta mas a la derecha. El 1 en 10b es el bit número 1, y el que esta mas a la izquierda en 1011b es el bit número 3. Numerar los bits de esta forma hará mas fácil hablar de uno en particular.

Un grupo de 8 digitos binarios se conoce como un **byte**, mientras que un grupo de 16 digitos binarios, o dos bytes, se llama **una palabra**. Se utilizarán estos terminos frecuentemente en el transcurso de este trabajo.

2.5 ARITMETICA EN EL 8088

Para conocer como es la aritmetica he adecimal del Debug y como es la aritmetica binaria del 8088, se comenzara con aprender como

el 8088 realiza estas tareas matematicas. Utilizando los cumandos internos del Debug llamados **intrucciones.**

2.5.1 REGISTROS COMO VARIABLES

El debug, sabe mucho acerca del Microprocesador 8088. A el se le puede ordenar que muestre el contenido de pequeñas piezas de memoria llamado **registros**, en los cuales se pueden almacenar numeros. Los registros son como variables en BASIC, pero, estos no son exactamente iguales. A diferencia del BASIC, el 8088 contiene un número fijo de registros, y estos registros no son partes de la memoria del computador. Ver seccion 1.3.?

Para ordenarles al Debug que muestre el contenido de los registros utilizamos el comando R (Register)

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100  NV UP DI PL NZ NA PO NC
3756:0100 E485       IN      AL,85
```

Probablemente en algunas computadoras se veran numeros diferentes en la segunda y tercera linea. Estos números reflejan la cantidad de memoria en una computadora.

Por ahora el debug nos ha dado cierta cantidad de informacion. se enfocara la atención en los cuatro primeros registros, AX,BX,CX, y DX, los cuales el debug informa que son iguales a cero. Estos registros son los **registros de propositos generales.** Los otros registros, SP,BP,SI,DI,DS,ES,SS,CS, e IP, son registros de propositos especiales.

Los cuatro digitos que siguen a cada nombre de registro estan en notacion hexadecimal. Se puede ver que cada uno de los registros del 8088 contienen una palabra, (16 digitos binarios o 4 digitos hexadecimales).
Anteriormente se dijo que los registros son como variables en BASIC. Esto significa que se puede cambiar su contenido. El comando R hace mas que mostrar el contenido de los registros. Segundo por el nombre de un registro, el comando le dice al Debug que desea ver el registro y luego cambiarlo, por ejemplo, para cobiar el contenido de AX se escribe lo siguiente:

```
-R AX
AX 0000
:3A7
-
```

Examinando los registros de nuevo para ver si el registro AX contiene ahora 3A7h, se observa el siguiente despliegue de registros:

```
-R
AX=03A7 BX=0000 CX=0000 DX=0000 SP=FEEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100  NV UP DI PL NZ NA PO NC
3756:0100 E485       IN     AL,85
```

De la misma manera se pueden colocar numeros he a dentro de los otros registros utilizando el comando **R** especificando el nombre del registro, escribiendo y entrando el nuevo numero despues de los dos puntos.

2.5.2 LA MEMORIA Y EL 8088

Se utilizara el Debug para darle instrucciones al 8088 para que sume numeros desde dos registros: Se colocara un número en el registro BX y luego se instruirá al 8088 para que sume el numero en el registro BX al número en le registro AX, y que coloque el resultado de regreso en el registor AX. Primero se debe de colocar un numero en el registro BX. Por esta vez se sumará 3A7h, contenido en AX y 92h contenido en BX, se utilizara de nuevo el comando R para almacenar 92h en BX.

El registro AX y el registro BX deberán contener, respectivamente 3a7h y 92Ah. Verificando con el comando R observaremos:

```
AX=03A7 BX=092A CX=0000 DX=0000 SP=FEEE BP=0000 SI=0000 DI=0000
DS=3776 ES=3776 SS=3776 CS=3776 IP=0100 NV UP DI PL NZ NA PO NC
3776:0100 E485       IN     AL,85
```

Ahora que ya se tienen los dos numeros en AX y BX, la pregunta es como decirle al 8088 que sume BX a AX. Para ello Se pondrán numeros dentro de la memoria del computador, estos numeros serán dos bytes de **código de máquina** que le indicaran al 8088 que sume el registro BX al registro AX. Luego se observará lo que sucede, cuando con la ayuda del debug se ejecute esta instrucción.

¿En que lugar de la memoria se deberá colocar los dos bytes de la instrucción? y ¿como se le dirá al 8088 que los encuentre? La respuesta es que el 8088 divide la memoria en "pedazos" de 64l llamados **segmentos**, anteriormente mencionados. Para accesar un byte en la memoria se utilizan dos números he a (de cuatro dígitos), uno para cada segmento de 64l y uno para cada byte, o **complemento**, (offset) dentro del segmento. Cada segmento comienza con un multiplo de 16 , así sumando el segmento y el complemento se obtiene la dirección deseada.

Todas las direcciones (etiquetas) que se utilizaran son complementos del inicio de un segmento. Las direcciones se escribiran como un numero de segmento seguido por el complemento dentro del segmento. Por ejemplo, 3756:0100 significa que se está

direccionando el byte 0100 del segmento 3756. Mas adelante se trataran mas en detalle los segmentos, pero por ahora se confiará en el debug para que el escoja el segmento, de manera que se pueda trabajar con un segmento sin tener que poner atencion al numero del segmento, las direcciones se referiran solo como el número de su complemento, cada una de estas direcciones apunta a un byte en el segmento, y las direcciones son secuenciales, es decir, 101h es el byte que sigue el byte 100 en la memoria.

Se escribirá ahora la instrucción de dos bytes para sumar BL a AL que la reconocera como ADD AL,BL. Se colocará esta instrucción en la localización 100h y 101h, en cualquier segmento que el debug comience a utilizar. Al hacer referencia a está instrucción se dira que está en la localización 100h, puesto que es la localización del primer byte de la instrucción.

El comando del debug para examinar y cambiar la memoria es llamado E (por ENTER). Se utilizará éste comando para entrar los dos bytes de la instrucción ADD, de la manera siguiente:

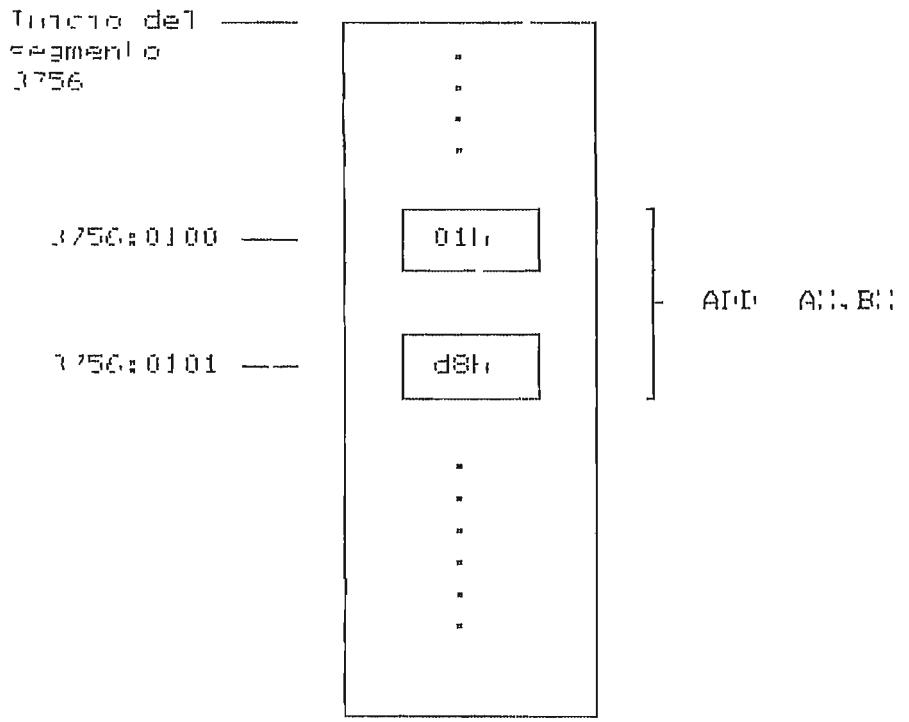


Figura 2.1 Representación gráfica del contenido de la memoria

```
-E 100
3756:0100 E4.01
-E 101
3756:0101 85.D8
-
```

Los numeros E4 y 85 junto con el punto que aparece al final de cada número aparecen por si solos. Luego se introduce el codigo de la instrucción.

Los numeros 01h y D8h son la instrucción en lenguaje de máquina del 8088 para la instrucción ADD y se encuentra en las localizaciones 3756:0100 y 3756:0101. Es probable que el número de segmento en otras computadoras sea distinto pero la diferencia no afectara el programa. De la misma manera es probable que se muestren numeros diferentes para cada uno de los comando E. Estos numeros (E4h y 85h en nuestro ejemplo) son los números que se encontraban en la memoria en esas localizaciones, es decir números que fueron dejados en la memoria por otros programas.

2.5.3 SUMANDO AL ESTILO DEL 8088

Ahora los registros al ser mostrados se verán como esto:
AX=03A7 BX=023A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3776 ES=3776 SS=3776 CS=3776 IP=0100 NV UP DI PL NZ NA PO NC
3776:0100 01D8 ADD AX,BX

La instrucción ADD se encuentra ahora colocada en la memoria, exactamente donde debe de estar. Los primeros dos numeros, 3756:100, dan la dirección 100h para el primer número de la instruccion ADD. Segundo de esto se ven dos Bytes para ADD: 01D8. el byte igual a 01 está en la dirección 100h, mientras que D8 está en la dirección 101h. Ver figura 2.1. Finalmente, puesto que se entraron las instrucciones en lenguaje de máquina- números que no tienen significado para los humanos, pero que el 8088 los interpreta como la instrucción de sumar- el mensaje ADD AX,BX confirma que se ha entrado la instrucción correctamente.

Aun que ya se ha colocado la instrucción ADD en la memoria, no se está completamente listo para correrla. Primero, se requiere decirle al 8088 donde hallar la instrucción.

El 8088 encuentra el segmento y el complemento de una dirección en dos registros especiales, CS e IP, los cuales se ven listados en el despliegue precedente de registros. El numero de segmento es almacenado en CS, (Code Segment). Si se observa el despligue de registros, el debug ya tiene establecido el registro CS de antemano (CS=3756, en nuestro ejemplo). Sin embargo la dirección completa del inicio de la instruccion es 3756:0100.

La segunda parte de esta dirección (el complemento dentro del segmento 3756) esta almacenada en IP (Instruction pointer). El 8088 utiliza el complemento almacenado en el registro IP para hallar la primera instrucción. Asi se le puede indicar donde buscar la primera instrucción estableciendo el valor de IP=0100. Pero el registro IP ya esta puesto a 100h. El debug establece el valor de IP a 100h siempre que se comienza con el. Sabiendo esto

deliveradamente se escogio 100h como la direccion de la primera instrucción y se eliminó así la necesidad de poner el registro IP en pasos separados. Si IP no contiene 100h se deberá de colocar éste valor utilizando el comando R IP, de manera similar a como se hizo para colocar numeros en AX y BX en el ejemplo de la sección 2.5.1.

Ahora ¿como se le dice al debug que ejecute una instrucción?. Para esto se utiliza el comando T (por Trace) el cual ejecuta una instrucción a la vez y luego despliega los registros. Después de cada Trace, el IP deberá de apuntar a la siguiente instrucción, en éste caso apuntará a la instrucción 102h. No se tiene que poner una instrucción en 102h, así en la última línea del despliegue de registros se verá una instrucción dejada por algún otro programa.

Para ordenarle al debug para que ejecute la instrucción utilicemos el comando T

```
-T
AX=09D1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=377E ES=377E SS=377E CS=377E IP=0102 NV UP DI PL NZ NA PO NC
377E:0102 58            POP AX
-
```

Es decir el registro AX ahora contiene 09D1, lo cual es la suma de 3A7h y 92Ah. Y el registro IP apunta a la dirección 102h, así la última línea del despliegue de registros muestra alguna instrucción en la localización 102h. (5A en éste caso, dejado por algún programa que ocupo la memoria anteriormente).

Como ya se mencionó anteriormente, los registros IP y CS siempre apuntan a la siguiente instrucción a ser ejecutada. Si se escribe T de nuevo, se ejecutará la siguiente instrucción, pero cuidado con hacerlo-el 8088 se podría perder en el limbo.

Si se desea ejecutar la instrucción ADD de nuevo, para sumar 92Ah a CD1h y almacenar el nuevo resultado en AX. Se debe de cambiar el registro IP con el cumando R de nuevo al valor de 100h y luego dar el comando T.

La figura 2.2 y 2.3 muestran grafricamente lo que sucede en los registros antes y despues de ejecutar la instrucción T

```
          AX: 0317      BX: 092A

IP:100 =  ADD   AX,BX

          POP BX
```

fig. 2.2 antes de ejecutar la instrucción ADD

```
AX= 0CD1     BX= 092A

ADD   AX,BX

IP=100 =   POP DX
```

fig 2.3 despues de ejecutar la instrucción ADD

2.5.4 RESTA AL ESTILO DEL 8088

A continuación se escribirá una instrucción para restar BX de AX de manera que después de dos restas se tendría 3A7h en AX.
Cuando se colocaron los dos bytes para la instrucción ADD, se escribió el comando E dos veces: una vez con 0100h para la primera dirección, y otra vez con 0101h para la segunda dirección.
Es posible entrar un segundo Byte sin necesidad de otro comando E si se separa del primer byte con un espacio. Cuando se ha finalizado de entrar bytes se presiona la tecla ENTER para salir del comando E.
Este método se ilustra para la instrucción de resta:

```
-E 100
3756:0100   01.29  D8.D8
```

El despliegue de registros debera mostrar ahora SUB AX,BX, la cual resta el registro BX del registro AX y el resultado lo coloca en AX, ejecutando esta instrucción con el comando T (recordando establecer IP a 100h). Antes de ejecutar el comando T, AX debera contener CD1, que fué el resultado de la suma que se realizo en la sección anterior. Despues de ejecutar el comando T el contenido de AX será 03A7, no olvidar que BX permanece con su valor original 092A inalterado

2.5.5 BYTES EN EL 8088

Toda la aritmética hasta aquí utilizada, se ha hecho en palabras, es decir, con cuatro dígitos hexa. ¿Sabe como realizar el 8088 operaciones matematicas con bytes ? Si, lo sabe.

Puesto que una palabra está formada por dos bytes, cada registro de propósito general puede ser dividido en dos bytes, conocidos como **byte alto** (los primeros dos dígitos hexa-los de la izquierda) y el **byte bajo** (los segundos dos dígitos hexa-los de la derecha). Cada uno de estos registros puede ser llamado por

en letra (de la A a la D), seguido por X para una palabra, H para el byte alto, o L para el byte bajo. Por ejemplo DL y DH son registros que almacenan un byte y DX es un registro que almacena una palabra. ver figura 2.4

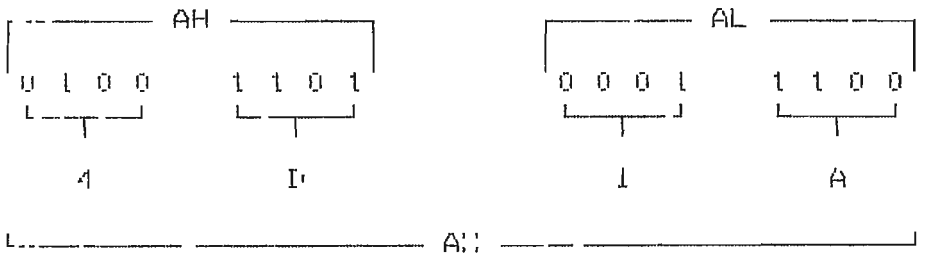


Figura 2.4 División de AX en dos registros de un byte

Para probar las operaciones matemáticas con bytes con una instrucción ADD, se colocarán dos bytes 00h y C4h, comenzando en la localización 100h. Al final del despliegue de registros se verá la instrucción ADD AH,AL, la cual suma los dos bytes de registro AX y coloca el resultado en el byte alto, AH.

Lo siguiente será cargar el registro AX con 0102h. Esto coloca 01h en en registro AH y 02h en el registro AL. Al establecer IP a 100h, y ejecutar el comando T, se hallará que AX ahora contiene 0302h. El resultado de 01h + 02h es 03h, y éste valor está en el registro AH. Es necesario aclarar que el debug no permite que se cambien bytes individualmente, es decir no se puede cambiar solamente el registro AH o el AL, si no que se debe de cambiar completamente todo el registro AX.

2.5.6 MULTIPLICACION Y DIVISION AL ESTILO 8088.

La instrucción de multiplicación es llamada MUL, y el código de máquina para multiplicar AX por BX es F7h E3h.
Al multiplicar dos números de 16 bits puede dar como resultado un número de 32 bits como respuesta, de manera que la instrucción **MUL** necesitara dos registros para almacenar su resultado, DX y AX. Los 16 bits más significativos son colocados en DX, y los menos significativos son colocados en AX. Esta combinación de registros se puede escribir DX:AX.
Al entrar la instrucción de multiplicación, F7h E3h, en las localizaciones 100h y 101h, y almacenando en AX 7C4Bh y en BX 100h, si se despliegan los registros se verá la instrucción como **MUL** BX, al hacer referencia al registro AX.Para multiplicar palabras, el 8088 siempre multiplica el registro que se indique en la instrucción por el registro AX, y almacena la respuesta en DX y AX.
Al realizar la multiplicación de 7CB4 * 100h el resultado sera de 7CB400h éste resultado es demasiado largo para almacenarlo en un

solo registro de manera que se dividirá en dos: así DX almacenará 007Ch y en AX 4B00h.
Por otra parte,El código de máquina para la instrucción de dividir **(DIV)** es F7h F3h. Del mismo modo que la instrucción **MUL** la instrucción **DIV** utiliza los registros DX:AX sin tener que mencionarlos. así en el despliegue de registros se verá **DIV BX**. Supongamos que se desea dividir 7C4B12h / 100h el resultado de esta división será de 7C4Bh con un residuo de 12h. Para realizar esta división se deberá de establecer los valores de los registros de la siguiente manera:

DX= 007C
AX= 4B12
BX= 0100

Después de ejecutar la instrucción de dividir los registros terminarán con los siguientes valores.

DX= 0012
AX= 7C4B
BX= 0100

Es decir los datos para la división deberán ser colocados de la siguiente manera :

DX= 16 bits más significativos del dividendo
AX= 16 bits menos significativos del dividendo
BX= divisor

El resultado vendrá dado de la siguiente manera:

AX= cociente
DX= Residuo
BX= no se altera

2.6 INTERRUPCIONES

A las cuatro instrucciones matemáticas anteriores, se agregará una nueva instrucción llamada INT n(por interrupt). Una interrupción para el presente propósito, se puede considerar similar a la instrucción GOSUB de BASIC. Se utilizará la instrucción INT 21 para lograr que el DOS imprima el caracter A en la pantalla.
Antes de aprender como INT n trabaja, se verá un ejemplo. utilizando el Debug colocando 200h en AX y 4h en DX, cuyo significado se discutirá mas adelante. La interrupción del DOS que se utilizará es INT 21h- en código de máquina: CDh 21h. Esta instrucción es de dos bytes. Se pondrá la instrucción INT 21h en la memoria, comenzando en la localización 100h, y utilizará el comando R para confirmar que la instrucción que se lee sea INT 21h (no se debe olvidar poner el registro IP a 100h si no está con ese valor).

Ahora se está listo para ejecutar esta instrucción. Sin embargo no se puede utilizar el comando Trace como se hizo anteriormente. El comando Trace ejecuta una sola instrucción al mismo tiempo, para la instrucción INT se llama un largo programa del DOS y se ejecuta, y no se desea ejecutar y ver una por una todas estas instrucciones.

Como lo que se desea es correr este programa y que se detenga antes de ejecutar la instrucción en la localización 102h. Esto se puede lograr con el comando G (por Go) del debug, seguido por la dirección en la cual se detendrá el programa, a esta dirección se le conoce con el nombre de **punto de ruptura.**

A continuación se utilizará el comando G con un punto de ruptura en la localización 102h.

```
G 102
A
AX=0241 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3070 SS=3970 CS=3070 IP=0102  NV UP DI PL NZ NA PO NC
3970:0102 8BE5          MOV     SP,BP
```

El DOS ha mostrado el caracter A como se deseaba, y luego retornó el control al debug.
(recordar, que la instrucción en 102h es solamente dato dejado por otro programa, probablemente se podría ver algo distinto)

¿Como sabe el DOS que se desea imprimir la letra A? El 02h en el registro AH le dice al DOS que imprima un caracter. Otro número en AH le diría al DOS que ejecute una diferente función (se puede hallar una lista de funciones en el manual de referencia técnica del DOS)
El DOS utiliza el número en el registro DL como el código ASCII del caracter que se desea imprimir, el 41h es el código ASCII para la letra A.

INTERRUPCION 20h

La interrupción anterior fue la 21h, si se cambia el 21h al 20h se tendrá la interrupción 20h, esta interrupción se utiliza para terminar un programa de manera que el DOS u otro programa pueda tomar el control de nuevo. En este caso, la INT 20h regresará el control al Debug, puesto que se está ejecutando el programa desde el debug y no desde el DOS. Cuando se ejecuta INT 20h todos los registros se regresan a sus valores originales, es decir que el registro IP vuelve a comenzar con 100h.

2.6.1 UN PROGRAMA DE DOS LINEAS- COLOCANDO LAS PIEZAS JUNTAS

Comenzando en la localización 100h, se entrarán las dos instrucciones INT 21h, INT 20h (CD 21h CD 20h) una después de

otra (desde ahora en adelante, siempre se comenzarán los programas en la localización 100h).
Cuando se tenía una sola instrucción, se podía "listar" esa instrucción con el comando R, pero ahora se tienen dos. Para verlas, se utilizará el comando U (Unassemble), el cual actua como el comando LIST en BASIC. Este comando muestra catorce líneas de instrucciones a partir la localización que se utiliza como argumento del comando U. Por ejemplo el comando U 100 mostrará lo siguiente:

```
-U 100
3970:0100  CD21            INT     21
3970:0102  CD20            INT     20
3970:0104  D98D46D250B8    ESC     09,[DI+D246][DI+B850]
3970:010A  8D00            LEA     AX,[BX+SI]
3970:010C  50              PUSH    AX
3970:010D  E82A23          CALL    SP+BP
3970:0110  8BE5            MOV     SP,BP
3970:0112  83C41A          ADD     SP,+1A
3970:0115  5D              POP     BP
3970:0116  C3              RET
3970:0117  55              PUSH    BP
3970:0118  83EC02          SUB     SP,+02
3970:011B  8BEC            MOV     BP,SP
3970:011D  823E0E0000      CMP     BYTE PTR [000E],00
```

las primeras dos instrucciones son las que se han entrado. Las otras son instrucciones que fueron dejadas por otros programas en la memoria.

El comando **U 100 10C** Solo mostrara las instrucciones en el rango especificado, es decir, las instrucciones comprendidas entre las localizaciones 100 y 10C, ambas inclusive. Al volver a escribir el comando U sin argumentos el despliegue de instrucciones continuará a partir de la localización que quedó pendiente de mostrarse con el comando U anterior.
Ahora, se colocará 02h en AH y en DL el código ASCII del caracter que se desee imprimir (así como se hizo cuando se cambio el registro AX anteriormente), luego simplemente se escribe el comando G para ver el caracter,sin especificar la dirección donde la ejecución debe detenerse Por ejemplo, si se coloca 41h en DL, se verá:

```
-G
A
Program terminated normally
-
```

2.6.2 ENTRANDO PROGRAMAS.

Desde aquí en adelante, la mayoría de los programas tendrán mas que una instrucción y para presentar los programas se utilizará

el comando U. El ultimo programa aparecería como sigue:

```
3970:0100  CD21                INT       21
3970:0102  CD20                INT       20
```

Las instrucciones se han colocado directamente como numeros, tales como CDh, 21h. Pero esto toma bastante esfuerzo y, ademas, existe una forma mucho mas facil de escribir instrucciones.
Ademas del comando U (Unassemble), el Debug incluye el comando A (Assemble), el cual permite entrar las instrucciones en nemónico directmente. Asi en lugar de escribir los códigos numéricos de cada instrucción, se puede utilizar el comando A para entrar los siguiente:

```
-A 100
3970:0100  INT 21
3970:0102  INT 20
3970:0104
-
```

Cuando se ha finalizado, todo lo que se tiene que hacer es presionar la tecla ENTER, y el indicador del Debug reaparece. Aqui, la A le indica al debug que se desea escribir instrucciones en nemónico y el 100 le indica que se escriban estas instrucciones a partir de la localización 100h.

2.6.3 MOVIENDO DATOS DENTRO DE LOS REGISTROS

La instrucción que se utiliza para mover datos dentro de los registros es la instrucción MOV. Si por ejemplo se utiliza el comando A para entrar la siguiente instrucción:

```
396F:0100 88D4      MOV   AH,DL
```

Esta instrucción mueve el número en DL dentro de AH, colocando una copia de DL en AH; AL no se ve afectado. Es decir que la instrucción MOV anterior es similar a la instrucción en BASIC LET AH=DL. MOV copia el contenido del segundo registro en el primero y por esta razon se escribe AH antes que DL. Aunque hay algunas restricciones que se discutirán mas adelante, se puede utilizar la instrucción MOV para copiar números entre otros pares de registros por ejemplo si se escribe:

```
396f:0100  89C3     MOV   BX,AX
```

Se ha movido una palabra y no un byte como en el primer caso. El comando MOV siempre trabaja entre palabra y palabra o entre byte y byte; nunca entre palabras y bytes. Los registros tambien se pueden cargar con un valor por medio de la instrucción MOV, observe el siguiente ejemplo:

```
396F:0100    B402        MOV   AH,02
```

Esta instrucción mueve 02 dentro del registro AH sin afectar AL.

Aplicando lo que se ha aprendido hata el momento, se hará un programa completo para escribir un asterísco *, sin necesidad de cambiar los valores de los registros (AH y DL). El programa quedaría de la siguiente manera:

```
396F:0100   B402      MOV   AH,02
396F:0102   B22A      MOV   DL,2A
396F:0104   CD21      INT   21
396F:0106   CD20      INT   20
```

Al escribir el programa anterior y chequearlo con el comando U, ademas, asegurarse que IP apunte a 100. Al escribir el comado G para correr el programa completo se verá aparecer el asterísco en la pantalla de la manera siguiete:

```
-G
*
Program terminated normally
-
```

Ahora que se tiene un programa completo, este se escribirá en un disco como un programa .COM, así será posible ejecutarlo directmente desde el DOS. Se puede correr un programa desde el DOS simplemente escribiendo su nombre. Para darle un nombre a un programa se utiliza el comando N (por Name). Por ejemplo:

```
-N ASTER.COM
```

Para darle el nombre de ASTER.COM al programa. Este comando no escribe el archivo en el disco simplemente le da un nombre.

Lo siguiente que se debe hacer es darle al Debug el número de bytes que el programa ocupa. Si se utiliza el comando U para ver el listado del programa, luego se puede utilizar el comando H (Hex aritmetic) para calcular la longitud del programa de la siguiente manera:

```
-H 108 100
```

Donde 108 es la dirección que se encuentra después de la instrucción INT 20 y 100 es la dirección de inicio del programa, el resultado de la diferencia de estas direcciones será 8.

Una vez que se tiene la longitud del programa, se necesita escribir este número en algun lado. El debug utiliza el par de registros BX:CX para guardar la longitud del archivo a grabar, de manera que se debera de escribir 8h en CX, ya que la longitud del programa es de solamente 8 bytes se pondrá BX a cero.

Cuando ya se ha establecido el nombre y la longitud del programa, se debe escribirlo en el disco con el comando W (por Write)

```
-W
Writing 0000 bytes
-
```

Ahora ya se tiene un programa en el disco cuyo nombre es ASTER.COM

2.6.4 ESCRIBIENDO UNA CADENA DE CARACTERES

Se utilizará la INT 21h con un diferente numero de función en el registro AH, para escribir una cadena completa de caracteres. Se tendra que almacenar la cadena de caracteres en la memoria y luego se le dirá el DOS donde se encuentra dicha cadena.

Como se vió anteriormente la función 02h para la INT 21h imprime un caracter en la pantalla. Otra función, la número 09h, imprime una cadena entera, y detiene la impresión de caracteres cuando encuentra el signo $ en la cadena. Se pondrá una cadena en la memoria. Se comenzará en la localización 200h, así la cadena no se mezclara con el programa. Al entrar los siguientes números, utilizando la instrucción E 200:

48	6F	6C	61
2C	20	6C	61
20	55	45	53
20	65	73	74
61	20	61	71
75	69	2E	24

el último número, el 24, es el código ASCII para el signo $, y le indica al DOS que es el fin de la cadena de caracteres. El programa encargado de mandar la cadena anterior se muestra a continuación:

```
396F:0100  B409          MOV   AH,09
396F:0102  BA0002        MOV   DX,0200
396F:0105  CD21          INT   21
396F:0107  CD20          INT   20
```

200h es la dirección de la cadena, por eso se carga 200h en el registro DX para decirle al DOS en donde encontrara la cadena que se desea imprimir. Al chequear el programa con el comando U y luego correrlo con el comando G, se obtiene

```
-G
Hola  la UES esta aqui.
Program terminated normally
```

Otro comando importante del Debug, es el comando D (por Dump, vaciar). El comando Dump muestra el contenido de la memoria en la pantalla. En la parte izquierda muestra los codigos ASCII y en la parte derecha muestra los caracteres que representan los respectivos codigos. Si se escribe el comando D 200 se observará la cadena de caracteres que se almacenan a partir de la localizacion 200, asi:

```
-D 200
396F:0200 48 6F 6C 61 2C 20 6C 61 20 55 45 53 20 65 73 74   Hola, la UES est
396F:0210 61 20 61 71 75 69 2E 24 5D C3 35 83 EC 30 8B EC   a aqui.$]ou.10.1
```

Despues de cada par de direcciones (tales como 396F:0200 en este ejemplo) se veran 16 digitos he adecimales, seguidos por los 16 caracteres ASCII correspondientes a esos bytes. Asi en la primera linea se ven la mayoria de los caracteres ASCII que se escribieron a partir de la localización 200. La cadena termina hasta que se encuentre el signo de $ que esta en la segunda linea: el resto de las lineas son caracteres dejados por otros programas.

Siempre que se vea un punto (.) en la ventana de los caracteres ASCII, este puede representar ya sea a un punto o a algún caracter especial, tal como la letra griega pi. El Debug despliega solo 96 de los 256 caracters en la tabla ASCII, asi el punto es utilizado para los restantes 160 caracteres.

Para grabar este programa en disco, se tiene que calcular la longitud del programa tomando en cuenta el tamaño y la localizacion de la cadena que se desea imprimir. El programa comienza en la linea 100h y del resultado del comando D, se puede ver que el primer caracter que le sigue al signo de $ (]) esta ubicado en la localizacion 218h. De nuevo se utiliza el comando H para hallar la diferencia entre estos dos números. Se encuetra la diferencia 218h-100h = 018h y se almacena este resultado en C:. otra vez se pone cero en B:. Se utiliza el comando N para darle un nombre (añadirle la e tension .COM) para correr el programa directamente desde el DOS). Luego se utiliza el comando W para escribir el progrma y los datos en el archivo, con el nombre que se le ha dado, en el disco.

2.7 ESCRIBIENDO NUMEROS BINARIOS

En esta sección se escribirá un programa para escribir números binarios en la pantalla como una cadena de ceros y unos. Se añadiran algunas instrucciones a las ya conocidas, incluyendo otra version de la instrucción ADD y algunas instrucciones que sirven para repetir una parte del progrma.

2.7.1 ROTACION CON BANDERA DE ACARREO

Se sabe que si se suma 1 a FFFFh el resultado deberia de ser 10000h, pero esto en realidad a nivel de registros no es asi. Solo los cuatro digitos hexadecimales menos significantes caben en una palabra (los que estan mas a la derecha), el uno no cabe. A este uno,en este caso, es lo que se denomina un rebalse (overflow) y no se pierde sino que se coloca en lo que se llama una **bandera**- la bandera de acarreo, o CF (Carry Flag). Las banderas contienen numeros de un bit, o sea que solo pueden almacenar o un cero o un 1, logico

Se analizara la tarea de escribir números binarios, para ver como la información en la bandera de acarreo puede ser util. Se imprimira un caracter a la vez, y se desea ir recogiendo los bits del número a imprimir uno por uno, desde la izquierda hasta la derecha. Por ejemplo el primer caracter que se desea escribir en el numero 10000000 es el uno. Lo que se debe hacer es ir moviendo el byte entero una posición a la izquierda, para que vaya callendo el digito mas significante en la bandera de acarreo. Luego se repite este proceso para cada uno de los bits restantes. Este procedimiento se puede llevar a cabo con la instrucción RCL (Rotate Carry Left). Para ver como trabaja se puede escribir el siguiente pequeño programa:

```
3985:0100 D0D3      RCL   BL,1
```

Esta instruccion rota el byte en BL hacia la izquierda en un bit (por eso el 1). La instrucción es llamada de rotacion, porque RCL mueve el bit mas significante a la bandera de acarreo mientras que el bit que se encuetra en ese momento en la bandera de acarreo se traslada al bit menos significante (posicion 0) de BL. Este proceso todos los demas bits de BL son movidos o rotados a la izquierda. Despues de varias rotaciones (17 para una palabra y 9 para un byte) los bit vuelven a ocupar sus posiciones originales.

Colocando B7h en BL, y luego utilizando el comando T para ejecutar la instruccion del programa anterior varias veces, al convertir el resultado a números binarios, el resultado sería el siguiente:

```
CARRY       BL
  0         1 0 1 1  0 1 1 1      B7h     Cuando se comienza
  1         0 1 1 0  1 1 1 0      6Eh
  0         1 1 0 1  1 1 0 1      DDh
  1         1 0 1 1  1 0 1 0      BAh
                    .
                    .
  0       1 0 1 1  0 1 1 1        B7h     Despues de 9
                                          rotaciones
```

Ahora falta saber como convertir un bit en la bandera de acarreo en un cero 0 en un 1.

2.7.2 SUMANDO CON LA BANDERA DE ACARREO

La instrucción normal ADD, por ejemplo ADD AL,BL, simplemente suma dos números. Otra instrucción, ADC (Add with Carry) sumar con acarreo, suma tres números: los dos operandos, como en la instrucción anterior, mas el bit de la bandera de acarreo. Al ver en la tabla ASCII, se encuentra que 30h es el caracter 0 y 31h es el caracter 1. Así, sumando 30 a la bandera de acarreo da como resultado el caracter 0 cuando el carry esta limpio y el caracter 1 cuando el carry esta puesto. Entonce, si, por ejemplo, DL=0 y la bandera de acarreo esta puesta (1), ejecutando:

```
ADC DL,30
```

Se suma DL (0) a 30h ("0") y el 1h (el carry) para dar 31h ("1"). Así con una instrucción se ha combertido la bandera de acarreo en el caracter que se desea imprimir,este resultado queda en DL. Lo que resta todavia es una instrucción mas para poder repetir las instrucciones anteriores 8 veces, es decir una vez para cada bit.

2.7.3 LAZOS

La instrucción para realizar los lazos es la instrucción LOOP. La cual trabaja de manera similar a la instrucción FOR-NEXT de BASIC. Para indicar cuantas veces se tiene que repetir un lazo se coloca el contador de repeticion en el registro CX. Cada vez que se ejecuta el lazo, el 8088 resta uno a CX, y cuado CX alcanza el valor de cero el lazo termina.

A continuación se presenta un programa que rota el registro BX a la izquierda 8 veces, es decir, mueve el registro BL a BH (pero no al reves, puesto que se rota a traves de la bandera de acarreo)

```
396F:0100 BBC5A3          MOV     BX,A3C5
396F:0103 B90800          MOV     CX,0008
396F:0106 D1D3            RCL     BX,1
396F:0108 E2FC            LOOP    0106
396F:010A CD20            INT     20
```

El lazo comienza en 106h (RCL BX,1) y termina con la instrucción LOOP. El número que sigue LOOP (106h) es la dirección de la instrucción RCL. Cuando se corre el progiama LOOP resta 1 de CX, luego salta a la dirección 106 si CX no es cero. La instrucción RCL BX,1 (rotar acarreo a la izquierda, una posicion) es ejecutada 8 veces.

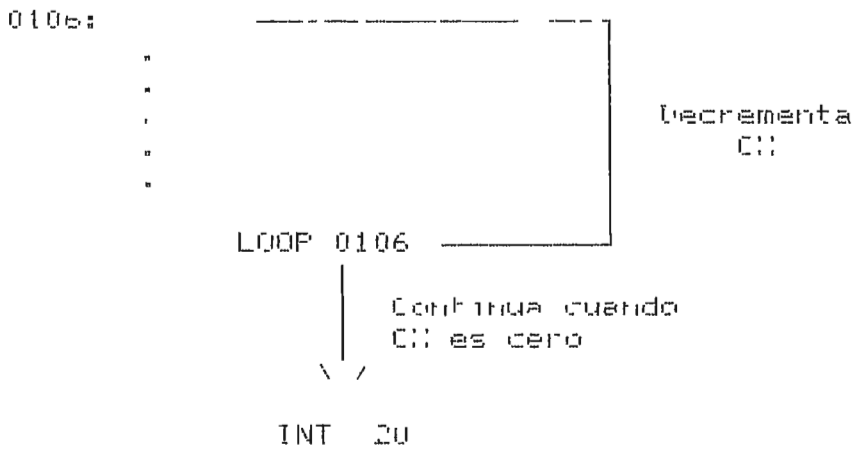


Figura 2.4 Ilustración del funcionamiento de la instrucción LOOP

Hay diferentes formas como se podría ejecutar este pequeño programa. Si simplemente se escribe G, No se verá ningún cambio en el despliege de registros, ya que el Debug salva todos los registros antes de comenzar a ejecutar las instrucciones. Luego, si se encuentra una instrucción INT 20 (Como ocurre en el programa anterior), recupera todos los registros. Es decir si se escribe G, y el programa termina con INT 20 al observar el registro IP, este almacenara 100h (donde se comenzo), y en los otros registros no se observará ninguna diferencia.
Una forma alternativa de ejecutar este programa es con el comando T (ejecucion paso a paso). Sin embargo esta forma es bastante lenta. La otra forma, y mas conveniente, es escribiendo G 10A para ejecutar el programa hasta la instrucion 0108h, o sea que no se incluye la instrucción INT 20 en la localización 10Ah; luego se despliegan los registros para ver los resultados del programa.

2.7.4 ESCRIBIENDO NUMEROS BINARIOS.

Anteriormente se vio como se desplazaban los digitos binarios uno a la vez y se convertian en caracteres ASCII. Si se agrega la instrucion INT 21h para imprimir los digitos binarios, el programa quedara completo. A continuación se presenta el programa. La primera instrucción pone 02 en AH para la llamada a la interrupción 21h (recuerde, 02 le dice al DOS que imprima el caracter que se encuentra en el registro DL):

```
3985:0100 B402      MOV   AH,02
3985:0102 B90800    MOV   CX,0008
3985:0105 B200      MOV   DL,00
3985:0107 D0D3      RCL   BL,1
3985:0109 80D230    ADC   DL,30
3985:010C CD21      INT   21
3985:010E E2F5      LOOP  0105
3985:0110 CD20      INT   20
3985:0100 B402      MOV   AH,02
```

la rotacion de BL (con la instrucción RCL BL,1) recoge los bits del número. El numero que se desea imprimir en la pantalla en binario se almacena en BL, luego al correr este programa con el comando R, y despues de la instrucción INT 20h, se restauran los registros a sus valores que tenian antes de ejecutarse el programa, asi BL todavia contiene el número que se imprimio en binario. La instrucción ADC DL,30 convierte al digito cero o uno en la bandera de acarreo es un caracter ASCII cero o uno. La instrucción MOV DL,0 coloca cero en DL al inicio del lazo, luego la instrucción ADC suma 30h a DL, y finalmente la añade el acarreo. Puesto que 30h es el codigo ASCII para cero, el y 31h el codigo ASCII para 1, el resultado de ADC DL,30 es el codigo ASCII para 0 cuando el acarreo esta limpio (NC) o 1 cuando el acarreo esta activado (CY).

Para correr este programa paso a paso se puede utilizar el comando T, pero es necesario recordar que cuando se alcanza la instrucción INT 21h se esta ejecutando no una sino varias instrucciones que componen la interrupcion 21h. Es por esto que llegado el momento de ejecutar dicha interrupcion lo mas conveniente es escribir el comando G 10E, 10E es la localizacion de la instrucción que continua despues de la interrupcion 21h, de esta manera se ejecutaran todas las instrucciones que forma la interrupcion 21h. Otra forma alternativa es utilizar en lugar del comando G 10E el comando P (Por proceed) el cual realiza la misma tarea pero tiene ventajas que se discutiran mas adelante.

2.8 IMPRESION DE NUMEROS HEXADECIMALES

Para lograr el obegetivo de imprimir números en hexadecimal, se necesita aprender algunas instrucciones adicionales. Aunque por el momento, a la hora de escribir programas se repetiran algunas instrucciones, mas adelante se estudiara la manera de escribir subrutinas similares a las que se utilizan en BASIC para evitar el trabajo de repetir secciones del programa.

2.8.1 COMPARACIONES Y BIT DE ACARREO

Ademas de la bandera de acarreo, mencionada anteriormente, y representado por CY o NC, el Debug muestra tambien las otras banderas, las cuales son igualmente utilizadas para conocer el estado de la última operación aritmetica. Existen ocho banderas, y se discutiran a medida que se vayan necesitando.

Recordar que CY significa que la bandera de acarreo esta en 1, o activada, mientra que NC significa que dicha bandera esta en 0 . En todas la banderas, 1 significa **cierto** y 0 significa **falso**
Por ejemplo, si se realizo una resta con un resultado de cero, la bandera conocida como bandera de cero se pondra en 1-Cierto y esto se vera en el area de banderas del despliegue de registros como ZR (Zero). Si el resultado no es cero, la bandera de cero estara en 0-NZ(Not Zero).
Si se resta uno de cero, el resultado es FFH, lo cual significa

l en complemento dos. Se puede saber del despliegue de registros si un número es positivo o negativo observando el estado de otra bandera, esta bandera llamada **bandera de signo** cambia entre NG(negative) y PL(Plus), y se pone en 1 cuando un numero es negativo en complemento dos.
Otra nueva bandera que será de interes es la bandera conocida como bandera de revalse, (overflow) la cual cambia entre OV(Overflow) cuando la bandera esta en 1 y NV (No Overflow) cuando la bandera esta en cero. La bandera de overflow es activada si el bit de signo cambia cuando no debería hacerlo. Por ejemplo, si se suman dos números positivos tales como 7000h y 7000h, el resultado es un número negativo, E000h, o -12288. Este error se debe a que el resultado sobrepasa la longitud de una palabra. El resultado debería de ser positivo, pero no es así, es por esto que el 8088 coloca la bandera de overflow.

Considere ahora, el conjunto de **instrucciones de salto condicional.** Estas instrucciones permiten chequear el estado de las banderas. La instrucción **JZ (Jump if Zero, Salte si Cero)** Salta a una nueva dirección si el resultado de la última operación aritmetica fue cero. Así, si despues de una instrucción SUB se coloca por ejemplo, la instrucción JZ 15A, un resultado de cero en la resta causará que el 8088 salte a, y comience a ejecutar, las instrucciones a partir de la dirección 15A en lugar de la siguiente instrucción despues de JZ.

La instrucción JZ prueba la bandera de cero, y si está activada (ZR),Se realiza el salto correspondiente. El opuesto de JZ es **JNZ (Jump if Not Zero).** Considere un ejemplo sencillo que utiliza JNZ para restar uno de un número hasta que el resultado es cero:

```
396F:0100  2C01          SUB  AL,01
396F:0102  75FC          JNZ  0100
396F:0104  CD20          INT 20
```

Se podrá haber notado que utilizando SUB para comparar dos numeros, tiene el efecto indeseable de cambiar el primer número. Otra instrucción,**CMP (Compare)** permite realizar la resta sin almacenar el resultado de la misma, es decir, sin alterar el primer numero, el resultado es utilizado solo para establecer las banderas.

2.8.2 IMPRESION DE UN DIGITO HEXADECIMAL

Se comenzará colocando un numero pequeño (entre 0 y Fh) en el registro BL. Puesto que cualquier número entre 0 y Fh es equivalente a un digito he adecimal, se podrá convertir el número elegido a un solo caracter ASCII y luego imprimirlo. Se Analizaran los pasos que se necesitan seguir para lograr esta conversion.
Los caracteres ASCII de 0 a 9 tienen valores de 30h a 39h; los

caracteres de la A a la F, sin embargo, tienen valores de 41h a 46h. Aquí resulta un problema: estos dos grupos de caracteres ASCII están separados por 7 caracteres. Como resultado, la conversión a ASCII será diferente para los dos grupos de números (0h a 9h y Ah a Fh), por lo tanto, se deben manipular cada grupo diferentemente. Un programa BASIC para hacer esta conversión de dos partes se vería como esto:

```
100 IF BL    %H0A
       THEN BL = BL + %H30
       ELSE BL = BL + %H37
```

Este programa BASIC es claramente sencillo. Desafortunadamente, el lenguaje de maquina del 8088 no incluye una instrucción ELSE; Por lo tanto hay que ser hábiles, y modificarlo. otro programa en BASIC que imita el metodo que se utilizará en el programa en lenguaje de maquina es:

```
100 BL = BL + %H30
110 IF  BL   = %H3A
       THEN BL = BL + %H7
```

La version en lenguaje de máquina de este programa contiene unos pocos pasos más. Se utiliza la instrucción CMP, así como también la instrucción de salto incodicional llamada **JL (Jump if Less Than-Salte si es memor que)**. El programa toma un solo dígito hexadecimal en el registro BL y lo imprime en Hexadecimal:

```
3985:0100  B402          MOV   AH,02
3985:0102  88DA          MOV   DL,BL
3985:0104  80C230        ADD   DL,30
3985:0107  80FA3A        CMP   DL,3A
3985:010A  7C03          JL    010F
3985:010C  80C207        ADD   DL,07
3985:010F  CD21          INT   21
3985:0111  CD20          INT   20
```

La instrucción CMP, como se dijo antes, resta dos números (DL 3Ah) para activar las banderas, pero no realiza cambios en DL. A si DL es menor que 3Ah, la instrucción JL 10F salta a la instruccion INT 21 en la localización 10F.

Para lograr una mejor comprension de la instruccion CMP lo que se hará es colocar un numero de un solo digito en el registro BL luego ir ejecutando cada instrucción con el comando T (paso a paso) sin olvidar que al llegar a la instrucción INT se debe de utilizar el comando G con un punto de ruptura, en lugar del comando T

2.8.3 OTRA INSTRUCCION DE ROTACION

El programa anterior trabaja cualquier número de un solo digito, pero si se desea escribir un número de dos digitos, se necesitaran unos pocos pasos mas. Se necesita aislar cada digito (cuatro bits, los cuales a menudo se llaman **nibble**) del numero de dos digitos. Considere como se realiza esta tarea.

Para comenzar, hay que recordar que la instrucción RCL rota un byte o una palabra a la izquierda, a través de la bandera de acarreo un cierto numero de veces. Anteriormente se utilizo la instrucción RCL BL,1 en la cual se le dice al 8088 que rote el registro BL en un bit. Se puede rotar en mas de un bit si se desea, como se ilustró antes. Pero, no se puede simplemente escribir la instrucción RCL BL,2. (NOTA: Aunque RCL BL,2 no es legal en las instrucciones del 8088, trabaja correctamente en el 80286. Pero ya que las computadoras mas antiguas IBM PCs son mas comunes, es mejor escribir el programa para este común denominador- las viejas 8088). Para rotaciones de mas de un bit, se debe de colocar un contador de rotación en el registro CL.

El registro CL se utiliza aquí en la misma forma que el registro CX se utilizo con la instrucción LOOP para determinar el numero de veces que se repite un lazo.

¿Como hacer para enlazar todo esto con el programa que imprime un numero hexadecimal de dos digitos?

El plan ahora es rotar el byte en el registro DL cuatro bits a la derecha. Para hacer esto, se utilizara una instrucción de rotación un poco distinta llamada **SHR (Shift right)** Utilizando SHR, se podrá mover los cuatro bit superiores del numero a los cuatro bits inferiores.
Se desean también que los cuatro bits superiores de DL se pongan en cero, de manera que el registro entero convierta al byte que se esta trasladando en el nibble de la derecha. Al utilizar SHR DL,1 se moverá el byte en DL un bit a la derecha, y **al mismo tiempo**, se moverá el bit cero a la bandera de acarreo, mientras se esta introduciendo un cero en el bit 7 (El mas significativo) Si se repite este procedimiento tres veces mas, los cuatro bits superiores (los mas significantes) terminaran en los cuatro bits inferiores (los menos significantes) mientras que los cuatro bits superiores tendrán almacenados ceros. Resumiendo lo dicho anteriormente se puede escribir un programa parcial para tomar un numero en el registro BL y luego imprimir el primer digito Hexadecimal:

```
3985:0100 B402          MOV   AH,02
3985:0102 88DA          MOV   DL,BL
3985:0104 B104          MOV   CL,04
3985:0106 D2EA          SHR   DL,CL
3985:0108 80C230        ADD   DL,30
```

continuación.

```
3995:0100 80FA3A        CMP   DL,3A
3998:0103 7C03          JL    0108
399A:0105 80C207        ADD   DL,07
399D:0108 CD21          INT   21
399F:010A CD20          INT   20
```

NOTA: Los programas del presente capitulo no llevan comentarios por que se han escrito con el Debug, cuando se comience a escribir programas utilizando el assembler se hará uso de este para recurso para una mayor aclaración de la lógica de los programas

2.8.4 LOGICA Y LA INSTRUCCION AND

Ahora que ya se sabe como imprimir el primero de los dos digitos de un numero hexa. Considere como se puede aislar e imprimir el segundo digito. Lo que se tiene que hacer, es poner en cero los cuatro digitos superiores del numero original (antes de rotarlo) dejando DL igual a los cuatro bits inferiores. Para poner los cuatro bits superiores a cero utilizará la instrucción llamada AND. Esta instrucción es una de las instrucciones logicas (Aquellas que tienen su origen en la logica formal). Considere como trabaja AND. Vease ane o de instrucciones del 8088 en el ane o

En la logica formal, se puede decir, "A es cierto, si B y C son ciertas ambas". Pero si ya sea C o B es falsa, entonces A debe tambien sea falsa. Si en estas instrucciones se sustituye uno por cierto y cero por falso, se veran la varias combinaciones de A, B y C. Se puede crear lo que se conoce como **tabla de verdad** la tabla de verdad para la instrucción AND es:

A	B	C = A and B
0	0	0
0	1	0
1	0	0
1	1	1

En la tabla anterior se puede ver que 0 AND 1 es 0, etc.

La instrucción AND trabaja sobre bytes y sobre palabras operando los bits de cada byte o palabra que estan en la misma posicion. Por ejemplo la instruccion AND BL,CL opera sucesivamente los bit cero de BL y CL, luego los bit 1, los bits 2, y asi sucesivamente, y coloca el resultado en BL. Haciendo esto mas claro con un ejemplo en binario:

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

Operando cualquier numero con 0F, se puede poner a cero los cuatro bits mas significantes. Ejemplo:

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

Esta logica se codifica en un corto programa que toma un numero en DL, aisla el digito hexadecimal menos significante operando con 0F y luego imprime el resultado como un caracter:

```
3985:0100 B402          MOV   AH,02
3985:0102 88DA          MOV   DL,BL
3985:0104 80E20F        AND   DL,0F
3985:0107 80C230        ADD   DL,30
3985:010A 80FA3A        CMP   DL,3A
3985:010D 7203          JL    0112
3985:010F 80C207        ADD   DL,07
3985:0112 CD21          INT   21
3985:0114 CD20          INT   20
```

COLOCANDO TODO JUNTO

Realmente no hay mucho que cambiar cuando se desea colocar los dos procedimientos en uno solo. Se necesita cambiar unicamente la direccion de la segunda instruccion JL que se utiliza para imprimir el segundo digito hexadecimal. El programa completo que realiza la tarea de escribir un numero hexadecimal de dos digitos se muestra a continuacion:

```
3985:0100 B402          MOV   AH,02
3985:0102 88DA          MOV   DL,BL
3985:0104 B104          MOV   CL,04
3985:0106 D2EA          SHR   DL,CL
3985:0108 80C230        ADD   DL,30
3985:010B 80FA3A        CMP   DL,3A
3985:010E 7204          JL    0114
3985:0110 80C207        ADD   DL,07
```

Continuacion.

```
3985:0113 CD21          INT   21
3985:0115 88DA          MOV   DL,BL
3985:0117 80E20F        AND   DL,0F
3985:011A 80C230        ADD   DL,30
3985:011D 80FA3A        CMP   DL,3A
3985:0120 7203          JL    0125
3985:0122 80C207        ADD   DL,07
3985:0125 CD21          INT   21
3985:0127 CD20          INT   20
```

Una vez que se ha escrito este programa, se tendrá que escribir el comando **U 100**. Despues de este comando se verá el listado completo en mnemonico. Note que se ha repetido un conjunto de 5 instrucciones: Las instrucciones de 100h hasta 113h, y de la 11Ah hasta la 125h. Mas adelante se vera como escribir un secuencia de instrucciones solo una vez utilizando una instruccion similar la la instrucción GOSUB de BASIC

2.9 LECTURA DE CARACTERES

Ahora que ya se sabe como imprimir un byte en notación hexadecimal, se hará un proceso inverso leyendo dos caracteres hexadecimales desde el teclado y convirtiendolos en un solo byte.

2.9.1 LEYENDO UN CARACTER

La función 1 de la interrupción INT 21 del DOS se utiliza para la lectura de caracteres desde el teclado. Anteriormente se dijo que el número de la función se debe de colocar en el registro AH antes de la llamada a la interrupcion. Ver tabla en anexo. Si se escribe lo siguiente en la localización 0100h:

```
3985:0100 CD21          INT   21
```

y luego se coloca 01h en AH y se escribe el comando G 102 o P para correr esta unica instruccion, se verá el cursor parpadeando, lo que significa que el DOS se ha detenido y esta esperando hasta que se presione un tecla. Una vez presionada una tecla, el DOS coloca el codigo ASCII para dicho caracter en el registro AL. Esta intrucción se utilizará mas tarde para leer los caracteres de un número hexadecimal. Por ahora observe que sucede cuando se presiona una tecla como F1.
P se utiliza cuando la instruccion a ejecutar es INT o es CALL, para que no se ejecute cada una de las instrucciones que componen ya sea la interrupcion o la llamada. Es equivalente a escribir G , en donde es la direccion de la intrucción que se encuentra depues de INT o de CALL segun sea el caso.

Si se presiona la tecal F1, el DOS retornará un cero en AL, y también se verá un punto y coma aparecer junto al guion indicador del Debug. Lo que ha sucedido es que F1 es una de un conjunto de teclas especiales con **codigo extendido** las cuales el DOS trata de manera diferente. Para cada una de estas teclas especiales el DOS envía **dos** caracteres, uno a la derecha del otro. El primer caracter retornado es siempre cero, indicando que un siguiente caracter es el **codigo scan** para la tecla especial.
Para leer ambos caracteres, se necesita ejecutar la interrupción INT 21h dos veces. Pero en el ejemplo, solo se ha leido el primer caracter, el cero, y se dejo el codigo scan. Cuando el debug finaliza con el comando G 102 (o P), comenzo a leer caracteres, y el primer caracter leido fue el codigo scan dejado anteriormente y el cual es 59, que es codigo ASCII para el punto y coma.

2.9.2 LECTURA DE UN SOLO DIGITO DE UN NUMERO HEXADECIMAL

Si se analiza en reversa la conversion utilizada anteriormente, cuando se transformó un solo digito he adecimal en un codigo ASCII por uno de los caracteres en el rango de 0 a 9 o de la A a la F, para convertir un caracter tal como C o D, desde un digito he adecimal en ASCII a un byte, se debe restar ya sea 30h (para los digitos del 0 al 9) o 37h (para los digitos de la A a la F). un simple programa que leerá un caracter ASCII y lo convertirá a un byte es:

```
3485:0100    B401        MOV     AH,01
3485:0102    CD21        INT     21
3485:0104    2C30        SUB     AL,30
3485:0106    3C09        CMP     AL,09
3485:0108    7E02        JLE     010C
3485:010A    2C07        SUB     AL,07
3485:010C    CD20        INT     20
```

la mayoría de estas instrucciones ya son familiares, pero ahora hay una nueva, **JLE (Jump than or Equal to-Salte si es menor o igual que).** En este programa, esta instrucción salta si AL es menor o igual a 9h.

Para ver la conversión del caracter ASCII a he adecimal, se necesita ver el registro AL e actamente antes de que la instrucción INT 20 sea ejecutada. Puesto que el Debug restaura los registros cuando se ejecuta dicha instrucción, se necesita establecer un punto de ruptura, como se hizo anteriormente. Aquí, se escribirá G 10C, y se verá que el registro AL contiene el número he adecimal convertido a partir del caracter.

Si se intenta escribir algunos caracteres tales como + o la letra minuscula d, que no son digitos he adecimales, se notará que este programa trabaja correctamente solo cuando la entrada es uno de los digitos del 0 al 9 o las letras mayusculas de la A a

la F. Mas adelante se corregiran esta fallas menores cuando se aprenda acerca de las subrutinas o procedimientos.

2.9.3 LECTURA DE UN NUMERO HEXADECIMAL DE DOS DIGITOS.

La lectura de dos digitos hexadecimales es mucho mas complicada que la lectura de uno solo, pero no se requieren muchas mas instrucciones. Se comenzará leyendo el primer digito, luego se colocará su valor hexadecimal en el registro DL y se multiplicará por 16. Para ejecutar esta multiplicacion, se rotará el registro DL 4 bits a la izquierda, colocando un digito hex a cero (los cuatro primeros bits) a la derecha del digito que se acaba de leer. La instrucción **SHL DL,CL**, con CL puesto a cuatro hace el truco de insertar ceros al lado derecho. En realidad, la intrucción SHL (Shift Left) es conocida como una **rotación aritmetica** por que tiene el mismo efecto que una multiplicación por dos, cuatro, ocho y asi sucesivamente, dependiendo del numero (tal como uno, dos o tres) en CL

Finalmente, con el primer digito rotados, se sumara el segundo digito hexadecimal al número en DL (el primer digito multiplicado por 16) lo que se resume en el siguiente programa:

```
3985:0100  B401        MOV  AH,01
3987:0102  CD21        INT  21
3989:0104  8AD3        MOV  DL,AL
398B:0106  80EA30      SUB  DL,30
398E:0109  80FA09      CMP  DL,09
3991:010C  7E03        JLE  0111
3993:010E  80EA07      SUB  DL,07
3996:0111  B104        MOV  CL,04
3998:0113  D2E2        SHL  DL,CL
399A:0115  CD21        INT  21
399C:0117  2C30        SUB  AL,30
399E:0119  3C09        CMP  AL,09
39A0:011B  7E02        JLE  011F
39A2:011D  2C07        SUB  AL,07
39A4:011F  02D0        ADD  DL,AL
39A6:0121  CD20        INT  20
```

Ahora que se ha escrito el programa, es buena idea revisar las condiciones extremas para confirmar que el programa esta trabajando correctamente. Para estas condiciones extremas, utilizar los números 00, 09, 0A, 0F, 90, A0, F0, y algunos otros números tales como 3C. hay que utilizar un punto de ruptura para correr el programa sin ejecutar la instrucción INT 20h (Ademas hay que asegurarse que las letras se escriban en mayusculas)

2.10 PROCEDIMIENTOS FAMILIARES DE LAS SUBRUTINAS

En esta sección, se aprenderá acerca de las subrutinas y del lugar especial para colocar números llamado stack.

2.10.1 PROCEDIMIENTOS

Un procedimiento es una lista de instrucciones que se pueden ejecutar y llamar desde muchos lugares diferentes de un programa, en lugar de tener que repetir la misma lista de instrucciones en cada lugar que se necesitan. En BASIC tales listas son llamadas **subrutinas**, pero en el caso de 8088 se llaman **procedimientos** por razones que que se aclararan mas adelante.

Para moverse desde y hacia un procedimiento, se hace igual que en BASIC. Se llama el procedimiento con la instrucción **CALL**, la que es equivalente al GOSUB de BASIC. Y para retornar del procedimiento se utiliza la instrucción **RET** la que es equivalente a la instruccion RETURN de BASIC.

Aqui se presenta un programa simple en BASIC que mas tarde se reescribirá en lenguaje de máquina. Este programa llama una subrutina 10 veces. Cada vez que se llama imprime un caracter comenzando con A y terminando con J:

```
  10  A = &H41
  20  FOR I=1 TO 10
  30  GOSUB 1000
  40  NEXT I
  50  END
1000  PRINT CHR$(A);
1010  A = A + 1
1020  RETURN
```

La subrutina, como es una práctica comun en BASIC, comienza en la linea 1000 para dejar espacio para añadir mas instrucciones en el programa principal sin afectar la subrutina. Se hará lo mismo con el procedimiento escrito en lenguaje de máquina colocandolo en la localización 200h, lejos del programa principal en la localización 100h. La instrucción CALL colocará IP a 200h, y el 8088 comenzará a ejecutar la instrucción 200h. La parte del FOR NEXT se hará como se hizo anteriormente con la instrucción LOOP. Las otras lineas del programa son ya familiares:

```
3985:0100  B041        MOV   DL,41
3985:0102  B90A00      MOV   CX,000A
3985:0105  E8F800      CALL  0200
3985:0108  E2FB        LOOP  0105
3985:010A  CD20        INT   20
```

La primera instrucción coloca 41h (ASCII para A) en el registr DL, ya que la instrucción INT 21h imprime el caracter dado por e codigo ASCII en DL. La instrucción INT 21h esta colocada lego de la localización del procedimiento, el cual se deberá coloca en la posición 200h:

```
0100:0100  B402          MOV   AH,02
0100:0102  CD21          INT   21
0100:0104  FEC2          INC   DL
0100:0106  C3            RET
```

Aqui hay dos instrucciones nuevas y dos viejas, recordemos que e 0,0h en AH le dice al DOS que imprima el caracter cuyo codig ASCII esta en DL cuando se ejecute la instrucción INT 21h. **IN DL,** la primera de las nuevas instrucciones, **incrementa** e registro DL. Es decir, suma uno a DL. La otra nuev instrucción, **RET, retorna** a la primera instrucción que sigu despues de CALL en el programa principal.

2.10.2 EL STACK Y LAS DIRECCIONES DE RETORNO

La instrucción CALL necesita salvar la **dirección de retorno** en algun lugar de manera que el 8088 sepa donde regresar y continuar ejecutando las instrucciones cuando encuentre la instrucción RET. Para el lugar donde se almacena esta direccion, se tiene una porción de la memoria conocida como **stack.** Para saber donde esta ubicado el stack hay dos registros que se pueden ver en el despliegue de registros: el registro **SP (stack Pointer),** el cual apunta a la parte superior del stack, y el registro **SS (stack segment),** el cual almacena un segmento de la dirección del stack.

La operación del Stack para el 8088,y en general para todo microprocesador, se asemeja a un pila de trastes en un cafeteria, donde se coloca un traste sobre de otro. El último traste en colocar en la pila es el primero en salir de ella, es por eso que el otro nombre para el stack es **LIFO,** por Last In, First Out (Ultimo en entrar primero en salir). En general LIFO es precisamente lo que se necesita para recuperar las direcciones de retorno despues de que se han **anidado** llamadas (CALL) como en el siguiente caso:

```
096F:0100  E81D00        CALL  0200
                            .
                            .
0200:0200  E80D00        CALL  0300
0203:0203  C3            RET
```

Continuación

```
CS:0300  E8FD00        CALL  0400
CS:0303  C3            RET

CS:0400  C3            RET
```

Aquí la instrucción en la 100h llama una en la 200h, la cual llama una en 300h, la que a su ver llama una en la 400h, en donde finalmente se retorna con la última instrucción RET. Esta instrucción retorna a la instrucción que continua despues de la pre la instrucción de llamada (CALL) en 300h, así el 8088 regresa a ejecutar la instruccion 303h, la cual e trae la siguiente vieja dirección (203)h del stack. luego el 8088 regresa a ejecutar la instruccion en 203h, que es un RET y así sucesivamente. Cada RET **recupera** la dirección de retorno que esta en el tope del stack, así cada RET sigue la misma trayectoria de regreso dejada por las llamadas que se hicieron anteriormente. Ver las figura 2.5

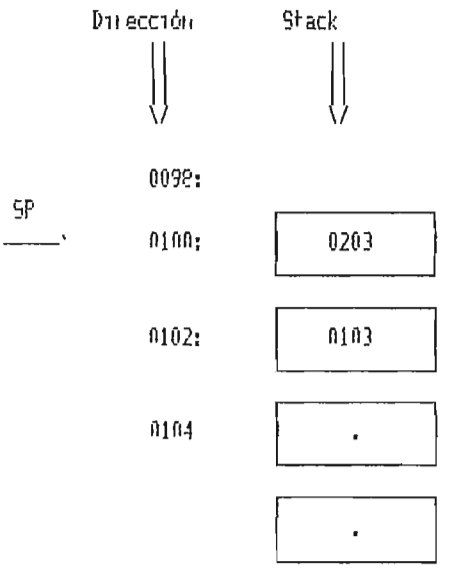


Figura 2.5 El stack exactamente antes de ejecutar CALL 400

2.10.3 PONIENDO Y SACANDO DEL STACK

El stack es un lugar utilizado para almacenar datos temporalmente. Si se utiliza el stack, se debe tener mucho cuidado de restaurarlo antes de una instrucción RET. Ya se vio que la instrucción CALL pone la dirección de retorno (una palabra) en el tope del stack, mientras que la instrucción RET recupera esta palabra del tope del stack, y la carga dentro del

registro IP, y deja en el tope la palabra que fue almacenada antes que dicha instrucción de retorno se ejecute. Se pueden hacer muchas cosas con el stack utilizando la instrucciones **PUSH** y **POP**, las cuales permiten poner y sacar palabras del stack. ¿Cuando se podría querer hacer esto?

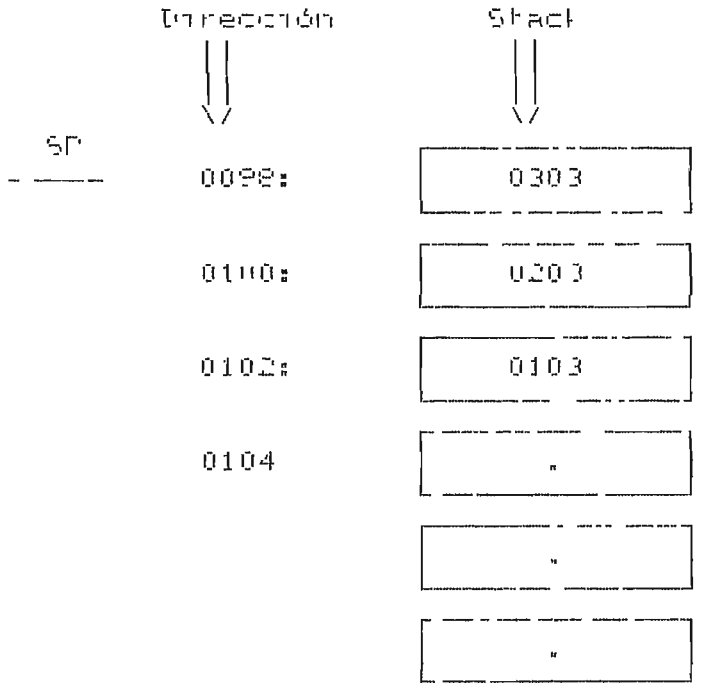


Fig 2.6 El stack exactamente despues de ejecutar CALL 400

Es conveniente salvar los valores de registros al inicio de un procedimiento y recuperarlos despues de terminarlo, exactamente antes de la instrucción RET, así se podrían utilizar libremente los registros de la forma que se desee dentro de los procedimientos, siempre que se recuperen sus valores al final.

Los programas estan construidos con muchos niveles de procedimientos, con cada nivel llamando procedimientos en el siguiente nivel inferior. Salvando los registros al inicio del procedimiento y recuperandolos al final, así se evitan interacciones indeseadas entre procedimientos en los diferentes niveles, y se hace mucho mas facil el trabajo del programador. A continuación se muestra un ejemplo en el cual se salva y se recupera CX y DX: (NO CORRERLO)

```
3FAF:0200    51              PUSH    CX
3FB0:0201    52              PUSH    DX
3FB1:0202    B90800          MOV     CX,0008
```

Continuación

```
xxxx:0205    E8F800          CALL    0300
xxxx:0208    FEC2            INC     DL
xxxx:020A    E2F9            LOOP    0205
xxxx:020C    5A              POP     DX
xxxx:020D    59              POP     CX
xxxx:020F    C3              RET
```

Note que las instrucciones POP está en orden invertido de las instrucciones PUSH, ya que POP recupera la palabra que se almacenó más recientemente en el stack, y el antiguo valor de DX se recupera primero que el valor de CX.

Salvar y restaurar CX y DX permite cambiar estos registros en procedimiento que comienza en 200h. Una vez que se han salvado CX y DX se puede utilizar estos registros para almacenar variables locales.(variables que pueden utilizarse dentro de este procedimiento) sin afectar los valores utilizados por el programa que realiza la llamada.

Se utilizan tales variables locales para simplificar la tarea de pogramación, siempre que se tenga cuidado de restaurar los valores originales, no hay que preocuparse por los cambios de cualquier registro utilizado por el programa que llama el procedimiento.

2.10.4 LECTURA DE NUMEROS HEXADECIMALES MEDIANTE UN PROCEDIMIENTO

Se desea crear un procedimiento que mantenga la lectura de caracteres hasta que reciba uno que pueda convertirlo a un número hexadecimal entre 0 y Fh . No se desea mostrar cualquier caracter invalido. Se enmarará la entrada utilizando una nueva función para la instrucción INT 21h, la función 8, que lee un caracter pero no lo muestra en la pantalla . De esta forma se pueden desplegar los caracteres solo si son válidos.

Utilizando esta función, el programa puede leer caracteres sin mostrarlos hasta que lea un digito hexadecimal válido (de 0 a 9 ó de A a F), los cuales se mostrarán en pantalla. A continuación se muestra el procedimiento para hacer esto y convertir el caracter hexadecimal a un número hexadecimal:

```
3985:  0200 52              PUSH    DX
3987:  0201 B408            MOV     AH,08
3989:  0203 CD21            INT     21
398B:  0205 3C30            CMP     AL,30
398D:  0207 72FA            JB      0203
398F:  0209 3C46            CMP     AL,46
```

Continuación

```
0900:0208 77F6          JA      0200
0905:020A 3C39          CMP     AL,39
0907:020C 77DA          JA      01E8
0909:020E B402          MOV     AH,02
090B:0210 8AD0          MOV     DL,AL
090D:0212 CD21          INT     21
090F:0214 2C30          SUB     AL,30
0911:0216 5A            POP     DX
0912:0217 C3            RET
0913:0218 3C41          CMP     AL,41
0915:021A 72E4          JB      0200
0917:021C B402          MOV     AH,02
0919:021E 8AD0          MOV     DL,AL
091B:0220 CD21          INT     21
091D:0222 2C37          SUB     AL,37
091F:0224 5A            POP     DX
0920:0225 C3            RET
```

El procedimiento lee un caracter en AL (con INT 21h en 203h) y chequea para ver si es válido con el salto condicional JA o JE asociado con CMP, si el caracter que se acaba de leer no es un caracter válido, la instrucción del salto condicional lo envía de regreso a la localización 203, donde INT 21h lee otro caracter (JA (Jump if above, y JB es Jump if Below), ambos tratán los números como numeros sin signo, mientras que la instrucción JL se usa anteriormente para numeros con signo)

Por la línea 20Fh, se sabe si se tiene un dígito válido entre 0 y 9 y así restar el código para el caracter 0, y devolver el resultado en AL; recordando de recuperar el registro DX el cual fue salvado al inicio del procedimiento. El proceso para digitos hexadecimales de la A a la F es bastanto parecido.

A continuación se muestra un programa simple para mostrar el procedimiento:

```
0920:0100 E8FD00        CALL    0200
0923:0103 CD20          INT     20
```

Como se hizo anteriormente, se utiliza el comando G con un punto de ruptura, ó se utiliza el comando P si se desea ejecutar la instrucción CALL 200h sin ejecutar la instrucción INT 20h, así se pueden ver los registros antes de que el programa termine y sean restaurados los registros.

El comando P ejecuta la misma función que el comando G con un punto de ruptura en la siguiente instrucción. Se utiliza este comando cuando se ejecuta una llamada a una subrutina, una

interrupción o un lazo. EL formato del comando P es el siguiente:

P [=dirección][,valor]

NOTA: Se puede utilizar el comando G con un punto de ruptura adecuado, es decir poner como punto de ruptura la dirección de la instrucción que este despues de la última instrucción que se desee ejecutar.

Si se especifica una dirección, el Debug ejecuta la instrucción en la dirección especificada, si se especifica valor, el Debug ejecuta el numero de instrucciones especificado por ese valor. Por ejemplo:

P =100 ejecuta la instrucción que se encuentra en la localización 100h, que es una llamada a una subrutina, la cual se ejecutará completamente.

Se verá el cursor en el lado izquierdo de la pantalla esperando pacientemente por un caracter. Si se escribe una X, el cual es un caracter invalido no sucederá nada. Ahora si se escribe cualquier caracter hexadecimal en letras mayusculas, se verá el valor del caracter hexadecimal en AL y el caracter mismo se mostrará en la pantalla.

Ahora que ya se tiene este procedimiento, el programa para leer un numero hexadecimal de dos digitos, sin errores de manipulación, esta claramente al alcance:

```
0F75:0100 E8F500          CALL    0200
0F75:0103 8AD0            MOV     DL,AL
0F75:0105 B104            MOV     CL,04
0F75:0107 D2E2            SHL     DL,CL
0F75:0109 E8F400          CALL    0200
0F75:010C 02D0            ADD     DL,AL
0F75:010E B402            MOV     AH,02
0F75:0110 CD21            INT     21
0F75:0112 CD20            INT     20
```

Aquí se puede ver la razón de salvar el registro DX en el procedimiento. El programa almacena el numero hexadecimal en DL, y no se desea que el procedimiento en 200h cambie DL. Por otro lado, el procedimiento en 200h utiliza el mismo DL para mostrar caracteres. Así, por la utilización de la instrucción PUSH DX en el inicio del procedimiento y, POP DX en el final, se evitarón problemas.

2.11 PROGRAMACION CON EL ASSEMBLER

En la sección anterior, se mencionó la estructura de un programa en lenguaje de ensamble cuando se utiliza el assembler. Recordar que existen dos tipos de instrucciones para el Assembler: Las directivas, y las instrucciones de lenguaje de ensamble. En esta sección se darán suficientes ejemplos para ver como se relacionan en un programa ambos tipos de instrucciones y para aclarar posibles dudas que aun permanezcan en lo que se refiere al assembler.

2.11.1 UN PROGRAMA SIN EL DEBUG

Hasta este punto, los programas se han hecho utilizando el Debug. Ahora se dejara el Debug a un lado, y los programas se escribirán utilizando un editor para crear el programa fuente. En la sección 2.6.3 se hizo un programa llamado ASTER.COM, en esta sección se hará un version de este programa utilizando el assembler. A continuación se muestra la version original hecha con el debug:

```
3960:0100 B402    MOV AH,02
3962:0102 B22A    MOV DL,2A
3964:0104 CD21    INT 21
3966:0106 CD20    INT 20
```

Utilizando un editor, se escribirá las siguientes instrucciones en el archivo llamado ASTER.ASM (La extension ASM significa que se trata de un archivo fuente en lenguaje de ensamble). Aunque la utilización de letras mayusculas o minusculas es indiferente, se utilizaran las primeras para evitar confundir la letra l (ele) y el numero 1 (uno):

```
CODE_SEG    SEGMENT
            MOV AH,2h
            MOV DL,2Ah
            INT 21h
            INT 20h
CODE_SEG    ENDS
            END
```

Nota que hay una h despues de cada número hexadecimal. Esta h le dice al assembler que el numero esta en hexadecimal. A diferencia del Debug, el cual asume que los numeros estan en hex, el assembler asume que todos los numeros estan en decimal.

NOTA: Todo numero hexadecimal que comience con letra, deberá de ser precedido por el numero 0 para evitar confusiones en el assembler, es decir un número como ACh se deberá de escribir como 0ACh.

Los operadores que utilizan el archivo fuente, son unicamente para efecto de hacer mas claros los programas.

Las directivas (o tambien llamadas seudo operaciones) se utilizan para definir las fronteras del segmento y el tamaño del archivo fuente

2.11.2 CREACION DE ARCHIVOS FUENTE

Una vez escrito el archivo fuente que contiene el programa ASTER.ASM, es necesario mencionar que el assembler utiliza archivos que contienen caracteres ASCII estandar. Si utiliza un procesador de palabras hay que recordar que no todos los procesadores de palabra escriben archivos utilizando unicamente el codigo ASCII estandar. WordStar y Microsoft Word, son algunos de estos procesadores. Para cualquiera de estos dos procesadores se debe de utilizar el modo de NO documento o Modo NO formateado para la creación de archivos que se utilizaran con el assembler.

A continuación se mostrara por medio de un ejemplo los pasos que se deben de seguir para compilar un programa fuente.

Para compilar el archivo ASTER.ASM se escribe el siguiente comando (asegurarse de escribir el punto y coma)

```
A MASM ASTER;
The IBM Personal Computer Assembler
Version 1.00 (C) Copyright IBM Corp 1981

Warning Severe
Errors  Errors
0       0

A
```

Despues de observar lo que se mostro anteriormente, todavía no esta concluido el trabajo, el assembler ha producido un archivo llamado ASTER.OBJ, el cual estara en el disco que este habilitado. Este es un archivo intermedio llamado **archivo objeto.** Contiene información para otro programa del DOS llamado **linker.**

2.11.3 EL ENCADENADOR (LINK)

Una vez creado el archivo objeto, en el paso 1, se creará el archivo ejecutable (extension .EXE) (Se debe de copiar el archivo LINK.EXE en el disco o directorio que contiene el archivo fuente y el compilado) Luego se escribirá el siguiente comando:

```
A LINK ASTER;
IBM Personal Computer Linker
Version 1.10 (C) Copyright IBM Corp 1982

Warning: No STACK segment

There was 1 error detected.

A
```

¿Un error?. En realidad NO; el linker cuenta las advertencias como errores (en algunas versiones de MS-DOS, el linker no reporta las advertencias como errores.) Aunque el linker advierta que no hay segmento para el stack, no se necesita en este ejemplo. En este punto ya se tiene un archivo EXE, pero todavía no es el último paso, hay un paso mas. Se tiene que crear un archivo .COM el cual es el que fue creado con el Debug. Mas adelante se aprenderá cuando se necesitan todos estos pasos.

Para crear el archivo .COM se necesita el programa EXE2BIN.EXE, este archivo lo que hace es convertir un archivo .EXE en .COM. Escribiendo el siguiente comando se crea el archivo .COM

```
A EXE2BIN ASTER ASTER.COM

A
```

Es posible que este procedimiento parezca muy tedioso, pero entre otras cosas no se ha tenido que preocuparse en que lugar de la memoria se colocó el programa, ni se tuvo que estar manipulando el registo IP. El uso de procedimientos se hará mucho más fácil como se verá mas adelante. De todos estos detalles se encargan las directivas.

Si se regresa momentaneamente al Debug, y se escribe el siguiente comando:

```
A DEBUG ASTER.COM
-U
397F:0100 B402    MOV   AH,02
397F:0102 B20A    MOV   DL,2A
397F:0104 CD21    INT   21
397F:0106 INT     INT   20
```

Es exactamente igual al que se escribió en la sección 2.6.3. Esto indica que todo lo que el debug ve es el programa ASTER.COM. Ninguna de las directivas ha aparecido en el programa .COM. Las directivas unicamente suministran información para que el assembler pueda realizar su trabajo.

2.11.4 COMENTARIOS

Cuando se trabaja con el assembler se puede agregar comentarios que se colcan despues de un punto y coma (;). Estos comentarios son de gran utilidad para clarificar los programas o de definir el proposito de un procedimiento completo. Considere como se veria el programa anterior si se utilizan comentarios.

```
CODE_SEG    SEGMENT
          MOV AH,2h       ;Selecciona función 2, Salida de caracter
          MOV DL,2Ah      ;Carga codigo ASCII para '*' a imprimir
          INT 21h         ;Imprime '*' con INT 21
          INT 20h         ;Salida al DOS
CODE-SEG    ENDS
          END
```

2.11.5 ETIQUETAS

Las etiquetas son otra de las caracteristicas que hacen la programación mas clara.
Hasta ahora cuando se tuvo que realizar una bifurcación se ha tenido que colocar la dirección a la que se desea saltar. En la programación diaria, la inserción de nuevas lineas cambia dicha direccion. El assembler toma cuidado de estos cambios utilizando las etiquetas, las cuales le dan nombre a la dirección de la instrucción o localización de memoria. Una etiqueta toma el lugar de una direccion. Tan pronto como el assembler ve una etiqueta la reemplaza con la dirección correcta antes de enviarla al 8088.

las etiquetas pueden tener hasta 31 caracteres de longitud pueden contener letras, números, y cualquiera de los siguientes simbolos: ?, .., @, _, $. Pero no puede comenzar con ningun digito entre 0 y 9. El punto se puede utilizar unicamente como el primer caracter. Como ejemplo practico, se tomará el programa de la sección 2.9.3, el cual contien dos saltos, JLE 0111 y JLE 011F. Ciertamente no es muy obvio lo que ese programa hace, a continuación se verá una version mucho mas clara utilizando el assembler.

```
CODE_SEG     SEGMENT

        ASSUME CS:CODE_SEG

        MOV   AH,1h           ;Función 1 del DOS, leer caracter
        INT   21h             ;Lee caraRter almacena cod ASCII en AL
        MOV   DL,AL           ;Mueve codigo ASCII a DL
        SUB   DL,30h          ;Convierte en digito de 0 a 9
        CMP   DL,9h           ;¿Es un digito de 0 a 9?
        JLE   DIGIT1          ;Si, se tiene primer digito (4 bits)
        SUB   DL,07h          ;No, convertir a letra de la A a la F
```

Continuación

```
DIGIT1:
        MOV   CL,4h             ;Prepara para trasladar 4 bits
        SHL   DL,CL             ;Convierte 4 bits superiores
        INT   21h               ;Consigue el siguiente caracter
        SUB   AL,30h            ;Repite conversion
        CMP   AL,9h             ;¿Es digito entre 0 y 9
        JLE   DIGIT2            ;Si, obtener el siguiente digito
        SUB   AL,7h             ;No, restar 7
DIGIT2:
        ADD   DL,AL             ;Suma el segundo digito
        INT    20h              ; terminar y salir
COD_SEG     ENDS
        END
```

Las etiquetas, DIGIT1 y DIGIT2, son del tipo conocido como NEAR, esto es porque terminan con dos puntos (:) El termino NEAR tiene que ver con segmentos, es decir que dicha localización se encuentra en el mismo segmento que el programa fuente. ASSUME, SEGMENT, Y ENDS son directivas que dan nombre al registro CS y definen el segmento que contiene el archivo fuente respectivamente

2.12 PROCEDIMIENTOS Y EL ASSEMBLER

En esta sección se verá que es mucho mas facil escribir procedimientos con el assembler. Se construirá un programa que será util cuando se escriba un programa mas completo para explorar el contenido de los discos.

Se comenzará con dos procedimientos para imprimir un byte en hexadecimal y se utilizarán mas directivas.

2.12.1 LOS PROCEDIMIENTOS DEL ASSEMBLER

Anteriormente en la sección 2.10.1 se escribió un programa utilizando la instrucción CALL para imprimir las letras de la A a la J. Se tuvo que dejar espacio entre el procedimiento y el programa principal para evitar que se anularan entre si las instrucciones. Cuando se utiliza el assembler estos problemas no existen ya que el assembler se encargará de estos detalles.

El programa de la sección 2.10.1 se muestra a continuacion:

```
3985:0100 B241      MOV   DL,41
3985:0102 B90A00    MOV   CX,000A
3985:0105 E8F800    CALL  0200
3985:0108 E2F8      LOOP  0105
3985:010A CD20      INT   20
```

Continuación.

```
3985:0200 B402      MOV   AH,02
3985:0202 CD21      INT   21
3985:0204 FEC2      INC   DL
3985:0206 C3        RET
```

El siguiente programa es la version assembler de este programa

```
CODE_SEG        SEGMENT
    ASSUME      CS:CODE_SEG
    ORG         100h            ; Hacer un archivo .COM

PRINT_A_J       PROC    NEAR
        MOV     DL,'A'          ;Comenzar con letra A
        MOV     CX,10           ;Establece contador a 10
PRINT_LAZO:
        CALL    ESCRIBE         ;Imprime caracter, prepara siguiente
        LOOP    PRINT_LAZO      ;Continua para 10 caracteres

PRINT_A_J       ENDP

ESCRIBE         PROC    NEAR
        MOV     AH,2            ;Función 2 caracter de salida
        INT     21h             ;Imprime caracter listo en DL
        INC     DL              ;Prepara el siguiente caracter
        RET                     ;Retorno al programa principal
ESCRIBE         ENDP

CODE_SEG        ENDS
        END     PRINT_A_J
```

Hay cuatro oreradores en este programa:ASSUME esta relacionado con la definición de segmentos. ORG esta relacionado a la forma en que el DOS carga los programas. Es necesario colocar ORG cuando se creará un archivo .COM. PROC y ENDP son directivas que definen el inicio y el final de un procedimiento respectivamente. La etiqueta que se escribe antes de estas directivas es el nombre que se le da a cada procedimiento que ellas definen. El programa principal definido en el procedimiento PRINT_A_J reemplaza la instrucción CALL 200 por CALL escriba que es mas descriptiva. El assembler se encargará de colocar la dirección de este procedimiento.
NEAR y FAR (aqui no utilizada) son directivas que suministran información al assembler sobre el uso de los segmentos. Hay dos tipos de llamadas (CALL) y retornos (RET): Cercanas y lejanas (near y far). Una llamanda lejana la cual no se utiliza en este ejemplo llama un procedimiento que esta contenido en otro segmento. Una llamada cercana,por otra parte, llama procedimientos que estan contenidos en el mismo segmento. Recordar que los segmentos se definen con las directivas SEGMENT y ENDS.

Finalmente,puesto que el programa tiene dos procedimientos, se necesita decirle al assembler cual se utilizará como programa principal, en donde el 8088 comenzará a ejecutar el programa. La directiva END se encarga de este detalle. Al escribir END PRINT_A_J se le esta diciendo al assembler que el programa principal es PRINT_A_J. y que a partir de ahi se comenzará a ejecutar el programa.

Para generar el archivo version .COM se debera de seguir los mismos pasos del ejemplo anterior:

```
MASM PRINTAJ;
LINK PRINTAJ;
EXE2BIN PRINTAJ PRINTAJ.COM
```

Si se corre la version .EXE no se observan los resultados deseados ya que la directiva ORG 100h se coloca cuando se creara un archivo .COM

Se puede cargar la version .COM con el Debug y observar el trabajo que ha realizado el assembler el cual elimina los espacios entre el programa principal y los procedimientos.

2.12.2 PROCEDIMIENTOS PARA SALIDAS EN HEXADECIMAL

Anteriormente ya se han visto procedimientos para salida en hexadecimal. En la sección 2.8.2 se mostró un programa que imprimia un digito hexadecimal. En esta sección se escribirá un programa que imprimirá un caracter.
Se utilizara un procedimiento central para un caracter en la pantalla, se puede cambiar la forma en que este procedimiento escribe caracteres sin afectar el resto del programa. Este programa se modificará posteriormente varias veces.
El siguiente programa se guardara en el archivo VIDEO_IO.ASM:

Listado 2.2 Programa VIDEO_IO.ASM

```
CODE_SEG      SEGMENT

     ASSUME   CS:CODE_SEG
     ORG      100h


PRUEBA_HEX:   PROC   NEAR
     MOV      DL,3Fh        ;Prueba con 3Fh
     CALL     ESCRIBE_HEX
     INT      20h           ;Retornar al DOS
PRUEBA_HEX:   ENDP
```

Continuacion listado 2.2

```
    PUBLIC   ESCRIBE_HEX
;-- -- -------------------------------------------------------------;
;Este procedimiento convierte un byte en DL a he a y escribe       ;
;los dos digitos he a en la posición del cursor                    ;
;                                                                  ;
;   DL   Byte a convertir en he adecimal                           ;
;                                                                  ;
;Utiliza:    ESCRIBE_DIGITO_HEX                                    ;
;------------------------------------------------------------------;

ESCRIBE_HEX  PROC   NEAR     ;Punto de entrada
   PUSH      CX              ;Salva registro utilizados
   PUSH      DX
   MOV       DH,DL           ;Hace copia del byte
   MOV       CX,4            ;Obtener nibble superior en DL
   SHR       DL,CL           ;
   CALL      ESCRIBE_DIGITO_HEX ;Despliega primer digito he/a
   MOV       DL,DH           ;Obtiene nibble inferior en DL
   AND       DL,0Fh          ;Remover en nibble superior
   CALL      ESCRIBE_DIGITO_HEX ;Muestra segundo digito he a
   POP       DX
   POP       CX
   RET
ESCRIBE_HEX  ENDP

   PUBLIC  ESCRIBE_DIGITO_HEX
;------------------------------------------------------------------;
; Este procedimiento convierte los cuatro bits inferiores de DL    ;
; a digito he a y los escribe en la pantalla.                      ;
;                                                                  ;
;   DL   4 bits inferiores contiene número a imprimirse en he a    ;
;                                                                  ;
; Utiliza: ESCRIBE_CAR                                             ;
;------------------------------------------------------------------;

ESCRIBE_DIGITO_HEX   PROC   NEAR
      PUSH    DX                 ;Salvar registros utilizados
      CMP     DL,10              ;Es el nibble   10
      JAE     HEX_LETRA          ;No, convierta a letra
      ADD     DL,"0"             ;Si, convierta a digito
      JMP     Short SALIDA_DIG

HEX_LETRA:
      ADD     DL,"A"-10        ;Convierte a letra he a
SALIDA_DIG:
      CALL    ESCRIBE_CAR      ;Despliega letra en la pantalla
      POP     DX               ;Recupera el valor de DX
      RET
ESCRIBE_DIGITO_HEX  ENDP
```

Continuacion listado 2.2

```
        PUBLIC  ESCRIBE_CAR
;-------------------------------------------------------------;
;Este procedimiento imprime un caracter en la pantalla        ;
;utilizando una llamado a la función 2 del DOS                ;
;                                                             ;
;       DL      Byte a imprimir en la pantalla                ;
;-------------------------------------------------------------;

ESCRIBE_CAR     PROC    NEAR
        PUSH    AX
        MOV     AH,2    ;Llamada para salida de caracter
        INT     21h     ;Salida de caracter en el registro DL
        POP     AX      ;Recupera antiguo valor de AX
        RET
ESCRIBE_CAR     ENP

CODE_SEG        ENDS

      END       PRUEBA
```

La directiva PUBLIC le dice al assembler que genere información adicional para el encadenador. El encadenador permite traer modulos de programas compilados en diferentes archivos fuente y colocarlos en uno solo. PUBLIC informa al assembler que el procedimiento definido como publico deberá de estar disponible para procedimientos en otros archivos.

El procedimiento PRUEBA_HEX que se incluye en como procedimiento principal en el programa anterior, se utiliza unicamente para probar los otro tres procedimientos. Una vez verificado el funcionamiento de estos procedimientos PRUEBA será removido del programa VIDEO_IO.ASM.

En este punto se deberá de crear la versión .COM del programa Video_io, y utilizar el debug para probar completamente este programa. Cambiando 3Fh en la localización de memoria 101h Conviene probar valores limite como: 0, 9, 0A, 0F, etc. Luego se utilizará el comando G para correr el programa de nuevo.
En lo posterior se utilizaran muchos programas simples de prueba para verificar nuevos procedimientos. De esta forma se puede construir un programa parte por parte, en lugar de intentar construirlo y depurarlo todo de una solo vez. Este metodo incremental es mucho mas eficiente y rapido, puesto que confina a los errores solamente al codigo nuevo.

2.12.3 COMIENZO DE DISEÑOS MODULARES

Deberá notarse que en cada procedimiento del programa Video_io, se incluyo un bloque de comentarios que describen brevemente la

función de cada procedimiento. Lo que es mas importante, estos comentarios dicen cuales registros se utilizan para pasar información hacia y desde el procedimiento. También se indica cuales son los otros procedimientos que se utilizan. El bloque de comentarios permite utilizar un procedimiento viendo su descripción. No se necesita reaprender como trabaja el procedimiento. Esto hace claramente más fácil reescribir un procedimiento sin tener que reescribir algún procedimiento que se llame.

Hay que recordar que se deben de salvar y recuperar los registros utilizados de manera que no haya que preocuparse de las interacciones complejas entre procedimientos. Cada procedimiento es libre de utilizar tantos registros como se desee, el salvar y restaurar el contenido de estos registros es el precio que se paga para añadir simplicidad. Sin salvar y restaurar registros la tarea de reescribir un procedimiento generaría un verdadero dolor de cabeza.

2.12.4 EL ESQUELETO DE UN PROGRAMA

Como se ha visto en esta y en las secciones precedentes, el assembler impone un cierto número de información adicional. En otras palabras, se necesita escribir una cuantas directivas que le digan al assembler lo basico. Para futuras referencias a continuación se muestra lo mínimo que se necesita para escribir programas:

```
CODE_SEG        SEGMENT
        ASSUME  CS:CODE_SEG
        ORG     1000H

PROCE1          PROC    NEAR
                .
                .
                .
                INT     20H
PROCE1          ENDP

PROCE2          PROC    NEAR
                .
                .
                .
PROCE2          ENP

PROCEn          PROC    NEAR
                .
                .
PROCEn          ENDP
    CODE_SEG    ENDS
        END     PROCE1
```

PROC1, PROC2..... PROCn Representan los nombres que se le daran a los procedimientos.

Se agregaran algunas directivas mas a medida que se vayan necesitando.

2.13 IMPRESION EN DECIMAL

En esta sección se escribirá un procedimiento que tomara una palabra y la imprimirá en decimal. ESCRIBE_DECIMAL utiliza nuevos trucos que si se memorizan se pueden utilizar para hacer programas mas cortos y rapidos. En este prodecimiento se agregaran mas operaciones logicas.

2.13.1 RECORDANDO LA CONVERSION

Dividir es la clave para convertir un palabra a digitos decimales. Hay que recordar que la instrucción DIV calcula el cociente entero y el residuo. Asi, al calcular 12345/10 el cociente es 1234 y el residuo es 5. En este ejemplo, el 5 es el digito menos significante del dividendo. Si se vuelve a dividir de nuevo por 10 se obtiene el siguiente digito menos significante. Al repetir la division por 10 a traves de los digitos de izquierda a derecha el residuo almacena cada uno de los digitos.
Por supuesto que los digitos vienen en orden inverso, pero en lenguaje de ensamble se tiene una solución para eso. Recordar que el stack es como una pila de trastes: el primero en salir es el ultimo en haber entrado a la pila. Si se sustituyen los digitos por los trastes y se colocan los digitos uno a continuación del otro en el stack en el orden en que aparecen los residuos, se tiene resuelto el problema, al recuperarlos del stack los digitos vendran en el orden correcto.

ESCRIBE_DECIMAL se debera de escribir junto con el procedimiento para escribir un byte en hexadecimal. Asegurarse de colocar ESCRIBE_DECIMAL despues de PRUEBA_HEX, el cual será reemplazado por PRUEBA_DEC. Para ahorrar trabajo, ESCRIBE_DECIMAL utiliza ESCRIBE_DIGITO_HEX para convertir un nibble (cuatro bits) a un digito:

Listado 2.3 agregar a VIDEO_IO.ASM

```
        PUBLIC ESCRIBE_DECIMAL
;--------------------------------------------------------------;
; Este procedimiento escribe un número sin signo de 16 bits en ;
; notacion decimal                                             ;
;          IN:       N: numero sin signo de 16 bits            ;
;                                                              ;
; Utiliza: ESCRIBE_DIGITO_HEX                                  ;
;--------------------------------------------------------------;
```

Continuación listado 2.3

```
ESCRIBE_DECIMAL  PROC  NEAR
         PUSH    AX          ;Salva registros que se utilizarán
         PUSH    CX
         PUSH    DX
         PUSH    SI
         MOV     AX,DX
         MOV     SI,10       ;Dividirá por 10 utilizando SI
         XOR     CX,CX       ;Contador de digitos en stack
NO_CERO:
         XOR     DX,DX       ;Establece palabra sup. de N a 0
         DIV     SI          ;Calcula N/10 y residuo
         PUSH    DX          ;Pone el residuo en el stack
         INC     CX          ;Un digito mas añadido
         OR      AX,AX       ;Es N=0 todavía ?
         JNE     NO_CERO     ;No, continue
LAZO_ESCRIBE_DIGITOS:
         POP     DX                          ;Obtiene digitos en orden
         CALL    ESCRIBE_DIGITO_HEX          ;inverso
         LOOP    LAZO_ESCRIBE_DIGITOS
FIN_DECIMAL:
         POP     SI
         POP     DX
         POP     CX
         POP     AX
         RET
ESCRIBE_DECIMAL  ENDP
```

Notar que se incluyo el registro SI (Source Inde). Mas tarde se mostrará porque se tiene ese nombre, y mencionará su registro hermano, el DI, o Destination Inde . Ambos registros tiene usos especiales, pero se pueden utilizar como registros de proposito general, como se hizo en este ejemplo, y se explico en el cap. 1.
Antes de probar el nuevo procedimiento, se necesita hacer otros cambios a VIDEO_IO.ASM. Primero, se debe de eliminar el procedimiento PRUEBA_HEX y colocar el siguiente procedimiento en su lugar:

Listado 2.4 Reemplazar PRUEBA_HEX en VIDEO_IO con este procedimiento

```
PRUEBA_DECIMAL   PROC NEAR
       MOV     DX,12345
       CALL    ESCRIBE_DECIMAL
       INT     20
PRUEBA_DECIMAL   ENDP
```

Este procedimiento prueba ESCRIBE_DECIMAL con el número 12345 (El cual el assembler convierte a la palabra 3039h).
Segundo, se necesita cambiar la instrucción END en el final del

programa Video_10 de manera que diga: END PRUEBA_DECIMAL, esto es porque PRUEBA_DECIMAL es ahora el procedimiento principal.

Ahora se deberá de compilar este programa y producir la version .COM del mismo. Si ocurren errores, revisar el archivo fuente, corregirlos y volver a compilar. Si se desea las correcciones se pueden hacer directamente al archivo .COM utilizando el Debug.

2.13.2 ARTIFICIOS UTILIZADOS EN EL PROGRAMA

En el procedimiento ESCRIBE_DECIMAL se encuentran dos nuevos artificios utilizados comunmente en programación. El primero es el que pone a cero AX con la instrucción:

```
XOR     AX,AX
```

la cual se pudo escribir como MOV AX,0. Pero es común encontrar la instrucción anterior. Para comprender ésta instrucción, a continuación se muestra su tabla de verdad:

A	B	C= A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Es cierto solo si A y B son distintos, de lo contrario el resultado es falso. Lo que quiere decir que si se aplica la operación Or exclusiva de un número a el mismo, el resultado será cero. Por otra parte, a menudo se encuentran instrucciones como SUB AX,AX que también realizan la tarea de poner a cero un registro.

El otro truco es como se utiliza la instrucción OR para ver si un registro es cero. Para hacer ésto, se podría utilizar la instrucción CMP AX,0. Pero no se hizo, para mostrar una de las instrucciones que aparecen muy seguido , la instrucción OR AX,AX y continuación se colocó la instrucción JNE (Salte si no es igual).

La instrucción OR, como las intrucciones matemáticas, afecta las banderas, incluyendo la bandera de cero. La tabla de verdad de la instrucción OR se muestra a continuación:

A	B	C = A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Esto quiere decir que si se opera con la instrucción OR a un

número con el mismo, el resultado será el mismo número (A or A A). Esta instrucción también se utiliza para colocar unos e determinados bits.

2.13.3 COMO TRABAJA EL PROGRAMA

Para ver como ESCRIBE_DECIMAL realiza su tarea, es necesario estudiar detenidamente el listado. En ésta sección solo se cubrirá lo más relevante.

Primero el registro CX se utiliza para contar cuantos dígitos se han colocado en el stack, así se sabe cuantos se tendrán que remover. El registro CX es particularmente conveniente para ésta tarea, porque se puede construir un lazo con la instrucción LOOP y utilizar el registro CX como la variable de control de lazo.

Lo siguiente a considerar son las condiciones límites. La condición límite en 0 no es un problema. La otra condición límite es 65535, o FFFFh, la cual se puede probar fácilmente con el Debug. Simplemente se escribe DEBUG VIDEO_IO.COM y se cambia el 12345 (3039h) en 101h por 65535 (FFFFh), esto es porque en 101 esta colocado el valor de prueba.

Llegado a éste punto seguramente se habrá notado un pequeño problema. El debug trabaja la mayoría de veces con byte (el comando E lo hace) pero lo que se desea cambiar es una palabra. Se debe de tener cuidado, puesto que el 8088 almacena los bytes en un orden diferente. A continuación se muestra un listado de la instrucción MOV:

```
3905:0100 BA3930    MOV   DX,3039
```

De aquí se puede ver que el byte 101 es 39h y el byte 102 es 30 (BA es la instrucción MOV). Los dos bytes son los bytes de 3039 pero aparecen en orden inverso ¿Confuso?. El orden es lógico con la siguiente explicación.

Una **palabra** consiste de dos partes, un byte inferior y un byte superior. El byte inferior es el menos significante (39h en 3039h), mientras que el byte superior es la otra parte (30h). En éste sentido, se coloca el byte inferior en la dirección de memoria inferior. (Algunas computadoras invierten el orden, y ésto puede confundir si se utilizan diversos tipos de computadoras). Para una mejor comprension del programa se sugiere correrlo con papel y lapiz para un par de valores

2.14 SEGMENTOS, PROGRAMAS .COM Y PROGRAMAS .EXE

En los capítulos precedentes ya se mencionó algo acerca de los segmentos y de las directivas que se utilizan para definirlos. En ésta sección se estudiará la forma en que el 8088 maneja la memoria haciendo uso de segmentos, y se trabajará con los registros que manipulan segmentos, estudiando la forma en que estos trabajan en archivos .COM y en arhivos .EXE. Ademas se

discutirá similitudes y diferencias entre estos dos tipos de archivos.

2.14.1 SECCIONAMIENTO DE LA MEMORIA DEL 8088

Una de las funciones principales que tienen los registros de segmentos, es poder manejar más de 64k de memoria, que es el límite de una palabra, puesto que 65535 es el límite mayor que una palabra puede sostener. Los diseñadores de Intel del 8088 utilizan segmentos y registros de segmento para resolver éste problema.
Hasta éste momento, no ha habido problema, ya que se ha utilizado el registro IP para almacenar la dirección de la siguiente dirección que ha de ser ejecutada por el 8088, desde que se comenzó a utilizar el Debug. También se recordará que la dirección completa está formada por los registros CS e IP. A continuación se dicutirá como estos y otros registros se encargan de manejar la estructura de memoria segmentada del 8088

Aunque la dirección completa está formada por los dos registros, el 8088 no forma un número de dos palabras para las direcciones. Si se tomara CS:IP como un número de 32 bits, el 8088 sería capaz de manejar alrededor de 4 billones de bytes. El método que utiliza el 8088 es distinto: el registro CS suministra la dirección de partida para localizar determinada dirección absoluta en determinado segmento, donde un segmento está formado por 64k. A continuación se discute como trabajan estos registros.

Como se puede ver en la figura 2.7, el 8088 divide la memoria en muchos segmentos traslapados, con un nuevo segmento comenzando cada 16 bytes. El primer segmento (segmento 0) comienza en la localización 0; el segundo (segmento 1) comienza en 10h (16); el tercero comienza en 20h (32), y así sucesivamente.
La dirección a la que apuntan el par CS:IP está detemrinado por la relación CS*16 + IP. Por ejemplo si el registro CS contiene 3FA0 e IP contiene D017, la dirección absoluta es:

```
CS*16:    0 0 1 1  1 1 1 1  1 0 1 0  0 0 0 0  0 0 0 0
   IP:             1 1 0 1  0 0 0 0  0 0 0 1  0 1 1 1
-------------------------------------------------------
          0 1 0 0  1 1 0 0  1 0 1 0  1 0 0 1  0 1 1 1
```

Se multiplica por 16 solo para rotar CS cuatro bits hacia la izquierda, y colocar cero a la derecha.
El 8088 tiene cuatro registros para segmetos: CS (Code Segment), DS (Data Segment), SS (Stack Segment), y ES (Extra Segment). El registro CS como se vió anteriormente se utiliza para el segmento en donde la siguiente instrucción está almacenada. De manera similar, El registro DS contiene el segmento para los datos, y SS es donde el 8088 coloca el segmento de stack.

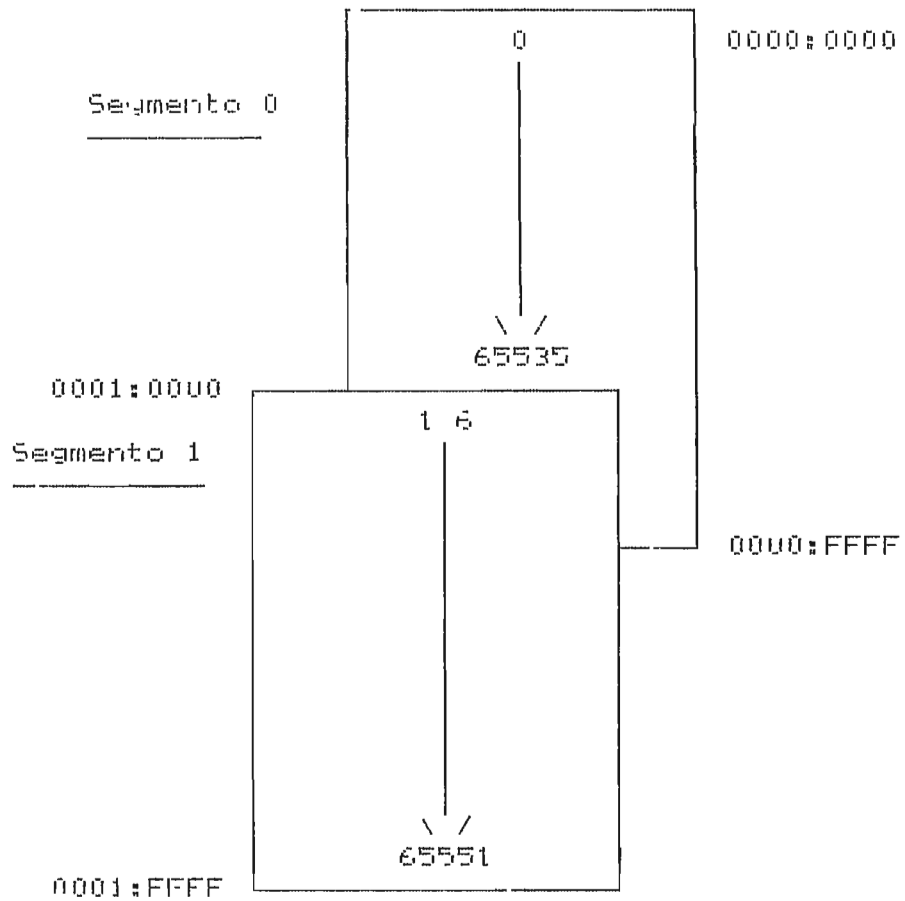


Figura 2.7 Segmentos traslapados comenzando cada 16 bytes

Para lograr una mejor comprensión de la función de los segmentos, antes de continuar, se estudiará un programa, bastante diferente a los que se han estudiado anteriormente, éste programa utiliza dos diferentes segmentos. Escribir el siguiente programa en el archivo llamado PRUEBA_SEG.ASM:

Listado 2.5 Programa PRUEBA_SEG.ASM

```
CODE_SEG      SEGMENT
   ASSUME     CS:CODE_SEG

PRUEBA_SEG    PROC  NEAR

      MOV     AH,4Ch        ;Pregunta por la función de salida al DOS
      INT     21h           ;Retorna al DOS

PRUEBA_SEG    ENDP

CODE_SEG      ENDS
```

Continuación Traslado 2.5

```
STACK_SEGMENT SEGMENT STACK
      DB      10 DUP ("Stack    ") ;Hay 3 espacios después de Stack

STACK_SEGMENT ENDS

      END     PRUEBA_SEG
```

Luego se deberá de compilar y encadenar el archivo Prueba_Seg, pero no se deberá de generar un archivo .COM. El resultado será PRUEBA_SEG.EXE, el cual es algo diferente a un archivo .COM

Nota: Se ha utilizado un forma diferente para salir de un archivo EXE. Para los archivo .COM INT 20h trabaja perfectamente bien, pero no para los archivos EXE ya que su organización es bastante diferente, más adelante se tratarán algunas de estas diferencias.

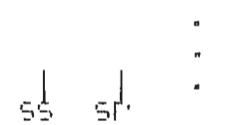
Cuando se utiliza el Debug en un archivo .COM, El Debug establece todos los registros de segmentos al mismo número, con el programa comenzando en un offset (complemento) de 100h a partir del inicio del segmento. Los primeros 256 bytes (100h) son utilizados como área de datos.

Ahora al cargar PRUEBA_SEG.EXE con el Debug, para ver lo que sucede con los segementos en un archivo .EXE se tendrá:

```
A:\ DEBUG PRUEBA_SEG.EXE
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=0050 BP=0000 SI=0000 DI=0000
DS=1985 ES=1975 SS=1996 CS=1995 IP=0000 NV UP DI PL NZ NA PO NC
1995:0000 CD 20       INT 20
```

El valor de los registros SS y CS son diferentes de aquellos en DS y ES. En el programa, se han definido dos segmentos. El STACK_SEGMENT es donde se coloca el stack (de aquí la palabra STACK después de la palabra SEGMENT). Se ha definido el stack de 80 bytes de longitud: La instrucción DB 10 DUP ("Stack ") le dice al assembler que convierta la cadena encerrada entre comillas a bytes, y repetir la cadena 10 veces en la memoria. DB (Define byte) le dice al assembler que se estan definiendo byte de memoria. En éste caso se ha inicializado el stack con repeticiones de los códigos ASCII de la palabra Stack y tres espacios. Los códigos para ésto son 53 74 61 63 6B 20 20 20, así si se observa el segmento del stack se verá estos números repetidos 10 veces. Si se le ordena al Debug, con el comando D, que muestre ésta área de memoria,partir de la localización definida por SS:0, Se observará lo siguiente

```
-D SS:0
3996:0000  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20 Stack   Stack
3996:0010  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20 Stack   Stack
3996:0020  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20 Stack   Stack
3996:0030  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20 Stack   Stack
3996:0040  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20 Stack   Stack ..
3996:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```



La dirección para el tope del stack, está dada por SS:SP. SP es el stack pointer, y es el offset (complemento) dentro del segmento del Stack. Realmente, "tope del stack" es un nombre equivocado, porque el stack crece a partir de la las localizaciones de memoria superiores hacia las inferiores. Así, el tope del stack es realmente el fondo del stack en la memoria, y las nuevas entradas al stack, son colocadas progresivamente en las localizaciones inferiores. En éste ejemplo, SP almacena 50h que es 80 en decimal, ésto es porque se definió un área de stack de 80 byte de longitud.

Al observar el despliegue de registros, se habrá notado que los registros ES y DS contienen 3985, 10h menos que el segmento en donde el programa comienza (3995). Multiplicando 10h por 16 (10h) para calcular el número de bytes, se puede ver que hay en un área de 100h (256) bytes antes del inicio del programa. Esta es la misma área colocada al inicio de un archivo .COM. Esta área se conoce como un PSP (Program Segment Prefix) y contiene información utilizada por el DOS. En otras palabras, el programador no debe de asumir que ésta área, que es un área de datos, se encuentra a su disposición.

Esta área contiene información que el DOS utiliza cuando se sale de un programa, ya sea con la instrucción INT 20h o con la instrucción INT 21h función 4Ch. Por razones que no están claras, la instrucción 20h espera que registro CS apunte al inicio del área de datos, lo cual es cierto para un programa .COM pero no para un programa .EXE.

El programa en un archivo .COM debe de comenzar siempre con un offset (complemento) de 100h (IP=0100h) a partir del Segmento de código (segmento en donde comienza el programa, definido por CS) para dejar espacio a los 256 bytes del área de datos. Esto diferente en un archivo .EXE el cual tiene su programa con un offset (complemento) de 0 (IP=0000), ya que el segmento de código (segemento en donde comienza el programa, definido por CS) comienza 100h bytes después del inicio del área de datos. Ver figura 2.0.

Hay que recordar que en los archivos .COM creados anteriormente, se ha tenido que colocar explícitamente la directiva ORG 100h al

inicio del programa para apartar 100h bytes para el área de datos. La directiva ORG 100h establece el origen del programa a 100h. Ésto es todo lo que hace, pero se continuará colocándola, ya que los archivos que se utilizarán en éste trabajo son .COM. S ha presentado la estructura de un archivo .EXE solamente par aprender algo acerca de los segmentos.

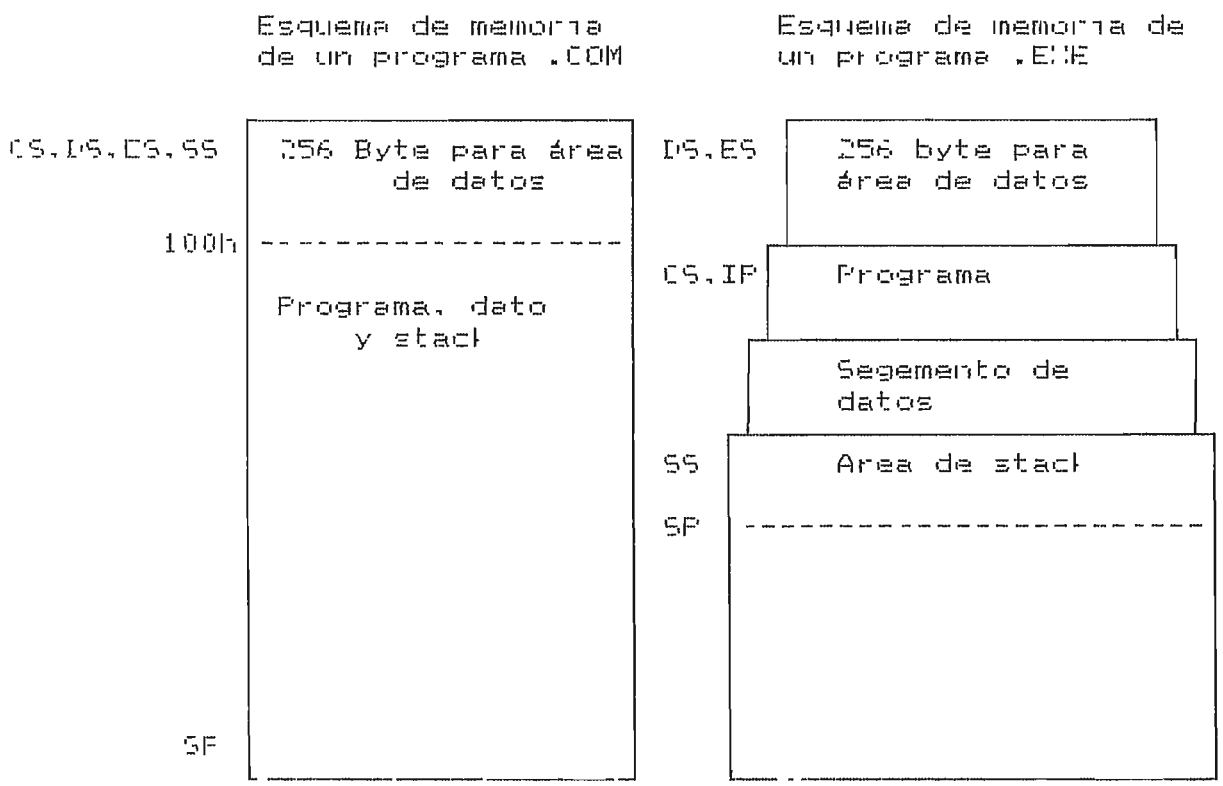


Figura 2.8 Archivo .COM vs archivo . EXE

2.15 DISEÑO MODULAR DE PROGRAMAS

Uno de los propositos de este capitulo es el de crear un programa que de aqui en adelante se le llamará DSK, y que será capaz de e plorar y modificar el contenido de un disLette, este programa no se presentará de una sola vez, si no que se presentaran programas de prueba mas cortos, que poco a poco le iran dando forma al programa final.

Puesto que este programa tratará con inforamación en disco, será por ahi donde se comenzará.

2.15.1 DISKETTES, SECTORES Y EL PROGRAMA DSK

La información en los discos magnéticos está dividida en sectores, en cada sector se pueden alamcenar hasta 512 bytes de información. En un disco de doble lado formateado con el DOS 2.1 o una versión posterior, se pueden almacenar un total de 720 sectores, o 368,640 bytes. Si se pudiera ver directamente estos sectores, se podría examinar el directorio, o se podrían ver los archivos en el disco. El usuario no puede hacer esto por el mismo, pero utilizando el programa DSK si se podrá. En esta sección se utilizará el debug para aprender algo mas acerca de los sectores y tener una mejor idea de lo que hará el programa DSK al desplegar los sectores en la pantalla. Ver ane o para mas detalles acerca de los discos. El debug tiene un comando, L (load), para leer sectores del disco a la memoria, en donde se puede observar su contenido. Como ejemplo, se verá el directorio que comienza en el sector 5 en un disco de doble lado. Para cargar el sector 5 del disco que esta en el driver A (que para el DeBug es el drive 0) se utiliza el siguiente comando

```
L 100 0 5 1
  |   | | |
  |   | | |_ -- Numero de sectores a leer
  |   | |___ -- Numero del sector a leer
  |   |_____ -- Driver que contiene el disco (0=A, 1=B, etc )
  |_________ -- Dirección en memoria a donde se trasladará
              el(los) sector(es) leido(s)
```

Como se puede ver, este comando carga sectores en la memoria, comenzando con el sector 5 y continuando a través de un sector. Para desplegar este sector se puede utilizar el siguiente comando:

```
D 100
```

Luego se verá parte del contenido de este sector. Si se utiliza el comando D, sin argumentos nuevamente, continuará el despliegue del sector que quedo pendiente con el comando D anterior, y así sucesivamente.

El formato que se utilizara para el programa DSK será parecido a lo que muestra el comando D, pero con muchas mejoras. Este programa será equivalente a un editor de pantalla completa para sectores de discos. Se podrá desplegar un sector a la vez en la pantalla y mover el cursor a través del sector desplegado, cambiando números o caracteres cuando se desee. También se podrá escribir el sector alterado de regreso al disco.

El programa DSK es la motivación para los procedimientos que se escribirán, pero eso no significa que ese sea el único fin. En este programa se encontraran muchos procedimientos que serán muy utiles a la hora de escribir programas. Esto significa que este programa contará con muchos procedimientos de propositos generales, tales como: despliegue de salidas, manipulación de la

pantalla, entradas desde el teclado y mas. En la siguiente subsección se aprenderá como dividir un programa en varios archivos fuentes diferentes y en la proxima sección se comenzara con el diseño serio del programa DSI

2.15.2 COMPILACION SEPARADA

En la sección 2.14 se añadieron los procedimientos ESCRIBE_DECIMAL a VIDEO_IO.ASM, y tambien se añadio un corto procedimiento de prueba llamado PRUEBA_DECIMAL. Se tomará ese procedimiento del archivo VIDEO_IO.ASM y se colocará en un archivo propio llamado PRUEBA.ASM. Luego, se compilara esos dos archivos separadamente y posteriormente se encadenaran en un solo programa. A continuación se muestra el archivo PRUEBA.ASM:

Listado 2.6 Archivo PRUEBA.ASM

```
CODE_SEG  SEGMET  PUBLIC
     ASSUME  CS:CODE_SEG
        ORG  100h

     EXTRN  ESCRIBE_DECIMAL:NEAR

PRUEBA_DECIMAL   PROC  NEAR
        MOV    DX,12345
        CALL   ESCRIBE_DECIMAL
        INT    20h
PRUEBA_DECIMAL   ENDP

CODE_SEG      ENDS

        END PRUEBA_DECIMAL
```

La mayoría de este archivo ya se ha visto, pero aparecen cosas nuevas. Primero la palabra PUBLIC aparece despues de SEGMENT. Esta palabra le dice al assembler que se desea que este segmento (CODE_SEG) se combine con otros segmentos que tengan el mismo nombre para formar uno solo. el CODE_SEG en este caso (segmento de instruuciones o codigo). El assembler pasará esta informacion al encadenador, el cual como su nombre lo indica, encadena varios archivos.

El archivo tambien contiene la directiva EXTRN. la instrucción EXTRN ESCRIBE_DECIMAL:NEAR le dice al assembler dos cosas: que ESCRIBE_DECIMAL esta en otro archivo, externo, y que esta definido como un procedimiento NEAR en ese otro archivo, que para el encadenador no importa cual sea, asi que debera estar en el mismo segmento. el assembler genera asi una llamada NEAR para este procedimiento. Esos son algunos de los cambios que se necesitan para separar achivos fuente, cuando se comience almacenar datos en la memoria se introducirán otros cambios. Lo que sigue es modificar el programa VIDEO_IO.ASM, y luego compilar y encadenar los dos archivos.

Hay que remover el procedimiento PRUEBA_DECIMAL de VIDEO_IO, su contenido se ha colocado en PRUEBA.ASM, así es que ya no se necesita en Video_io. Luego, remover la instrucción ORG 100H de Video_io. Esto se ha trasladado también al procedimiento PRUEBA.ASM, el cual tiene el primer procedimiento en el programa. Lo siguiente es poner la palabra PUBLIC despues de SEGMENT, de la manera siguiente:

```
CODE_SEG    SEGMENT PUBLIC
```

así el encadenador sabrá que debera de combinar este segmento con el mismo segmento en PRUEBA.ASM.

Finalmete, se debe de cambiar END PRUEBA_DECIMAL en el final de VIDEO_IO.ASM y dejar solamente END. Esto es porque el procedimiento principal se ha movido a PRUEBA.ASM. Los procedimientos en VIDEO_IO.ASM son ahora procedimientos e termus. Es decir, que ellos no funcionan por ellos mismos; si no que deben de ser encadenados a procedimientos que puedan llamarlos desde otros archivos. No se necesita un nombre despues de la directiva END en VIDEO_IO, ya que el programa principal está ahora en PRUEBA.ASM.

Cuando se haya terminado de realizar estos cambios, el programa VIDEO_IO.ASM se verá parecido a lo que sigue:

```
CODE_SEG    SEGMENT  PUBLIC
      ASSUME  CS:CODE_SEG

      PUBLIC ESCRIBE_DIGITO_HEX
           .
           .
ESCRIBE_DIGITO_HEX    ENDP

      PUBLIC ESCRIBE_HEX
           .
           .
ESCRIE_HEX    ENDP

      PUBLIC  ESCRIBE_CAR
           .
           .
ESCRIBE_CAR   ENDP
PUBLIC  ESCRIBE_DECIMAL
           .
           .
ESCRIBE_DECIMAL  ENDP
CODE_SEG    ENDS
      END
```

Ahora se deberá compilar estos dos archivos como se compiló Video_io anteriormete. PRUEBA.ASM sabe todo lo que necesita saber acerca de VIDEO_IO.ASM por medio de la directiva EXTRN. El resto vendrá cuando se encadenen los dos archivos. En este punto ya se

deberan de tener los archivo PRUEBA.OBJ y VIDEO.OBJ. El comando que se debe de utilizar para encadenar estos dos archivos en un programa llamado PRUEBA.EXE es el siguiente:

```
A> LINK PRUEBA VIDEO_IO;
```

El LINK combinara los procedimientos en estos dos archivos para crear un archivo que contenga el programa completo. Se utiliza el nombre del primer archivo como nombre para el archivo .EXE resultante, es decir que despues de ejecutar el comando anterior se tiene el archivo PRUEBA.EXE.
Finalmente se deberá de crear el archivo .COM como se hizo anteriormente, escribiendo:

```
A> EXE2BIN PRUEBA PRUEBA.COM
```

Con esto se completa el procedimiento para crear un archivo a partir de otros dos. Este procedimiento será utilizado frecuentemente en las programas secciones con lo que ahorrará gran cantidad de esfuerzo mental. Pero antes de continuar se discutirá algo sobre el diseño modular.

2.15.3 LAS TRES LEYES DEL DISEÑO MODULAR

Estas leyes, que en realidad son sugerencias, haran el trabajo mucho mas facil.

1. Salvar y restaurar todos los registros, a menos que el procedimiento retorne un valor en ese registro.

2. Ser consistente acerca de cuales registros se utilizan para pasar información. Por ejemplo:
   - DL,DX: Enviar valores en byte y en palabras
   - AL,AX: Retornar valores en byte y en palabras
   - BX:AX: Retornar valores de palabra doble
   - DS:DX: Enviar y retornar direcciones
   - CX: Contador de repeticion
   - CF: Indica cuando hay un error; el codigo de error podría ser retornado en uno de los registros, tales como AL o AX.

3. Definir **todas** las interacciones e ternas en comentarios de encabezados:
   - Información necesaria como entrada
   - Inforamción retornada (registros modificados)
   - Procedimientos llamados
   - Variables utilizadas (Leidas, escritas, etc)

2.16 MOSTRANDO LA MEMORIA

De aquí en adelante, se comenzará a construir el programa DSI. Algunas de las instrucciones pueden parecer poco familiares: se explicarán brevemente a medida que se utilicen; pero para información detallada se deberá recurrir al anexo 2, en lugar de cubrir detalladamente las instrucciones, se estudiará nuevos conceptos tales como los distintos modos de direccionar la memoria, y algunas características de las computadoras IBM PC y sus familiares cercanos.

2.16.1 MODOS DE DIRECCIONAMIENTO

Ya se han visto aplicados en ejemplos anteriores, dos modos de direccionamiento; estos modos son conocidos como direccionamientos de **registro e inmediato.** El primero que se aprendió fue el modo de registro, el cual utiliza registros como variables. Por ejemplo la instrucción:

```
MOV   AX,BX
```

Utiliza los registros AX y BX como variables.
El modo de direccionamiento inmediato mueve un número directamente a un registro, por ejemplo:

```
MOV   AX,2
```

Este modo permite leer la palabra colocada inmediatamente despues de la instruccion , pero no permite cambiar la memoria. Por esta razon, se necesita otro modo de direccionamiento. Este se explicará por medio de un ejemplo. El siguiente programa lee 16 bytes de la memoria, un byte al mismo tiempo, y despliega cada byte en notación hexadecimal, con un espacio entre cada número hexadecimal. Este programa se escribirá en el archivo DESP_SEC.ASM.

Listado 2.7 programa DESP_SEC.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG  ;Agrupar dos segmentos
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT     PUBLIC
        ORG     100h
        EXTRN   ESCRIBE_HEX:NEAR
        EXTRN   ESCRIBE_CAR:NCAR
;------------------------------------------------------------;
;Este es un programa de prueba que muestra 16 bytes de       ;
;memoria como números hexadecimales, todos en una linea      ;
;------------------------------------------------------------;
DESP_LINEA   PROC    NEAR
        XOR   BX,BX             ;Pone BX a 0
```

Listado 2.7 Continuación

```
        MOV   CX,16             ;Mostrar 16 bytes
LAZO_HEX:
        MOV   DL,SECTOR[BX]     ;Obtener un bytes
       CALL   ESCRIBE_HEX       ;Mostrar este byte en hexa
        MOV   DL,' '            ;Escribe un espacio entre números
       CALL   ESCRIBE_CAR
        INC   BX
       LOOP   LAZO_HEX
        INT   20h
DESP_LINEA  ENDP

CODE_SEG    ENDS

DATA_SEG    SEGMENT PUBLIC
        PUBLIC  SECTOR
SECTOR DB    10h, 11h, 12h, 13h, 14h, 15h, 16, 17h  ; Muestra
       DB    18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh ; de prueba
DATA_SEG    ENDS

        END   DESP_LINEA
```

Hay que notar que se ha colocado el segmento de datos (DATA_SEG) despues del segmento de codigo (CODE_SEG). Se ha colocado al final del archivo asi el encadenador cargará los datos en la memoria al final del programa.

A este programa se le han agregado unos nuevos trucos, y por esta razón se necesita hacer algunos pequeños cambios a VIDEO_IO.ASM. Primero, remover la instrucción ASSUME en Video_io y colocar las siguientes dos líneas en el inicio de VIDEO_IO.ASM:

```
CGROUP  GROUP  CODE_SEG      ;Agrupa dos segmentos juntos
        ASSUME CS:CGROUP
```

Estas dos líneas se deberan de colocar al inicio de cada archivo de ahora en adelante, con una sola variación. En la cual se deberá de escribir

```
CGROUP   GROUP  CODE_SEG, DATA_SEG   ;Agrupa dos segmentos
         ASSUME CS:CGROUP, DS:CGROUP
```

(Con segmento de datos) Siempre que se tenga ambos, segmento de codigo y segmento de datos en el archivo.

El nuevo ASSUME reemplaza al ASSUME antiguo, y mas harde se explicará como funciona, pero por ahora, se estudiará como funciona el programa, y luego compilar ambos Desp_sec y Video_io. Ahora que ya se esta listo para encadenar DESP_SEC.OBJ y VIDEO_IO.OBJ y correr el resultado atraves de E exbin. Asi primero se utiliza en LINK para crear el archivo .EXE llamado DESP_SEG.EXE. El primer nombre en el comando LINK debe de ser el

del nombre del archivo que contenga el procedimiento principa (Desp_sec en este caso),y el punto y coma debe de aparecer a final de la lista de archivos, escribir entonces:

```
A LINK DESP_SEC VIDEO_IO;
```

El encadenamiento siempre será igual, si hay mas archivo. Est se colocan antes del punto y coma, pero el procedimient principal siempre debe de colocarse al principio.
Ahora, convirtiendo el archivo .EXE en .COM se tiene:

```
A EXEBIN DESP_SEC DESP_SEC.COM
```

Una vez se tiene la version .COM, esta se puede correr, y s debera de ver en la pantalla:

```
10 11 12 13 14 15 1C 17 18 19 1A 1B 1C 1D 1E 1F
```

Ahora se vera como Desp_sec trabaja. La instrucción :

```
  MOV  DL,SECTOR[BX]      ;Obtiene 1 byte
```

utiliza un nuevo modo de direccionamiento conocido como **Modo de direccionamiento indirecto**, o simplemente **de base relativa** e este modo, la memoria se direcciona atravez de un registro Bas que almacena un complemento (offset).
La etiqueta SECTOR se encuetra en un segemto llamado DATA_SEG Este es un nuevo segment utilizado para variables en la memoria Cada vez que se desee almacenar y leer datos en la memoria, s reservara un espacio en este segmento. Antes de continuar con la variables en la memoria, se estudiará un poco mas acerca de lo segmentos.
ASSUME DS:CGROUP le dice al assembler donde hallar la variables en la memoria. Se podría preguntar para que se dese ASSUME DS:DATA_SEG. Como lo que se quiere crear es un archiv .COM, este se debe de construir en un solo segemento. Aun asi es conveniente trabajar con dos: uno para el codigo (e programa), y uno para los datos. Es aqui en donde interviene l directiva GROUP. Esta agrupa dos segmentos diferentes en l efectivamentes es un segmento, con el nombre que se le da ante de la directiva GROUP. De esta manera la instrucción :

```
CGROUP GROUP  CODE_SEG,DATA_SEG
```

Combina los dos segmentos CODE_SEG y DATA_SEG en un sol segmento de 64k con el nombre de CGROUP.
Es tiempo de regresar al modo de direccionamiento relativo. La dos lineas:

```
SECTOR  DB   10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h  ;Muestra de prueba
        DB   18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
```

Reservan 16 byte de memoria en el segmento de datos comenzando en SECTOR, la cual el assembler convierte en una dirección. DB, como se recordara, se utiliza para definir bytes: los numeros despues de DB son los valores iniciales. De esta manera, cuando se comienza con DESP_SEC.COM, la memoria que comienza en SECTOR tendrá almacenado 10h, 11h, 12h, y así sucesivamente. Si se escribe:

```
MOV  DL,SECTOR
```

esta instrucción movería el primer byte (10h) en el registro DL. Esto se conoce como modo de direccionamiento directo. Pero eso no fue lo que se hizo, si no que se colocó [BX] despues de SECTOR. Esto puede verse como un indice en un arreglo, como la instrucción en BASIC:

```
I = L(10)
```

la cual mueve el decimo elemento de L a I.

En realidad, la instrucción MOV es muy parecida, el registro BX contiene un complemento de memoria para SECTOR. Así si BX es cero, la instrucción MOV DL,SECTOR[BX] mueve el primer byte (en este caso 10h) a DL, si BX es 0Ah, la instrucción MOV traslada el onceavo byte (1Ah recordar que se comenzo con 0) a DL.

Hay otros modos de direccionamieto, los cuales se resumen en la siguiente tabla.

Tabla 3.1 Modos de direccionamiento

Modo	Formato de dirección	Registros utilizados
Registro	Registros (tales como AX)	ninguno
Inmediato	Datos (tales como 12345)	ninguno
Registro Indirecto	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
De base relativa¹	etiqueta[BX]	DS
	etiqueta[BP]	SS
Indexado indirecto	etiqueta[DI]	DS
	etiqueta[SI]	DS
Indexado base	etiqueta[BX+SI]	DS
	etiqueta[BX+DI]	DS
	etiqueta[BP+SI]	SS
	etiqueta[BP+DI]	SS

Comandos de cadena   lee desde DS:DI
(MOVSW,LODSB, etc)   escriben en ES:DI

* mi_metal...] puede ser reemplazado por [desp+...], donde desp es un desplazamiento. Así, se podría escribir [10+BX] y la dirección sería 10+BX.

3.16.2 AÑADIENDO CARACTERES A LOS DIGITOS HEXA

Con el programa de la sección anterior se tiene un procedimiento para mostrar el contenido de la memoria similar a lo que hace el comando 'd' del Debug' en el siguiente paso se le agregará los caracteres que representan los números he adecimales que se muestran en la pantalla. No son muchos los cambios, así que sin más preámbulo aquí está la nueva versión de DESP_LINEA (en DESP_SEL.ASM) con un segundo lazo agregado para desplegar los caracteres:

```
; Cambio de nombre a DESP_LINEA en DESP_SEL.ASM


DESP_LINEA  PROC  NEAR
        MOV     BX,0              ;Pone BX a 0
        MOV     CX,16             ;Contador de caracteres
LAZO_HEX:
        MOV     DL,SECTOR[BX]     ;Obtiene un byte
        CALL    ESCRIBE_HEX       ;Muestra el byte en he o
        MOV     DL,' '            ;Escribe espacio entre números
        CALL    ESCRIBE_CAR
        INC     BX
        LOOP    LAZO_HEX

        MOV     DL,' '
        CALL    ESCRIBE_CAR
        MOV     CX,16
        XOR     BX,BX
LAZO_ASCII:
        MOV     DL,SECTOR[BX]
        CALL    ESCRIBE_CAR
        INC     BX
        LOOP    LAZO_ASCII

        RET
DESP_LINEA  ENDP
```

El programa anterior se deberá de compilar, y encadenar con Vinculo.to, luego con e2bin se obtiene la versión .COM. Cuando se corra este programa, además de los caracteres he adecimales se deberán ver los respectivos codigos ASCII.
Si se cambian los datos y se incluye un 00h o un 0Ah se verá un desplegue extraño de caracteres, esto es porque 0Ah y 00h son

los caracteres para el avance de línea y para el retorno del carro. El DOS interpreta estos caracteres como comandos para mover el cursor. Pero lo que se desea es que se vean como caracteres ordinarios para esta parte del despliegue. Para hacer esto, se tendrá que cambiar ESCRIBE_CAR para que imprima todos los caracteres sin aplicar ningún significado especial. Esto se hará posteriormente, pero por ahora, se reescribirá ESCRIBE_CAR de manera que imprima un punto en lugar de los caracteres comprendidos entre 0 y 1Fh.

Reemplazar ESCRIBE_CAR en VIDEO_IO.ASM con este nuevo procedimiento:

Listado 2.7 Un nuevo ESCRIBE_CAR en VIDEO_IO.ASM

```
   PUBLIC  ESCRIBE_CAR
; - --- -- ------------------------------------------------------;
;Este es un procedimiento que imprime un caracter en la pantalla;
;utilizando una función del DOS. ESCRIBE_CAR reemplaza los       ;
;caracteres de 0 a 1F por un punto                               ;
;                                                                ;
;   AL byte a imprimir en la pantalla                            ;
; - -  - -- ------------------- ------------------------------;

ESCRIBE_CAR   PROC  NEAR
        PUSH    AX
        PUSH    DX
        CMP     DL,32           ;Este caracter antes que espacio?
        JAE     SE_IMPRIME      ; No, imprimalo como esta
        MOV     DL,'.'          ; Si, reemplacelo con un punto
SE_IMPRIME:
        MOV     AH,2            ;Llamar salida de caracter
        INT     21h             ;Salida de caracter en DL
        POP     DX              ;Recuperar valores anteriores
        POP     AX
        RET
ESCRIBE_CAR   ENDP
```

2.16.3 MOSTRANDO 256 BYTES DE MEMORIA

Una vez que se han mostrado 16 bytes, el siguiente paso es mostrar 256 bytes de memoria. Esta cantidad representa la mitad de un sector, lo que significa que se esta trabajando para construir un despliegue de medio sector.

Se necesitan dos nuevos procedimientos y modificar la versión de DESP_LINEA. Los nuevos procedimientos serán DESP_MEDIO_SECTOR, el cual prontamente evolucionara a un procedimiento finalizado para desplegar medio sector, y ENVIA_CRLF, el cual solamamente envia el curso a la siguiente línea (CRLF significa Carriage Return_Line Feed, el par de caracteres que mueven el cursor a la siguiente linea).

ENVIA_CRLF es muy simple. Se comenzara colocando el siguiente procedimiento en un archivo llamado CURSOR.ASM:

Listado 2.10 Archivo CURSOR.ASM

```
CR          EQU   13       ;retorno del carro
LF          EQU   10       ;avance de linea

CGROUP  GROUP  CODE_SEG
        ASSUME      CS:CGROUP

CODE_SEG    SEGMENT PUBLIC
```

Continuación listado 2.10

```
        PUBLIC  ENVIA_CRLF
;-- - -- ----------------------------------------------------------;
; Esta rutina solo envia un retorno de carro y un avance de      ;
; de linea utilizando rutinas del DOS                            ;
;-- - ----------------------------------------------------------;
ENVIA_CRLF  PROC NEAR
        PUSH    AX
        PUSH    DX
        MOV     AH,2
        MOV     DL,CR
        INT     21h
        MOV     DL,LF
        INT     21h
        POP     DX
        POP     AX
        RET
ENVIA_CRLF  ENDP

CODE_SEG    ENDS

        END
```

Este procedimiento utiliza la función 2 del DOS para enviar caracteres. La instruccion:

```
CR    EQU   13
```

Utiliza la directiva EQU para definir el nombre CR igual a 13. Asi la instruccion MOV DL,CR es equivalente a MOV DL,13. Es decir que el assembler siempre sustituye 13 cuando ve CR, de manera similar sustituye 10 siempre que ve LF.

El archivo Desp_sec necesita mucho trabajo. A continuación se muestra la nueva version de DESP_SEC.ASM. **De aqui en adelante, las adiciones a los programas se mostraran en negritas; el texto que se deberá de eliminar se mostrará PRECEDIDO DEL SIGNO **

Listado 2.11 Nueva versión de DESP_SEC.ASM

```
CGROUP GROUP  CODE_SEG, DATA_SEG
       ASSUME CS:CGROUP, DS:CGROUP

CODE_SEG       SEGMENT PUBLIC
     ORG       100H

   PUBLIC      DESP_MEDIO_SECTOR
   EXTRN       ENVIA_CRLF:NEAR
;-------------------------------------------------------------;
;Este procedimiento despliega medio sector (256 bytes)        ;
;                                                             ;
;Utiliza:     DESP_LINEA, ENVIA_CRLF                         ;
;-------------------------------------------------------------;

DESP_MEDIO_SECTOR   PROC  NEAR
      XOR    DX,DX              ;Comienza en el inicio del sector
      MOV    CX,16              ;Despliega 16 lineas
MEDIO_SECTOR:
      CALL   DESP_LINEA
      CALL   EVIA_CRLF
      ADD    DX,16
      LOOP   MEDIO_SECTOR
      INT    20H
DESP_MEDIO_SECTOR  ENDP

      PUBLIC DESP_LINEA
      EXTRN  ESCRIBE_HEX:NEAR
      EXTRN  ESCRIBE_CAR:NEAR
;-------------------------------------------------------------;
; Este procedimiento despliega una linea de datos, o 16 byte  ;
; primero en hexa, luego en ASCII.                            ;
;                                                             ;
;      DS:DX  Complemento a partir de sector en bytes         ;
;                                                             ;
;  Utiliza: ESCRIBE_CAR, ESCRIBE_HEX                          ;
;  Lee:     SECTOR                                            ;
;-------------------------------------------------------------;

DESP_LINEA   PROC  NEAR
    XOR   BX,BX
    PUSH  BX
    PUSH  CX
    PUSH  DX
    MOV   BX,DX          ;Complemento utilizado se pone en BX
    MOV   CX,16          ;Muestra 16 bytes
    PUSH  BX             ;Salva el complemento para LAZO_ASCII

LAZO_HEX:
    MOV   DL,SECTOR[BX]  ;Obtiene un byte
    CALL  ESCRIBE_HEX    ;Muestra este byte en hexa
```

Continuacion Listado 2.11

```
     MOV   DL,' '          ;Escribe espacio entre numero
     CALL  ESCRIBE_CAR
     INC   BX
     LOOP  LAZO_1
     MOV   DL,' '          ;Otro espacio  entre caracteres
     CALL  ESCRIBE_CAR
     MOV   CX,16
     POP   BX              ;Se recupera complemento dentro de SECTOR
     XOR   DI,DI           ;Para el lazo ASCII
LAZO_ASCII:
     MOV   DL,SECTOR[DI]
     CALL  ESCRIBE_CAR
     INC   DI
     LOOP  LAZO_ASCII
     POP   DX
     POP   CX
     POP   BX
     RET
     INT   20h
DESP_LINEA  ENDP

CODE_SEG    ENDS

DATA_SEG    SEGMENT PUBLIC
     PUBLIC SECTOR

  SECTOR    DB 10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h ;Muestra de
            DB 18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh ;  prueba

  SECTOR    DB 16 DUP(10h)
            DB 16 DUP(11h)
            DB 16 DUP(12h)
            DB 16 DUP(13h)
            DB 16 DUP(14h)
            DB 16 DUP(15h)
            DB 16 DUP(16h)
            DB 16 DUP(17h)
            DB 16 DUP(18h)
            DB 16 DUP(19h)
            DB 16 DUP(1Ah)
            DB 16 DUP(1Bh)
            DB 16 DUP(1Ch)
            DB 16 DUP(1Dh)
            DB 16 DUP(1Eh)
            DB 16 DUP(1Fh)
DATA_SEG    ENDS
        END DESP_MEDIO_SECTOR
```

los cambios son todos claros. En DESP_LINEA, se ha agregado PUSH DI y POP DI en el LAZO_HEX, ya que se desea reutilizar los complementos iniciales en el LAZO_ASCII.
También se han agregado instrucciones PUSH y POP para salvar y recuperar todos los registros que se utilizan dentro de DESP_LINEA. En este momento DESP_LINEA esta casi terminado; los cambios que se harán despues son practicamte esteticos para mostrar espacios y caracteres graficos de manera que la pantalla se vea mas atractiva.
Cuando se encadenan los archivos, hay que recordar que se tienen tres archivos: Desp_sec, Video_io, y Cursor. Desp_sec deberá de ser el primero en la lista. Despues de correr obtener la versión .COM utilizando E e2bin, se vera un despliegue como el de la figura 2.9

Se tendran muchos archivos que seran hechos después, pero por ahora, lo que se hara en la siguiente sección es aprender a leer un sector directamente desde el disco antes de mostrar medio sector. El cual se colocara a partir de la localización identificada como SECTOR.

```
A   desp_sec
    10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
    11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
    12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
    13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
    14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
    15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
    17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
    18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
    19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
    1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
    1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B
    1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
    1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D
    1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E
    1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F

A
```

Figura 2.9 Salida de Desp_sec.

2.17 MOSTRANDO SECTORES DEL DISCO

Ahora que ya se tiene un programa que muestra 256 bytes de memoria, se pueden agregar algunos procedimientos para leer un sector de de el disco y colocarlo en la memoria comenzando en SECTOR. Luego, con el procedimiento para mostrar bytes, se mostrara la primera mitad de este sector.

2.17.1 HACIENDO LA VIDA MAS FACIL

Con los tres archivos fuente que se tuvieron en la ultima sección, las cosas se vuelen mas complicadas. ¿Se tienen que compilar los tres archivos cuando en realidad solo se han modificado dos?, probablemente se compilen los tres, en lugar de revisar para ver si se ha realizado un cambio desde la ultima compilación.

Para la compilación de todos los archivos fuente cuando solo se han cambiado uno de ellos es muy lenta y será mas lenta cuando el programa DSI se hagan más grande. Lo que se debe hacer es compilar solo los archivos que se han modificado.

Afortunadamente, si se esta utilizando uno de los mas recientes Macro Assembler (O si se tiene un compilador C), existe una forma de realizar esta tarea. Estos incluyen un programa llamado MAKE que hace exactamente lo que se desea hacer. Para utilizarlo se deberá crear un archivo que se llamara DSI que le diga a MAKE como hacer su trabajo, simplemente escribiendo:

```
A> MAKE DSK
```

Make compilara solo los archivos que se han modificado.

El archivo que se creara (DSI) le dice a Make cuales archivos dependen de cuales otros archivos. Cada vez que se cambie un archivo, el DOS actualiza el tiempo de modificacion para este archivo (esto se puede ver en el despliegue del DIR) Make simplemente verá ambas versiones de un archivo, la versión ASM y la OBJ. Si la versión .ASM es mas reciente que la versión .OBJ, Make sabe que se necesita compilar el archivo de nuevo.

Hay algo que se debe de aclarar, Make trabajara correctamene solo si se tiene cuidado de colocar hora y la fecha cada vez que se carga el sistema operativo. Sin esta inforamción, Make no podrá saber cuando se han realizado cambios en los archivos.

2.17.2 FORMATO DEL ARCHIVO PARA EL PROGRAMA MAKE

El formato para el archivo que se llamara DSI, y que será utilizado por Make es claramente simple y se muestra a continuacion:

Listado 2.12 El archivo DSI para Make

```
desp_sec.obj:  desp_sec.asm
          masm desp_sec;
video_io.obj:  video_io.asm
          masm video_io;
cursor.obj:    cursor.asm
          masm cursor;

desp_sec.com: desp_sec.obj video_io.obj cursor.obj
          link desp_sec video_io cursor;
          ren desp_sec.exe desp_sec.com
```

Cada entrada tiene un nombre de archivo al lado izquierdo (antes de los dos puntos) y uno o mas nombres de archivos al lado derecho. Si algunos de los archivos en el lado derecho (tal como DESP_SEC.ASM en la primera linea) es mas reciente que el primer archivo (DESP_SEC.OBJ), Make ejecutará todos los comandos indentados que aparecen en las siguientes lineas. (NOTA: Las lineas de comandos se deben de indentar con un tab, y no con espacios.)

Si el compilador tiene el programa Make, para realizar la tarea de compilar nuevamente solo los archivos modificados, el comando que se deberá escribir es el siguiente:

```
C:\> MAKE DSK
```

Make hará la mínima cantidad de trabajo necesaria para reconstruir el programa.

2.17.3 MODIFICANDO DESP_SEC

Así como se dejó DESP_SEC.ASM, incluye una versión de DESP_MEDIO_SECTOR, el cual se utilizó como procedimiento de prueba, y como procedimiento principal. En este momento se cambiará DESP_MEDIO_SECTOR en un procedimiento ordinario de manera que se pueda llamar desde un procedimiento que se llamará Disk_io. El procedimiento de prueba estará en Disk_io, junto con una versión de prueba del procedimiento para leer un sector del disco.

Primero se debe modificar Desp_sec para hacerlo un archivo de procedimientos, exactamente como se hizo con Video_io. Cambiando la instrucción END DESP_MEDIO_SECTOR y dejar solamente END. Puesto que el programa principal estará en DISK_IO. Luego se removerá la instrucción ORG 100h de CODE_SEG , por que se ha movido a otro archivo.

Se planea leer un sector en la memoria comenzado en SECTOR, no hay necesidad de suministrar estos datos de prueba, y se puede reemplazar todas las instrucciones DB 16 despues de SECTOR con la siguiente linea:

```
SECTOR  DB  8192 DUP (0)
```

La cual reserva 8192 bytes para almacenar un sector.

Pero si se recuerda, anteriormente se dijo que los sectores tienen un longitud de 512 bytes. ¿Porque se necesita un area de almacenamiento tan grande?. Esto es porque algunos discos duros (300 Megabyte, por ejemplo) utilizan sectores de tamaño muy largo. Este tamaño de sectores no es muy comun. Pero, se desea tener la seguridad de que al leer un sector este no sea muy grande para el area reservada. En lo que sigue de este trabajo se asumirá que los sectores tienen solamente 512 bytes de longitud.

Lo que sigue ahora es escribir una nueva versión de DESP_MEDIO_SECTOR. La versión anterior es solamente un

procedimiento de prueba que se utilizo para probar DESP_LINEA. En la nueva versión, se desea suministrar un complemento dentro de un sector de manera que se pueda mostrar 256 bytes comenzando en cualquier parte del sector. Entre otras cosas, esto significa que se podran mostrar la primera mitad, la ultima mitad, o los 256 bytes del medio. De nuevo este complemento estara en DX. A continuación se muestra la version final de DESP_MEDIO_SECTOR en Desp_sec:

Listado 2.13 Versión final de DESP_MEDIO_SECTOR
en DESP_SEC.ASM

```
        PUBLIC  DESP_MEDIO_SECTOR
        EXTRN   ENVIA_CRLF:NEAR
;-------------------------------------------------------------
; Este procedimiento despliega medio sector (256 bytes)      ;
;                                                            ;
;   DS:DX  complemento de sector,en bytes multiplicar por 16  ;
;                                                            ;
; Utiliza:  DESP_LINEA, ENVIA_CRLF                          ;
;------------------------------------------------------------;
DESP_MEDIO_SECTOR  PROC  NEAR
      PUSH    DX
      PUSH    CX
      MOV     CX,16
MEDIO_SECTOR:
      CALL    DESP_LINEA
      CALL    ENVIA_CRLF
      ADD     DX,16
      LOOP    MEDIO_SECTOR
      POP     CX
      POP     DX
      RET
      INT     20h
DESP_MEDIO_SECTOR   ENDP
```

2.17.4 LECTURA DE UN SECTOR

En esta primera versión de LEE_SECTOR deliberadamente se ignoraran errores, tales como no tener disco en el disk drive. Esto no es buena práctica, pero esta no es la versión final de LEE_SECTOR. A continuación se muestra el listado del archivo DISK_IO.ASM

Listado 2.14 Archivo DISK_IO.ASM

```
CGROUP GROUP    CODE_SEG, DATA_SEG
      ASSUME    CS:CGROUP, DS:CGROUP
```

Listado 3.14 continuación

```
CODE_SEG        SEGMENT  PUBLIC
                ORG      100h

                EXTRN    DESP_MEDIO_SECTOR:NEAR
;---------------------------------------------------------------
; Este procedimiento lee  el primer sector del disco A y       ;
;  muestra la primera mitad del sector                         ;
;---------------------------------------------------------------;
LEE_SECTOR      PROC  NEAR
                MOV     AL,0                ;Disk driver A (número 0)
                MOV     CX,1                ;Leer solo un sector
                MOV     DX,0                ;Leer sector numero 0
                LEA     BX,SECTOR           ;Donde almacenar este sector
                INT     25h                 ;Leer el sector
                POPF                        ;Recuperar registro de estado
                XOR     DX,DX               ;Poner complemento de sector a 0
                CALL    DESP_MEDIO_SECTOR   ;Muestra primer medio sector
                INT     20h                 ;Retornar al DOS
LEE_SECTOR      ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT  PUBLIC
                EXTRN   SECTOR:BYTE
DATA_SEG        ENDS

                END     LEE_SECTOR
```

En este procedimiento hay tres nuevas instrucciones. La primera:

```
LEA    BX,SECTOR
```

Mueve la dirección, o complemento, de SECTOR (desde el inicio de CGROUP) en el registro BX: LEA significa Load Effective Address. Despues de esta instrucción LEA, DS:BX contiene la dirección completa de SECTOR, y el DOS utiliza esta dirección para la segunda nueva instrucción, la instrucción INT 25h, la cual se discutiraá más adelante. (LEA carga el complemento en BX sin colocar nada en DS: el programador debe asegurarse que DS apunte al segmento correcto).
SECTOR no esta en el mismo archivo fuente que LEE_SECTOR. Se encuentra en DESP_SEC.ASM. ¿Que hacer para decirle al assembler donde esta?. Para eso se utiliza la directiva EXTRN como se muestra:

```
DATA_SEG     SEGMENT PUBLIC
             EXTRN   SECTOR:BYTE
DATA_SEG     ENDS
```

Este conjunto de instrucciones le dicen al assembler que SECTOR está definida en DATA_SEG, el cual se encuentra en otro archivo fuente, y que SECTOR es una variable de bytes (en lugar de palabras). EXTRN será utilizada a menudo en las siguientes secciones; esta es la forma de utilizar la misma variable en varios archivos fuente. Solo se necesita tener cuidado en definir las variables en un solo archivo.

Retornando a la instrucción 25h, esta es una función especial que llama al DOS para que lea sectores de un disco. Cuando el DOS recibe una llamado de INT 25h, utiliza la información de los registros siguientes:

AL	Número del Drive (0=A, 1=B, etc)
CX	Número de sectores a leer
DX	Número de el sector que se leerá (el primero es 0)
DS:BX	Dirección de transferencia: Donde se colocará el sector leído

El número en el registro AL determina el driver del cual el DOS leerá los sectores. Si AL = 0, el DOS lee del driver A.

El DOS puede leer mas de un sector con una sola llamada, y leerá el número de sectores indicado en CX. En el ejemplo, se establece CX a uno así el DOS leerá solo un sector de 512 bytes.

Se pone DX a cero, así el DOS leerá el primer sector del disco. Se puede cambiar este número si se desea leer un sector diferente; mas tarde se hará.

DS:BX es la dirección completa para el área de memoria en donde el DOS colocará el(los) sector(es) leído(s). En este caso, DS:BX apuntan a la dirección de SECTOR, de manera que se pueda llamar DISP_MEMORIA SECTOR para que muestre la primera mitad del primer sector leído desde el disco colocado en el driver A.

Finalmente se notará la instrucción POPF inmediatamente despues de la instrucción INT 25h. Como se mencionó anteriormente, el 8088 tiene un registro llamado registro de estado que contiene la varias banderas, tales como la bandera de cero y la de acarreo. POPF es un instrucción POP especial que recupera una palabra en el registro de estado; ¿Porque se necesita esta instrucción POP?

La instrucción INT 25h pone primero en el stack el registro de estado, luego la dirección de retorno en el mismo stack. Cuando el DOS retorna de esta INT 25h, deja el registro de estado en el stack. El DOS hace esto de manera que pueda poner la bandera de acarreo al retornar si hay un error en el disco tal como intentar leer de un driver que no tiene disco. En este ejemplo no se chequeará errores pero se tiene que remover el registro de estado del stack. Esta es la razon de la instrucción POPF.
(NOTA: la instruccion 25h, junto con la instruccion 26h la cual escribe un sector en el disco, son las unicas rutinas del DOS que dejan el registro de estado en el stack.)

Ahora ya se puede compilar DISK_IO.ASM, y recompilar DISP_SEC.ASM. Luego, encadenar los cuatro archivos Inst_io,

Desp_sec, Video_io, y cursor, con Disk_io colocado primero. O si se tiene el programa MAKE, agregar estas dos líneas al archivo DOS:

```
Disk_io.obj:    disk_io.asm
      masm disk_io;
```

y cambiar las últimas tres líneas a:

```
disk_io.com:    disk_io.obj desp_sec.obj video_io.obj cursor.obj
      link disk_io desp_sec video_io cursor;
      exe2bin disk_io disk_io.com
```

Una vez creada la versión .COM de Disk_io, si se ejecuta este programa se verá la salida que se muestra en la figura 2.10 (Posteriormente se le agregará más a Disk_io, pero por ahora es suficiente). En la siguiente sección se harán cambios puramente estéticos para embellecer la salida de este programa.

```
A disk_io
1B D1 00 49 42 D4 20 20 33 2E 31 0A 02 02 01 00
0C 70 00 D0 02 FD 02 00 09 00 07 00 0A 0A 00 00
0A 0A 0A C4 5C 8A 33 ED B8 C0 07 8E D8 33 C9 A0
D2 79 E0 89 1E 1E 00 8C 06 20 00 88 16 22 0A B1
02 8E C5 0E D5 BC 00 7C 51 FC E1 36 C5 36 78 00
BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00
72 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1
E0 40 D8 80 00 E8 86 0A BB 00 05 53 B0 01 E8 AB
0A 5F BE 73 01 B9 0B 00 90 73 A6 75 62 83 C7 15
B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B
00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96
A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 E1
6F 01 E8 D7 00 E8 64 00 2B F0 76 01 E8 2A 00 5D
F7 26 0B 00 03 D8 5A EB E9 C1 11 B9 02 00 D3 E0
80 E4 03 74 04 FE 64 8A CC 5B 58 FF 2E 6F 01 BE
89  0 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18
```

Figura 2.10 Agregando Caracteres ASCII

2.18 ENSANCHADO EL DESPLIEGUE DEL SECTOR

En este capítulo se agregarán varios procedimientos a Video_io, también se modificará DESP_LINEA en Desp_sec. La mayoría de las modificaciones y adiciones que se harán serán para mejorar la apariencia de la salida, pero una de ellas agregará nueva información: añadirá números a la izquierda que actuan como las direcciones del comando D del Debug.

2.18.1 AÑADIENDO CARACTERES GRAFICOS

Las computadoras IBM PC tienen un numero de caracteres que se utilizan para trazar lineas, estas líneas se pueden utilizar para dibujar cuadros alrededor de varias partes de la pantalla de salida del programa. Se trazará un cuadro alrededor de los valores hex y otra alrededor de los caracteres ASCII que aparecen a la derecha de la pantalla de salida.

Se deberán de escribir las siguientes definiciones cerca del topo del archivo DESP_SEC.ASM, entre la directiva ASSUME y la primera directiva SEGMENT, dejando una o dos lineas en blanco antes y despues de las definiciones:

Listado 2.15 Agregar al inicio de DESP_SEC.ASM

```
;-- ------------------------------------------------------------;
; Caracteres graficos para el borde del sector                 ;
;-------------------------------------------------------------- -;
VERTICAL_BAR     EQU 0BAh
HORIZONTAL_BAR   EQU 0CDh
SUPERIOR_IZQ     EQU 0C9h
SUPERIOR_DER     EQU 0BBh
INFERIOR_IZQ     EQU 0C8h
INFERIOR_DER     EQU 0BCh
TUPE_T           EQU 0CBh
PUNTA_T          EQU 0CAh
TUPE_SUP         EQU 0D1h
PUNTA_SUP        EQU 0CFh
```

Estas son las definiciones para los caracteres graficos. Notar que se ha puesto un 0 antes de cada digito hexadecimal para que el ensamblador no los confunda con etiquetas.

Se podria haber escrito los numeros hexadecimales en lugar de las definiciones en el procedimiento, pero las definiciones hacen los procedimientos mucho mas faciles de comprender. Por ejemplo, si se comparan las siguietes dos instrucciones:

```
MOVE DL,VERTICAL_BAR
MOV  DL,0BAh
```

La mayoria de las personas hayará mas clara la primera.

A continuación se muestra el nuevo procedimiento DESP_LINEA para separar las diferentes partes del despliegue con el caracter VARTICAL_BAR, cuyo numero es 186 (0BAH). Como antes las adiciones se muestra en negrita:

Listado 2.16 Cambios a DESP_LINEA en DESP_SEC.ASM

```
DESP_LINEA  PROC  NEAR
     PUSH   B ;
     PUSH   C;
     PUSH   D;
```

Continuación listado 3.14

```
    MOV     BX,D
    MOV     DL,' '              ;Complemento se utiliza en BX
    CALL    ESCRIBE_CAR         ;Escribe separador
    MOV     DL,VERTICAL_BAR
    CALL    ESCRIBE_CAR         ;Traza lado izquierdo del cuadro
    MOV     DL,' '
    CALL    ESCRIBE_CAR
                                ;Ahora escribe 16 bytes
    MOV     CX,10               ;Cuenta 16 byte
    PUSH    DX                  ;Sal a complemento para los ASCII
LAZO_HEX:
    MOV     DL,SECTOR[BX]       ;Obtiene un byte
    CALL    ESCRIBE_HEX         ;Muestra byte en hex
    MOV     DL,' '              ;Escribe espacio entre numeros
    CALL    ESCRIBE_CAR
    INC     BX
    LOOP    LAZO_HEX

    MOV     DL,VERTICAL_BAR     ;Escribe separador
    CALL    ESCRIBE_CAR
    MOV     DL,' '
    CALL    ESCRIBE_CAR

    MOV     CX,10
    POP     BX                  ;Recupera complemento de SECTOR
LAZO_ASCII:
    MOV     DL,SECTOR[BX]
    CALL    ESCRIBE_CAR
    INC     BX
    LOOP    LAZO_ASCII
    MOV     DL,' '              ;Traza lado derecho del cuadro
    CALL    ESCRIBE_CAR
    MOV     DL,VERTICAL_BAR
    CALL    ESCRIBE_CAR

    POP     BX
    POP     CX
    POP     AX
    RET
DESP_LINEA ENDP
```

Luego se debe de compilar esta nueva versión de Desp_sec y encadenar los cuatro archivos (recordar colocar Desp_sec primero en la lista de los archivos que siguen al comando LINK). El resultado de estos cambios se muestra en la figura 3.11

La siguiente mejora consiste en agregarle direcciones al desplegue del sector, las que serán muy utiles posteriormente.

```
A disl_
EB 21 90 49 42 D4 20 20 33 2E 31 00 02 02 01 00  δ!ÉIBM  3.1.....
02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00  .p.%╨.².........
00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 A0  ...-\.3╪╕└•╪3╔ 
D2 79 E0 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1  πy.e...î. .ê.".▒
02 9E C5 9E D5 BC 00 7C 51 FC E1 36 C5 36 78 00  .Ä╞Ä╒╝.|Q³ß6┼6x.
BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00  ┐#|╣..≤ñ.ê.,.á...
A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1  ó'.┘x.╕#|½æ½í..┬
E0 40 E9 90 00 E8 86 00 BB 00 05 53 B0 01 E8 AB  α@Θÿ.Φå.╗..S ░☺½
00 5F BE 73 01 B9 0B 00 90 73 A6 75 62 83 C7 15  ._╛s.╣..Éªubâ╟§
B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B  ▒.ÉÉ≤ªuW&ïG.öï..
00 03 C1 43 F7 F1 80 3E 71 01 60 75 02 B0 14 36  ..┴C≈±Ç>q.'u.°.6
A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 E1  í..▒.╙ΦΦ;. 6..─ß
6F 01 E3 39 00 E8 64 00 2B F0 76 0D E8 26 00 52  o.π9.Φd.+≡v.Φ&.R
F2 2E 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0  ≥..... ┼Z68=.╣..╙α
80 E4 03 74 04 FE 64 9A CC 5B 58 FF 2E 6F 01 BE  Çπ.t.■d▄╠[X .o.╛
99 00 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18  ö.δUÉ.....◄▲..├í↑
```

Figura 2.11 agregando barras verticales

2.18.2 AGREGANDO DIRECCIONES AL DESPLIEGUE

La siguiente es agregar direcciones he adecimales hacia abajo de lado izquierdo de la pantalla de salida. Estos numeros seran e complemento (offset) desde el inicio de el sector, asi el primer numero sera 00, el siguiente 10, luego 20, y asi sucesivamente.

El proceso es claramente simple, Puesto que ya se tiene e procedimiento ESCRIBE_HE.' Para escribir un número en he a. Pero se tiene un problema al tratar con sectores de 512 bytes de longitud: ESCRIBE_HE1 imprime solo números he adecimales de dos digitos, mientras que se necesitan tres digitos para números mayores que 255.

La solución es la siguiente: puesto que los numeros estaran entre 0 y 511 (0h a 1FFh), el primer digito será un espacio, si el numero dial como BCh) es menor que 100h, o será un uno. Luego si el numero es mayor que 255, simplemente se escribirá un uno, seguido por el numero he adecimal del byte inferior. De lo contrario se escribira un espacio primero. A continuación se muestra las adiciones a DESP_LINEA que imprimiran esta columna de numeros he adecimales de tres digitos:

Listado 2.17 Adiciones a DESP_LINEA en DESP_SEC.ASM

```
DESP_LINEA    PROC   NEAR
        PUSH    B;
        PUSH    C;
        PUSH    D;
        MOV     B;,D;
        MOV     DL,' '

        CMP     BX,100h
```

Listado 2.17 continuacion

```
        JB      ESCRIBE_UNO
        MOV     DL,'1'
ESCRIBE_UNO:
        CALL    ESCRIBE_CAR
        MOV     DL,BL
        CALL    ESCRIBE_HEX

        MOV     DL,' '
        CALL    ESCRIBE_CAR
        MOV     DL,VERTICAL_BAR
                .
                .
                .
                .
```

Los resultados se pueden ver en la figura 2.12.

```
Addr_10
00 | EB 21 90 49 42 D4 20 20 33 2E 31 00 02 02 01 00 | δ!ÉIBM  3.1.....
10 | 02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 | .p.╥▓.².........
20 | 00 00 00 C4 5C 08 33 ED B8 C0 07 3E D8 33 C9 A0 | ...-\.3╫╕└.A╪3╔.
30 | D2 79 E0 99 1E 1E 00 8C 06 20 00 88 1E 22 00 B1 | π╤.e...ì. .ê.".▒
40 | 02 9E C5 8E D5 BF 00 7C 51 FC E1 3E C5 36 78 00 | .Ã╪A╒┘.|Q".6┼6x.
50 | BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 13 00 | ┐#|╣..≤.ê...à...
60 | A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1 | ó'.└x.╕#|½æ½ó..┬
70 | E0 40 E8 80 00 E8 86 00 BB 00 05 53 B0 01 E8 AB | α@Φ╟.Φå.┐..S≡.Φ½
80 | A0 5F BE 73 01 B9 0B 00 90 73 AE 75 62 83 C7 15 | á_╪s.╣..╔≤ûubâ╟.
90 | B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B | ▒.ÉÉ≤ûuW&ïG.oò..
A0 | 00 03 C1 49 F7 F1 90 3E 71 01 60 75 02 B0 14 96 | ..┬╫±╟?q.'ü. .ê
B0 | A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 E1 | ó..▒ └ΦΦ;. 6.._..
C0 | 6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52 | o.Φ9.Φd.+≡b.Φ&.ı
D0 | F2 26 0B A0 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0 | ≈&...╪Z68=.╣..╙α
E0 | 80 E4 03 74 04 FE 64 8A CC 5B 58 FF 2E 6F 01 BE | ÇΣ.ƒ.ⁿ-è╪[x .o.╪
F0 | 89 00 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18 | å.ΦUÉ.......╪æ..
```

Figura 2.12 Añadiendo números al lado izquierdo

De esta forma este programa se acerca al despliegue que se busca para presentar los sectores en forma agradable. Pero en la pantalla este despligue no esta muy centrado. Se necesita moverlo a la derecha unos tres espacios. Al hacer este último combio se tendra la versión final de DESP_LINEA.

Podria hacerse este cambio llamando ESCRIBE_CAR tres veces con un espacio, pero no se hará asi. En lugar de eso, se agregará otro procedimiento llamado ESCRIBE_CAR_N_VECES, a Video_io, como su nombre lo implica, este procedimiento escribe un caracter N veces. Es decir, Se coloca el número N en el registro CX y el codigo del caracter en el registro DL, y se llama ESCRIBE_CAR_N_VECES para escribir N copias de el caracter cuyo

codigo ASCII esta colocado en DL. De esta manera se podra escribir tres espacios colocando 3 en CX y 20h (el codigo ASCII para el caracter espacio) en DL.
El siguiente procedimiento se le agregara a VIDEO_IO.ASM:

Listado ...IO Agregar este procedimiento a VIDEO_IO.ASM

```
; PUBLIC ESCRIBE_CAR_N_VECES
;-------------------------------------------------------;
; Este procedimiento escribe N copias de un caracter    ;
;                                                       ;
;  DL  Codigo del caracterer                            ;
;  CX  Numero de veces a escribir el caracter           ;
;                                                       ;
; Utiliza: ESCRIBE_CAR                                  ;
;-------------------------------------------------------;
ESCRIBE_CAR_N_VECES  PROC NEAR
     PUSH   CX
N_VECES:
     CALL   ESCRIBE_CAR
     LOOP   N_VECES
     POP    CX
     RET
ESCRIBE_CAR_N_VECES  ENDP
```

Como se puede ver este procedimientoes muy simple, por que ya se cuenta con ESCRIBE_CAR. Se ha agregado este procedimiento, para hacer el programa DSP mucho mas claro cuando se llama a ESCRIBE_CAR_N_VECES, en lugar de escribir un lazo corto para imprimir copias de un caracter. Ademas este procedimiento se utilizara de nuevo varias veces.

A continuación se muestran los cambios para DESP_LINEA para agregar tres espacios a lado izquierdo de la pantalla. Los cambios de deben de hacer en DESP_SEC.ASM

```
     PUBLIC DESP_LINEA
     EXTRN  ESCRIBE_HEX:NEAR
     EXTRN  ESCRIBE_CAR:NEAR
     EXTRN  ESCRIBE_CAR_N_VECES:NEAR
;-------------------------------------------------------;
;  Este procedimiento despliega un linea de datos, o 16 bytes ;
;  primero en hex, y luego en ASCII.                    ;
;                                                       ;
;    DS:DX  Complemento a partir de sector, en bytes    ;
;                                                       ;
;Utiliza:  ESCRIBE_CAR, ESCRIBE_HEX, ESCRIBE_CAR_N_VECES ;
;Lee    :  SECTOR                                      ;
;-------------------------------------------------------;
DESP_LINEA  PROC NEAR
    PUSH    DX
    PUSH    CX
```

Continuación

```
    PUSH    DX
    MOV     BX,DX                   ;El complemento se utiliza en BX
    MOV     DL,' '
    MOV     CX,3                    ;Escribe 3 espacios
    CALL    ESCRIBE_CAR_N_VECES
                                    ;Escribir complemento en he a
    CMP     BX,100h                 ;¿Es el primer digito uno?
    JB      ESCRIBE_UNO             ;No, espación listo en DL
    MOV     DL,'1'                  ;Si, colocar 1 en DL
ESCRIBE_UNO:
                .
                .
                .
```

Se han realizado cambios en tres partes. Primero, se ha agregado una instrucción ERTRN para ESCRIBE_CAR_N_VECES, ya que este procedimiento reside en Video_io, y no en este archivo. También se ha cambiado el bloque de comentarios, para mostrar el uso de un nuevo procedimiento. El tercer cambio, son las dos líneas que utilizan ESCRIBE_CAR_N_VECES, la cuales no necesitan explicación. Lo que resta agregar mas características a la pantalla de salida para hacerla mucho mas fácil de leer.

2.19.3 AGREGANDO LINEAS HORIZONTALES

Agregar líneas horizontales a la pantalla de salida no es tan simple como suena, ya que hay que pensar en algunos casos especiales. Se tiene los extremos de la líneas en donde se debe de formar una esquina, y también hay que colocar un caracter en forma de T (┬,┴) en la parte superior e inferior de la separación entre los caracteres hexa y los caracteres ASCII.

Se podría escribir una larga lista de instrucciones para crear una línea horizontal, pero no se hará, ya que existe un forma mas corta. Se introducirá otro procedimiento llamado ESCRIBE_PATRON, el cual escribirá un patron de caracteres en la pantalla. Luego, todo lo que se necesitara en una pequeña area en la memoria para almacenar la descripción de cada patron. Utilizando este nuevo procedimiento, fácilmente se pueden añadir marcas de separación para dividir la ventana de los números hexadecimales, como se verá cuando se finalice esta sección.

ESCRIBE_PATRON utiliza nuevas instrucciones como LODSB y CLD que se describiran mas adelante, por ahora se deberá de introducir el siguiente procedimiento en el archivo VIDEO_IO.ASM:

Estudio 2.1ª Agregar este procedimiento a Video_in.asm

```
    PUBLIC  ESCRIBE_PATRON
;  ---  ---------------------  ------------------------------;
; Este procedimiento escribe una linea en la pantall, basandose ;
; en la forma de los datos                                      ;
;                                                               ;
;    DB  Numero de veces que se escribe un caracter             ;
; Pos:DI Direccion anterior del segmento                        ;
;                                                               ;
; Utiliza: ESCRIBE_CAR_N_VECES                                  ;
;  - - ---------------------------------------------------------;
ESCRIBE_PATRON  PROC NEAR
    PUSH    AX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    PUSHF                   ;Salva la bandera de direccion
    CLD                     ;Pone bandera de dirección p/ incremento
    MOV     SI,DX           ;Mueve incremento detro de SI para LODSB
LAZO_PATRON:
    LODSB                   ;Obtiene caracter en DL
    OR      AL,AL           ;¿Es el fin de datos?
    JZ      FIN_PATRON      ;Si, retorne
    MOV     DL,AL           ;No, Escribir caracter N veces
    LODSB                   ;Obtiene el cotador de repetición en AL
    MOV     CL,AL           ;Y coloca en CX para ESCRIBE_CAR_N_VECES
    XOR     CH,CH           ;Pone a cero byte superión de CX
    CALL    ESCRIBE_CAR_N_VECES
    JMP     LAZO_PATRON
FIN_PATRON:
    POPF                    ;Recupera bandera de direccion
    POP     SI
    POP     DX
    POP     CX
    POP     AX
    RET
ESCRIBE_PATRON  ENDP
```

Antes de ver como trabaja este procedimiento, se deberá de escribir los datos para el patrón de la linea, se colocarán los datos para el patrón de la linea superior en el archivo Desp_Sec, el cual es donde se usará la trayectoria. Para este fin, se agregará otro procedimiento, llamado INI_DESP_SEC , para inicializar el despliegue del sector, luego se modificará LEE_SECTOR para llamar al procedimiento INI_DESP_SEC.
Primero se deberá colocar los siguientes datos exactamente despues de SECTOR (en DESP_SEC.ASM), dentro del segmento de datos:

Listado 2.20. Adiciones a DESP_SEC.ASM

```
PATRON_LINEA_SUP  LABEL   BYTE
        DB     ' ',7    '
        DB    SUPERIOR_IZQ,1
        DB    HORIZONTAL_BAR,12
        DB    TOPE_SEP,1
        DB    HORIZONTAL_BAR,11
        DB    TOPE_SEP,1
        DB    HORIZONTAL_BAR,11
        DB    TOPE_SEP,1
        DB    HORIZONTAL_BAR,12
        DB    TOPE_T,1
        DB    HORIZONTAL_BAR,18
        DB    SUPERIOR_DER,1
        DB    0

PATRON_LINEA_INF  LABEL BYTE
        DB ' ',7
        DB INFERIOR_IZQ,1
        DB HORIZONTAL_BAR,12
        DB FONDO_SEP,1
        DB HORIZONTAL_BAR,11
        DB FONDO_SEP,1
        DB HORIZONTAL_BAR,11
        DB FONDO_SEP,1
        DB HORIZONTAL_BAR,12
        DB FONDO_T,1
        DB HORIZONTAL_BAR,18
        DB INFERIOR_DER,1
        DB 0
```

Cada instrucción DB contiene partes de los datos para una línea
El primer byte es el carcter a imprimir; el segundo byte le dic
a ESCRIBE_PATRON cuantas veces repetir ese caracter. Por
ejemplo, se comenzó la línea superior con siete espacios
seguidos por un caracter de esquina superior izquierda, seguid
luego por doce carcateres de barra horizontal, y as
sucesivamente. El último DB es un solitario cero hexadecimal, e
cual marca el final del patrón.
Se continuarían las modificaciones y se verá el resultado despue
que se discuta el trabajo interno de ESCRIBE_PATRON. A
continuación se muestra la versión de prueba de INI_DESP_SECT.
este procedimiento escribe la línea superior, el despliegue d
medio sector y finalmente la línea inferior. Colocar est
procedimiento en el archivo DESP_SEC.ASM, exactamente antes d
DESP_MEDIO_SECTOR:

Listado 2.21 agregar este procedimiento a DESP_SEC.ASM

```
 PUBLIC  INI_DESP_SEC
  EXTRN ESCRIBE_PATRON:NEAR, ENVIA_CRLF:NEAR
```

```
;       ---- -------- ----------------------------------------------;
;Este procedimiento inicia el despliegue de medio sector:          ;
;utiliza:    ESCRIBE_PATRON, ENVIA-CRF,DEP_MEDIO_SECTOR
;lee    :    PATRON_LINEA_SUP, PATRON_LINEA_INF
;    -   - -- -------- ----------------------------------------------;
INI_DESP_SEC    PROC   NEAR
 PUSH    DI
 LEA     DI,PATRON_LINEA_SUP
 MOV     ESCRIBE_PATRON
 CALL    ENVIA_CRLF
 MOV     DI,DI                  ; inicializa el comienzo del sector
 CALL    DESP_MEDIO_SECTOR
 LEA     DI,PATRON_LINEA_INF
 CALL    ESCRIBE_PATRON
 POP     DI
 RET
INI_DESP_SEC     ENDP
```

Se utilizó la instrucción LEA para cargar una dirección en el registro DI, así ESCRIBE_PATRON sabe donde encontrar los datos. Finalmente se necesita hacer un pequeño cambio a LEE_SECTOR en el archivo DISK_IO.ASM para llamar INI_DESC_SEC, en lugar de ESCRIBE_MEDIO_SECTOR, así el cuadro completo se trazara alrededor del despliegue del medio sector.

Listado 2.22 cambios a LEE_SECTOR en DISK_IO.ASM

```
 EXTRN INI_DESP_SEC:NEAR

;    -----  - ---------- ----------------------------------------- ;
; Este procedimiento lee el primer sector del disco A, y muestra;
; la primera mitad de este sector                                 ;
;-          -    -  --  ---------- ------------------   - --  ;

LEE_SECTOR      PROC   NEAR
 MOV            AL,0        ; disk driver a (0)
 MOV            CX,1        ; leer solo un sector
 MOV            DX,0        ; leer sector cero
 LEA            BX,SECTOR   ; donde almacenar este sector
 INT            25h         ; leer el sector
 POPF                       ;  sacar las banderas puestas  en  el
                            ;  satck   por DOS
 XOR            DX,DX       ; poner complemento a cero dentro de
                            ; sector
 CALL           INI_DESP_SEC ; mostrar primera mitad
 INT            20h         ; retorna al DOS
 LEE_SECTOR     ENDP
```

Esto es todo lo que se necesita para escribir las lineas inferior y superior para el despliegue del sector.
A continuación se deberá compilar y encadenar todos estos archivos (recordar compilar los tres archivos que se han

modificado), luego correr el resultado a través de E edosbin. y probar el resultado. La figura 2-13 muestra la salida que se deberá tener en pantalla.

A continuación se verá como trabaja ESCRIBE_PATRON. como se mencionó, se utilizan dos nuevas instrucciones. LODSB que significa LOAD STRING BYTE, la cual es una de las instrucciones para cadenas de caracteres. Esto no es exactamente lo que se esta haciendo aquí, pero el 8088 no tiene cuidado si se esta tratando con un string de caracteres ó solo con números, así LODSB es conveniente para lo que se desea hacer.

LODSB Mueve (carga) un solo byte en el registro AL, de la localización de memoria dada por DS:SI, un par de registros que se han utilizado anteriormente. Todos los registros segmentos, en el archivo .COM apuntan al inicio de un segmento, CGROUP, así DS ya está apuntando al segmento. Y antes de la instrucción LODSB se ha colocado el complemento dentro del registro SI con la instrucción MOVE SI,DI.

La instrucción LODSB es algo parecido a la instrucción MOV, pero más poderosa.

Con una instrucción LODSB, el 8088 mueve un byte dentro del registro AL, y luego incrementa ó decrementa el registro SI. Incrementando el registro SI se apunta al siguiente byte en memoria; decrementando el registro se apunta al byte previo en memoria.

A>disk_io

```
00 | EB 21 90 49 42 M4 20 2M 33 2E 31 0M 02 02 01 00 | δ!ÉIBM 3.1.....
10 | 02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 | .p.%╨.².........
20 | 00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 A0 | ...-\.3ø╢└.Ä╪3╔.
30 | D2 7º E0 99 1E 1E 00 8C 06 20 00 88 16 22 00 B1 | πy.e...î. .ê."▒▒
40 | 02 8E C5 9E D5 BC 00 7C 51 FC E1 36 C5 36 78 00 | .Ä╪Ä┌.!0".6╪6x.
50 | BF 23 7C B9 0B 00 F3 A4 1F 89 0E 2C 00 A0 18 00 | ┐#╡╣..≤.ê...à...
60 | A2 27 00 BF 7º 00 Bº 23 7C AB 91 AB A1 16 00 D1 | ó'.└x.╢#½╗▒½¿..┬
70 | E0 40 E9 80 00 E3 86 00 BB 00 05 53 B0 01 E8 AB | α@ΦÇ.πå.╖..S ░☺Ω½
80 | 00 5F BE 73 01 B3 0B 00 90 73 A6 75 E2 83 C7 15 | ._╛s.╢..É≤ªuΓâ╟.
90 | B1 0B 90 90 F3 A6 75 57 26 9B 47 1C 99 8B 0E 0B | ▒.ÉÉ≤ªuW&¢G.Öï..
A0 | 00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96 | ..┬╪÷±Ç>q.'u. .ê
B0 | A1 11 00 B1 04 D3 E8 E8 3B 00 FF 3E 1E 00 C4 E1 | í..▒╘╙╪╪;. 6.._..
C0 | 6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52 | o.Φ╔9.Φd.+≡b.Φ╓.r
D0 | F2 2E 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0 | ≈╣...╪ZδΘ=.╞..╞α
E0 | 90 E4 03 74 04 FE 64 9A CC 5B 59 FF 2E 6F 01 BE | ÇΣ.ƒ.ⁿ-è╞{x .o.╞
F0 | 93 00 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18 | ô.ΦUÉ.....╞ß..
```

fig 2.13 El despliegue encerrado en cuadros

El incremento anterior es exactamente lo que se decea hacer, se desean ir através del patrón un byte a la vez, comenzando desde el inicio, y esto es lo que la instruccion LODSB hace a causa del empleo de la porta nueva instrucción, CLD (clear Direction Flag) para limpiar la bandera de dirección. Si se tiene puesta la

banderas de dirección, la instrucción LODSB decrementaría el registro CX.
Aparte de LODSB y CLD, hay que notar que también se utilizó PUSHF y POPF, para salvar y restaurar el registro de banderas. Esto se hizo porque más tarde se utilizará la bandera de dirección en el procedimiento llamado PATRON.

2.18.4 AGREGANDO NUMEROS AL DESPLIEGUE

Antes de comenzar hacer cosas mas grandes y mejores hay que notar que al despliegue le hace falta una fila de numeros a traves de la parte superior. Numeros tales como 00,01,02,03 etc., esto recordara ver hacia abajo en las columnas y encontrar la direccion para algún byte. A continuacion se escribira un procedimiento para escribir esta fila de números:
Se agregará el procedimiento ESCRIBE_GUION_FILA_NUM a DES_SEL.ASM, e inmediatamente después de INI_DES_SEC:

Listado 7.11 Agregar este procedimiento a DES_SEL.ASM

```
PUBLIC ESCRIBE_CAR_N_VECES,ESCRIBE_HEX,ESCRIBE_CAR,NE
PUBLIC CONVIERTE_DIGITO_HEX,ENVIA_CRLF
;-------------------------------------------------------;
;  El procedimiento escribe numeros indice (0 a F) en la parte  ;
;  superior del despliegue del medio sector                     ;
;Llamar: ESCRIBE_CAR_N_VECES, ESCRIBE_HEX,ESCRIBE_CAR
;        ESCRIBE_DIGITO_HEX,ENVIA_CRLF
;-------------------------------------------------------;
ESCRIBE_FILA_NUM  PROC NEAR
 PUSH    CX
 PUSH    DX
 MOV     DL,' '                  ; escribe nueve espacios  al  lado
                                 ; izquierdo
 MOV      CX,9
 CALL    ESCRIBE_CAR_N_VECES
 XOR     DH,DH                   ;comenzar con cero
LAZO_NUMERO_HEX:

 MOV     DL,DH
 CALL    ESCRIBE_HEX
 MOV     DL,' '
 CALL    ESCRIBE_CAR
 ADD     DH,1
 CMP     DH,10H                  ;ya hecho aun?
 JB      LAZO_NUMERO_HEX
 MOV     DL,' '                  ;escribir numeros de a sobre ventana
                                 ;ASSCI
 MOV     CX,2
 CALL    ESCRIBE_CAR_N_VECES
 POP     DX,CX
```

Listado 2.11 Continuación

```
LA_U_DIGITO_HEX :

 CALL     ESCRIBE_DIGITO_HEX
 INC      DI
 CMP      DI,8000H
 JB       LA_U_DIGITO_HEX
 CALL     ENVIA_CRLF
 POP      DI
 RET
 ESCRIBE_FILA_NUM ENDP
```

Debemos modificar INI_DES_SEC (también DESF_SEC.ASM) como sigue y así llamar ESCRIBE_FILA_NUM antes de desplegar el medio sector

Listado 2.12 cambios a INI_DES_SEC en DES_SEC.ASM

```
;-------------------------------------------------------------
; Utilizará: ESCRIBE_PATRON, ENVIA_CLRF,DESC_MEDIO_SECTOR     ;
; ESCRIBE_FILA_NUM                                            ;
; LEE PATRON_LINEA_SUP, PATRON_LINEA_INF                      ;
;-------------------------------------------------------------

INI_DES_SEC          PROC NEAR
 PUSH                DI
 CALL                ESCRIBE_FILA_NUM
 LEA                 DI,PATRON_LINEA_SUP
 CALL                ESCRIBE_PATRON
 CALL                ENVIA_CRLF
 MOV                 DI,00                ; comenzar en el inicio
                                          ; del sector
 CALL                DESPLIEGA_MEDIO_SECTOR
 LEA                 DI,PATRON_LINEA_INF
 CALL                ESCRIBE_PATRON
 POP                 DI
 RET
INI_DES_SEC          ENDP
```

Ahora que se ha completado el despliegue del medio sector como puede observarse en la figura 2.14.
Todavía hay unas diferencias entre este despliegue y la versión final.
Posteriormente se cambiara ESCRIBE_CAR de manera que se puedan imprimir todos los 256 caracteres que se pueden desplegar en una IBM PC, y además se limpiara la pantalla y se centrara este despliegue verticalmente utilizando las rutinas del la ROM BIOS dentro de la IBM personal.

4 disk_io

```
     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
-------------------------------------------------------------------------
000  EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00  δ!ÉIBM  3.1.....
010  02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00  .p.╨²└.♫.......
020  00 00 00 C4 5C 09 33 ED B8 C0 07 8E D8 33 C9 A0  ...─\.3Ø╡└.Ä╪3╔á
030  D2 79 E0 89 1E 1E 00 8C 06 20 00 88 1E 22 00 B1  π y.e....ì. .ê.".▒
040  02 8E C5 8E D5 BC 00 7C 51 FC E1 36 C5 36 78 00  .Ä╪Ä╒╝.|Q■ß6┼6x.
050  BF 23 7C B7 0B 00 F3 A4 1F 8E 0E 2C 00 A0 18 00  ┐#|╖..≤ñ.Ä♫,.á↑..
060  A2 27 00 BF 78 00 B3 23 7C AB 91 AB A1 16 00 D1  ó'.┘x.│#|½æ½í..╤
070  E0 40 E9 80 00 E8 8E 00 BB 00 05 53 B0 01 E8 AB  α@ΘÇ.Φä.╗..S ☺Φ½
080  00 5F BE 73 01 B9 0B 00 90 73 A6 75 62 83 C7 15  ._╛s.╣..Éªubâ╟§
090  B1 0B 90 90 F3 AE 75 57 26 8B 47 1C 99 8B 0E 0B  ▒♫ÉÉ≤«uW&ïG.öï♫♫
0A0  00 03 C1 49 F7 F1 30 3E 71 01 60 75 02 B0 14 96  ♥♥┴I╫±0>q☺`u☻ °¶û
0B0  A1 11 00 B1 04 D3 E8 E6 3B 00 FF 36 1E 00 C4 E1  í◄.▒♦╙Φ╡;.  6▲..─ß
0C0  6F 01 E9 39 00 E9 64 00 2B F0 76 0D E9 26 00 52  o☺Θ9.Θd.+≡v♪Θ&.R
0D0  F2 26 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0  ≥&♂..♥╪Zδ Θ═◄╣☻.╙α
0E0  90 E4 03 74 01 FE 64 8A CC 5B 58 FF 2E 6F 01 BE  ÉΣ♥t☺■d è╠[X .o☺╛
0F0  89 00 EB 55 90 01 0E 1E 00 11 2E 20 00 C3 A1 13  ë.δUÉ♪♫▲..◄.. .├í‼
```

figura # 14 Despliegue de medio sector

2.19 RUTINAS DEL ROM BIOS

Dentro de una computadora estan algunos chips, ICs (Circuitos Integrados) conocidos como ROM (Read Only Memory). Una de estas ROM contiene un numero de rutinas, similares a los procedimientos, que suministran todas las rutinas basicas para varias diferentes tareas de entrada y salida de la computadora. Ya que esta ROM suministra rutinas para ejecutar operaciones de entrada y salida al mas bajo nivel, frecuentemente se conoce como BIOS (Basic Input Output System). En esta sección se hara un gran uso de las rutinas del BIOS, la cuales incluyen rutina que de otra forma no se podrían accesar sin trabajar directamente con el hardware.

2.19.1 RUTINAS DEL BIOS, Y VIDEO_IO

Se hara referencia a los elementos del BIOS como rutinas para distinguirlas de los procedimientos. Los procedimientos se utilizan con la instrucción CALL, mientras que las rutinas se llaman con la instrucción INT. Se utilizará la instrucción INT 10h, por ejemplo, para llamar rutinas de entrada/salida en pantalla (Video I/O), de la misma manera que se utilizo la instrucción 21h para llamar rutinas del DOS.

En esta sección, se utilizaran rutinas del ROM BIOS para añadir

nuevos procedimientos al programa PST: una para limpiar la pantalla, y la otra para mover el cursor a cualquier localización de la pantalla.

La instrucción INT 10h es el número de entrada a diferenetes funciones. hay que recordar que cuando se utiliza la instrucción del DOS INT 21h, se seleccionó una función particular colocando el numero de la función en el registro AH. La funciones de video_io se seleccionan de la misma forma: colocando el numero de función en el registro AH (Una lista completa de estas funciones se muestra en la tabla 2.2)

Tabla 2.2 Funciones de INT 20h

(AH)=0	Arreglo del modo display, el registro AL contiene el modo numerico
	MODO TEXTO
	(AL)=0 40 x 25, Para el modo en blanco y negro (AL)=1 40 x 25, Para color (AL)=2 80 x 25, Para blanco y negro (AL)=3 80 x 25, Para color (AL)=7 80 x 25, para adaptador de despliegue monocromatico.
	MODO GRAFICO
	(AL)=4 320 x 200, color (AL)=5 320 x 200, blanco y negro (AL)=6 640 x 200, blanco y negro
(AH)=1	Fija el tamaño del cursor
	(CH) Inicia el escaneado del cursor por lineas. La parte superior de la linea es cero para ambos tipos de despliegue grafico:monocromatico y color, mientras que la linea inferior es 7, para el adaptador grafico y 13 para el adaptador monocromático.Los rangos validos son de 0 a 31.
	(CL) programo escaneado del cursor por lineas.
	La activación para el adaptador grafico a color estdado por CH=6 y CL=7. Para el adaptador gafico monocromatico es: CH=11 y CL=12
(AH)=2	Fija la posición del cursor
	(DH,DL) Posiciona el cursor en la nueva fila y columna: tomando la esquina inferior izquierda como (0,0).

Tabla 2.2 continuación

(BH) Numero de página. este es el número de despliegue de página.Los adpetadores gráf cos poseen mucho espacio para el despliegu de paginas; pero algunas programa utilizán la página cero

(AH) = 3 **Lectura de la posición del cursor**

(BH) Numero de página
on e ib (DH,DL) fila,columna del cursor
(CH,CL) tamaño del cursor.

(AH)=4 **lectura de la posición del lápiz óptico**
(ver manual técnico de referencia)

(AH)=5 **Desplazamiento de página hacia arriba**

(AL) Número de lineas en blanco de la parte inferior de la ventana.
(CH,CL) Fila,columna de la esquina inferior izquierda de la ventana
(DH,DL) Fila, columna de la esquina superior derecha de la ventana
(BH) despliegue de los atributos utilizados para las lineas en blanco.

(AH)=7 **Despalzamiento hacia abajo**

Igual que la función 6, pero las lineas son blanqueadas a la izquierda de la parte superior de la entana.

(AH)=8 **lectura de atributos y caracteres sobre el cursor**

(BH) Despliegue de página (solo en el modo te to)
(AL) Lectura de caracter
(AH) Atributo de lectura de caracteres(solo en el modo te to)

(AH)=9 **Atributo de lectura de caracteres sobre el cursor**

(BH) Despliegue de pagina(solo en el modo te to)
(CX) Número de veces a escribir un caracter y atributos de pantalla.
(AL) caracter a escribir
(BL) Atributo de escritura

(AH)=10 **Escribe caracter bajo cursor (con atributo normal)**

tabla 2.2 continuación

	(BH)	página a desplegar
	(CX)	numero de veeces a escribir el caracter
	(AL)	caracter a escribir
(AH)=11 a 13	**varias funciones graficas** (ver manual técnico de referencia)	
(AH)=14	**escribe teletipo** Escribe un caracter en la pantalla y mueve el cursor a la siguiente posición	
	(AL)	Caracter a escribir
	(BL)	color del carcter (solo modo gráfico)
	(BH)	página a desplegar (solo en el modo te to)
(AH)=15	**Retorna el estado de video actual**	
	(AL)	Modo de despliegue actual
	(AH)	numero de caracteres por linea
	(BH)	página desplegada ártiva

Se utilizara la función 6 de la instrucción INT 10h, SCROLL ACTIVE PAGE UP (Desplazar hacia arriba pagina activa) para limpiar la pantalla. En este momento no se desea desplazar la pantalla, pero esta función tambien se utiliza para limpiar la pantalla. A continuación se muesta el procedimiento para limpiar la pantalla: el cual se deberá escribir en el archivo CURSOR.ASM:

Listado 2.23 Agregar este procedimiento a CURSOR.ASM

```
        PUBLIC  LIMPIA_PANTALLA
;--------------------------------------------------------------;
; Este procedimiento limpia la pantalla completamente          ;
;--------------------------------------------------------------;
LIMPIA_PANTALLA  PROC NEAR
     PUSH   AX
     PUSH   BX
     PUSH   CX
     PUSH   DX
     XOR    AL,AL       ;Limpiar ventana entera
     MOV    CX,0000     ;Esquina superior izq. esta en (0,0)
     MOV    DH,24       ;Linea inferior esta en linea 24
     MOV    DL,79       ;Lado derecho esta en columna 79
     MOV    BH,7        ;Utiliza atributo normal para limpiar
     MOV    AH,6        ;Llama la función DESPLAZAR HACIA ARRIBA
     INT    10h         ;Limpia la pantalla
```

Continuación listado 2.23

```
        POP     DI
        POP     SI
        POP     BX
        POP     AX
        RET
LIMPIA_PANTALLA  ENDP
```

En este procedimiento aparece la función 6 de la intrucción 10H necesita bastante información, aunque lo que se pretende es limpiar toda la pantalla. Esta funcion es mas poderosa: Puede limpiar cualquier parte rectangular de la pantalla (**conocida como ventana**). En este procedimiento se defino la ventana como la pantalla completa colocando la primera y la última línea a 0 y 24, y establece las columnas de 0 a 79. La rutinas que se estan utilizando tambien pueden limpiar la patalla a todo blanco (para utilizarlo con caracteres negros), o todo negro (para utilizarlo con caracteres blancos) que es lo que se indica con la instrucción MOV BH,7. Luego se coloca 0 en AL, en el registro AL se coloca el número de líneas que se desea desplazar. Es decir que con 0 en AL se esta diciendo que se desea limpiar sin desplazar linea.

Lo que sigue es modificar el procedimiento de prueba, LEE_SECTOR para que llame a LIMPIA_PANTALLA e actamente antes de comenzar a desplegar el sector. Para modificar LEE_SECTOR, agregar una declaracion EXTRN para LIMPIA_PANTALLA e insertar CALL LIMPIA_PANTALLA. A cotinuación se muestra los cambios a realizar.

Listado 2.24 Cambios para LEE_SECTOR en DISK_IO.ASM

```
    EXTRN  INI_DESP_SEC:NEAR, LIMPIA_PANTALLA:NEAR
;--------------------------------------------------------;
; Este procedimiento lee el primer sector del disco A y  ;
; muestra la mitad de este segmento                      ;
;--------------------------------------------------------;
LEE_SECTOR  PROC NEAR
    MOV    AL,0           ;Disk drive A (Numero 0)
    MOV    CX,1           ;Lee solo un sector
    MOV    DX,0           ;Lee sector número 0
    LEA    BX,SECTOR      ;Donde almacenar el sector
    INT    25h            ;Lee el sector
    POPF                  ;Saca registro de bandera
    XOR    DX,DX          ;Colocar complemeto de SECTOR a 0

    CALL   LIMPIA_PANTALLA
    CALL   INI_DESP_SEC   ;Muestra el primer medio sector
    INT    20h            ; Retorna al DOS
LEE_SECTOR  ENDP
```

Antes de correr esta nueva versión de Disk_io, hay que nota
donde esta localizado el cursor. Luego correr. Disk_io. l
pantalla se limpiará, y Disk_io comenzará a escribir el sector e
el lugar que el cursor estaba antes de que se corriera e
programa.
Aunque la pantalla fue limpiada, no se hizo nada para mover e
cursor a la parte superior de la pantalla. En BASIC, el comand
CLS limpia la pantalla en dos pasos: Limpia la pantalla, lueg
mueve el cursor a la parte superior de la pantalla. E
procedimiento escrito anteriormete no hace eso: el movimiento de
cursor se hace por cuenta del programador.

2.19.2 MOVIMIENTO DEL CURSOR

La función 2 de la instrucción 10h establece la posición de
cursor de la misma manera que lo hace la instrucción LOCATE d
BASIC. En esta sección se escribirá un procedimiento llamad
GOTO_XY para mover el cursor a cualquier parte de la pantall
(tal como la parte superior despues de haber limpiado l
pantalla).
El siguiente procedimiento se debera escribir en el archiv
CURSOR.ASM:

Listado 2.25 Agregar este procedimiento a CURSOR.ASM

```
        PUBLIC  GOTO_XY
;       - -- --   - ---- ------------------------------------------;
; Este procedimiento mueve el cursor                                ;
;                                                                   ;
;    DH   Fila (Y)                                                  ;
;    DL   Columna (X)                                               ;
;                 - -  ---- -----   --- ----------------------------;
GOTO_XY PROC NEAR
    PUSH AX
    PUSH BX
    MOV  BH,0          ;Despliega pagina 0
    MOV  AH,2          ;Llamada para establecer posición del cursor
    INT  10h
    POP  BX
    POP  AX
    RET
GOTO_XY ENDP
```

Se utilizara GOTO_XY en una versión revisada de INI_DESP_SEC.
Para mover el cursor a la segunda línea e actamente antes de
escribir el medio sector. A continuación se muestra las
modificaciones a INI_DESP_SEC en DESP_SEC.ASM

Listado 2.36 Cambios a INI_DESP_SEC en DESP_SEC.ASM

```
        PUBLIC  INI_DESP_SEC
        EXTRN   ESCRIBE_PATRON:NEAR, ENVIA_CRLF:NEAR
        EXTRN   GOTO_XY:NEAR
;--------------------------------------------------------------
; Este procedimiento inicializa el despliegue del medio sector
;
; Utiliza:  ESCRIBE_PATRON, ENVIA_CRLF, DESP_MEDIO_SECTOR
;           ESCRIBE_FILA_NUM, GOTO_XY
; Lee:      PATRON_LINEA_SUP, PATRON_LINEA_INF
;--------------------------------------------------------------
INI_DESP_SEC  PROC  NEAR
    PUSH  DX
    XOR   DL,DL         ; Mover cursor a  posición de inicio
    MOV   DH,2          ; de tercera linea
    CALL  GOTO_XY
    CALL  ESCRIBE_FILA_NUM
    LEA   DX,PATRON_LINEA_SUP
          .
          .
          .
```

Despues de correr el programa con los cambios anteriores se ver que el despligue del medio sector esta correctamente centrado. Como se puede ver es muy facil trabajar con la pantalla cuando se tienen las rutinas del ROM BIOS. En la siguiente sección, se utilizara otra rutina de ROM BIOS para mejorar ESCRIBE_CAR, de manera que se pueda escribir cualquier caracter en la pantalla. Pero antes de continuar se haran algunos otros cambios al programa, y se finalizará la sección con la escritura de un procedimiento llamado ESCRIBE_ENCABEZADO. El cual escribira una linea de estado en la parte superior de la pantalla, para mostrar el disco habilitado y el sector que se esta leyendo.

2.19.3 REESCRIBIENDO LAS VARIABLES UTILIZADAS

Ya se han realizado tantos cambios en el programa que es necesario rehacerlo antes de crear ESCRIBE_ENCABEZADO. Asi como esta solo procedimientos, muchos tienen números un poco dificiles de modificar los; por ejemplo LEE_SECTOR, lee el sector 0 de el driver A. Lo que se desea es colocar el sector y el driver en variables de memoria de manera que mas de un procedimiento pueda leerlas.

Se necesitara cambiar estos procedimientos de manera que utilicen variables en memoria, pero se comenzara colocando todas las variables en un archivo llamado DSI.ASM, para hacer el trabajo mas simple. DSI.ASM será el primero archivo en el programa DSK, las variables en memoria seran facilmente halladas en este archivo. A continuación se muestra el archivo DSI.ASM, que se completara con una larga lista de variables en memoria.

Listado ...7 El nuevo archivo DSI.ASM

```
CGROUP  GROUP  CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG    SEGMENT  PUBLIC
            ORG   100h

    EXTRN   LIMPIA_PANTALLA:NEAR, LEE_SECTOR:NEAR
    EXTRN   INI_DESP_SEC:NEAR

DISI        PROC  NEAR
     CALL   LIMPIA_PANTALLA
     CALL   LEE_SECTOR
     CALL   INI_DESP_SEC
      INT   20h
DISI        ENDP

CODE_SEG    ENDS

DATA_SEG    SEGMENT PUBLIC

      PUBLIC SECTOR_OFFSET
;-- ---------------------------------------------------------------;
; SECTOR OFFSET es el offset de el medio sector desplegado       ;
; Debe de ser un multiplo de 16, y no mayor que 256              ;
;      --- ---------------------------  ----------------------    ;
SECTOR_OFFSET DW  0

      PUBLIC  SECTOR_ACTUAL_NO, DISI_DRIVE_NO
SECTOR_ACTUAL_NO  DW  0       ;Inicializar Sector 0
DISI_DRIVE_NO     DW  0       ;Inicializa Drive A

      PUBLIC  LINEAS_DESPUES_SECTOR, LINEA_ENCABEZADO_NO
      PUBLIC  ENCABEZADO_PARTE_1, ENCABEZADO_PARTE_2
;                 -  -   ---------- ------------------------ --------;
; LINEAS DESPUES_SECTOR es el número de lineas en el tope        ;
; de la pantalla antes del despliegue del medio sector           ;
;          -  ---------- ------------------------------------    ;
LINEAS_DESPUES_SECTOR  DB  2
LINEA_ENCABEZADO_NO    DB  0
ENCABEZADO_PARTE_1     DB  'Disco '0
ENCABEZADO_PARTE_2     DB  '          Sector',0

       PUBLIC SECTOR
;  - -- ---------- ------------------------ ------ ----------------;
; El sector entero (hasta 8192 bytes) esta almacenado en esta    ;
; parte de memoria.                                              ;
;  -   ---------- ------------------------ --- ------------------;
SECTOR  DB  8192 DUP (0)
DATA_SEG    ENDS
     END    DISI
```

El procedimiento principal, DISK, llama a los otros tres procedimientos. Lo que continua ahora es reescribir LEE_SECTOR y INIT_DESP_SEC para utilizar las variables que se colocaron en el segmento de datos.

Antes que se pueda utilizar DISK, se necesita modificar Desp_sec, para reemplazar la definición de SECTOR con un EXTRN. También se necesita alterar Disk_io, para cambiar LEE_SECTOR a un procedimiento ordinario que se pueda llamar desde Disk.

Ocupandose de SECTOR primero, puesto que se a colocado en DISK.ASM como una variable en memoria, se necesita cambiar la definición de SECTOR en Desp_sec a una declaración EXTRN. Haciendo estos cambios en DESP_SEC.ASM:

Listado 2.28 Cambios a DESP_SEC.ASM

```
DATA_SEG  SEGMENT PUBLIC
      EXTRN  SECTOR:BYTE
      PUBLIC SECTOR
  SECTOR DB  512 DUP (0)

PATRON_LINEA_SUP  LABEL BYTE
      DB   ' ',7
      DB   SEPERIOR_IZQ,1
             .
             .
```

Reescribiendo el archivo DISK_IO.ASM de manera que contenga solo procedimientos, y que LEE_SECTOR utilice variables en memoria (no numeros fijos) para el sector y para el número del disk driver. A continuación se muestra la nueva versión de DISK_IO.ASM:

Listado 2.29 Cambios a DISK_IO.ASM

```
CGROUP  GROUP CODE_SEG, DATA_SEG
     ASSUME   CS:CGROUP, DS:CGROUP

CODE_SEG     SEGMENT PUBLIC
     ORG     100h
     PUBLIC LEE_SECTOR
DATA_SEG     SEGMET PUBLIC
     EXTRN   SECTOR:BYTE
     EXTRN   DISK_DRIVE_NO:BYTE
     EXTRN   SECTOR_ACTUAL_NO:WORD
DATA_SEG     ENDS
     EXTRN   INIT_DESP_SEC:NEAR, LIMPIA_PANTALLA:NEAR
; ------------------------------------------------------ ;
; Este procedimiento lee un sector (512 bytes) en SECTOR  ;
;                                                         ;
; Lee: SECTOR_ACTUAL_NO,DISK_DRIVE_NO                     ;
;Escribe: SECTOR                                          ;
```

```
;-------------------------------------------------------------------;
Listado 2.29 Continuacion

LEE_SECTOR  PROC NEAR
      PUSH    AX
      PUSH    BX
      PUSH    CX
      PUSH    DX
      MOV     AL,DISK_DRIVE_NO     ;Numero del driver
      MOV     CX,1                 ;Lee solo un sector
      MOV     DX,SECTOR_ACTUAL_NO  ;Numero de sector a leer
      LEA     BX,SECTOR            ;Donde almacenar el sector
      INT     25h                  ;Lee el sector
      POPF                         ;Saca registro banderas del stac
      MOV     DX,0                 ;Pone offset a 0 dentro de secto
      CALL    LIMPIA_PANTALLA
      CALL    INI_DESP_SEC
      INT     20h
      POP     DX
      POP     CX
      POP     BX
      POP     AX
      RET
LEE_SECTOR  ENDP

CODE_SEG    ENDS

  DATA_SEG    SEGMENT PUBLIC
    SECTOR    DB 512 DUP(?)
  DATA_SEG    ENDS
      END
```

Esta nueva versión de Disk_io utiliza las variables DISK_DRIVE_NO y SECTOR_ACTUAL_NO para el disk driver y el sector a leer. Puesto que estas variables esta definidas en DSK.ASM, no se necesita cambiar Disk_io cuando se comience a leer diferentes sectores en otros discos.

Si se esta utilizando el programa MAKE para reconstruir DSK.COM se necesita hacer algunas adiciones al archivo llamado DSK.

Listado 2.30 La nueva versión de DSK

```
dsk.obj:        dsk.asm
   masm         dsk;

disk_io.obj:    disk_io.asm
   masm     disk_io;

desp_sec.obj:   desp_sec.asm
   masm desp_sec;
```

```
video_io.obj:    video_io.asm
   masm   video_io;

cursor.obj:      cursor.asm
   masm   cursor;


dsk.com: dsk.obj disk_io.obj desp_sec.obj video_io.oj cursro.obj
     link dsk disk_io desp_sec video_io cursor;
     exe2bin dsk dsk.com
```

Si no se utiliza make, hay que asegurarse de recompilar todos los tres archivos cambiados (Dsk, Disk_io, y Desp_sec), y encadenar todos los cinco archivos, con Dsk listado primero:

```
LINK DSK DISK_IO DESP_SEC VIDEO_IO CURSOR;
EXE2BIN DSK DSK.COM
```

Ahora que se han realizado varios cambios, hay que probar Dsk para asegurarse de que trabaja correctamente, antes de continuar.

2.19.4 ESCRIBIENDO EL ENCABEZADO

Ahora que ya se han convertido los numeros fijos (del sector y el driver) a localizaciones de memoria, Se escribira un procedimiento llamado ESCRIBE_ENCABEZADO Para escribir la linea de estado, o encabezado, en la parte superior de la pantalla. El encabezado se vera como esto:

```
Disco A       Sector 0
```

ESCRIBE_ENCABEZADO utilizará ESCRIBE_DECIMAL para escribir el numero del sector que se esta accesando en decimal. Tambien escribira dos cadenas de caracteres, Disco y Sector (cada una seguido por un espacio), y la letra que identifica al disco, tal como A. Este procedimiento se colocara en el archivo VIDEO_IO.ASM.

Para comenzar, puesto que se hara referencia al segmento de datos (DATA_SEG), se deberá de cambiar la primera linea (la instrucción GROUP) en VIDEO_IO.ASM para que se lea:

```
CGROUP GROUP  CODE_SEG, DATA_SEG
```

Colocar el siguiente procedimiento en VIDEO_IO.ASM:

Listado 2.31 añadir este procedimiento a VIDEO_IO.ASM

```
        PUBLIC ESCRIBE_ENCABEZADO
DATA_SEG    SEGMENT  PUBLIC
        EXTRN  LINEA_ENCABEZADO_NO:BYTE
```

Listado 2.31 Continuación.

```
        PUBLIC  ENCABEZADO_PARTE_1:BYTE
        PUBLIC  ENCABEZADO_PARTE_2:BYTE
        PUBLIC  DISK_DRIVE_NO:BYTE
        PUBLIC  SECTOR_ACTUAL_NO:WORD
DATA    ENDS
        PUBLIC  GOTO_XY:NEAR

;--------------------------------------------------------------;
; Este procedimiento escribe el encabezado con en número del   ;
; del disk driver y el del sector                              ;
;                                                              ;
; Utiliza: GOTO_XY,ESCRIBE_CADENA,ESCRIBE_CAR,ESCRIBE_DECIMAL  ;
;          LINEA_ENCABEZADO_NO,ENCABEZADO_PARTE_1              ;
;          ENCABEZADO_PARTE_2,DISK_DRIVE_NO, SECTOR_ACTUAL_NO  ;
;--------------------------------------------------------------;
ESCRIBE_ENCABEZADO  PROC NEAR
        PUSH    DX
        XOR     DL,DL                   ;Mover cursor a linea de
        MOV     DH,LINEA_ENCABEZADO_NO  ;encabezado
        CALL    GOTO_XY
        LEA     DX,ENCABEZADO_PARTE_1
        CALL    ESCRIBE_CADENA
        MOV     DL,DISK_DRIVE_NO
        ADD     DL,'A'                  ;Imprimir disco A, B, ...
        CALL    ESCRIBE_CAR
        LEA     DX,ENCABEZADO_PARTE_2
        CALL    ESCRIBE_CADENA
        MOV     DX,SECTOR_ACTUAL_NO
        CALL    ESCRIBE_DECIMAL
        POP     DX
        RET
ESCRIBE_ENCABEZADO   ENDP
```

Listado 2.32 Agregar este procedimiento a VIDEO_IO.ASM

```
      PUBLIC  ESCRIBE_CADENA
;--------------------------------------------------------------;
;  Este  procedimiento  escribe una cadena de  caracteres en la ;
;  pantalla. La cadena debe de terminar con DB 0               ;
;                                                              ;
;  DS:DX Direcci¢n de la cadena                                ;
;                                                              ;
; Utiliza   ESCRIBE_CAR                                        ;
;--------------------------------------------------------------;
ESCRIBE_CADENA  PROC NEAR
  PUSH   AX
  PUSH   DX
  PUSH   SI
  PUSHF
```

Listado 2.32 Continuación.

```
   CLD
   MOV     SI,DI
LAZO_CADENA:
   LODSB
   OR      AL,AL
   JZ      FIN_CADENA
   MOV     DL,AL
   CALL    ESCRIBE_CAR
   JMP     LAZO_CADENA
FIN_CADENA:
   POPF
   POP     SI
   POP     DI
   POP     AX
   RET
ESCRIBE_CADENA  ENDP
```

Hasta donde se ha llegado, ESCRIBE_CADENA escribirá caracteres cuyos codigos ASCII sean menores que 32 como un punto (.), porque no se tiene una versión de ESCRIBE_CAR que escriba cualquier caracter. En la siguiente sección se tomará cuidado de este detalle.

Despues del trabajo de esta sección, se cambiara DISI que esta en DSI.ASM para que incluya la llamada a ESCRIBE_ENCABEZADO:

Listado 2.33 Cambios a DISI en DSI.ASM

```
    EXTRN  LIMPIA_PANTALLA:NEAR, LEE_SECTOR:NEAR
  EXTRN  INI_DESP_SEC:NEAR, ESCRIBE_ENCABEZADO:NEAR
DISI     PROC  NEAR
    CALL    LIMPIA_PANTALLA
    CALL    ESCRIBE_ENCABEZADO
    CALL    LEE_SECTOR
    CALL    INI_DESP_SEC
    INT     20h
DISI        ENDP
```

Ademas de lo que se observo en la figura 2.14 en la parte superior se debera de observar lo siguiente:

Disco A Sector 0

2.20 LA ULTIMA VERSION DE ESCRIBE_CAR

Hasta el momento se ha hecho una buena utilización de las rutinas del BIOS, como limpiar la pantalla y mover el cursor, pero hay muchos mas usos para estas rutinas, y se mostrarán algunos en esta sección.

Utilizando el DOS solamente, no se ha podido desplegar todos los

256 caracteres que una IBM PC o compatibles, es capaz de mostrar. En este capitulo, se presentará una nueva versión de ESCRIBE_CAR que muestre cualquier caracter, gracias a otra función de BIOS.

Luegos se escribira otro util procedimiento, llamado LIMPIA_HASTA_FIN_DE_LINEA Que limpia la linea desde la posición del cursor hasta el borde de la pantalla. Esto se utilizara en ESCRIBE_ENLABECADO, para limpiar el resto de la linea. Esto se utiliza, por ejemplo, se va de sector 10 al 9, un cero es dejado por el numero 10, y es necesario borrarlo para que no se leea sector 9h.

2.20.1 NUEVA VERSION DE ESRCRIBE_CAR

La función de la instrucción INT 10h del BIOS escribe un caracter, su **atributo** en el lugar en donde se encuentre el cursor. Los atributos controlan caracteristicas tales como subrayado, parpadeo (blinking) y color. Para el programa DSI se utilizaran solo dos atributos: el atributo 7, el cual es el atributo normal, y el atributo 70h, en cual indica un caracter negro sobre un fondo blanco y produce lo que se conoce con el nombre de video inverso (caracteres blancos sobre fondo negro). Los atributos se pueden establecer para cada caracter, y es lo que se hara mas tarde para crear un bloque de cursor en video inverso conocido como **cursor fantasma** .

A continuación se muestra la nueva versión de ESCRIBE_CAR, la cual escribe un caracter y luego mueve el cursor un caracter a la derecha. Entrar lo siguiente en el archivo VIDEO_IO.ASM:

Listado 2.34 Cambios a ESCRIBE_CAR en VIDEO_IO.ASM

```
        PUBLIC  ESCRIBE_CAR
        EXTRN   POSICION_CURSOR:NEAR
;-------------------------------------------------------------------;
; Este procedimiento muestra un caracter en la pantalla;
; utilizando rutinas del BIOS, asi caracteres como espacio;
; hacia atras son tratados como cualquier otro caracter y son;
; desplegados.                                                      ;
; Este procedimiento se encarga de actualizar la posición del;
; cursor.                                                           ;
;                                                                   ;
; DL  byte a imprimir en la pantalla                                ;
;                                                                   ;
; Utiliza POSICION_CURSOR.                                          ;
;-------------------------------------------------------------------;
ESCRIBE_CAR  PROC NEAR
  PUSH  AX
  PUSH  BX
  PUSH  CX
```

Listado 2.34 Continuación.

```
  PUSH  DX
  MOV   AH,9              ;Llama salida de caracter/atributo
  MOV   BH,0              ;Establece despliegue pagina 0
  MOV   CX,1              ;Escribe solo un caracter
  MOV   AL,DL             ;Caracter a escribir
  MOV   BL,7              ;Atributo normal
  INT   10h               ;Escribe caracter y atributo
  CALL  POSICION_CURSOR ;Mover cursor siguiente posición
  POP   DX
  POP   CX
  POP   BX
  POP   AX
  RET
ESCRIBE_CAR  ENDP
```

Al leer a través de este procedimiento se puede observar l instrucción MOV AH,0. Si se tiene un monitor con adaptadc gráfico, el adaptador tiene cuatro paginas en el modo de te t normal. Aqui solo se utilizara la primera pagina, la pagina 0 por eso esta instrucción.

El procedimiento anterior utiliza otro procedimiento llamad POSICION_CURSOR para mover el cursor una posición a la derecha al inicio de la siguiente línea si el movimiento del cursor pas de la columna 79. El siguiente procedimiento se deberá de coloca en CURSOR.ASM:

Listado 2.35 Agragar este procedimiento al CURSOR.ASM

```
    PUBLIC POSICION_CURSOR
;--  ------------------------------------------------------;
;Este procedimiento mueve el cursor una posición a la derecha o;
;a la siguiente línea si el cursor pasa del fin de la línea    ;
;                                                             ;
; Utiliza    ENVIA_CRLF                                       ;
;            ------------------------------------------------;
POSICION_CURSOR  PROC NEAR
  PUSH  AX
  PUSH  BX
  PUSH  CX
  PUSH  DX
  MOV   AH,3        ;Lee la posición actual del cursor
  MOV   BH,0        ;En pagina 0
  INT   10h         ;Lee la posición del cursor
  MOV   AH,2        ;Colocar nueva posición del cursor
  INC   DL          ;Establecer columna a la siguiente posicion
  CMP   DL,79       ;Asegurase de que columna sea  = 79
  JBE   BIEN
  CALL  ENVIA_CRLF ;Ir a la siguiente línea
  JMP   ECHO
```

Listado 2.35 Continuación.

```
BIEN:
    INT   10h
ECHO:
    POP   DI
    POP   CX
    POP   BX
    POP   AX
    RET
POSITION_CURSOR ENDP
```

CURSOR_POSICION utiliza dos nuevas funciones de la instrucción INT 10h. La función 3 lee la posición del cursor, y la función 2 cambia la posición del cursor. El procedimiento primero utiliza la función 3 para hallar la posición del cursor, la cual es retornada y dos bytes, el número de la columna en DL, y el número de línea en DH. Luego, CURSOR_POSICION incrementa en número de columna en DL y mueve el cursor. Si DL apunta a la ultima columna (79), el procedimiento envia un retorno de carro y un avance de línea para mover el cursor a la siguiente línea.

Con todos estos cambios, DSI deberá mostrar todos los 25 caracteres.

Lo que sigue ahora es algo mas interesante: Se escribirá u procedimiento para limpiar la línea del cursor hasta el final.

2.20.2 LIMPIANDO HASTA EL FINAL DE LA LINEA

En la sección anterior, se utilizo la instrucción INT 10h, con l función 6 para limpiar toda la pantalla en el procedimient LIMPIA_PANTALLA. En esa ocasión, se menciono que la función 6 s podria utilizar para limpiar cualquier ventana rectangular. Est capacidad se aplica aun si la ventana tiene solo una línea d altura y menor que una línea de longitud, es decir, que se pued utilizar la funcon 6 para limpiar parte de la línea.

El lado izquierdo de la ventana, en este caso, es la columna de cursor, la cual se obtiene con la función 3 el lado derecho de l ventana es siempre la columna 79. A continuación se muestra e procedimiento llamado LIMPIA_HASTA_FIN_DE_LINEA: el cual se deberá de colocar en CURSOR.ASM:

Listado 2.36 Agragar este procedimiento a CURSOR.ASM

```
  PUBLIC  LIMPIA_HASTA_FIN_DE_LINEA
;-------------------------------------------------------------;
; Este procedimiento limpia la línea desde la posición actual  ;
; del cursor hasta el final de la línea                        ;
;-------------------------------------------------------------;
LIMPIA_HASTA_FIN_DE_LINEA  PROC NEAR
    PUSH  AX
    PUSH  BX
```

Listado 2.36 Continuación.

```
    PUSH  CX
    PUSH  DX
    MOV   AH,3    ;Función leer posición del cursor
    XOR   BH,BH   ;en pagina 0
    INT   10h     ;obtener (X,Y) en DL,DH
    MOV   AH,6    ;Funcio para limpiar ventana
    XOR   AL,AL   ;Limpiar ventana
    MOV   CH,DH   ;Todo en la misma línea
    MOV   CL,DL   ;Comenzar en la posición del cursor
    MOV   DL,79   ;Y detenerse en el final de la línea
    MOV   BH,7    ;Utilizar atributo normal
    INT   10h
    POP   DX
    POP   CX
    POP   BX
    POP   AX
    RET
LIMPIA_HASTA_FIN_DE_PANTALLA  ENDP
```

Este procedimiento se utilizara en ESCRIBE_ENCABEZADO, para limpia el resto de la línea en donde se ha comenzado a leer otro sector (esto se hará muy pronto). No hay ninguna forma para ver como trabaja LIMPIA_HASTA_FIN_DE_LINEA con ESCRIBE_ENCABEZADO hasta que se agregue el procedimiento que permita leer un sector diferente y se actualice el despliegue, antes de continuar se deberan de hacer los siguientes cambios a ESCRIBE_ENCABEZADO en VIDEO_IO.ASM, para llamar LIMPIA_HASTA_FIN_DE_LINEA al final del procedimiento:

Listado 2.37 Cambios a ESCRIBE_ENCABEZADO en VIDEO_IO

```
  PUBLIC ESCRIBE_ENCABEZADO
DATA_SEG    SEGMENT  PUBLIC
  EXTRN     LINEA_ENCABEZADO_NO:BYTE
  EXTRN     ENCABEZADO_PARTE_1:BYTE
  EXTRN     ENCABEZADO_PARTE_2:BYTE
  EXTRN     DISK_DRIVE_NO:BYTE
  EXTRN     SECTOR_ACTUAL_NO:WORD
DATA_SEG

  EXTRN     GOTO_XY:NEAR LIMPIA_HASTA_FIN_DE_LINEA:NEAR
;--- ------------------------------ ------------------------
; Este procedimiento escribe el encabezado con el disk driver
; y el numero de sector
;
; Utiliza:  GOTO_XY, ESCRIBE_CADENA, ESCRIE_CAR, ESCRIBE_DECIMAL
;           LIMPIA_HASTA_FIN_DE_LINEA
; Lee:      LINEA_ENCABEZADO_NO, ENCABEZADO_PARTE_1,
;           ENCABEZADO_PARTE_2, DISK_DRIVE_NO, SECTOR_ACTUAL_NO
;----- ---------------------------------------------------------
```

Listado 2.47 Continuación.

```
ESCRIBE_ENCABEZADO  PROC_NEAR
   PUSH  DX
   CMP   DL,DL
   MOV   DH,LINEA_ENCABEZADO_NO
   CALL  GOTO_XY
   LEA   DX,ENCABEZADO_PARTE_1
   CALL  ESCRIBE_CADENA
   MOV   DL,DISK_DRIVE_NO
   ADD   DL,'A'
   CALL  ESCRIBE_CAR
   LEA   DX,ENCABEZADO_PARTE_2
   CALL  ESCRIBE_CADENA
   MOV   DX,SECTOR_ACTUAL_NO
   CALL  LIMPIA_HASTA_FIN_DE_LINEA  ; Limpia resto de sector
   POP   DX
   RET
ESCRIBE_ENCABEZADO  ENDP
```

Esta revisión marca la versión final de ESCRIBE_ENCABEZADO y también se completa el archivo CURSOR_ASM. Todavía están pendientes varias partes importantes de DSK, en la siguiente sección se agregarán los comandos centrales para DSK, con los que se podría leer otros sectores presionando F1 y F2.

2.21 ARCHIVO DSK

En esta sección se hara que el program DSK sea mas interactivo, se escribirá un simple procedimiento para entrada desde el teclado y un control central para DSK llamado MASTER. El trabajo de master será llamar el procedimiento correcto para cada tecla presionada. Por ejemplo, cuando se presione F1 se leerá y se desplegara el sector previo, el master llamará un procedimiento llamado SECTOR_PREVIO. para hacer eso, se harán varios cambios a DSK. Se comenzará creando master, y algunos otros procedimientos para formatear el despliegue.

2.21.1 EL MASTER

El master será el control central para DSK, de esta manera todas las entradas del teclado y las ediciones se harán por medio de el. El trabajo de MASTER será leer caracteres y llamar otros procedimientos para hacer el trabajo, pronto se verá como como master hace este trabajo, pero primero hay que ver como se ajusta a DSK.

Master tendrá su propio indicador (prompt), exactamente bajo el despliegue del medio sector en donde el cursor esperará por una entrada desde el teclado.

A continuación se muestran las primeras modificaciones a DSK.ASM; estas agregan los datos para la línea del indicador (prompt)

Listado 2.38 Agregar a DATA_SEG en DSI_ASM

```
  LINEA_ENCABEZADO_NO      DB  0
  ENCABEZADO_PARTE_1       DB  'Disco ',0
  ENCABEZADO_PARTE_2       DB            Sector ',0
      PUBLIC  LINEA_INDICADOR_NO, INDICADOR_EDITOR
LINEA_INDICADOR_NO   DB  21
INDICADOR_EDITOR     DB  'Presione tecla de funcio, o ENTER'
                     DB  'Caracter o byte hexa: ',0
```

Se han agregado nuevos indicadores, mas tarde, se tomara cuidado de asunto tales como entradas de un nuevo numero de sector, e trabajo se hará mas simple utilizando un procedimiento de us comun llamado ESCRIBE_LINEA_INDICADOR, para escribir cada línea d indicaciones. Cada procedimiento que utilic ESCRIBE_LINEA_INDICADOR debera de suministrar la dirección de l indicación a mostrar (Por ejemplo, la dirección d INDICADOR_EDITOR), y luego escribir la indicación en la línea 2 (porque LINEA_INDICADOR_NO es 21). La nueva versión de DISK e DSI.ASM utiliza ESCRIBE_LINEA_INDICADOR.

Listado 2.39 Agregar a DISK en DSI.ASM

```
  EXTRN LIMPIA_PANTALLA:NEAR, LEE_SECTOR:NEAR
  EXTRN INI_DESP_SEC:NEAR, ESCRIBE_ENCABEZADO:NEAR
  EXTRN ESCRIBE_LINEA_INDICADOR:NEAR, MASTER:NEAR
DISK    PROC NEAR
  CALL  LIMPIA_PANTALLA
  CALL  ESCRIBE_ENCABEZADO
  CALL  LEE_SECTOR
  CALL  INI_DESP_SEC
  LEA   DX,INDICADOR_EDITOR
  CALL  ESCRIBE_LINEA_INDICADOR
  CALL  MASTER
  INT   20h
DISK    ENDP
```

El master es un programa claramente simple, pero utiliza alguno nuevos trucos. El siguiente listado es la primera versión de e archivo CONTROL.ASM en el cual se encontrará el procedimiento MASTER.

Listado 2.40 El nuevo archivo CONTROL.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
    ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG    SEGMENT  PUBLIC
    PUBLIC  MASTER
    EXTRN   LEE_BYTE:NEAR
```

Listado 2.40 Continuación

```
;- ------------------------------------------------------------;
;Este es el control central. Durante la edición normal y;
;durante se observan los sectores, este procedimiento lee;
;caracteres desde el teclado y, si el caracter es una tecla;
;de comando (tales como las tecla de cursor), MASTER llama;
;los procedimientos que realizan el trabajo indicado. Este;
;progama esta echo para las teclas especiales listadas en la;
;TABLA_MASTER, donde el procedimiento direccionado esta;
;almacenado e actamente despues del nombre de la tecla.      ;
; Si el caracter no es una tecla especial, entoces se debera;
;de colocar en el buffer de edicion_ es decir el modo de;
;edición.                                                    ;
;                                                           ;
;Utiliza:  LEE_BYTE                                         ;
;- ---- ------------------------------------------------------;
MASTER  PROC NEAR
  PUSH  AX
  PUSH  BX
LAZO_CONTROL:
  CALL  LEE_BYTE            ;lee caracter en AX
  OR    AH,AH               ;AH=0 si no se ha leido caracter
                            ;-1 para el codigo e tendido
  JZ    LAZO_CONTROL        ;No ha leido caracter trate de nuevo
  JS    TECLA_ESPECIAL      ;lee codigo e tendido
;No hacer nada con el caracter por ahora
  JMP   LAZO_CONTROL        ;leer otro caracter
TECLA_ESPECIAL:
  CMP   AL,68               ;F10 salir?
  JE    FIN_CONTROL         ;Si, salga
  LEA   BX,TABLA_MASTER
LAZO_ESPECIAL:
  CMP   BYTE PTR [BX],0     ;Fin de tabla?
  JE    NO_EN_TABLA         ;Si, tecla no esta en tabla
  CMP   AL,[BX]             ;Esta esta entrada en la tabla
  JE    CONTROL             ;Si, luego ir a control
  ADD   BX,3                ;No, pruebe con la siguiente entrada
  JMP   LAZO_ESPECIAL       ;Revidar la siguiente entrada

CONTROL:
  INC   BX                  ;Apuntar a dirección del procedimiento
  CALL  WORD PTR [BX]       ;llamar procedimiento
  JMP   LAZO_CONTROL        ;Esperar por otra tecla

NO_EN_TABLA:
  JMP   LAZO_CONTROL
FIN_CONTROL:
  POP   BX
   POP  AX
   RET
MASTER   ENDP
```

Listado 3.40 Continuacion

```
CODE_SEG ENDS

DATA_SEG  SEGMENT  PUBLIC

CODE_SEG  SEGMENT PUBLIC
   EXTRN  SECTOR_PROXIMO:NEAR     ;En DISK_IO.ASM
   EXTRN  SECTOR_PREVIO:NEAR      ;En DISK_IO.ASM
CODE_SEG  ENDS

;- - ---------------------------------------------------------;
; Esta tabla contiene las tecla permitidas en codigo ASCII    ;
; y las direcciones de los procedimientos que se llamaran     ;
; cuando cada tecla es presionada.                            ;
;  El formato de la tabla es                                  ;
;    DB  72           ;Codigo para cursor hacia arriba        ;
;    DW  OFFSET CGROUP:FANTASMA_ARR                           ;
;- -- ---------------------------------------------------------;
TABLA_MASTER  LABEL BYTE
   DB  59                                ;F1
   DW  OFFSET CGROUP:SECTOR_PREVIO
   DB  60                                ;F2
   DW  OFFSET CGROUP:SECTOR_PROXIMO
   DB  0                                 ;Fin de tabla
DATA_SEG   ENDS
        END
```

TABLA_MASTER contiene los codigos ASCII e tendido para F1 y F2 cada codigo es seguido por la dirección de el procedimiento qu master debera llamar cuando lea determinado codigo e tendido. Po ejemplo, cuando LEE_BYTE, el cual es llamado por MASTER, lee F1 MASTER llama al procedimiento SECTOR_PREVIO.

La dirección del procedimiento que se desea llamar desde master esta en TABLA_MASTER, lo que lleva a utilizar una nueva directiva, OFFSET, para coseguir dicha dirección. La línea

```
DW OFFSET CGROUP:SECTOR_PREVIO
```

le dice al assembler que utilice el complemento del procedimiento SECTOR_PREVIO1. La locacización de este segmento es relativa al inicio de CGROUP, es por esto que se necesita CGROUP antes del nombre del procedimiento. Si no se hubiera puesto CGROUP, el assembler calcularia la dirección de SECTOR_PREVIO a partir del inicio del segmento de codigo (CODE_SEG), y esto podria no ser lo que se desea (Aunque asi como esta este programa este CGROUP no es absolutamente necesario, porque el segmento de codigo esta cargado primero. Aun asi, por el interes de claridad se escribe de cualquier manera).

Hay que notar que tabla_master contiene ambos tipos de datos, bytes y palabras. Esto lleva a unas pocas consideraciones.

Anteriormente, siempre se trataba con tablas de datos de un solo tipo: ya sean todos bytes o todos palabras. Pero aquí, se tienen ambos, es por esto que se le tiene que decir al assembler que tipo de datos esperar cuando utilice CMP o una instrucción CALL. En el caso de una instrucción escrita como esto:

```
CMP [BX],0
```

El assembler no sabrá si se desea comparar palabras o bytes. Pero si la instrucción se escribe como esto:

```
CMP BYTE PTR [BX],0
```

Se le está diciendo al assembler que BX apunta a un byte, y que se desea una comparación entre bytes. Similarmente la instrucción CMP WORD PTR [BX],0 compararía palabras. Por otro lado, una instrucción como CMP AL,[BX] no causa ningún problema, ya que AL es un registro de un byte.

Hay que recordar también que una instrucción CALL puede ser ya sea cerca (near) o lejana (far). Una llamada cercana necesita una palabra para la dirección, mientras que una llamada lejana necesita dos. La instrucción:

```
CALL WORD PTR [BX]
```

le dice al assembler, con WORD PTR, que [BX] apunta a una palabra, así es que debera de generar una llamada cercana y utilizar la palabra direccionada por [BX] como la dirección, que la dirección que se almacenó en tabla_master. (para una llamada lejana, la cual utiliza una dirección de dos palabras, se utilizaría la instrucción CALL DWORD PTR [BX]. DWORD significa doble palabra).

Antes de poder probar este nuevo procedimiento se deberan de escribir los otros procedimientos que todavía estan pendientes. LEE_BYTE es un procedimiento que lee caracteres y codigos extendidos ASCII desde el teclado. La versión final sera capaz de leer teclas especiales, tales como las teclas de cursor, caracteres ASCII, y números de dos digitos hexadecimales. En este punto se escribirá una versión simple de LEE_BYTE para leer ya sea un caracter o una tecla especial. A continuación se muestra la primera versión de KBD_IO.ASM, el cual tendrá todos los procedimientos para leer desde el teclado:

Listado 2.41 Nuevo archivo KBD_IO.ASM

```
CGROUP GROUP CODE_SEG
   ASSUME  CS:CGROUP, DS:CGROUP
```

Listado 2.41 Continuación.

```
CODE_SEG    SEGMENT  PUBLIC
    PUBLIC  LEE_BYTE
;--------------------------------------------------------;
;Este procedimiento lee un solo caracter ASCII. Esta es solo ;
;una version de prueba de LEE_BYTE
;
; Retorna byte en AL  Codigo de caracter (a no ser que  AH=0);
;                     AH  1 Si lee caracter ASCII
;                        -1 Si lee una tecla especial
;--------------------------------------------------------;
LEE_BYTE  PROC NEAR
   MOV    AH,7
   INT    21H
   OR     AL,AL
   JZ     CODIGO_EXTENDIDO
NO_EXTENDIDO:
   MOV    AH,1
LECTURA_HECHA:
   RET
CODIGO_EXTENDIDO:
    INT   21H
    MOV   AH,0FFH
    JMP   LECTURA_HECHA
LEE_BYTE  ENDP
CODE_SEG  ENDS
    END
```

Ahora se agregara ESCRIBE_LINEA_INDICADOR a VIDEO_IO.ASM como sigue:

Listado 2.42

```
    PUBLIC  ESCRIBE_LINEA_INDICADOR
    EXTRN   LIMPIA_HASTA_FIN_DE_LINEA:NEAR
    EXTRN   GOTO_XY:NEAR
DATA_SEG    SEGMENT  PUBLIC
    EXTRN   LINEA_INDICADOR_NO:BYTE
DATA_SEG    ENDS
;--------------------------------------------------------;
;Este procedimiento escribe la línea de indicaciones en la   ;
;pantalla y limpia hasta el final de la línea
;
;  DS:DI  Dirección de la línea de indicaciones
;  Lee:   LINEA_INDICADOR_NO
;--------------------------------------------------------;
ESCRIBE_LINEA_INDICADOR PROC NEAR
    PUSH  DI
    XOR   DL,DL                    ;Escribe la línea de indicaciones y
    MOV   DH,LINEA_INDICADOR_NO;   mueve el cursor ahí
```

Listado 2.42 Continuación.

```
    CALL   GOTO_XY
    POP    DX
    CALL   ESCRIBE_CADENA
    CALL   LIMPIA_HASTA_FIN_DE_LINEA
    RET
ESCRIBE_LINEA_INDICADOR ENDP
```

Realmente no hay mucho en este procedimiento. Mueve el cursor a inicio de la línea de indicaciones, la cual fue puesta (en DISP.ASM) a la línea 21, luego escribe la línea de indicaciones y limpia el resto de la línea.

2.21.2 Lectura de otros sectores

Finalmente se necesitan los dos procedimientos SECTOR_PREVIO y SECTOR_SIGUIENTE próximo para leer y desplegar el SECTOR_PREVIO y SECTOR_SIGUIENTE en el disco. Se deberá agregar estos procedimientos a DISP_IO.ASM:

Listado 2.43 añadir este procedimiento a DISP_IO.ASM

```
     PUBLIC     SECTOR_PREVIO
          EXTRN     INI_DESP_SEC:NEAR,ESCRIBE_ENCABEZADO:NEAR
          EXTRN     ESCRIBE_LINEA_INDICADOR:NEAR
DATA_SEG            SEGMENT PUBLIC
          EXTRN     SECTOR_ACTUAL_NO:WORD,INDICADOR_EDITOR:BYTE
DATA_SEG            ENDS
;-------------------------------------------------------------
;Este procedimiento lee el SECTOR_PREVIO si es posible
;
;Utiliza:   ESCRIBE_ENCABEZADO,LEE-SECTOR,INI_DESP_SEC
;           ESCRIBE_LINEA_INDICADOR
;    Lee:   SECTOR_ACTUAL_NO,INDICADOR_EDITOR
; Escribe:  SECTOR_ACTUAL_NO
;-------------------------------------------------------------
SECTOR_PREVIO       PROC NEAR
     PUSH   AX
     PUSH   DX
     MOV    AX,SECTOR_ACTUAL_NO  ; obtiene numero de sector actual
     OR     AX,AX    ; no decremente si ya es cero
     JZ     NO_DECREMENTE_SECTOR
     DEC    AX
     MOV    SECTOR_Actual_NO,AX  ; salvar nuevo número de sector
     CALL   ESCRIBE_ENCABEZADO
     CALL   LEE_SECTOR
     CALL   INI_DESP_SEC         ; muestra nuevo sector
     LEA    DX,INDICADOR_EDITOR
     CALL   ESCRIBE_LINEA_INDICADOR
NO_DECREMENTE_SECTOR:
```

Listado 3.43 continuación

```
    POP   DI
    POP   AX
    RET
SECTOR_PROCIMO   ENDP

    PUBLIC    SECTOR_PROCIMO
    EXTRN   INI_DESP_SEC:NEAR,ESCRIBE_ENCABEZADO:NEAR
    EXTRN     ESCRIBE_LINEA_INDICADOR:NEAR

DATA_SEC      SEGMENT  PUBLIC
    EXTRN     SECTOR_ACTUAL_NO:WORD, INDICADOR_EDITOR:BYTE
DATA_SEC      ENDS

;--  - -- ------------------------------------------------------
;
;UTILIZA: ESCRIBE_ENCABEZADO, LEE_SECTOR, INI_DESP_SEC
;           ESCRIBE_LINEA_INDICADOR
;LEE     : SECTOR_ACTUAL_NO, INDICADOR_EDITOR
;ESCRIBE: SECTOR_ACTUAL_NO
;-- -- ---------------------------------------------------------
SECTOR_PROCIMO  PROC NEAR
      PUSH  AX
      PUSH  DX
      MOV   AX,SECTOR_ACTUAL_NO
      INC   AX
      MOV   SECTOR_ACTUAL_NO,AX
      CALL  ESCRIBE_ENCABEZADO
      CALL  LEE_SECTOR
      CALL  INI_DESP_SEC
      LEA   DX,INDICADOR_EDITOR
      CALL  ESCRIBA_LINEA_INDICADOR
      POP   DX
      POP   AX
      RET
SECTOR_PROCIMO  ENDP
```

Ahora ya se esta listo para compilar todos los archivos creados o cambiados: DSI.ASM, VIDEO_IO.ASM, IBP_IO.ASM, CONTROL.DISI_IO. cuando se encadene los archivos del programa DSI, hay que recordar que ahoera hay siete archivos : DSI.ASM, DESI_SEC.ASM, DISI_IO.ASM, VIDEO_IO.ASM,IBD_IO.ASM, CONTROL.ASM, y CURSOR.ASM. Si se esta utilizando MAKE aqui estan las adiciones que se necesitan hacer al archivo DSI (la plica invertida al final de la cuarta linea apartir del fondo le dice a MAKE que se continuará la lista de archivos en la siguiente linea)

Listado 3.14 cambios a el archivo DSI utilizado por MAKE

```
cursor.obj:     cursor.asm
      masm cursor;
```

```
control.obj:    control.asm
     masm  control;

fbd_io.obj:     fbd_io.asm
     masm  fbd_io;

dsl.com:    dsl.obj disl_io.obj desp_sec.obj video_io.obj \
       cursor.obj  control.obj fbd_io.obj
       link dsl disl_io desp_sec video_io cursor control fbd_io;
       e:\bin\dsl dsl.com
```

Si no se tiene el programa MAKE es conveniente escribir el siguiente archivo .bat para encadenar y crear el archivo .COM .

```
LINK DSL DISL_IO DESP_SEC VIDEO_IO CURSOR CONTROL FBD_IO
E:\BIN\DSL DSL.COM
```

2.21.3 EL CURSOR FANTASMA

En esta sección se escribirá un procedimiento para escribir y borrar un cursor fantasma en la ventana hexadecimal y otro en la ventana ASCII. Un cursor fantasma es llamado así porque no es el cursor propio de una PC; aunque mas bien una sombra inusual puesto que invierte el caracter encendiendo el fondo a blanco y el caracter a negro (video inverso), en la ventana hexadecimal, se tiene el espacio para hacer este cursor de cuatro caracteres de ancho de manera que pueda ser fácil de leer, en la ventana ASCII, el cursor fantasma sera solamente de un caracter de ancho porque no hay espacio entre caracteres. Se tendrán que cubrir una serie de procedimientos y de instrucciones en esta sección de manera que se irán describiendo brevemente.

INT DESP_SEC es el único procedimiento que tiene que cambiar el despliegue del sector. Se comenzará el trabajo colocando una llamada a ESCRIBE_FANTASMA en INI_DESP_SEC . De esta forma se escribirá el cursor fantasma cada vez que se escriba un nuevo sector.

A continuación se muestra la versión final de INI_DESP_SEC en DESP_SEC.ASM:

Listado 2.45 cambios a INI_DESP_SEC en DESP_SEC.ASM

```
    PUBLIC  INI_DESP_SEC
    EXTRN   ESCRIBE_PATRON:NEAR, ENVIA_CRLF:NEAR
    EXTRN   GOTO_XY:NEAR, ESCRIBE_FANTASMA:NEAR
DATA_SEG    SEGMENT PUBLIC
    EXTRN   LINEAS_DESPUES_SECTOR:BYTE
    EXTRN   SECTOR_OFFSET:WORD
DATA_SEG    ENDS
```

Listado 2.45 Continuación.

```
;-- -- -- -- ------------------------------------------------------
;este procedimiento inicializa el despliegue del medio sector
;
;UTILICA: ESCRIBE_PATRON, ENVIA_CRLF, DESPLIEGA_MEDIO_SECTOR
;         ESCRIBE_FILA_NUM, GOTO_XY, ESCRIBE_FANTASMA
;    LEE: PATRON_LINEA_SUP, PATRON_LINEA_INF,
;         LINEAS_DESPUES_SECTOR
; ESCRIBE: SECTOR_OFFSET
; -  ------------------------------------------------------------
INI_DESP_SEC  PROC  NEAR
    PUSH  DX
    SUB   DL,DL       ; mueve cursor a posición
    MOV   DH,LINEA_DESPUES_SECTOR
    CALL  GOTO_XY
    CALL  ESCRIBE_FILA_NUM
    LEA   DX,PATRON_LINEA_SUP
    CALL  ESCRIBE_PATRON
    CALL  ENVIA_CRLF
    XOR   DX,DX
    MOV   SECTOR_OFFSET,DX  ;pone complemento de sector a cero
    CALL  DESPLIEGA_MEDIO_SECTOR
    LEA   DX,PATRON_LINEA_INF
    CALL  ESCRIBE_PATRON
    CALL  ESCRIBE_FANTASMA  ; escribe cursor fantasma
    POP   DX
    RET
INI_DESP_SEC  ENDP
```

Note que se a actualizado INI_DESP_SEC para usar e inicializa variables y ahora se establece sector offset ha cero, para desplegar la primera mitad de un sector.
continuaremos con ESCRIBE_FANTASMA el cual tomará un poco de esfuerzo. Ya se han escrito completamente seis procedimientos incluyendo ESCRIBE_FANTASMA. La idea es claramente simple, primero se movera el cursor real a la posición del cursor fantasma en la ventana he a y se cambiarán el atributo de los siguientes cuatro caracteres a video inverso (atributo 70h) esto crea un bloque blanco de cuatro caracteres de ancho con los numeros he adecimales en negro. Luego se hara lo mismo en la ventana ASCII pero para un solo caracter. Finalmente, moveremos el cursor real de regreso a donde estaba al inicio. Todos los procedimientos para el cursor fantasma estarán en FANTASMA.ASM. con escepción de ESCRIBE_ATRIBUTO_N_VECES, el procedimiento que establece los atributos de los caracteres.
escribir los siguientes procedimientos en el archivo FANTASMA.ASM

Listado 2.46 nuevo archivo FANTASMA.ASM

```
CGROUP    GROUP CODE_SEG, DATA_SEG
          ASSUME  CS:CGROUP,DS:CGROUP
```

Listado de continuacion

```
CODE_SEG  SEGMENT PUBLIC
          PUBLIC    MOVER_A_POSICION_HEXA
          EXTRN     GOTO_XY:NEAR
DATA_SEG  SEGMENT  PUBLIC
          EXTRN     LINEAS_DESPUES_SECTOR:BYTE
DATA_SEG  ENDS
;--------------------------------------------------------------
; Este procedimiento mueve el cursor real a la posición del
; cursor fantasma en la ventana hexa
;
;UTILIZA: GOTO_XY
;LEE      :       LINEA_DESPUES_SECTOR,      CURSOR_FANTASMA_X,
;                 CURSOR_FANTASMA_Y
;--------------------------------------------------------------
MOVER_A_POSICION_HEXA  PROC  NEAR
     PUSH  AX
     PUSH  CX
     PUSH  DX
     MOV   DH,LINEA_DESPUES_SECTOR ; hallar fila de fantasma (0,0)
     ADD    DH,2                   ;  mas la fila de hexa y de  barra
                                   ;  horizontal
     ADD   DH,CURSOR_FANTASMA_Y ;  DH fila del cursor fantasma
     MOV   DL,8                 ;  sangrar el lado   izquierdo
     MOV   CL,3                 ;  cada columa usa 3 caracteres as
     MOV   AL,CURSOR_FANTASMA_X ;  que se debe de multiplicar por
     MUL   CL
     ADD   DL,AL                ; suma a la sangria para obtener
     CALL  GOTO_XY              ; columna para cursor fantasma
     POP   DX
     POP   CX
     POP   AX
     RET
MOVER_A_POSICION_HEXA  ENDP

     PUBLIC MOVER_A_POSICION_ASCII
     EXTRN  GOTO_XY:NEAR
DATA_SEG    SEGMET PUBLIC
     EXTRN  LINEAS_DESPUES_SECTOR:BYTE
DATA_SEG    ENDS
;--------------------------------------------------------------;
; Este procedimiento mueve el cursor  real  al inicio del cursor ;
; fantasma en la ventana ASCII.                                  ;
;                                                                ;
; Utiliza: GOTO_XY                                               ;
; Lee     : LINEAS_DESPUES_SECTOR,      CURSOR_FANTASMA_X        ;
;           CURSOR_FANTASMA_Y                                    ;
;--------------------------------------------------------------;
MOVER_A_POSICION_ASCII  PROC NEAR
      PUSH  AX
      PUSH  DX
```

Listado 2.46 Continuación

```
        MOV     DH,LINEAS_DESPUES_SECTOR ;hallar fila cursor fantasma
        ADD     DH,2                     ;mas fila he  y barra horizo.
        MOV     DH,CURSOR_FANTASMA_Y     ;DH=fila de cursor fanatasma
        MOV     DL,59                    ;  sangrar el lado izquierdo
        ADD     DL,CURSOR_FANTASMA_X     ;  sumar CURSOR_X para obtener
                                         ;  posición X.
        CALL    GOTO_XY                  ;  para cursor fantasma
        POP     DX
        POP     AX
        RET
MOVER_A_POSICION_ASCII  ENDP

        PUBLIC  SALVAR_CURSOR_REAL
;-------------------------------------------------------------------;
; Este procedimiento salva la posición del cursor real en dos       ;
; variables CURSOR_REAL_X, y CURSOR_REAL_Y.                         ;
;                                                                   ;
; Entradas: CURSOR_REAL_X, CURSOR_REAL_Y                            ;
;-------------------------------------------------------------------;
SALVAR_CURSOR_REAL  PROC NEAR
        PUSH    AX
        PUSH    DX
        PUSH    CX
        PUSH    BX
        MOV     AH,3                ; Leer posición del cursor
        XOR     BH,BH               ; en pagina 0
        INT     10h                 ; y retorna en DL, DH
        MOV     CURSOR_RAAL_Y,DL    ; Salvar la posicion
        MOV     CURSOR_REAL_X,DH
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
SALVAR_CURSOR_REAL   ENDP

        PUBLIC RECUPERA_CURSOR_REAL
        EXTRN  GOTO_XY:NEAR
;-------------------------------------------------------------------;
; Este procedimiento recupera el cursor real en su posición         ;
; inicial, guardada en CURSOR_REAL_X, y CURSOR_REAL_Y.              ;
;                                                                   ;
; Utiliza:  GOTO_XY                                                 ;
; Lee     :   CURSOR_REAL_X, CURSOR_REAL_Y                          ;
;-------------------------------------------------------------------;
RECUPERA_CURSOR_REAL  PROC NEAR
        PUSH  DX
        MOV   DL,CURSOR_REAL_Y
        MOV   DH,CURSOR_REAL_X
        CALL  GOTO_XY
```

Listado 7.46 Continuación.

```
    POP   DI
    RET
RECUPERA_CURSOR_REAL  ENDP

     PUBLIC ESCRIBE_FANTASMA
     EXTRN  ESCRIBE_ATRIBUTO_N_VECES:NEAR
;-------------------------------------------------------------;
; Este procedimiento utiliza CURSOR_X y CURSOR_Y,  por medio de ;
; MOVER_A_.... como coordenadas para el cursor fantasma.        ;
; ESCRIBE_FANTASMA escribe este cursor fantasma                 ;
;                                                               ;
; Utiliza: ESCRIBE_ATRIBUTO_N_VECES, SALVAR_CURSOR_REAL
;          RECUPERAR_CURSOR_REAL, MOVER_A_POSICION_HEXA
;          MOVER_A_POSICION_ASCII
;-------------------------------------------------------------;
ESCRIBE_FANTASMA  PROC NEAR
     PUSH  CX
     PUSH  DX
     CALL  SALVAR_CURSOR_REAL
     CALL  MOVER_A_POSICION_HEXA     ;Coordenadas de cursor en
                                     ;ventana hexa
     MOV   CX,4                      ;Cursor fantasma 4 caracter
     MOV   AL,70h
     CALL  ESCRIBE_ATRIBUTO_N_VECES
     CALL  MOVER_A_POSICION_ASCII    ;Coord. cursor ventana ASCII
     MOV   CX,1                      ;Cursor un caracter de ancho
     CALL  ESCRIBE_ATRIBUTO_N_VECES
     CALL  RECUPERA_CURSOR_REAL
     POP   DX
     POP   CX
     RET
ESCRIBE_FANTASMA   ENDP

     PUBLIC BORRA_FANTASMA
     EXTRN  ESCRIBE_ATRIBUTO_N_VECES:NEAR
;-------------------------------------------------------------;
; Este procedimiento borra el cursor fantasma                   ;
;                                                               ;
; Utiliza: ESCRIBE_ATRIBUTO_N_VECES, SALVAR_CURSOR_REAL         ;
;          RECUPERAR_CURSOR_REAL, MOVER_A_POSICION_HEXA         ;
;          MOVER_A_POSICION_ASCII                               ;
;-------------------------------------------------------------;
BORRA_FANTASMA   PROC NEAR
     PUSH  CX
     PUSH  DX
     CALL  SALVAR_CURSOR_REAL
     CALL  MOVER_A_POSICION_HEXA     ;Coord. cursor ventana hexa
     MOV   CX,4                      ;regresar blanco sobre negro
     MOV   AL,7
     CALL  ESCRIBE_ATRIBUTO_N_VECES
```

Listado 2.46 Continuación

```
    CALL   MOVER_A_POSICION_ASCII
    MOV    CX,1
    CALL   ESCRIBE_ATRIBUTO_N_VECES
    CALL   RECUPERA_CURSOR_REAL
    POP    DI
    POP    CX
    RET
BORRA_FANTASMA   ENDP

CODE_SEG    ENDS

DATA_SEG  SEGMENT  PUBLIC
  CURSOR_REAL_X     DB      0
  CURSOR_REAL_Y     DB      0
    PUBLIC  CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y
CURSOR_FANTASMA_X  DB      0
CURSOR_FANTASMA_Y  DB      0
DATA_SEG   ENDS

    END
```

ESCRIBE_FANTASMA y BORRA_FANTASMA son similares. En realidad la única diferencia es el atributo utilizado: ESCRIBE_FANTASMA utiliza el atributo 70h, mientras que BORRA_FANTASMA utiliza el atributo 7.

Ambos procedimientos salvan la posición inicial del cursor con SALVAR_CURSOR_REAL, el cual utiliza la instrucción INT 10h con la función 3 para leer la posición del cursor y luego salvar esta posición en los dos bytes CURSOR_REAL_X y CURSOR_REAL_Y.

Después de salvar la posición del cursor real, después, ambos procedimientos ESCRIBE_FANTASMA y BORRA_FANTASMA llaman a MOVER_A_POSICION_HEXA, el cual mueve el cursor al inición del cursor fantasma en la ventana hexadecimal, después, ESCRIBE_ATRIBUTO_N_VECES escribe el atributo de video inverso para cuatro caracteres comenzando en el cursor y moviendose a la derecha. Esto escribe el cursor fantasma en la ventana hexadecimal. De manera similar, ESCRIBE_FANTASMA escribe después un cursor fantasma de un caracter de longitud en la ventana ASCII. Finalmente, RECUPERA_CURSOR_REAL recupera la posición inicial del cursor real, es decir, la posición en donde se encontraba antes de llamar ESCRIBE_FANTASMA.

El único procedimiento que aun no se ha escrito es ESCRIBE_ATRIBUTO_N_VECES, de manera que se tratara a continuación

2.21.4 CAMBIANDO LOS ATRIBUTOS DE LOS CARACTERES

El procedimiento ESCRIBE_ATRIBUTO_N_VECES se utilizara para hacer tres cosas. Primero, leera el caracter bajo la posición del cursor. Esto se hara porque la función 9 de la instrucción 10h que se utiliza para poner los atributos de los caracteres escribe ambos el caracteres y el atributo bajo el cursor. De esta manera ESCRIBE_ATRIBUTO_N_VECES cambiará el atributo a medida que se vayan escribiendo los nuevos atributos junto con los nuevos caracteres que se acaban de leer. Finalmente, el procedimiento mueve el cursor a la derecha a la siguiente posición así se puede repetir el procedimiento N veces. Los detalles se pueden ver en el procedimiento mismo: hay que colocar ESCRIBE_ATRIBUTO_N_VECES en el archivo VIDEO_IO.ASM:

```
Listado 2.4' Agregar este procedimiento a VIDEO_IO.ASM

     PUBLIC  ESCRIBE_ATRIBUTO_N_VECES
     EXTRN   POSICION_CURSOR:NEAR
;--- ------------------------------------------------------ ;
; Este procedimiento establece los atributos de N caracteres ;
; comenzando con el de la posición actual del cursor         ;
;                                                            ;
;    CX   número de caracteres a establecer atributos       ;
;    DL   nuevo atributo para el caracter                   ;
;                                                            ;
; Utiliza: POSICION_CURSOR                                   ;
;--- ---------- ---------- ---------- ------------------------;
ESCRIBE_ATRIBUTO_N_VECES  PROC  NEAR
     PUSH  AX
     PUSH  BX
     PUSH  CX
     PUSH  DX
     MOV   BL,DL                    ;coloca nuevo atributo
     XOR   BH,BH                    ;poner despliegue a pagina 0
     MOV   DX,CX                    ;CX utilizada para rutina BI
     MOV   CX,1                     ;atributo para un caracter
LAZO_ATRIBUTO:
     MOV   AH,8                     ;lee caracter bajo cursor
     INT   10h
     MOV   AH,9                     ;escribir atributo /caracter
     INT   10h
     CALL  POSICION_CURSOR
     DEC   DX                       ;poner atributos para n
                                    ;caracteres?
     JNZ   LAZO_ATRIBUTO
     POP   DX
     POP   CX
     POP   BX
     POP   AX
     RET
ESCRIBE_ATRIBUTOS_N_VECES ENDP
```

Con esto se obtiene la versión final de VIDEO_IO.ASM
Si se corre el programa DSI ahora se verá el despliegue del sector con los dos cursores fantasmas, una línea de emulazado y una línea de indicaciones como se muestra en la figura 2.15

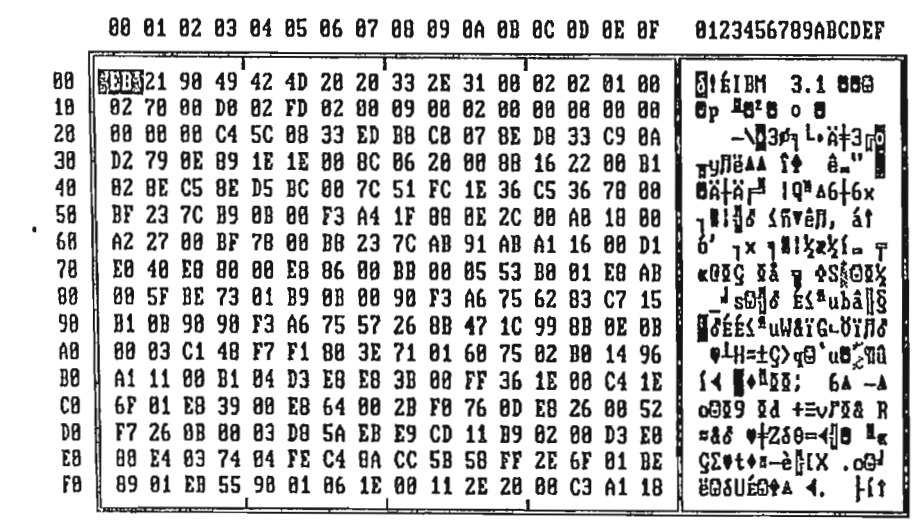


Figura 2.15 despliegue de pantalla con cursores fantasmas

2.22 EDICION SIMPLE

Ya casi se ha alcanzado el punto en el cual se podrá editar el despliegue del sector (cambiar numeros en el medio sector desplegado). Dentro de poco se agregarán simples versiones de procedimientos para editar bytes en el despliegue, pero antes de hacerlo se necesita alguna forma para mover el cursor fantasma a diferentes bytes dentro del despliegue del medio sector. Esta tarea resulta claramente simple ahora que ya se tienen dos procedimientos: BORRAR_FANTASMA y ESCRIBIR_FANTASMA.

2.22.1 MOVIENDO EL CURSOR FANTASMA

El movimiento del cursor fantasma en alguna dirección depende de tres pasos básicos:borrar el cursor fantasma de su posición actual ; cambiando la posición del cursor segun se modifique una de las variables , CURSOR_FANTASMA_X o CURSOR_FANTASMA_Y ; y utilizando ESCRIBE_FANTASMA para escribir el cursor fantasma a su nueva posición. En el proceso, sin embargo, se debe de tener el cuidado de no permitir que el cursor se mueva fuera de la ventana la cual tiene 16 bytes de ancho y 16 bytes de altr.
Para mover el cursor fantasma, se necesitaran cuatro nuevos procedimientos,uno para cada una de las teclas de flechas del teclado. MASTER no necesita cambio, ya que toda la información sobre procedimientos y códigos e tendidos está en la TABLA_MASTER, solo se necesita agregar los códigos ASCII e tendidos y las direcciones de los procedimientos para cada teclas de flechas. A continuación se mostraran las adiciones para CONTROL.ASM que le dará vida a las teclas de cursor .

Listado 2.48 Cambios a CONTROL.ASM

```
DATA_SEG   SEGMENT  PUBLIC
CODE_SEG   SEGMENT  PUBLIC
    EXTRN  SECTOR_PROXIMO:NEAR                  ;En DISK_IO.ASM
    EXTRN  SECTOR_PREVIO:NEAR                   ;En DISK_IO.ASM
    EXTRN  FANTASMA_ARR:NEAR, FANTASMA_ABA:NEAR
    EXTRN  FANTASMA_IZQ:NEAR, FANTASMA_DER:NEAR
CODE_SEG   ENDS
;-------------------------------------------------------;
; Esta tabla contiene las tecla permitidas en codigo ASCII   ;
; y las direcciones de los procedimientos que se llamaran    ;
; cuando cada tecla es presionada.                           ;
;  El formato de la tabla es                                 ;
;    DB  ??           ;Codigo para cursor hacia arriba       ;
;    DW  OFFSET CGROUP:FANTASMA_ARRIBA                       ;
;    .    ------------------------------------------------;
TABLA_MASTER  LABEL BYTE
    DB  59                                  ;F1
    DW  OFFSET CGROUP:SECTOR_PREVIO
    DB  60                                  ;F2
    DW  OFFSET CGROUP:SECTOR_PROXIMO
    DB  72                                  ;Cursor arriba
    DW  OFFSET CGROUP:FANTASMA_ARR
    DB  80                                  ;Cursor abajo
    DW  OFFSET CGROUP:FANTASMA_ABA
    DB  75                                  ;Cursor izquierda
    DW  OFFSET CGROUP:FANTASMA_IZQ
    DB  77                                  ;Cursor derecha
    DW  OFFSET CGROUP:FANTASMA_DER
    DB  0                                   ;Fin de tabla
DATA_SEG    ENDS
        END
```

Como se puede ver, es simple agregar mas comandos a CONTROL: Simplemente se coloca el nombre del procedimiento en TABLA_MASTER y se escribe el procedimiento. Lo que sigue ahora es escribir los procedimientos que se encargaran de mover el cursor. Dichos procedimientos se muestran en el siguiente listado:

Listado 2.10 Agregar estos procedimientos a FANTASMA.ASM

```
;--------------------------------------------------------------;
; Estos cuatro procedimientos mueven el cursor fantasma        ;
;                                                              ;
; Utiliza: BORRA_FANTASMA, ESCRIBE_FANTASMA                    ;
; Lee    : CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y                ;
; Escribe: CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y                ;
;--------------------------------------------------------------;
        PUBLIC  FANTASMA_ARR
FANTASMA_ARR  PROC NEAR
        CALL    BORRA_FANTASMA          ;Lo borra de la posición actual
        DEC     CURSOR_FANTASMA_Y       ;Mueve cursor 1 línea arriba
        JNS     NO_TOPE                 ;Cursor no esta e tremo superior
        MOV     CURSOR_FANTASMA_Y,0     ;esta en e tremo, no se mueve
NO_TOPE:
        CALL    ESCRIBE_FANTASMA        ;Escribir fantasma nueva posicion
        RET
FANTASMA_ARR  ENDP

        PUBLIC  FANTASMA_ABA
FANTASMA_ABA  PROC NEAR
        CALL    BORRA_FANTASMA          ;Lo borra de la posición actual
        INC     CURSOR_FANTASMA_Y       ;Mueve cursor 1 línea abajo
        CMP     CURSOR_FANTASMA_Y,16    ;Cursor no esta e tremo infer.?
        JB      NO_FONDO                ;No, escriba cursor fantasma
        MOV     CURSOR_FANTASMA_Y,15    ;Si, esta en e tremo, no se mueve
NO_FONDO:
        CALL    ESCRIBE_FANTASMA        ;Escribir fantasma nueva posicion
        RET
FANTASMA_ABA  ENDP

        PUBLIC  FANTASMA_IZQ
FANTASMA_IZQ  PROC NEAR
        CALL    BORRA_FANTASMA          ;Lo borra de la posición actual
        DEC     CURSOR_FANTASMA_X       ;Mueve cursor 1 columna a izq.
        JNS     NO_IZQ                  ;Cursor no esta e tremo izquierdo
        MOV     CURSOR_FANTASMA_X,0     ;esta en e tremo, no se mueve
NO_IZQ:
        CALL    ESCRIBE_FANTASMA        ;Escribir fantasma nueva posicion
        RET
FANTASMA_IZQ  ENDP
        PUBLIC  FANTASMA_DER
FANTASMA_DER  PROC NEAR
        CALL    BORRA_FANTASMA          ;Lo borra de la posición actual
        INC     CURSOR_FANTASMA_X       ;Mueve cursor 1 columna a derecha
```

Listado 2.49 Continuacion

```
      CMP   CURSOR_FANTASMA_[X],16 ;Esta cursor en e tremo derecho?
      JB    NO_DER                 ;Cursor no esta e tremo derecho
      MOV   CURSOR_FANTASMA_[X],15 ;esta en e tremo, no se mueve
NO_DER:
      CALL  ESCRIBE_FANTASMA       ;Escribir fantasma nueva posicion
      RET
FANTASMA_DER  ENDP
```

FANTASMA_IZQ y FANTASMA_DER ya son las versiones finales, pero se tendrá que cambiar FANTASMA_ARR y FANTASMA_ABA cuando se comience a desplazar la pantalla.

Así como esta del ahora, solo se puede ver el primer medio sector. En la sección 2.26, se harán algunos cambios a DOS para que se pueda observar el otro medio sector. Hasta entonces, se cambiará FANTASMA_ARR y FANTASMA_ABA para desplazar la pantalla cuando se trate de mover el cursor mas alla del tope o del fondo de la pantalla. Por ejemplo, cuando el cursor este en el fondo del despliegue del medio sector, y se presione la tecla flecha hacia abajo de nuevo deberá de desplazarse el despliegue una línea hacia arriba, agregando otra línea en el fondo, de manera que se vean los siguientes 16 bytes. El desplazamiento es un poco confuso, sin embargo, estos procedimientos se mantendrán hasta casi el final.

2.22.2 EDICIONES SIMPLES

Listado 2.50 Cambios a MASTER en CONTROL.ASM

```
      PUBLIC  MASTER
      EXTRN  LEE_BYTE:NEAR, EDITAR_BYTE:NEAR
;  ---- ---------- ---------- ---------- ---------- ----------;
;Este  es  el control central. Durante la  edición  normal  y;
;durante  se  observan los sectores, este  procedimiento  lee;
;caracteres  desde el teclado y, si el caracter es una  tecla;
;de  comando (tales como las tecla de cursor),  MASTER  llama;
;los  procedimientos que realizan el trabajo  indicado.  Este;
;programa echa echo para las teclas especiales listadas en  la;
;TABLA MASTER  donde  el  procedimiento  direccionado  esta;
;almacenado e actamente despues del nombre de la tecla.      ;
; Si el caracter no es una tecla especial, entoces se  debera;
;de  colocar  en el buffer de edicion_ es decir  el  modo  de;
;edicion.                                                    ;
;                                                           ;
;Utiliza:  LEE_BYTE, EDITAR_BYTE                            ;
;---------- ---------- ---------- ---------- ----------- ---;

MASTER  PROC NEAR
  PUSH  AX
  PUSH  BX
  PUSH  DX
```

Listado 3.5b continuación

```
LAZO_CONTROL
   CALL   LEE_BYTE              ;lee caracter en AL
   OR     AL,AH                 ;AL=0 si no se ha leido caracter
                                ;-1 para el codigo e tendido
   JZ     LAZO_CONTROL          ;No ha leido caracter trate de nuevo
   JS     TECLA_ESPECIAL        ;lee codigo e tendido
    ;No hacer nada con el caracter por ahora
   MOV    DL,AL
   CALL   EDITAR_BYTE           ; fue caracter normal, editar
   JMP    LAZO_CONTROL          ;leer otro caracter
TECLA_ESPECIAL:
   CMP    AL,+68                ;F10 salir?
   JE     FIN_CONTROL           ;Si, salga

   LEA    BX,TABLA_MASTER
LAZO_ESPECIAL:
   CMP    BYTE PTR [BX],0       ;Fin de tabla?
   JZ     NO_EN_TABLA           ;Si, tecla no esta en tabla
   CMP    AL,[BX]               ;Esta esta entrada en la tabla
   JE     CONTROL               ;Si, luego ir a control
   ADD    BX,3                  ;No, pruebe con la siguiente entrada
   JMP    LAZO_ESPECIAL         ;Revidar la siguiente entrada

CONTROL:
   INC    BX                    ;Apuntar a dirección del procedimiento
   CALL   WORD PTR [BX]         ;llamar procedimiento
   JMP    LAZO_CONTROL          ;Esperar por otra tecla

NO_EN_TABLA:
   JMP    LAZO_CONTROL

FIN_CONTROL:
    POP    DX
    POP    BX
    POP    AX
    RET
MASTER    ENDP
```

EL procedimiento EVITAR_BYTE consistirá casi enteramente en llamar otros procedimientos, esto es una característica del diseño modular. Con el diseño modular, se pueden escribir procedimientos complejos simplemente dando una lista de llamadas a otros procedimientos que hagan el trabajo. Muchos de los procedimientos que se utilizán en EDITAR_BYTE trabajan con un caracter en el registro DL, pero este ya está puesto cuando se llama a EDITAR_BYTE . la única instrucción que no es una instrucción CALL (o PUSH,POP) es la instrucción LEA para establecer la dirección de la línea de indicaciones para DESPTBP_LINEA_INDICADOR la mayoría de los procedimientos llamados en editar byte son para actualizar el despliegue cuando se edite

un byte. Se verán otros detalles de editar byte en los procedimientos que se listarán en unos momentos puesto que EDITAR_BYTE cambia un byte en la pantalla, se necesitará otro procedimiento, ESCRIBE_A_MEMORIA, para cambiar el byte en SECTOR. ESCRIBE_A_MEMORIA utilizará las coordenadas en CURSOR_FANTASMA_X y CURSOR_FANTASMA_Y para calcular el complemento a partir de SECTOR del SECTOR_FANTASMA luego escribirá el caracter byte en el registro DL a ell byte correcto dentro de SECTOR.
A continuación se mostrará el nuevo archivo EDITOR.ASM el cual contiene la versión final de EDITAR_BYTE y ESCRIBE_MEMORIA

Listado 2.51 el nuevo archivo EDITOR.ASM

```
CGROUP  GROUP  CODE_SEG, DATA_SEG
        ASSUME CS:CGROUP, DS:CGROUP
CODE_SEG        SEGMENT PUBLIC
DATA_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   CURSOR_FANTASMA_X:BYTE
        EXTRN   CURSOR_FANTASMA_Y:BYTE
DATA_SEG  ENDS
;-------------------------------------------------------------;
;Este procedimiento escribe un byte a sector en la localizacion;
;de    memoria    apuntada    por    el    cursor    Fantasma    ;
;                                                             ;
; DL byte a escribir en sector                                ;
;                                                             ;
;El complemento es calculado por                              ;
;OFFSET=SECTOR_OFFSET+(16 * CURSOR_FANTASMA_Y)+CURSOR_FANTASMA_X;
;                                                             ;
; lee: CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y, SECTOR_OFFSET    ;
; escribe: SECTOR                                             ;
;-------------------------------------------------------------;
ESCRIBE_A_MEMORIA  PROC NEAR
    PUSH  AX
    PUSH  BX
    PUSH  CX
    MOV   BX,SECTOR_OFFSET
    MOV   AL, CURSOR_FANTASMA_Y
    XOR   AH,AH
    MOV   CL,4                  ; multiplicar CURSOR_FANTASMA_Y
                                ; por 16
    SHL   AX,CL
    ADD   BX,AX                 ; BX= SECTOR_OFFSET + ( 16 * Y)
    MOV   AL,CURSOR_FANTASMA_X
    XOR   AH,AH
    ADD   BX,AX                 ; esta es la dirección
    MOV   SECTOR[BX],DL         ; ahora almacenar el byte
    POP   CX
    POP   BX
    POP   AX
```

Listado 2.5I continuación

```
    RET
ESCRIBE_A_MEMORIA  ENDP
  PUBLIC  EDITAR_BYTE
  EXTRN  SALVAR_CURSOR_REAL:NEAR, RECUPERA_CURSOR_REAL:NEAR
  EXTRN  MOVER_A_POSICION_HEXA:NEAR,MOVER_A_POSICION_ASCII:NEAR
  EXTRN   ESCRIBE_FANTASMA:NEAR, ESCRIBE_LINEA_INDICADOR:NEAR
  EXTRN   POSICION_CURSOR:NEAR, ESCRIBE_HEX:NEAR
  EXTRN   ESCRIBE_CAR:NEAR
DATA_SEG  SEGMENT  PUBLIC
  EXTRN  INDICADOR_EDITOR:BYTE
DATA_SEG  ENDS
;-- ------------------------------------------------------------------;
; este procedimiento cambia un byte en la memoria y en la           ;
; pantalla                                                          ;
;                                                                   ;
; DL byte a escribir en sector y cambiar en la pantalla             ;
;                                                                   ;
; utiliza: SALVAR_CURSOR_REAL, RECUPERAR_CURSOR_REAL                ;
;          MOVER_A_POSICION_HEXA, MOVER_A_POSICION_ASCII            ;
;          ESCRIBE_FANTASMA, ESCRIBE_LINEA_INDICADOR                ;
;          POSICION_CURSOR, ESCRIBE_HEX,ESCRIBE_CAR                 ;
;          ESCRIBE_MEMORIA                                          ;
;                                                                   ;
;lee   : INDICADOR_EDITOR,                                          ;
;   --    - ------------------------------------------------------- ;
EDITAR_BYTE  PROC NEAR
    PUSH  D
    CALL  SALVAR_CURSOR_REAL
    CALL  MOVER_A_POSICION_HEXA  ; moverse al numero hexa en la
    CALL  POSICION_CURSOR        ; ventana hexa
    CALL  ESCRIBE_HEX            ; escribe un nuevo número
    CALL  MOVER_A_POSICION_ASCII; mover al car en ventana ASCII
    CALL  ESCRIBE_CAR            ; escribe el nuevo caracter
    CALL   RECUPERA_CURSOR_REAL ; regresar  cursor  a  posición
                                 ; inicial
    CALL  ESCRIBE_FANTASMA       ; reescribir el cursor fantasma
    CALL   ESCRIBE_A_MEMORIA     ;salvar estos nuevoas  byte  en
                                 ;SECTOR
    LEA   DX, INDICADOR_EDITOR  ;
    CALL  ESCRIBE_LINEA_INDICADOR
    POP   DX
    RET
EDITAR_BYTE  ENDP
CODE_SEG  ENDS
     END
```

Si se utiliza la versión actual de DSI, y se presiona cualquier tecla se verá un cambio en el número y en el caracter bajo el cursor fantasma. La edición trabaja, pero aún no es tan segura, puesto que se puede cambiar un byte al presionar cualquier

teclado se necesita agregar algun tipo de seguridad, tales como insertar MODE para cambiar un byte; así se evitará hacer cambios no deseados.

2.23 ENTRADAS DECIMALES Y HEXADECIMALES

En esta sección se encontrarán dos nuevos procedimientos para entradas desde el teclado: Un procedimiento para leer un byte ya sea por la lectura de un número hexadecimal de dos dígitos ó un caracter individual y otro para leer una palabra para leer los caracteres de un numero decimal: estos serán los procedimientos de entrada hexadecimal y decimal.

2.23.1 entrada hexadecimal

Se comenzara reescribiendo LEE_BYTE .LEE_BYTE leería ya sea un caracter ordinario ó una tecla de función especial y retorna un byte a CONTROL. CONTROL luego llama al editor: si LEE-BYTE lee un caracter ordinario y EDITAR_BYTE modifica el byte a el cual apunta el cursor fantasma. De otra forma, CONTROL ve si se trata de una tecla de función especial en TABLA_MASTER para ver si el byte está ahí: si es así, CONTROL llama al procedimiento nombrado en la TABLA .
LEE_BYTE se cambiará de manera que no retorne el caracter que se escriba hasta que se presione la tecla ENTER. Esto se logrará utilizando la función 0Ah de la interrupción 21H del DOS que lee una cadena de caracteres. El DOS solo retornará esta cadena cuando se presione ENTER.
Para ver exactamente como los cambios afectan a LEE_BYTE, se necesita escribir un programa de prueba para probar LEE_BYTE aisladamente de manera, de que si algo extraño sucede, se sabrá que el problema esta en LEE_BYTE y no en alguna otra parte de DSI
El trabajo de escribir el procedimiento de prueba será muy simple si se utilizán unos pocos procedimientos de IBM_IO, VIDEO_IO, Y CURSOR para imprimir información para el progreso de LEE_BYTE, se utilizarán procedimientos como ESCRIBE_HEX, y ESCRIBE_DECIMAL para imprimir el código de caracter retornado y el numero de caracteres leidos. Los detalles se mostrarán a continuación:

Listado 2.52 programa de prueba PRUEBA.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
ASSUME CS:CGROUP, DS:CGROUP
CODE_SEG  SEGMENT PUBLIC
ORG  100H
    EXTRN ESCRIBE_HEX:NEAR, ESCRIBE_DECIMAL:NEAR
    EXTRN ESCRIBE_CADENA:NEAR, ENVIA_CRLF:NEAR
    EXTRN LEE_BYTE:NEAR
PRUEBA   PROC   NEAR
    LEA  DX,INDICADOR_ENTER
```

LISTADO 2.52 continuación

```
    CALL   ESCRIBE_CADENA
    CALL   LEE_BYTE
    CALL   ENVIA_CRLF
    LEA    DX, INDICADOR_CARACTER
    CALL   ESCRIBE_CADENA
    MOV    DL,AL
    CALL   ESCRIBE_HEX
    CALL   ENVIA_CRLF
    LEA    DX,INDICADOR_CARACTERES_LEIDOS
    CALL   ESCRIBE_CADENA
    MOV    DL,NUM_CAR_LEIDOS
    XOR    DH,DH
    CALL   ESCRIBE_DECIMAL
    CALL   ENVIA_CRLF
    INT    20h
PRUEBA  ENDP
CODE_SEG   ENDS
DATA_SEG    SEGMENT PUBLIC
INDICADOR_ENTER    DB 'ENTRAR CARACTERES: ',0
INDICADOR_CARACTER DB ' CODIGO DE CARCTER: ',0
INDICADOR_CARACTERES_LEIDOS DB ' NUMERO DE CARACTERES LEIDOS:',0

   ; y ahora variables finales
   PUBLIC   LINEA_ENCABEZADO_NO,DISK_DRIVE_NO,ENCABEZADO_PARTE_1
   PUBLIC   ENCABEZADO_PARTE_2,LINEA_INDICADOR_NO,SECTOR_ACTUAL_NO
LINEA_ENCABEZADO_NO DB   0
DISK_DRIVE_NO       DB   0
ENCABEZADO_PARTE_1  DB   0
ENCABEZADO_PARTE_2  DB   0
LINEA_INDICADOR_NO  DB   0
SECTOR_ACTUAL_NO    DB   0
DATA_SEG   ENDS
  END PRUEBA
```

A continuación habrá que encadenar este procedimiento con las versiones actuales de IBM_IO, VIDEO_IO,y CURSOR (colocar PRUEBA primero en la lista de LINK). Si se presiona alguna tecla de función especial PRUEBA informará que ha leido 255 caracteres. Porque? se ha colocado el -1 de AH en DL y se puso el byte superior de DX a cero, dejando DX puesto a 255 (FFFh), no 1 (1FFFh) . Antes de reescribir LEE_BYTE, hay que tener en cuenta una caracteristica de PRUEBA.ASM : su definición de variables . La definición de variables al final de PRUEBA estan incluidas solamente para satisfacer al LINKER . cuando se encadena PRUEBA con IBM_IO, VIDEO_IO y CURSOR, el encadenador busca a un grupo de variables utilizadas por IBM_IO y CURSOR . Estas variables se definieron en DSK pero puesto que no se esta encadenando DSK, se necesita redefinir estas variables en PRUEBA.ASM. Estas variables no se utilizaran ya que no se llama ningún procedimiento en VIDEO_IO y cursor que las requiera. Pero de cualquier forma esta

variables se necesitan, para satisfacer al encadenador.
Cuando se reescriba LEE_BYTE para hacepta una cadena de caracteres, no solo salvara de equivocaciones cuando se utilice DEL, si no que tambien servira para utilizar la tecla Backspare para eliminar caracteres si se cambia de parecer acerca de lo que se desea escribir. LEE_BYTE utilizara el procedimiento LEE_CADENA para leer una cadena de caracteres, ademas de este procedimiento tambien se utilizaran: CADENA_A_MAYUSCULA, CONVERTIR_DIGITO_HEX, y HEX_A_BYTE.
CADENA_A_MAYUSCULA y HEX_A_BYTE ambos trabajan con cadenas. CADENA_A_MAYUSCULA convierte todas las letras en minusculas de una cadena a mayusculas. Lo que significa que se puede escribir f3 o F3 para el número he adecimal F3h.

HEX_A_BYTE toma una cadena leida por el DOS, despues se llamara CADENA_A_MAYUSCULA, y se convertira los dos digitos de la cadena he adecimal a un número de un solo byte. HEX_A_BYTE hace uso de CONVERTIR_DIGITO_HEX para convertir cada digito he adecimal a un numero de cuatro bits.
¿Como se hara para asegurarse de que el DOS no lea mas de dos digitos he adecimales ? El DOS con la función 0Ah lee una cadena de caracteres a una area de memoria definida como esto:

```
LIMITE_NUM_CAR          DB      0
NUM_CAR_LEIDOS          DB      0
CADENA                  DB      80 DUP (0)
```

El primer byte se asegura que no se lean muchos caracteres, LIMITE_NUM_CAR le dice al DOS cuantos caracteres, a lo sumo, leer. Si se establece esto a tres, el DOS leerá hasta dos caracteres, mas el retorno del carro (el DOS siempre cuenta el retorno del carro). Cualquier caracter que se escriba despues no sera tomado en cuenta, quedara fuera, y por cada caracter e tra, el DOS sonara un beep, para indicar que se ha sobrepasado el limite. Cuando se presione la tecla Enter, el DOS establece el segundo byte, NUM_CAR_LEIDOS, a el número de caracteres que se han leido, no incluyendo el retorno del carro.
CADENA_A_MAYUSCULA, LEE_BYTE, y CONVIERTE_DIGITO_HEX, todas utilizan a NUM_CAR_LEIDOS. Por ejemplo, LEE_BYTE la utiliza para saber si se ha escrito un solo caracter o un número he adecimal de dos digitos. Si NUM_CAR_LEIDOS esta en uno, LEE_BYTE retorna un solo caracter en el registro AL. Si NUM_CAR_LEIDO esta en dos, LEE_BYTE utiliza HEX_A_BYTE Para convertir la cadena de dos digitos he adecimales a un byte.
A continuación se muestra el archivo LBD_IO.ASM, con los nuevos cuatro procedimientos:

Listado 2.53 La nueva versión de LBD_IO.ASM

```
CGROUP GROUP CODE_SEG, DATA_SEG
       ASSUME CS:CGROUP, DS:CGROUP
```

```
Listado  .5.3  continuación

CODE SEG  SEGMENT PUBLIC
  PUBLIC  CADENA_A_MAYUSCULA
;-------------------------------------------------------------;
; Este procedimiento convierte una cadena a Letras Mayusculas  ;
;                                                             ;
;    DS:DI  dirección de la cadena en el buffer               ;
;-------------------------------------------------------------;
CADENA_A_MAYUSCULA PROC NEAR
  PUSH AX
  PUSH BX
  PUSH CX
  MOV  BX,DI
  INC  BX                       ; apunta a contador de caract.
  MOV  CL,[BX]                  ;contador de caracter en segundo
  XOR  CH,CH                    ;limpia byte superior de contador
LAZO_MAYUSCULA:
  INC  BX                       ; apuntar al siguiente caracter  en
                                ;  buffer
  MOV  AL,[BX]
  CMP  AL,'a'                   ; ver si es una letra minuscula
  JB   NO_MINUSCULA             ; no lo es
  CMP  AL,'z'
  JA   NO_MINUSCULA
  ADD  AL  'A'-'a'              ; convertir a mayuscula
  MOV  [BX],AL
NO_MINUSCULA:
  LOOP  LAZO_MAYUSCULA
  POP  CX
  POP  BX
  POP  AX
  RET
CADENA_A_MAYUSCULA  ENDP

PUBLIC  CONVERTIR_DIGITO_HEX
;-------------------------------------------------------------;
; Este procedimiento convierte un caracter de ASCII (he a) a un  ;
; nibble ( 4 bits)                                            ;
;                                                             ;
;               AL caracter a convertir                       ;
; retorna:      AL  nibble                                    ;
;               CF puesta si hay error, limpia si no lo hay   ;
;-------------------------------------------------------------;

CONVERTIR_DIGITO_HEX  PROC NEAR
  CMP  AL,'0'                   ; es un digito legal?
  JB   MAL_DIGITO               ; no lo es
  CMP  AL,'9'                   ; aun no se esta seguro
  JA   PRUEBA_HEXA              ; podria ser digito he a
  SUB  AL '0'                   ; es decimal, convertirlo a
                                ; nibble
```

```
listado 2.53  continuación

  CLC                                   ; limpiar el carry no error
  RET

PROC BH_HEXA:
  CMP  AL,'A'                           ; no se esta seguro todavia
  JB   MAL_DIGITO                       ; no es hexa
  CMP  AL,'F'                           ; aun no esta seguro
  JA   MAL_DIGITO                       ; no es hexa
  SUB  AL,'A'-10                        ; es hexa convertirlo a nibble
  CLC                                   ; limpiar carry no error
  RET
MAL_DIGITO:
  STC                                   ; poner carry error
  RET
CONVERTIR_DIGITO_HEX  ENDP

  PUBLIC HEXA_A_BYTE
;-------------------------------------------------------------------;
; Este procedimiento convierte los dos caracteres en DS:DX          ;
; de hexa a un byte                                                 ;
;                                                                   ;
;  DS:DX  dirección de los dos caracteres para número hexa          ;
;                                                                   ;
;  retorna: AL byte                                                 ;
;           CF 1 si hay error, 0 si no lo hay                       ;
;  utiliza: CONVERTIR_DIGITO-HEX                                    ;
;                                                                   ;
;                                                                   ;
;-------------------------------------------------------------------;
HEXA_A_BYTE  PROC NEAR
  PUSH BX
  PUSH CX
  MOV  BX,DX                            ;poner dirección en BX para
                                        ; direccionamiento indirecto
  MOV  AL,[BX]                          ; obtener primer digito
  CALL CONVERTIR_DIGITO-HEX
  JC   MAL_HEXA                         ; digito malo si carry igual 1
  MOV  CX,4                             ; ahora multiplicar por  16
  SHL  AL,CL
  MOV  AH,AL                            ; retener una copia
  INC  BX                               ; obtener segundo digito
  MOV  AL,[BX]
  CALL CONVETIR_DIGITO_HEX
  JC   MAL_HEXA
  OR   AL,AH                            ;combinar dos nibbel
  CLC                                   ;limpiar carry
HECHO_HEXA:
  POP  CX
  POP  BX
  RET
```

Listado 2.53 continuación

```
MAL_HEXA:
   STC                                  ; poner carry hay error
   JMP  HECHO_HEXA
HEXA_A_BYTE ENDP
;  ---------------------------------------------------------------;
;Esta es una version simple de LEE_CADENA                         ;
;       DS:DX direccion de el area de la cadena                   ;
;                                                                 ;
;  ---------------------------------------------------------------;
LEE_CADENA  PROC  NEAR
   PUSH AX
   MOV  AH,0Ah                          ;llamar entrada del teclado con
                                        ; buffer
   INT  21h                             ;llamar función DOS al buffer
   POP  AX
   RET
LEE_CADENA  ENDP
PUBLIC LEE_BYTE
;  ---------------------------------------------------------------;
; Este procedimiento lee ya sea un solo caracter ASCII o un       ;
; número hexadecimal de dos digitos,esta es solo una versión de   ;
; prueba de LEE_BYTE                                              ;
;                                                                 ;
; retorna  byte en  AL codigo de caracter (a no ser AH=0)         ;
;                   AH 1 si se lee caracter ASCII                 ;
;                      0 si no se lee caracter                    ;
;                     -1 si se lee una tecla especial             ;
;                                                                 ;
; utiliza:  HEXA_A_BYTE, CADENA_A_MAYUSCULA,LEE_CADENA            ;
; lee     : ENTRADA_TECLADO,ETC...                                ;
; escribe: ENTRADA_TECLADO, ETC..                                 ;
;  ---------------------------------------------------------------;
LEE_BYTE  PROC NEAR
   PUSH DX
   MOV  LIMITE_NUM_CAR,3                ;permite solo dos caracteres
                                        ; ENTER
   LEA  DX,ENTRADA_TECLADO
   CALL LEE_CADENA
   CMP  NUM_CAR_LEIDOS,1                ;ver cuantos caracteres
   JE   ENTRADA_ASCII                   ; solo uno, tratelo como
                                        ;caracter ascii
   JB   NO_CARACTERES                   ;solamente se toco la tecla
                                        ; ENTER
   CALL   CADENA_A_MAYUSCULA            ;no,convertir cadena   a mayuscul
   LEA   DX,CARACTERES                  ;dirección  de  la  cadena   a
                                        ; convertir
   CALL HEXA_A_BYTE                     ;covertir cadena de he a abyte
   JC   NO_CARACTERES                   ;error, retornar'no lectura
                                        ; de caracteres'
   MOV  AH,1                            ;lectura de un caracter
```

Listado ..52 Continuacion.

```
LEE_LINEA_ENTRADA:
   PUSH DI
   PUSH CX
MOV_CARACTERES:
   MOV     AH,0AH                      ;poner a 'no   lectura   de
                                       ;caracteres'
   JMP  LECTURA_HECHA
ENTRADA_ASCII:
   MOV  AH,CARACTERES                  ;carga lectura de caracteres
   MOV  AH,1
   INT  LECTURA_HECHA
LEE_BYTE ENDP
CODE_SEG           ENDS

DATA_SEG           SEGMENT  PUBLIC
   PUBLIC ENTRADA TECLADO
TECLADO_BUFFER     LABEL    BYTE
LONGITUD_MAX_CAR   DB       0           ;longitud del buffer
NUM_CAR_LEIDOS     DB       0           ;numero de caracteres leidos
CARACTERES         DB       80 DUP (0);buffer  para entrada
DATA_SEG           ENDS
     END
```

Lo que sigue es recompilar LIBRARY_IO y encadenar los cuatro archivos PRUEBA, LIBRARY_IO,VIDEO_IO Y CURSOR y luego probar esta versión de LEE_BYTE.
En este punto, se tienen dos problemas con LEE_BYTE. Las teclas de función especial no se pueden leer con la función del DOS 0Ah. Simplemente el programa no trabaja. Se puede realizar una prueba presionando una tecla de función cuando se corra PRUEBA. El DOS no retorna dos bytes, con el primero puesto a cero como se podría esperar.

No se tiene una forma de leer códigos extendidos con la entrada con buffer del DOS (utilizando la función 0Ah), esta función se utilizó para que sea posible utilizar la tecla Back Space para eliminar caracteres antes de que se haya presionado la tecla ENTER, pero ahora, puesto que no se puede leer teclas de función especiales, se tendrá que escribir un procedimiento LEE CADENA propio. Por lo cual se tendrá que reemplazar la función 0Ah para asegurar que se puedan presionar teclas de función especial sin necesidad de presionar ENTER.
El otro problema con la función para entrada desde el teclado 0Ah del DOS tiene que ver con el caracter avance de línea (Line Feed). Si se presiona CONTROL-ENTER (Line Feed) después de que se escribió un caracter, y luego se presiona la tecla Back Space. Se hallará que el cursor se encuentra en la siguiente línea, sin ninguna forma de retornar a la línea anterior. La nueva versión de LIB_IO en las siguiente sección tratará el caracter Line Feed (Control-Enter) como un caracter ordinario; luego, al presionar

line feed el cursor no se moverá a la siguiente línea.

2.24 ENTRADA DESDE EL TECLADO MEJORADA

En esta sección se escribirá una nueva versión de LEE_BYTE y se colocará un error sutil dentro del programa para que, en la siguiente sección se discuta como encontrar y corregir dicho error (sugerencia: se deberá ser cuidadoso al probar todas las condiciones límites para LEEE_BYTE , cuando este anexado al programa INPUT) .

2.24.1 UNA NUEVA VERSION DE LEE_CADENA

En esta sección, LEE_CADENA utilizará un nuevo procedimiento, BACK_SPACE, para simular la función de la tecla Back Space. Cuando se presionó esta tecla, BACK_SPACE borrará el último carácter escrito, de la pantalla y de la cadena en la memoria.
En la pantalla, BACK_SPACE borrá el caracter moviendo el cursor un caracter a la izquierda, escribiendo un espacio sobre el, y luego se moverá un espacio a la derecha de nuevo .
En el buffer (región de memoria destinada para la cadena), BACK_SPACE borrará el caracter cambiando el puntero al buffer, DS:SI + BX , de esta manera se apunta al byte próximo inferior en la memoria. En otras palabras, BACK_SPACE simplemente decrementara BX. El caracter todavía estará en la memoria pero el programa no podrá verlo puesto que BX es quien determina cuantos caracteres pertenecen a la cadena .
Hay que ser cuidadoso de no borrar caracteres cuando el buffer este vacío, recordar que el area de datos reservada para la cadena se verá como esto:

```
LIMITE_NUM_CAR    DB  0
NUM_CAR_LEIDOS    DB  0
CADENA            DB  DUP(0)   ; buffer
```

El buffer para la cadena comienza en el segundo byte de esta area de datos dicho de otro modo con un offset de 2 , apartir del principio de manera que BACK_SPACE no podría borrar un caracter si BX esta puesto a 2, ya que este el inicio de el buffer para la cadena de manera que el buffer esta vacio cuando B es igual a 2.
A continuación se muestra el procedimiento BACK_SPACE el cual se deberá colocar en LIB_IO.ASM

Listado 2.54 agregar este procedimiento a LIB_IO.ASM

```
PUBLIC  BACK_SPACE
EXTRN ESCRIBE_CAR:NEAR
```

Listado 2.54 Continuación.

```
;     ---------------------------------------------------------------;
; Este procedimiento elimina caracteres, una a la vez, de el     ;
; buffer y de la pantalla cuando el buffer no esta vacio         ;
; Si el baffer esta vacio este procedimiento simplemete retorna ;
;                                                                ;
;   DS:SI + BX   Caracter mas reciente todavia en baffer         ;
;                                                                ;
; Utilizar ESCRIBE_CARB                                         ;
;          - - - ---------------------------------------------------;
BACK_SPACE  PROC  NEAR
      PUSH    AX
      PUSH    DX
      CMP     BX,2                ;Esta el buffer vacio?
      JE      FIN_BS              ;Si, lea el siguiente caracter
      DEC     BX                  ;Remover un caracter del buffer
      MOV     AH,2                ;Remover un caracter de la panta
      MOV     DL,BS
      INT     21h
      MOV     DL,20h              ;Escribir un espacio ahi
      CALL    ESCRIBE_CAR
      MOV     DL,BS               ;Regresar de nuevo
      INT     21h
FIN_BS:
      POP     DX
      POP     AX
      RET
BACK_SPACE  ENDP
```

Lo que sigue ahora es escribir una nueva versión de LEE_CADENA, a este procedimiento se le agragaran unas nuevas caracteristicas. Si se presiona la tecla Esc, LEE_CADENA limpiara la cadena del buffer y removera todos los caracteres de la pantalla.

LEE_CADENA utiliza tres tecla especiales: Backspace, Esc, y Enter. Se podria escribir los codigos ASCII para cada una de estas teclas en LEE_CADENA siempre que se necesiten, pero en lugar de eso se agragaran una pocas definiciones al inicio de IBM_IO.ASM para hacer LEE_CADENA mas legible. A continuación se muestra estas definiciones:

Listado 2.55 Adiciones a IBM_IO.ASM

```
CGROUP GROUP CODE_SEG, DATA_SEG
       ASSUME CS:CGROUP, DS:CGROUP

BS    EQU   8                     ;Caracter BackSpace
CR    EQU   13                    ;Caracter Retorno del carro
ESC   EQU   27                    ;Caracter Espacio

CODE_SEG   SEGMENT PUBLIC
```

.
.
.

A continuación se muestra LEE_CADENA. Aunque es un tanto largo, del listado se puede ver que no es muy complicado. Hay que reemplazar la vieja versión de LEE_CADENA en LIBRO_IO.ASM con esta nueva versión:

Listado 2.56 Nueva versión de LEE_CADENA en LIBRO_IO.ASM

```
        PUBLIC LEE_CADENA
        EXTRN  ESCRIBE_CAR:NEAR
;-------------------------------------------------------------------;
; Este procedimiento ejecuta una función muy similar a la           ;
; función 0Ah del DOS. Pero esta función retorna un caracter        ;
; especial si una tecla de función es presionada. ESC borrara       ;
; la entrada y comenzara de nuevo                                   ;
;                                                                   ;
;   DS:DX  dirección para el buffer. El primer byte debe de         ;
;          contener el maximo numero de caracteres a leer (mas      ;
;          uno para el Enter). El segundo byte sera utilizado       ;
;          por este procedimiento para retornar el numero de        ;
;          caracteres actualmente leidos                            ;
;           0       No se han leido caracteres                      ;
;          -1       Se ha leido un caracter especial                ;
;          otro     Se han leido nuemros                            ;
;                                                                   ;
; Utiliza:  BACK_SPACE, ESCRIBE_CAR                                 ;
;-------------------------------------------------------------------;

LEE_CADENA   PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    SI
        MOV     SI,DX                   ;Utiliza SI como registro indice
                                        ;BX para complemento
COMENZAR:
        MOV     BX,2
        MOV     AH,7                    ;Llamada a entrada si chequeo
        INT     21h                     ;para control-break y si eco
        OR      AL,AL                   ;Es un caracter ASCII extendido?
        JZ      EXTENDIDO               ;Si, lee el caracter extendido
NO_EXTENDIDO:
        CMP     AL,CR                   ;Es un retorno de carro?
        JE      FIN_ENTRADA             ;Si, la entrada esta echa
        CMP     AL,BS                   ;Es un Back_Space?
        JNE     NO_BS                   ;No lo es
        CALL    BACK_SPACE              ;Si, elimine caracter
        CMP     BL,2                    ;Esta el buffer vacio
        JE      COMENZAR                ;Si, ahora se puede leer ASCII
```

Listado 2.56 Continuacion

```
        JMP     SHORT LEE_SIG_CAR ;No, continue lectura normal
NO_ES:
        CMP     AL,ESC            ;Es un Esc, purgar buffer
        JE      PURGAR_BUFFER     ;Si, eliminar el buffer
        CMP     BL,[SI]           ;Ver si el buffer esta lleno
        JA      BUFFER_LLENO      ;Buffer esta lleno
        MOV     [SI+BX],AL        ;Si no, salvar caracter en buffer
        INC     BX                ;Apuntar siguiente byte en buffer
        PUSH    DX
        MOV     DL,AL             ;Eco de caracter en la pantalla
        CALL    ESCRIBE_CAR
        POP     DX
LEE_SIG_CAR:
        MOV     AH,7
        INT     21h
        OR      AL,AL             ;Un caracter ASCII e tendido no es
                                  ;valido cuando el buffer no esta
                                  ;vacio
        JNE     NO_EXTENDIDO      ;Caracter es valido
        MOV     AH,7
        INT     21h               ;Desechar el caracter e tendido
;-------------------------------------------------------------;
; Señala una condicion de error enviando un beep  a el despliegue;
;-------------------------------------------------------------;
BEEP:
        PUSH    DX
        MOV     DL,7              ;Suena indicador de error
        MOV     AH,2              ;
        INT     21h
        POP     DX
        JMP     SHORT LEE_SIG_CAR ;ahora leer siguiente caracter

;-------------------------------------------------------------;
; Vacia la cadena del buffer  y borra todos los caracteres    ;
; desplegados en la pantalla                                  ;
;-------------------------------------------------------------;
PURGAR_BUFFER:
        PUSH    CX
        MOV     CL,[SI]           ;Back_Space sobre el maximo numero
                                  ;de caracteres en buffer
        XOR     CH,CH
LAZO_PURGA:
        CALL    BACK_SPACE        ;Back_Space movera el cursor de
        LOOP    LAZO_PURGA        ;regreso
        POP     CX
        JMP     COMENZAR          ;Ya se puede leer e tendido ASCII
                                  ;puesto que el buffer esta vacio
```

Listado 2.56 Continuación.

```
;--------------------------------------------------------------;
; El buffer esta lleno, de manera que no se puede leer otro    ;
; caracter. Enviar un beep para alertar al usuario de esta     ;
; condición                                                    ;
;--------------------------------------------------------------;
BUFFER_LLENO:
      JMP     SHOR ERROR          ;El buffer esta lleno suene beep

;--------------------------------------------------------------;
;Lee el codigo ASCII e tendido y lo coloca en el buffer como el;
;el unico caracter, luego retorna un -1 como numero de caracter-;
;es leidos                                                     ;
;--------------------------------------------------------------;
FIN_MEDIO:
      MOV     AH,7                ;Leer un codigo ASCII e tendido
      INT     21h
      MOV     [SI+2],AL           ;Colocar solo este caracter en
      MOV     BL,0FFh             ;buffer, caracteres leidos = -1
      JMP     SHORT FIN_CADENA
;--------------------------------------------------------------;
;Salvar el contador de el numero de caracteres leidos y retornar;
;--------------------------------------------------------------;
FIN_ENTRADA:
      SUB     BL,2                ;Echo con entrada
FIN_CADENA:
      MOV     [SI+1]              ;retorna numero caracte. leidos
      POP     SI
      POP     BX
      POP     AX
      RET
LEE_CADENA    ENDP
```

Examinando a traves de este procedimiento, se puede ver que LEE_CADENA primero revisa si se ha presionado una tecla de función especial. Esto permite hacer solo cuando el buffer esta vacio. Por ejemplo, si se presiona F1 y despues se presiona la letra a, LEE_CADENA ignorara la tecla F1 y sonara el beep para decir que se ha presionado una tecla especial en tiempo equivocado. Si embargo, en este caso se puede presionar ESC para limpiar el buffer.

Si LEE_CARACTER lee un retorno de carro, coloca el número de bytes leidos en el segundo byte del area de la cadena y luego retorna. La nueva versión de LEE_BYTE lee este byte para saber cuantos caracteres acaba de leer LEE_CADENA.

Luego LEE_CADENA chequea para ver si se ha escrito un caracter back_space, si es asi se llama a BACK_SPACE para borrar un caracter. Si el buffer esta vacio (BX se pone a 2, que es el inicio de la cadena) luego se regresa al inicio en donde se puede leer una tecla especial si no simplemente se lee el siguiente caracter.

Finalmente, LEE_CADENA revisa en busca de la tecla ESC. BACK_SPACE borra caracteres solo cuando hay caracteres en el buffer, de manera que se puede limpiar el buffer llamando BACK_SPACE LIMITE_NUM_CAR veces, ya que LEE_CADENA no puede leer nunca, mas caracteres que LIMITE_NUM_CAR.

En la sección anterior se cambio LEE_BYTE de manera que no se pudiera leer teclas especiales. Ahora solo se necesita agregar unas pocas líneas para permitir que LEE_BYTE trabaje con la nueva version de ESCRIBE_CADENA, la cual puede leer teclas especiales. A continuación se muestra los cambios que hay que hacer a LEE_BYTE en IBD.ASM:

Listado 7.3 Cambios a LEE_BYTE en IBD.ASM

```
LEE_BYTE
;--------------------------------------------------------;
; Este procedimiento lee un solo caracter ASCII de un numero    ;
; hexadecimal                                                    ;
;                                                                ;
; retorna  byte en  AL codigo de caracter (a no ser AH=0)        ;
;                   AH 1 si se lee caracter ASCII                ;
;                      0 si no se lee caracter                   ;
;                      -1 si se lee una tecla especial           ;
;                                                                ;
; utiliza: HEXA_A_BYTE, CADENA_A_MAYUSCULA,LEE_CADENA            ;
; lee    : ENTRADA_TECLADO,ETC...                                ;
;--------------------------------------------------------;
LEE_BYTE  PROC NEAR
  PUSH DI
  MOV  LIMITE_NUM_CAR,3          ;permite solo dos caracteres
                                 ; ENTER
  LEA  DI,ENTRADA_TECLADO
  CALL LEE_CADENA
 CMP  NUM_CAR_LEIDOS,1           ;ver cuantos caracteres hay
  JE   ENTRADA_ASCII             ;Solo uno, tratelo como ASCII
                                 ;caracter asscii
  JB   NO_CARACTERES             ;solamente se toco la tecla
                                 ; ENTER
  CMP  BYTE PTR NUM_CAR_LEIDOS,00Fh  ;Es una tecla especial?
  JE   TECLA_ESPECIAL            ; Si

  CALL   CADENA_A_MAYUSCULA      ;no,convertir cadena  a mayuscul
  LEA    DI,CARACTERES           ;dirección de la cadena a
                                 ; convertir
  CALL HEXA_A_BYTE               ;covertir cadena de he a abyte
  JC   NO_CARACTERES             ;error, retornar'no lectura
                                 ; de caracteres'
  MOV  AH,1                      ;lectura de un caracter
LECTURA_HECHA:
  POP DI
  RET
NO_CARACTERES:
```

Listado 2.57 Continuacion.

```
   JMP     AH,AH                        ;poner a 'no   lectura   de
                                        ;caracteres'
   JMP   LECTURA_HECHA
ENTRADA_ASCII:
   MOV   AL,CARACTERES                  ;carga lectura de caracteres
   MOV   AH,1
   JMP   LECTURA_HECHA
TECLA_ESPECIAL:
   MOV   AL,CARACTERES[0]               ;Retorna codigo Scan
   MOV   AH,0FFh                        ;Indicar tecla especial con -1
   JMP   LECTURA_HECHA

LEE_BYTE ENDP

CODE_SEG            ENDS

DATA_SEG            SEGMENT   PUBLIC
ENTRADA_TECLADO     LABEL     BYTE
LIMITE_NUM_CAR      DB        0          ;longitud del buffer
NUM_CAR_LEIDOS      DB        0          ;numero de caracteres leidos
CARACTERES          DB        80 DUP (0);buffer para entrada

     END
```

El programa DS1 con la nueva versión de LEE_BYTE y de LEE_CADENA, debería de hacer el trabajo mucho mas facil, pero ahí esta un bicho (bug en inglés), como se dijo antes. Hay que tratar de hallarlo (borrar Del) y luego probar las condiciones límites para FIN_BYTE y NO_A_BYTE

2.25 EN BUSCA DE BICHOS

Si se prueba la nueva versión de DS1 con ag, el cual no es un numero hexadecimal, se notara que DS1 no hace nada cuando se presiona la tecla Enter. Puesto que ag no es un número hexadecimal, hay algo que anda mal en DS1, el programa debería, al menos borrarlo de la pantalla.

Este error es del tipo que se encuentra solo si se revisan las condiciones límites de un programa. El error en este caso no esta en una falla de LEE_BYTE, aunque apareció cuando se escribió este procedimiento. Si no que el problema esta en la forma que se escribió MASTER y EDITAR_BYTE.

EDITAR_BYTE esta diseñada para llamar ESCRIBE_LINEA_INICADOR para reescribir la línea de indicaciones y limpiar el resto de la línea. Esto removera cualquier caracter que se haya escrito anteriormente. Pero si se escribe una cadena como ag, LEE_BYTE reporta que ha leído una cadena de longitud cero, y CONTROL no llama a EDITAR_BYTE. Cual es la solución?

2.25.1 SOLUCIONANDO EL PROBLEMA EN MASTER

Una solución para este problema, sera reescribir la línea de indicaciones cada vez que LEA_BYTE reporte que ha leído una cadena de longitud cero.

Aquí están las modificaciones que se deberan hacer a MASTER (en CONTROL.ASM) para solucionar el problema:

Listado 2.58 Cambios a MASTER en CONTROL.ASM

```
CGROUP  GROUP  CODE_SEG, DATA_SEG
    ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG   SEGMENT  PUBLIC

    PUBLIC  MASTER
    EXTRN  LEE_BYTE:NEAR EDITAR_BYTE:NEAR
    EXTRN  ESCRIBE_LINEA_INDICADOR:NEAR
DATA_SEG   SEGMET PUBLIC
    EXTRN  INDICADOR_EDITOR:BYTE
DATA_SEG   ENDS

;  - ------ ------------------------------------------------;
;Este  es  el control central. Durante la  edicion  normal  y;
;durante  se  observan los sectores, este  procedimiento  lee;
;caracteres  desde el teclado y, si el caracter es una  tecla;
;de  comando (tales como las tecla de cursor).  MASTER  llama;
;los  procedimientos que realizan el trabajo  indicado.  Este;
;programa esta echo para las teclas especiales listadas en  la;
;TABLA MASTER  donde  el  procedimiento  direccionado  esta;
;almacenado e actamente despues del nombre de la tecla.      ;
; Si el caracter no es una tecla especial, entoces se  debera;
;de  colocar  en el buffer de edicion_ es decir  el  modo  de;
;edicion.                                                    ;
;                                                           ;
;Utiliza:  LEE_BYTE, EDITAR_BYTE, ESCRIBE_LINEA_INDICADOR    ;
;    lee:  INDICADOR_EDITOR                                  ;
;---- ------------------------------------------- - ------;
MASTER  PROC NEAR
  PUSH  AX
  PUSH  BX
  PUSH  DX
LAZO_CONTROL
  CALL  LEE_BYTE            ;lee caracter en AX
  OR    AH,AH               ;AX=0 si no se ha leido caracter
                            ;-1 para el codigo e tendido
  JZ    NO_CARACTER_LEIDO ;No hay caracter leido, tratar de nuevo
  JS    TECLA_ESPECIAL      ;lee codigo e tendido
  MOV   DL,AL
  CALL  EDITAR_BYTE         ;Fue caracter normal, editar byte
  JMP   LAZO_CONTROL        ;leer otro caracter
```

```
Listado   5B Continuacion

LAZO_ESPECIAL:
   CMP   AL,64                ;FIN salir?
   JE    FIN_CONTROL          ;Si, salga
   LEA   BX,TABLA_MASTER
LAZO_ESPECIAL:
   CMP   BYTE PTR [BX],0      ;Fin de tabla?
   JE    NO_EN_TABLA          ;Si  tecla no esta en tabla
   CMP   AL,[BX]              ;Esta esta entrada en la tabla
   JE    CONTROL              ;Si, luego ir a control
   ADD   BX,2                 ;No, pruebe con la siguiente entrada
   JMP   LAZO_ESPECIAL        ;Revisar la siguiente entrada
CONTROL:
   MOV   BX                   ;Apuntar a direccion del procedimiento
   CALL  WORD PTR [BX]        ;llamar procedimiento
   JMP   LAZO_CONTROL         ;Esperar por otra tecla
NO_EN_TABLA:
   JMP   LAZO_CONTROL
NO_CARACTER_LEIDO:
   LEA   DX,INDICADOR_EDITOR
   CALL  ESCRIBE_LINEA_INDICADOR ;Borra caracteres invalidos
   JMP   LAZO_CONTROL            ; Tratar de nuevo

FIN_CONTROL:
    POP  DI
    POP  BX
    POP  AX
    RET
MASTER   ENDP

CODE_SEG ENDS

DATA_SEG  SEGMENT  PUBLIC

CODE_SEG  SEGMENT PUBLIC
    EXTRN  SECTOR_PROXIMO:NEAR     ;En DISP_IO.ASM
    EXTRN  SECTOR_PREVIO:NEAR      ;En DISP_IO.ASM
CODE_SEG  ENDS

;     ---    -------- ----------------------------------------;
; Esta tabla contiene las tecla permitidas en codigo ASCII    ;
; y las direcciones de los procedimientos que se llamaran     ;
; cuando cada tecla sea presionada.
;  El formato de la tabla es
;    DB  ??             ;Codigo para cursor hacia arriba
;    DW  OFFSET CGROUP:FANTASMA_ARRIBA
;                 - ----------------------------------------;
TABLA_MASTER  LABEL BYTE
    DB  59                              ;F1
    DW  OFFSET CGROUP:SECTOR_PREVIO
    DB  60                              ;F2
```

Listado 2.98 Continuacion

```
    DW  OFFSET CGROUP:SECTOR_PROXIMO
    DB  0                                   ;Fin de tabla
DATA_SEG    ENDS
        END
```

Esta modificacion realmente no ha sido muy grande. Pero hizo que DSI sea menos elegante, y por lo general la elegancia va de la mano con la claridad. Otra solución hubiera sido hacer mas modular este programa, pero, se considera que por ser un ejemplo tipico de error, vale la pena presentarlo.

2.26 ESCRIBIENDO DE REGRESO AL DISCO LOS SECTORES MODIFICADOS

El programa DSI casi esta listo para ser usado. En esta seccion, se escribira un procedimiento para escribir el sector modificado de regreso al disco, y en la siguiente seccion, se escribira un procedimiento para mostrar la segunda mitad del sector. Aunque en ese punto este programa ya sera util, no se puede decir que el programa este terminado. Un programa nunca esta terminado, siempre se le pueden hacer mejorias.

2.26.1 ESCRIBIENDO AL DISCO

Escribir un sector moficado de regreso al disco puede ser desastroso si se hace accidentalemente. Todas las funciones del programa DSI se accionan por medio de las teclas F1, F2, F10 y las teclas de cursor. Pero cualquiera de estas teclas podria ser presionada sin intencion. Afortunadamente, esto no es posible si se utiliza una tecla de funcion combinada con la tecla de cambio (Shift). Es por esto que para escribir un sector de regreso al disco se utilizara la combinación de teclas Shift+F5. Esto impedira escribir un sector de regreso al disco, a menos que se desee hacerlo.

Los siguientes cambios se haran a CONTROL, para agragar ESCRIBE_SECTOR a la tabla:

Listado 2.99 Cambios a CONTROL.ASM

```
DATA_SEG  SEGMENT  PUBLIC

    EXTRN  SECTOR_PROXIMO:NEAR      ;En DISI_IO.ASM
    EXTRN  SECTOR_PREVIO:NEAR       ;En DISI_IO.ASM
    EXTRN  FANTASMA_ARR:NEAR, FANTASMA_ABA:NEAR
    EXTRN  FANTASMA_IZQ:NEAR, FANTASMA_DER:NEAR
    EXTRN  ESCRIBE_SECTOR:NEAR      ;En Disk_io.ASM

;  ---  --  ---------- -- ----------;
; Esta tabla contiene las tecla permitidas en codigo ASCII   ;
; y las direcciones de los procedimientos que se llamaran    ;
```

Listado 2.59 Continuación.

```
; cuando cada tecla sea presionado.                                    ;
;  El formato de la tabla es                                           ;
;    DB  72           ;Codigo para cursor hacia arriba                 ;
;    DW  OFFSET CGROUP:FANTASMA_ARRIBA                                 ;
;---  ------------------------------------------------------------------;
TABLA_MASTER  LABEL BYTE
    DB  59                                  ;F1
    DW  OFFSET CGROUP:SECTOR_PREVIO
    DB  60                                  ;F2
    DW  OFFSET CGROUP:SECTOR_PROXIMO
    DB  72                                  ;Cuersor hacia arriba
    DW  OFFSET CGROUP:FANTASMA_ARR
    DB  80                                  ;Cursor hacia abajo
    DW  OFFSET CGROUP:FANTASMA_ABA
    DB  75                                  ;Cursor hacia izquierda
    DW  OFFSET CGROUP:FANTASMA_IZQ
    DB  77                                 ;Cursor hacia la derecha
    DW  OFFSET CGROUP:FANTASMA_DER
    DB  88                                 ;Shift F5
    DW  OFFSET CGROUP:ESCRIBE_SECTOR
    DB  0
DATA_SEG    ENDS
        END
```

ESCRIBE_SECTOR es casi identico a LEE_SECTOR. El unico cambio e que se desea escribir, en lugar de leer, en sector. Mientras qu la instrucción INT 25h le indica al DOS que lee un sector, s función complementaria, INT 26h, le indica al DOS que escriba u sector a el disco. A continuación se muestra ESCRIBE_SECTOR: Ha que colocarlo en DISK_IO.ASM:

Listado 2.60 Agregar este procedimiento a DISK_IO.ASM

```
      PUBLIC ESCRIBE_SECTOR
;-------------------------------------------------------------------
; Este procedimiento escribe un sector de regreso al disco         '
;                                                                   '
; Lee:  DISK_DRIVE_NO, SECTRO_ACTULA_NO, SECTOR                     '
;-------------------------------------------------------------------
ESCRIBE_SECTOR   PROC   NEAR
      PUSH       AX
      PUSH       BX
      PUSH       CX
      PUSH       DX
      MOV        AL,DISK_DRIVE_NO      ;Numero del disk drive
      MOV        CX,1                  ;Escribe sector 1
      MOV        DX,SECTOR_ACTUAL_NO   ;Sector logico
      LEA        BX,SECTOR
      INT        26h                   ;Escribe sector a disco
      POPF                             ;Saca de stack reg. bandera
```

Listado 7.4 (continuación).

```
        PUSH        DX
        PUSH        CX
        PUSH        BX
        PUSH        AX
        RET
ESCRIBE_SECTOR    ENDP
```

Ahora habría que recompilar Control y Dsk_rw, pero todavía no se debe de probar con las teclas de función. Pero se puede hacer lo siguiente prueba: colocar un disco que no aprecie mucho en el driver A. Y poner el disco que contiene Dsk en otro driver tal como B. luego correr dsk desde el driver B (o el que se haya escogido), de manera que Dsk lee el primer sector del disco del driver A. Antes de continuar hay que asegurarse que el disco de Drive A no importe si se destruye.

Luego se cambiara un byte en el despliegue del sector y se anotara el byte que se cambio. Luego, se presiona Shift|F5. Se vera que la luz roja del driver enciende: lo que quiere decir que se ha escrito el sector modificado de regreso al disco en la driver A.

Luego si se presiona F2 para leer el sector 1, y luego F1 para leer de nuevo el sector 0 y se debera de ver el sector modificadoa. Una vez hecho esto se procedera a restaurar el byte modificado y volverlo a grabar de regreso en el disco

2.27 EL OTRO MEDIO SECTOR

Idealmente el Dsk debera de comportarse como un procesador de palabras cuando el cursor se encuentre en el fondo del despliegue del medio sector y se mueva hacia abajo: el despliegue se debera de desplazar hacia arriba una línea, y debera de aparecer una nueva línea en el fondo. Pero la vesión que se hara en estas secciones solamete desplazara grupos de 16 líneas de manera que se obeserve ya sea la primera o la segunda mitad del sector.

2.27.1 DESPLAZANDO MEDIO SECTOR

La antigua versión de FANTASMA_ARR y de FANTASMA_ABA detiene el movimiento del cursor cuando se trata de sobrepasar el fondo o el tope del despliegue. En esta sección se cambiara FANTASMA_ARR y FANTASMA_ABA de manera que se pueda llamar a DESPLAZA_ARR o DESPLAZA_ABA cuando el cursor se mueva fuera del fondo o del tope del despliegue. Estos dos procedimientos desplazaran el despliegue y colocara el cursor en su nueva posicion. A continuación se muestra la versión mofcada de FANTASMA_ARR y FANTASMA_ABA en FANTASMA.ASM

Listado 2.6 Cambios a FANTASMA.ASM

```
FANTASMA_ARR  PROC NEAR
      CALL   BORRA_FANTASMA          ;Lo borra de la posición actual
      DEC    CURSOR_FANTASMA_Y       ;Mueve cursor 1 línea arriba
      JNS    NO_TOPE                 ;Cursor no esta e tremo superior
      MOV    CURSOR_FANTASMA_Y,0     ;esta en e tremo, no se mueve
      CALL   DESPLAZA_ABA            ;Estaba en el tope, deplazar
NO_TOPE:
      CALL   ESCRIBE_FANTASMA        ;Escribir fantasma nueva posicion
      RET
FANTASMA_ARR  ENDP

      PUBLIC  FANTASMA_ABA
FANTASMA_ABA  PROC NEAR
      CALL   BORRA_FANTASMA          ;Lo borra de la posición actual
      INC    CURSOR_FANTASMA_Y       ;Mueve cursor 1 línea abajo
      CMP    CURSOR_FANTASMA_Y,16    ;Cursor no esta e tremo infer.?
      JB     NO_FONDO                ;No, escriba cursor fantasma
      MOV    CURSOR_FANTASMA_Y,15    ;Si, esta en e tremo, no se mueve
      CALL   DESPLAZA_ARR            ;Estaba en fondo, desplazar
NO_FONDO:
      CALL   ESCRIBE_FANTASMA        ;Escribir fantasma nueva posicion
      RET
FANTASMA_ABA  ENDP
```

No hay que olvidar cambiar el encabezado de comentarios de este archivo, ya que ahora se utilizan dos nuevos procedimientos

```
;--- ---------------------------------------------------------;
; Estos cuatro procedimientos mueven el cursor fantasma        ;
;                                                              ;
; Utiliza: BORRA_FANTASMA, ESCRIBE_FANTASMA                    ;
;          DESPLAZA_ARR, DESPLAZA_ABA                          ;
; Lee    : CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y                ;
; Escribe: CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y                ;
;-- -- ----- ------------------------------------------------;
```

DESPLAZA_ARR y DESPLAZA_ABA son dos procedimientos claramente simples, puesto que solo se encargarán de desplegar el otro medio sector. Por ejemplo, si se esta viendo el primer sector, y FANTASMA_ABA llama DESPLAZA_ARR, se vera la segunda mitad del sector. DESPLAZA_ARR cambia SECTOR_OFFSET a 256, que es el inicio del siguiente sector, luego mueve el cursor al inicio del despliegue del sector, a continuación se despliega el otro medio sector, y finalmente se escribe el cursor fantasma en la parte superior del despliegue.

Los detalles de los procedimientos antes mencionados se muestran en el siguiente listado:

Listado 2.6 Agregar estos procedimientos a FANTASMA.ASM

```
      EXTRN   DESP_MEDIO_SECTOR:NEAR, GOTO_XY:NEAR
```

```
DATA_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   LINEAS_DESPUES_SECTOR:BYTE
DATA_SEG        ENDS

;---------- -- ------ ---------- ------------------------------------;
;Estos dos procedimientos permite moverse entre los dos medios  ;
;sectores desplegados                                           ;
;                                                               ;
;Utilizan: ESCRIBE_FANTASMA, DESP_MEDIO_SECTOR,                 ;
;          BORRA_FANTASMA, GOTO_XY, SALVAR_CURSOR_REAL          ;
;          RECUPERA_CURSOR_REAL                                 ;
;    Lee: LINEAS_DESPUES_SECTOR                                 ;
;Escribe: SECTOR_OFFSET, CUSOR_FANTASMA_Y                       ;
;---------- -- ------ ---------- ------------------------------;
DESPLAZA_ARR    PROC    NEAR
        PUSH    DX
        CALL    BORRA_FANTASMA          ;Remover el cursor fantasma
        CALL    SALVAR_CURSOR_REAL      ;Salvar la posición de cursor
        XOR     DL,DL                   ;Pone cursor despliegue medio S
        MOV     DH,LINEAS_DESPUES_SECTOR
        ADD     DH,2
        CALL    GOTO_XY
        MOV     DX,256                  ;Despliega la segunda mitad
        MOV     SECTOR_OFFSET,DX
        CALL    DESP_MEDIO_SECTOR
        CALL    RECUPERA_CURSOR_REAL
        MOV     CURSOR_FANTASMA_Y,0 ;Al tope del segudo medio S
        CALL    ESCRIBE_FANTASMA
        POP     DX
        RET
DESPLAZA_ARR    ENDP
DESPLAZA_ABA    PROC    NEAR
        PUSH    DX
        CALL    BORRA_FANTASMA
        CALL    SALVAR_CURSOR_REAL
        XOR     DL,DL
        MOV     DH,LINEAS_DESPUES_SECTOR
        ADD     DH,2
        CALL    GOTO_XY
        SUB     DX,DX
        MOV     SECTOR_OFFSET,DX
        CALL    DESP_MEDIO_SECTOR
        CALL    RECUPERA_CURSOR_REAL
        MOV     CURSOR_FANTASMA_Y,15
        MOV     ESCRIBE_FANTASMA
        POP     DX
        RET
DESPLAZA_ABA    ENDP
```

CONCLUSIONES

Uno de los principales aspectos que se se deben tener en cuenta la hora de realizar un programa, es la tecnica que se utilizara In este trabajo se ha hecho enfasis en programar de form modular, de manera que cualquier error pueda ser dectectado cc relativa facilidad.

Aunque en los multiples programas de ejemplo de este trabajo, n se han visto todas las instrucciones del 8088, ni tampoco toda las directivas del macroassembler, la mayoria de programas e lenguaje de ensamble se pueden realizar con lo que se h estudiado. La mejor tecnica de aprendar mas, es tomar lo programas aqui desarrollados y modificarlos.

Una de las situaciones que es necesario aclarar, es que durant el desarrollo de este trabajo, algunos de los programas, qu aunque estaban bien escritos, despues de compilarlos n funcionaban correctamente. La causa estaba en errores de procesador de texto utilizado (WordStar) para escribirlos, ya qu el compilador no reconocia ciertos segmentos del programa aunqu estaban escritos. Esto se descubrio al utilizar el debug con lo programas ya compilados, rastreando el flujo de ejecucion observando la ejecucion paso a paso del programa, hasta llegar a punto en que se observo la falta de ciertas instrucciones. Lo qu se debe de hacer en un caso similar, es reescribir la parte qu no fue compilada. Al relaizar esta busqueda, es una buena idea crear un listado de cada procedimiento con sus respectiva direcciones.

BIBLIOGRAFIA

1. IAPX 88 Book, Intel, 1981

2. Norton, Peter, Guia del Programador Para El IBM PC
   Bogota, Colombia: Rei
   [1987]

3. Norton, Peter, Assembly Language Book for the IBM PC
   New York, USA: Brady
   [1986]

CAPITULO III

HARDWARE ALREDEDOR DE UN SISTEMA 286

Introducción

Debido al acelerado desarrollo de las computadoras en nuestro siglo, y tomando en cuenta el enfoque principal de esta tesis la cual es el estudio del microprocesador 8088, su arquitectura y programación a través del lenguaje assembler, es que se ha incluido en este capítulo la información del desarrollo tecnológico de los microprocesadores siendo el objetivo el de estudiar a un nivel básico los sistemas de hardware con el microprocesador 80286 que es un descendiente mas avanzado del 8088, y su interelación con otros dispositivos para formar un sistema completo de computación. Complementando de esta manera el enfoque principal de la tesis que es sobre aspectos de software del 8088 proveyéndose así una visión del desarrollo tecnológico e industria de hardware dentro de la familia 8086/8088.

3.1 Hardware con el 286 construcción de un sistema

El microprocesador 80286 es solamente la Unidad Central de Proceso (CPU), no un sistema completo de computación .Para construir un sistema completo de computación se deberá agregar dispositivos de memoria y I/O.Así como también un controlador de interrupciones para controlar y dar prioridad a determinadas interrupciones.Además si se requiere de una aritmética de punto flotante se tendrá que incluir el IC 287. Por lo tanto este capítulo describirá como conectar todos estos dispositivos al microprocesador 286.
En primer lugar se describirá como se comunica este micro con los dispositivos externos en general. Además se estudiará el sistema de tiempo de dicho micro.Después de cubrir en forma general esta información, se estudiarán dispositivos específicos. No se estudiará el funcionamiento específico de cada dispositivo sino por el contrario se hará énfasis en la interfase con el 286.
Primero se mostrará la forma como el 286 inicializa la operación de lectura con la memoria. Seguidamente se resolverá el problema de conexión con los dispositivos de I/O del 286, presentandose tres ejemplos: el primero un temporizador de intervalo programable. El segundo ejemplo será el estudio del Controlador de interrupciones programable 8259A. El ejemplo final será los dispositivos de I/O en interfase con el 286 a través del canal de memoria de alta velocidad de acceso directo (DMA).
Seguidamente se estudiará los tipos de memoria mas común utilizados en la computadoras personales así como también en sistemas de negocio hoy en día conocidas como Memoria de Acceso Aleatorio Dinámica, y su conexión con el 286. Después de estudiar

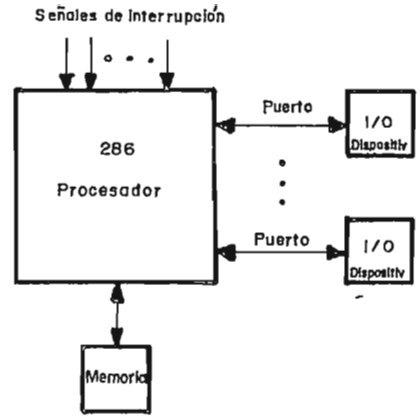


Fig 3 1 Diagrama logico de un sistema 286

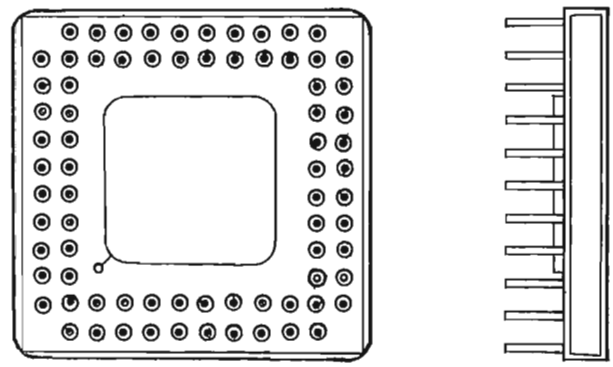


Fig 3 2 Dos vistas del pin-out del 286.

el funcionamiento de la memoria se procederá a estudiar la conexión con el 286: la interfase del IC 287 y el IC 286, concluyendo el capítulo con una descripción general de un sistema completo.

3.2 Introducción al bus 286

La figura 3.1 muestra el esquema lógico de un sistema de computación basado en un 286. El 286 es la unidad central y todos los dispositivos se unen a él por medio de líneas de comunicación separadas. En efecto esta conexión es imprácticа, porque el chip 286 solo tiene un número limitado de pines con los que interactúa con el mundo exterior (los dispositivos, ver fig. 3.2).Si el arreglo mostrado en la fig 3.1 fuera utilizado se tendría el problema de conexión con otros dispositivos; ya que el chip solo cuenta con un número determinado de pines, con lo cual se estaría limitado a conectar una cantidad específica de dispositivos(ver fig 3.3); para ello resulta mas conveniente conectar el 286 a un bus como lo muestra la fig 3.3. Un bus es un conjunto o colección de alambres en paralelo (actualmente un grupo de líneas metálicas en una tableta impresa), corriendo a través de todo el sistema, mediante el cual se pueden conectar varios dispositivos para interactuar con el microprocesador 80286. Las líneas del bus estan divididas en tres grupos: **el bus de direcciones, el bus de datos, y el bus de control.**
Cada dispositivo es asignado a una determinada dirección, o una posición de memoria o rangos de memoria. Para comunicarse con un dispositivo particular el 286 coloca un valor lógico(0 o 1) de las 24 líneas de direccionamiento del bus correspondiente al bits de la dirección del dispositivo respectivo.Así todos los dispositivos tendrán su posición en memoria, para la cual deberá de responder.
Si el 286 esta enviando información, entonces colocará en las 16 líneas del bus de datos la palabra correspondiente que se esta transmitiendo. Por otro lado si el 286 esta recibiendo información, entonces el dispositivo transmisor fijará el valor enviado en las líneas de datos, en cualquier caso, el 286 maneja las líneas del bus de control para indicar cuando se esta enviando ó recibiendo información ya sea, de un dispositivo de I/O o de una localización de memoria.

3.3 El pin-out del bus del 286

La figura 3.4 describe el pin-out del procesador 286 . El 286 coloca las direcciones en los pines A0 - A23. Para operaciones con memoria se emplea el direccionamiento físico. En el modo real, la dirección física es de solo 20 bits (en lugar de 24). Así de este modo la memoria solo necesitará los pines de A0 - A19 cuando esté trabajando solamente en el modo real. Para operaciones de entrada/salida, las direcciones serán las de los números de puertos. Como los números de puertos son de 16 bits,

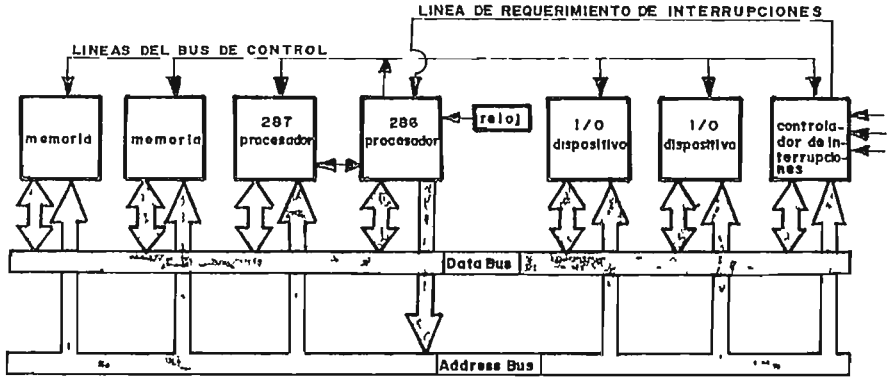


Figura 3 3 EL BUS DEL 286

los dispositivos necesitarán solo los pines del A0 - A15.
Los pines del D0 - D15 llevarán los datos que estan siendo enviados o recibidos en una transacción del bus. Una transacción en un bus es llamada **ciclo del bus.** Los cuatro pines de estado S1, S0, COD/INTA, y M/IO indicarán la naturaleza del ciclo del bus, y por lo tanto controlarán las líneas del bus.
La tabla 3.1 muestra como interpretar el estado de los pines para un ciclo de bus. La señal S0 y S1 indican si el 286 está recibiendo (lectura o escritura),enviando (escritura o salida), o haciendo algo diferente (interrupciones, paradas, o parar el trabajo . Cuando el 286 esta enviando o recibiendo, la señal M/IO indicara si la transacción es desde memoria o de dispositivos de entrada/salida. Si el microprocesador esta leyendo de memoria, COD/INTA, indicara si el 286 esta buscando (por ejemplo el código de operación) para ejecución o lectura de datos.

Tabla 3.1 Códigos de estado para un ciclo del bus

	S1	S0	M/IO	COD/INTA	CICLO DEL BUS
	0	1	0	1	ENTRADA I/O
286	0	1	1	0	LECTURA EN MEMORIA (DATOS)
RECIBIENDO	0	1	1	1	LECTURA EN MEMORIA (INSTRUCCIONES)
286	1	0	0	1	SALIDA I/O
ENVIANDO	1	0	1	0	ESCRITURA EN MEMORIA
OTROS	0	0	0	0	RECONOCIMIENTO DE INTERRUFCIONES
	0	0	1	0	PARAR Y TERMINAR *

* A1= 1 PARA PARAR, A1= 0 PARA TERMINAR

3.4 Los tiempos del ciclo del bus

Existen tres unidades utilizadas para la medida del tiempo en un sistema 286. Estas unidades son **sistema de ciclo de reloj, procesador de ciclo, y el ciclo del bus.**
La señal de tiempo fundamental en el 286 es el sistema de señal de reloj, o CLOCK. El período de repetición del sistema de señal de reloj es el ciclo de reloj. Para un sistema de reloj corriendo a 16 millones de ciclos por segundo (o 16 MHz), el ciclo del sistema de reloj será de 62.5 nanosegundos (62.5 10 - 9 segundos).

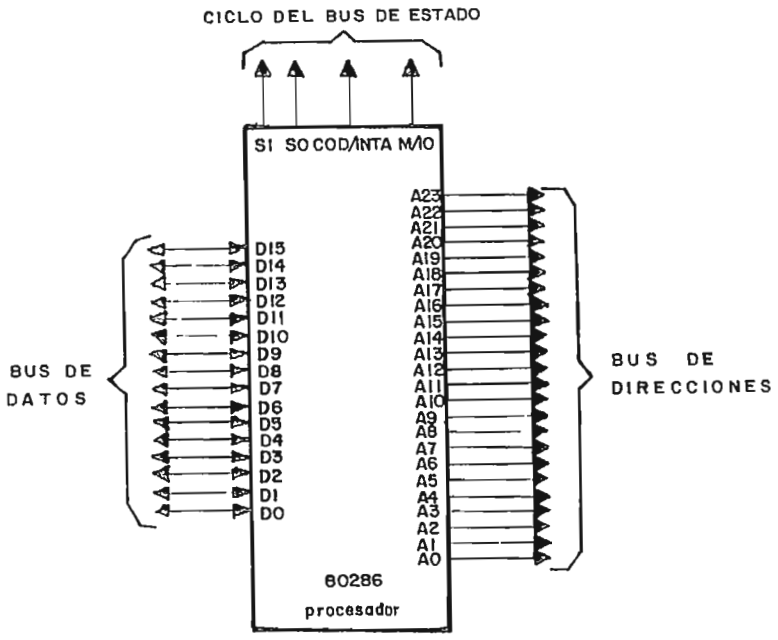


Figura 34 INTERFACE DEL BUS 286

Por lo tanto dos ciclos de sistema de reloj harán un ciclo de proceso. El primer ciclo del sistema de reloj es llamado fase 1, y el segundo es llamado fase 2 (ver fig 3.5). Así el procesador 286 correrá a la mitad de la frecuencia del sistema de reloj. Por lo tanto, un 286 corriendo a 8 MHz tendrá un sistema de reloj de 16 MHz y un ciclo de proceso de 125 nanosegundos. Cada ciclo de bus involucra dos acciones y desde aquí en adelante se asumirá dos ciclos de proceso por ciclo del procesador . Durante el primer ciclo de proceso el 286 envía el estado del bus y la información de la dirección , siendo este conocido como ciclo de estado. Los dispositivos conectados al bus deberán de responder, en este intervalo. Durante el siguiente ciclo de proceso, llamado ciclo de comandos, el 286 y los dispositivos de direccionamiento llevarán a cabo la transferencia de datos.
Algunos dispositivos (I/O o Memoria) son incapaces de responder antes del final de la parte de comandos del ciclo del bus. Una solución es disminuir el tiempo del sistema del CLOCI hasta que estos puedan responder. Para solventar este problema el 286 tiene un pin llamado READY, de esta forma si un dispositivo tiene la señal de no READY durante la parte de comando, el 286 extenderá la parte del comando, adicionando ciclos de procesos. Este ciclo adicional de proceso es llamado estado de espera (wait state), por lo tanto si un dispositivo sigue con la señal de no READY el 286 producirá estados de espera adicionales (ver fig 3.6)
Por lo tanto, en conclusión la unidad de medida mas pequeña de tiempo en un sistema 286 será el ciclo del sistema de reloj. Cada ciclo de proceso del 286 tomará exactamente dos ciclos de sistema de reloj. Cada ciclo de bus tomará como mínimo dos ciclos del procesador. De esta manera el ciclo del bus consistirá de una parte de estado, al menos para un ciclo de proceso, y una parte de comando, para uno o más ciclos de proceso. La parte de comando se extenderá mas allá del un ciclo de proceso agregando estados de espera, cuando es necesario.

3.5 Generación de pulsos de reloj

El sistema de pulsos de ciclo de reloj no es generado por el 286 sino que es generado por un chip externo generador de pulsos 82284. La figura 3.7 muestra dicha conexión al micro 286. La frecuencia de la señal del sistema generador de reloj es determinada por un cristal de cuarzo conectado al 82284.Por lo tanto para cambiar la frecuencia, solo se necesitara cambiar el cristal.
La señal de reloj del microprocesador (CLKOUT) procede del 82284, y éste de la señal de 16 o 20 MHz aplicada en su terminal de entrada EFI. En el 82284 esta señal es utilizada para sincronizar las órdenes de inicialización (RST) y alistamiento (READY) del CPU.
Además del sistema de reloj (CLOCI), el 82284 produce pulsos correspondientes al ciclo del procesador (PCLK) como un servicio al resto del sistema; así como también el monitoreo de la señal

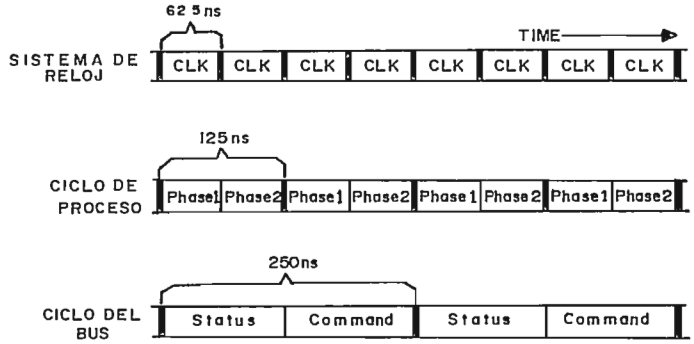


FIG. 3 5 TEMPORIZACION DEL BUS

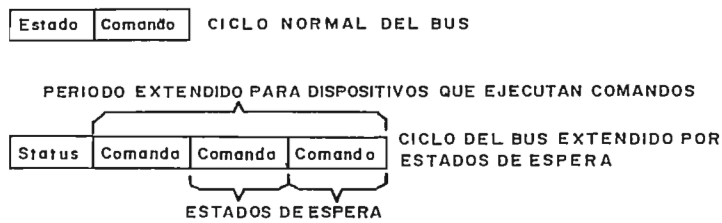


FIG 3 6 EJECUCION DE UN CICLO DE BUS CON ESTADOS DE ESPERA

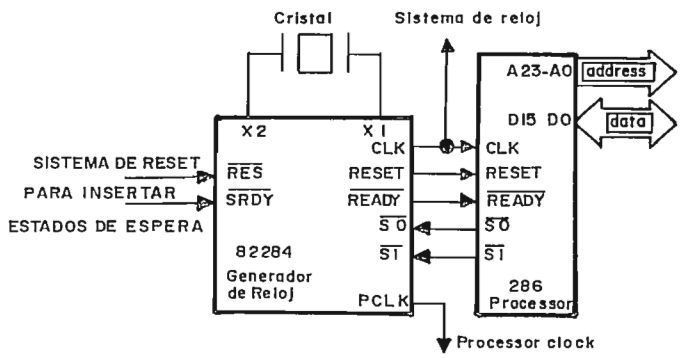


FIG 3 7 CONECCION DE UN RELOJ AL 286

de estado del bus desde el 286. Por lo tanto toda esta información es utilizada por el 82284 para mantener sincronía con el 286.

3.6 Bus de estructura segmentada

El rendimiento de un computador depende de su velocidad. Toda las otras partes son idénticas, la rapidez de tiempo de acceso memoria mas el costo por bit. El bus del 286 utiliza un artificio en el tiempo llamado estructura segmentada del bus (pipelined), con el cual se obtiene una alto rendimiento en la memorias lentas a cambio de un incremento entra en el costo de la complejidad del hardware.

En la discusión del ciclo del bus del 286 se asumió que el 286 colocaba una dirección dentro del bus de direcciones al inicio de la parte de estado de un ciclo del bus y la sostenía durante todo el ciclo del bus. Similarmente, implicaba que los datos eran colocados en el bus durante la parte de comando sosteniendolo hasta el final del ciclo del bus.En lugar de esto, se considera que los datos son colocados en el bus durante el inicio de la dirección que se tiene y que no se disprime de los datos hasta un instante después de la parte del comando del ciclo del bus. Por lo tanto estos dispositivos demandarán mas tiempo de respuesta. Examinando esto en mas detalle se observa en la figura 3.8 (a) lo que sucede actualmente en las tres porciones del bus del 286 (linea de estado, bus de direcciones, y bus de datos). El tiempo dado se asumirá de 8 MHz. Observando la figura se notará que la dirección es colocada en el bus por un sistema de ciclo de reloj (62.5 nanosegundos). similarmente, la retención de los datos es llevada a cabo por la misma cantidad.

Muchos dispositivos no trabajan si su dirección desaparece a la mitad del ciclo del bus. Pero la figura 3.8 (a) muestra lo que sucede exactamente en el bus del 286, como el ciclo anterior sostiene la dirección del próximo ciclo. Un dispositivo como un LATCH sostendra esta dirección hasta que el ciclo del bus finalice. Por lo tanto se asumirá que cada dispositivo esta conectado al bus de direcciones a través de un componente llamado latch. La figura 3.9 describe un CHIP latch ALS 573. Cuando una señal ocurre en el pin de strobe (pin c), los 8 bits en la entrada de datos D0-D7 del latch serán memorizados. El latch entonces colocará los bytes memorizados por un tiempo estable hasta que la entrada output enable (OE) sea alta.

Así, cambios en las entradas del latch no cambiarán las salidas hasta que la señal en OE sea activada de nuevo. De esta manera tres latch ALS 573 podrán memorizar todos los 24 bits del bus de direcciones, si fuera necesario.

Asumiendo que cada dispositivo posee su propio latch, y que la dirección del dispositivo es activada por el latch cuando la dirección primero aparece en el bus, se puede observar en la figura 3.8 que esta dirección es sostenida por el latch hasta que el dispositivo finaliza con el dato. En esta figura se asume que

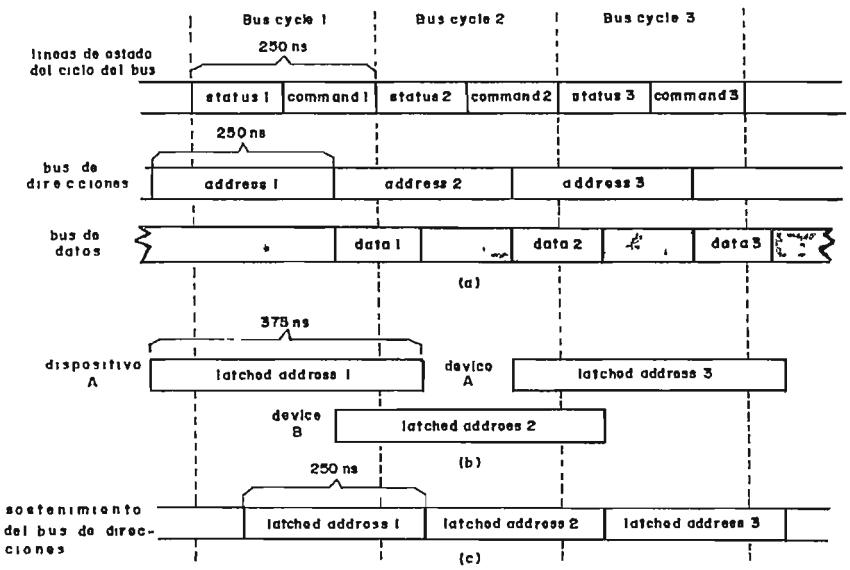


FIG 3 8 DIAGRAMA DE TIEMPOS DE LA ESTRUCTURA DE BUS SEGMENTADA

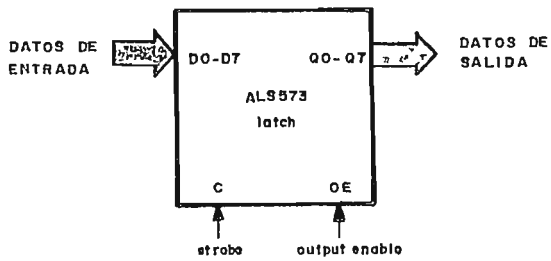


Figura 3 9 LATCH DE 8 BIT ALS573

el dispositivo A es direccionado en el primer ciclo de bus, e dispositivo B en el segundo, y el A de nuevo en el tercero. Así en lugar de contar con 250 ns (la longitud de un ciclo de bus para completar la transacción del bus, ahora cada dispositivo tendrá 375 ns. Por lo tanto se observa de que no hay necesidad de estados de espera e tras. ni de retrasos de reloj.
Esto daría la impresión de que se ha conseguido mucho para nada Además se sospecharía que se necesita una gran cantidad de latch La duda es que esto no trabaja si los dos ciclos de bus son direccionados por el mismo dispositivo, uno después de otro (observe la figura 3.8b).
En el caso de dispositivos de I/O, esto es fácil para el programador ya que se asegurará de que las instrucciones IN o OU del mismo puerto no ocurran muy cerca una de otra.
Pero para dispositivos de memoria esto debería ser una tarea de programación. Por lo tanto un circuito especial debera ser utilizado para insertar estados de espera dentro del segundo de cualquier par de ciclos de algún dispositivo de memoria, el cual será discutido en la sección correspondiente a memoria dinámica.
Por tanto la desventaja de la estructura de bus segmentada es que cada dispositivo deberá ser equipado con su propio latch.

3.7 Buffer del bus de datos

En un sistema grande, el 286 no podrá tener toda la potencia para manejar todos los componentes agregados al bus.
Este problema será aliviado direccionando el bus a través de un potente latch, como se mencionó en los parrafos anteriores.
Un transceiver (transmitter and receiver) es un amplificador bidireccional, el cual pasará señales en cualquier dirección obteniendo la señal de salida con mayor ganancia que la señal de entrada. De esta manera estos dispositivos se utilizarán para solventar los problemas de interfaz en un sistema 80286

3.8 Comandos del bus

La lógica de interfaz del bus es necesaria por dos razones:
1) la señal de los procesadores puede no ser lo suficientemente potentes para controlar al resto del sistema y 2) las señales producidas por los procesadores puede que no correspondan directamente a las señales que necesita el resto del sistema.
Observando la señal de activación del latch, la habilitación del transceptor, y la decodificación del estado del bus se observa que todos requieren un tiempo apropiado para su funcionamiento lo cual requerirá de circuitería compleja, involucrando muchos componentes. Pero en realidad, este lo hace un simple componente a saber el controlador de bus 82288. La figura 3.11 muestra como conectar el controlador de bus.
El controlador de bus 82288 es un chip encargado de realizar el

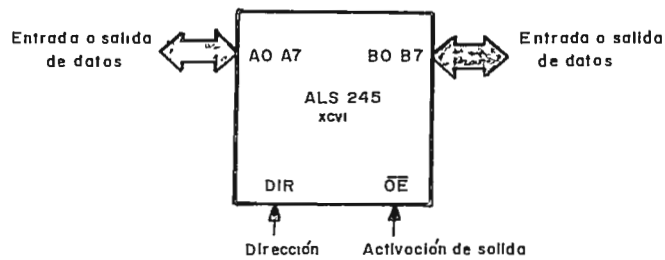


FIGURA 3 10 TRANSCEPTOR DE DATOS DE 8 BIT ALS245

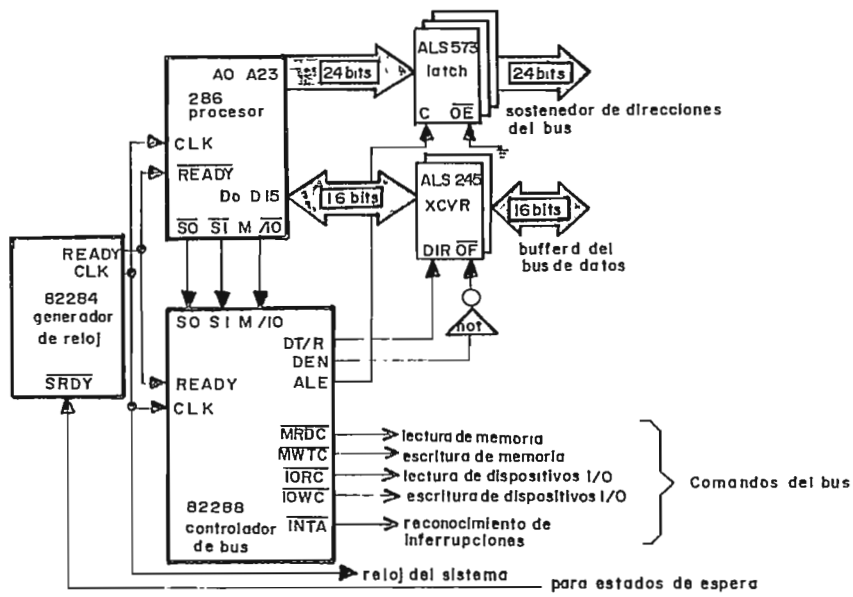


Fig 3 11 Conexion de un controlador de bus 82288

interfaz entre el procesador y el bus de control. Decodifica las señales de estado S0, S1 y M/IO y generá un conjunto completo de señales de control, como la de control de lectura en memoria (MRTC), control de escritura en memoria (MWTC), control de lectura de I/O (IORC), control de escritura de I/O (IOWC), latch de direcciones disponibles (ALE) y datos disponibles (DEN).Algunas de estas señales de control, como las de lectura y escritura tienen como destino el bus del sistema, mientras que otras, tales como las de direcciones o datos disponibles, son señales destinadas a los otros chip de interfase del procesador con los otros sub-buses (el transceptor y el latch).
La señal de activación de datos (DEN) del bus de control controlara la activación de el transceptor en el bus de datos cuando los datos esten siendo transferidos. La señal transmisor/receptor de datos (DT/R) indicara el flujo correcto de información en la transmisión de datos al bus. De esta forma los pasos para un procedimiento de lectura serán:
Por medio de la línea ALE el bus de direcciones queda enclavado en los latch de direcciones durante la porción inicial del ciclo de máquina. Esta señal es generada por la líneas de estado S0 y S1 del microprocesador.
La señal MR instruye a la memoria a colocar un byte en el bus de datos, de la localización que se encuentra presente en el bus de direcciones para que pueda ser leído por el microprocesador. De manera similar la señal MW se activará cuando el microprocesador desee escribir un byte, a través del bus de datos, en la localización de memoria especificada en el bus de direcciones.
La lectura o escritura en los puertos I/O se realiza mediante las señales IOR e IOW, de igual forma como se usan las señales MR y MW para la memoria, respectivamente. La diferencia se da en la línea M/IO del microprocesador, en donde con un estado alto se controla la transmisión o recepción de datos en la memoria (líneas MR y MW) y mediante el estado bajo en los puertos I/O (líneas de memoria y de puertos I/O, los cuales se describieron en la tabla 3.1. La señal DEN capacita los buffers para comunicar el bus de datos del microprocesador con el bus de datos del sistema. La dirección que tomen los datos en una operación de lectura o escritura es controlada por la señal DT/R (data transmit/receive) hacia los chips anteriores. Así, durante una operación de escritura (en memoria o en puertos I/O) DT/R se mantendrá en alto, y en una de lectura se conmutará al estado bajo.
La señal INTA (interrupt acknowledge) depende de la operación de la línea INTR (interrupt request) del microprocesador, la cual cambiara al estado alto cada vez que un periférico necesita comunicarse con él. Cuando esta solicitud es aceptada, el microprocesador provocará que el 82288 active la señal INTA produciendose el ingreso de un byte en el bus de datos que especificara cual de los 256 vectores de interrupción de baja memoria (0-3FF) que contiene parte de la dirección de rutina de manejo de interrupciones correspondiente al periférico que envió la solicitud de atención. La señal INTR la envia el chip controlador de interrupciones programable del cual se hablara en

la siguiente sección. De igual manera los pasos para un procedimiento de escritura serán:
1- La dirección es sustenida, la dirección de transmisión e sostenida, y el transceptor es activado.
2- La línea de comando apropiada (IOWC o MWTC) es activada.
3- El dispositivo direccionado podrá ahora captar los datos.
4- La línea de comando es puesta en bajo
5- El transceptor es desactivado.

3.9 Memoria de solo lectura

Las memorias de solo lecturas son utilizadas por el sistema de computación porque retiene los datos cuando falta la energía. Los chips de memoria de solo lectura programable (EPROM) son chip en blanco de fábrica, pero pueden ser programadas utilizando un dispositivo llamado programador PROM. Una vez que la PROM ha sido programada, esta no podrá ser borrada. Pero existe un tipo de memoria que puede ser programada y borrada conocida como EPROM, la cual posee una ventana donde se programa o borra a través de rayos ultravioleta. Una vez que ha sido borrada, esta podrá ser reprogramada eléctricamente utilizando un programador de EPROM. Un chip de memoria de acceso aleatorio no borrable (NVRAM) esta dividido en partes. Una parte es una memoria ordinaria RAM y la otra una memoria EPROM. La parte de la RAM pierde los datos cuando la potencia en la computadora es suspendida; pero la EPROM no la perdera. Una memoria NVRAM puede recibir comandos de la porción de RAM de lectura ó escritura. cuando un NVRAM recibe un comando especial STORE (ej. prioridad por perdida de potencia) el contenido de la porción de la RAM es transferido a la EPROM para la seguridad de almacenamiento. Así cuando un NVPAM recibe un comando especial RECALL (ej. despues de la restauración de la potencia) el contenido de la copia de respaldo en la EPROM es transferido a la RAM.

3.10 Inicialización de una PROM

Como un ejemplo del uso de la memoria RAM en esta sección se presentara la manera de como instalar una memoria PROM al bus 286, para almacenamiento y inicialización de un programa. Un chip EPROM 285166 es mostrado en la figura 3.12. El 285166 es una PROM de 16K porque puede alamcenar 2 14 = 16,384 bits de datos. Más específicamente, es una PROM de 2K x 8 porque los datos son organizados dentro de 2048 direcciones individuales de 8 bits cada una . La memoria PROM 285166 aceptará un direccionamiento de 11 bits (2 11=2,048) en los pines A0-A10 y salida de datos en los pines D0-D7. Los pines G1, G2, y G3 estan formadas por compuertas AND unidas para producir la señal de activación de la salida del chip.
Para el ejemplo se utilizarán dos chip PROM 285166, para proveer 2K x 16 palabras (4K bytes) de inicialización de memoria del

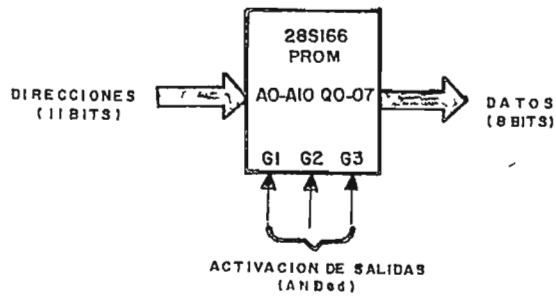


Figura 3.12 MODULO DE MEMORIA PROM DE 2K x 8 28816

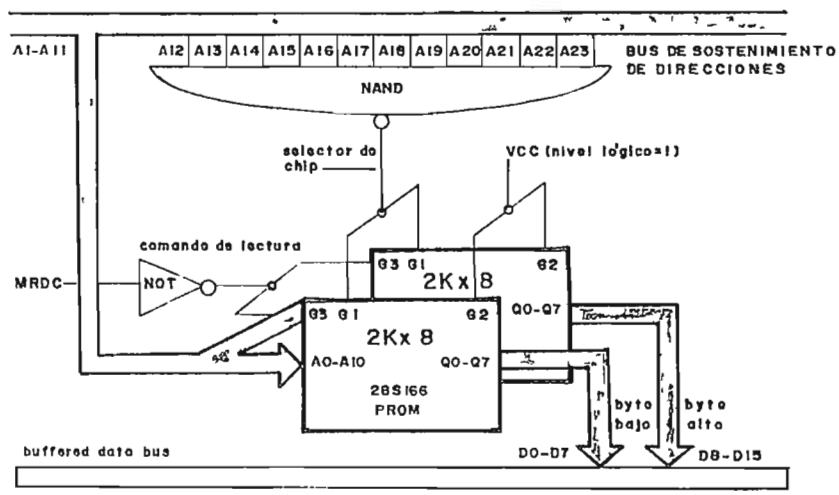


Figura 3.13 CONEXION DE 4 KBYTES DE MEMORIA PROM AL BUS

sistema. Ya que la inicialización de la memoria de solo lectura del sistema esta posicionada en la parte superior de los 16 megabytes del espacio de direccionamiento físico según requerimiento del 386 , se necesitará que la PROM responda solo a los 24 bits de direccionamiento cuyos 12 bits mas significativos son todos unos. El resto de los 11 bits será utilizado para direccionamiento interno de las memorias PROM.
La figura 3.13 muestra como instalar el chip PROM al bus. La compuerta NAND determinará si las direcciones en el bus estarán dentro del rango cubierto por las memorias PROM, por lo tanto es llamado decodificador de direcciones. Si la dirección esta dentro del rango, entonces la dirección decodificada producirá una señal llamada selector de chip.
Esta señal es operada a través de una compuerta AND a la señal de comando de lectura de memoria (MRDC) desde el controlador del bus para activar las PROMS. Así una PROM producirá los bytes de orden menor y la otra PROM producirá los bytes de orden mayor de la palabra transferida en el bus de datos.
Se podrá notar que el byte menos significante (A0) de la dirección no es utilizado. Esto es porque el 286 solo buscará palabras que comiencen con dirección par. Si una instrucción de programa del 286 lee una palabra en una dirección impar, entonces el 286 hara dos búsquedas: una a la palabra que contiene el byte impar (y un byte par inútil) y otra palabra que contiene el byte par (y un byte impar inútil). El procesador yuxtapone ambos bytes para formar la palabra, y elimina los dos restantes (ver figura 3.14) . La razon por la cual el 286 nunca buscará palabras de datos en direcciones impares es que este aprovechará hacer el interfazado simple con los dispositivos de memoria , I/O y el bus, aún así, no obstante capturando la ventaja de funcionabilidad de un bus de 16 bits sobre un bus de 8 bits en muchos casos. Por ejemplo, todas las búsquedas de instrucciones hechas por el 286, son efectuadas buscando palabras en direcciones pares. Solo los saltos a direcciones impares deberán ser beneficiosas para accesos de búsqueda a memoria complejos.

3.11 DISPOSITIVOS DE ENTRADA/SALIDA

3.11.1 Interface entre el timer de intervalos y el bus

Así como se discutió en el primer ejemplo la manera de instalar dispositivos de I/O al bus del 286, en esta sección se indicará como conectar un temporizador programable de intervalos 8254. El chip temporizador 8254 es descrito en la figura 3.15. Se observará en la figura que el chip contiene tres timers, numerados desde 0-2. los pulsos de reloj que activan el timer entran a través de los pines CLK 0, CLK 1 y CLK 2. Cada uno de los timers producirá una señal de interrupción a través de uno de los pines OUT0, OUT1, o OUT2.
El chip temporizador 8254 posee muchos modos de operación.

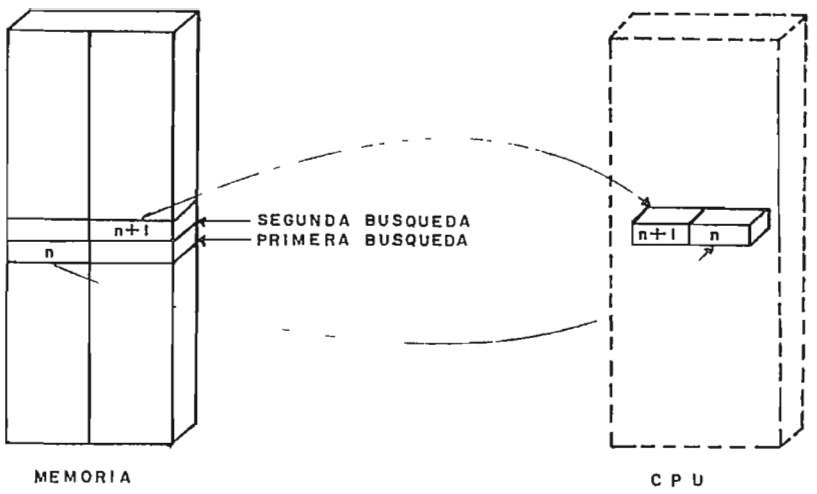
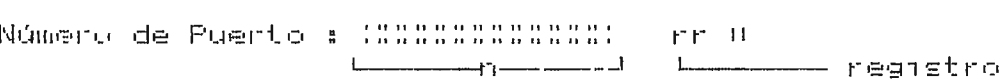


FIGURA 3 14 ACCESO A UNA PALABRA IMPAR

Para división de tiempo en aplicaciones de multitarea, el rango de generación debera ser cambiado. En este modo el 8254 periódicamente producirá señales de interrupción. La longitud del período puede ser fijado por software. No se discutirá los modos del 8254 en detalle. Para mayor información deberán de consultarse la bibliografía al final del capítulo.
El 8254 contiene un registro de control, los cuales establecen el modo, y registros que contendrán el período de los tres temporizadores. Todos los registros serán accesados a través de los pines D0-D7.
Los pines A0 y A1 seleccionarán cual de los cuatro registros serán conectados a las líneas D, como se muestra en la tabla 3.2. Los pines RD y WR determinarán la selección de los registros para lectura o escritura.
El pin CS es el selector de chip, el cual habilitará lecturas o escrituras ; por lo tanto en esta sección se conectarán cada uno de los registros a los diferentes puertos de I/O del 286, de manera que un programa pueda leer o escribir dichos registros con solo utilizar las instrucciones IN y OUT respectivamente.
De esta manera el comparador se podrá utilizar como un direccionador de memoria RAM .Así el comparador tendría la función de asignar cada posición de memoria a un circuito determinado en cada una de sus posiciones, en lugar de simplemente ponerlo en un bus de direcciones, a la vez que restringe el acceso de memoria a cada dispositivo para proteger el resto de memoria de trabajo. En concecuencia el timer seleccionaría el integrado específico que trabajaría en cada momento dando así el tiempo de respuesta apropiado.
Por ejemplo, seremos un poco vagos asignando números a los puertos como sigue:

8n	período 0
8n+2	período 1
8n+4	período 2
8n+6	registro de control

Además puesto que los números de los puertos son de 16 bits, los 13 bits de mayor orden serán n,los siguientes dos bits seleccionarán un registro y los bits de orden menor deberán de ser cero, de esta manera el formato para el número del puerto será:



Pero como se codificarán las direcciones ? Una forma sería construir un circuito externo de compuertas lógicas que reconociera el valor de n. Así la figura 3.16 describe el identificador comparador programable ALS 526, el cual podrá ser programado con cualquier número 0 de 16 bits dentro del ALS 526 idéntico a la programación de una PROM.
Si el número de 16 bits en los pines P0-P15 es igual al número

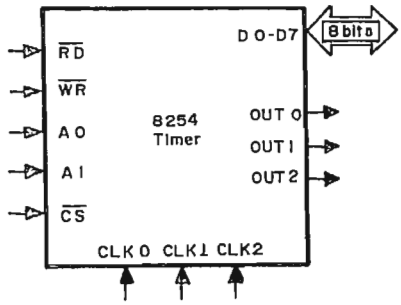
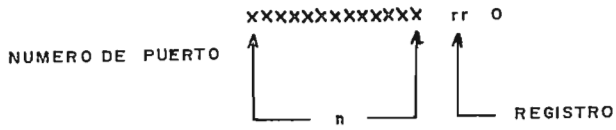


Fig 3 15 TEMPORIZADOR PROGRAMABLE DE INTERVALOS



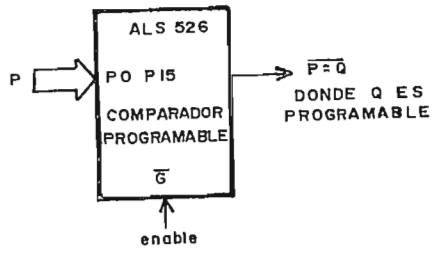
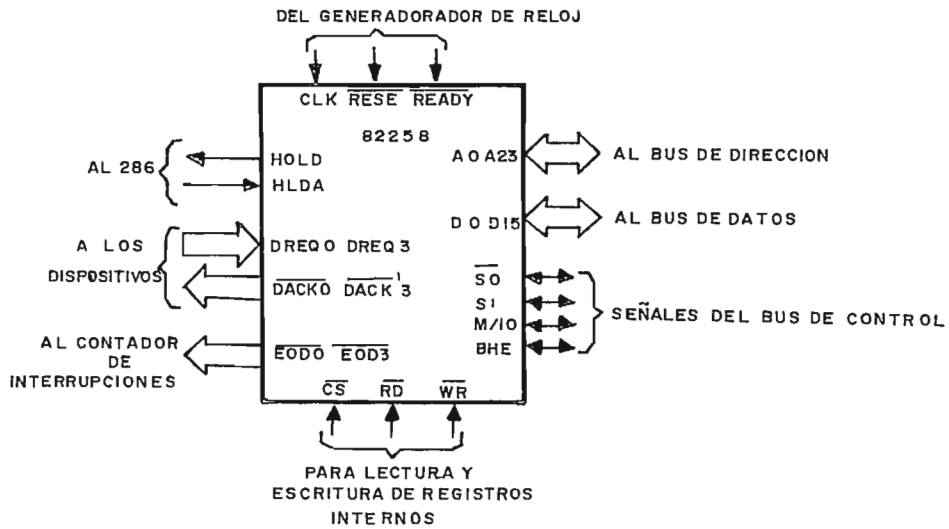


Fig 3 16 ALS 526 COMPARADOR PROGRAMABLE DE 16 BIT



Fig, 3 21 CONTROLADOR DE ACCESO DIRECTO A MEMORIA 82258

TABLA 3 2 DIRECCIONAMIENTO DE LOS REGISTROS DEL 8254

A_1	A_0	SELECTS
0	0	PERIODO 0
0	1	PERIODO 1
1	0	PERIODO 2
0	1	REGISTRO DE CONTROL

TABLA 3 3 DECODIFICACION DE ACCESO A MEMORIA

BHE	A_0	OPERACION
0	0	ACCESO A BYTE PAR
1	1	ACCESO A BYTE IMPAR
1	0	ACCESO A UNA PALABRA PAR
0	1	NADA OCURRE

programado 0, y si además el pin G es activado, entonces la salida de el chip deberá ser activada.
La figura 3.17 muestra como hacer la interfase del temporizador 8254 al bus. Los bits A1 y A2 del bus de direcciones seleccionarán el registro apropiado del 8254. Los bits A3-A15 son comparados con n,el cual se asumirá que ha sido programado dentro de los bits 03-015 dentro del comparador programable.
El bit A0 es comparado con 00, el cual se asume que ha sido programado con cero. G1, P1, y P2 no serán utilizados por lo tanto se tendrán que conectar a tierra; así como 01 y 02 deberán de ser programados a cero. En la figura 3.17 se muestra la conexión de las tres entradas de reloj a el procesador de señales de reloj PCLK (ya que no se necesita alta resolución CLK), y las señales de comando I/O de lectura y escritura (IORC y IOWC) desde el controlador del bus. Las salidas del temporizador 8254 deberán ir al controlador de interrupciones 8259A, el cual será descrito brevemente en la siguiente sección (ver referencias al final del capítulo).

3.12 E/S en mapa de memoria

Una alternativa para separar las instrucciones de E/S es utilizar las instrucciones de transferencia de datos de la memoria para comunicar con los dispositivos de E/S. Los usuarios podrían asignar un bloque de direcciones de memoria no utilizada para servir como direcciones del dispositivo. Este procedimiento denominado E/S en mapa de memoria, se suele utilizar en muchos sistemas de computadoras. Aunque este método reduce el area disponible de direcciones de memoria, puede reducir tanto los requerimientos de almacenamiento de programas como los tiempos en la ejecución de programas. Las instrucciones de memoria serían las de cargar datos (datos de mensaje de entrada o palabra de estado) y almacenar datos .
Supóngase que en la figura 3.17 las señales de comandos de memoria MRDC y MWTC se conectarán al temporizador 8254 en lugar de las señales de comando IORC y IOWC. Además supongase que no hay memorias en la direcciones físicas que corresponden a las localizaciones de los puertos de manera que dos dispositivos no respondan simultáneamente.Entonces aunque el 286 manejará la memoria de lectura/escritura en el bus idéntico a los dispositivos de I/O de lectura/escritura, los registros del 8254 serán considerados como localizaciones de memoria.
De esta forma, al leer la memoria por medio de un programa se estarían leyendo los registros del 8254, y al escribir en memoria se estarían escribiendo en los registros, siendo este método llamado E/S en mapa de memoria.
La ventaja del E/S en mapeo de memoria sobre el E/S ordinario es que se podrá utilizar las facilidades de traslación en modo virtual del 286 para poder colocar cada dispositivo mapeado en la memoria I/O dentro de su propio segmento. Entonces cada dispositivo podrá ser protegido separadamente, y aquellos

programas que necesiten manejar dispositivos sólo se les permitirá acceder los dispositivos para los cuales tienen control y no a todos ellos.La desventaja del mapeo de E/S sobre el ordinario es que algunas veces se necesita de mucho hardware decodificador de direcciones; ya que el número de puertos es de 16 bits, el circuito en la figura 3.17 solamente se observan líneas de direcciones A0-A15. Puesto que las direcciones físicas son de 24 bits, y el temporizador mapea la memoria, esto implica que aparezcan muchas direcciones físicas diferentes, a menos que el hardware sea agregado para checar los 8 bits e tra.

3.13 Conectando un controlador de interrupciones

El 286 posee dos pines de interrupción :NMI(reconocimiento de interrupción no enmascarable) y el INTR(requerimiento de interrupción).
Una señal NMI causa que el 286 invoque la interrupción de sostenimiento para la interrupción número 2. La señal INTR a través de dispositivos e ternos transmite un número de interrupcion de 1 byte al 286 vía el bus. En ésta forma, el 286 podrá sostener 256 tipos diferentes de interrupciones sin requerir de 256 pines de requerimiento de interrupción.

Típicamente, el pin de INTR es manejado por un chip controlador de interrupciones programable 8259A, en lugar de ser manejado directamente por dispositivos de interrupción.
El propósito de éste chip es el manejar los requerimientos de interrupciones de una multitud de dispositivos, y conducir las peticiones dentro del 286, una a la vez.
El 8259A acepta señales de peticiones con prioridades de interrupciones desde dispositivos e ternos(incluyendo otros 8259A) hacia el 286 a través del pin INTR.

Cuando se ha completado una instrucción el 286, finaliza la instrucción actualmente en ejecución, para luego inicializar dos ciclos de reconocimiento de interrupcion del bus (INTA).
El primer INTA del ciclo de bus informara al 8259A que el 286 ha reconocido el requerimiento de interrupcion, así como también la comunicación a través del tiempo con otros 8259A conectados a éste. Durante el segundo INTA de ciclo de bus, el 8259A enviará un número de interrupción de 8 bit al 286 a través del bus de datos.
El 286 entonces invocara el correspondiente tratamiento de interrupción.
La figura 3.18 muestra un controlador de interrupciones 8259A conectado dentro de un sistema 286.
Los 8 pines IR0-IR7 son los pines de requerimiento de interrupciones del 286. Así, un total de 8 clases de interrupciones (ocho diferentes tipos de interrupciones) podrán ser procesados por un único controlador 8259A, por lo tanto se necesitarán múltiples controladores 8259A, si existieran más de ocho tipos de interrupciones.
El 8259A requiere dos puertos de I/O para permitir al 286

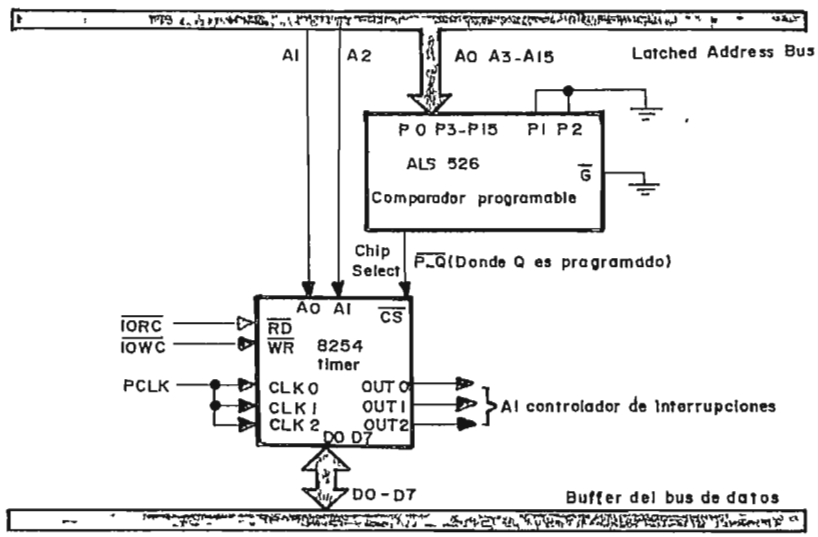


Fig. 3 17 Conección del Timer 8254 al bus

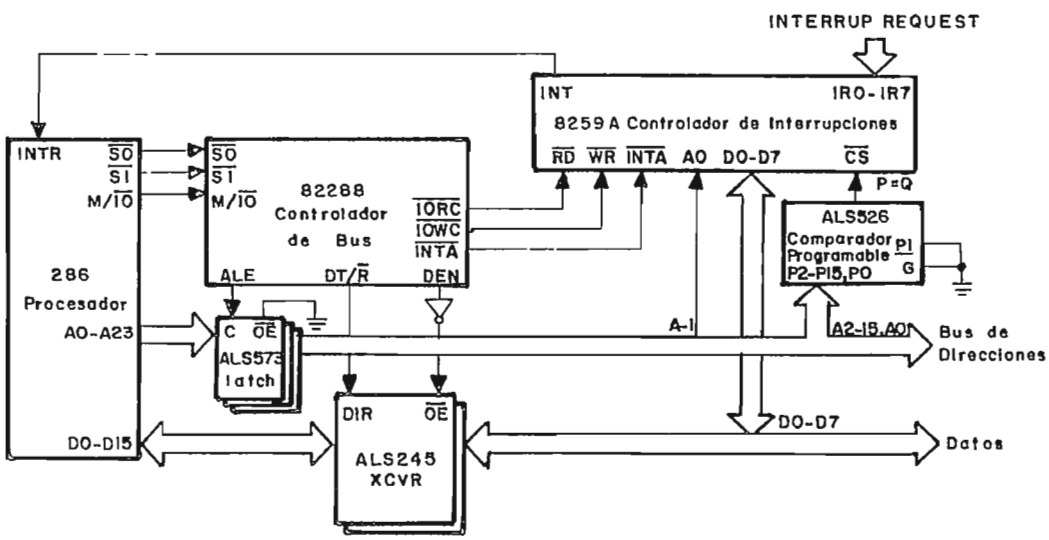


Figura 318 CONEXION UN CONTROLADOR DE INTERRUPCIONES 8259 AL 286

programas de lectura y para fijarlo en varios modos, acceso de registros internos y señales de finalización de la ejecución de un manejador de interrupción. El pin Ao especificará cual de los dos puertos serán direccionados, y los pines RD,WR,CS y D serán nombrados precisamente similares a los pines en el temporizador 8254. (Para mayor detalle consultar la bibliografía al final del capítulo).

3.14 Controladores de interrupción en cascada

Para poder procesar más de ocho tipos de interrupciones, se deberá utilizar más de un controlador.
Solamente un controlador de interrupciones, llamado el master, debería ser conectado directamente al pin INTR del 286. Los demás controladores llamados esclavos, deberán conectar sus pines de requerimiento de interrupción a los demás pines del master.
Por ejemplo, la figura 3.19 muestra como conectar cinco controladores de interrupción para poder procesar 36 diferentes tipos de interrupción.
Cuatro de los dispositivos de interrupción irán directamente al master, mientras el resto de los 32 interrupciones primero deberían ir a través de los esclavos, y luego al master; en ésta figura han sido omitidas las líneas de los cinco controladores al bus del 286. Las líneas CAS0, CAS1, y CAS2 son utilizadas para coordinar más de un 8259 .
La señal en el pin SP/EN en el chip controlador indicara si éste es el maestro o esclavo.
Adicionalmente, el master utiliza un registro interno para memorizar, cual de las líneas de requerimiento de interrupción será conectada al esclavo y cual será conectada directamente al dispositivo.
Cada esclavo sostiene dentro del registro interno el número del pin IR en el cual el master ha sido conectado.
Supongase un dispositivo requiriendo una interrupción a través de uno de los esclavos. El esclavo entonces solicitará la interrupción al 286 a través del master . Así el 286 responderá con dos ciclos de bus de reconocimiento de interrupcion (INTA).
Durante el primer INTA del ciclo del bus el master confirmará cual esclavo está solicitando la interrupción enviando a los pines IR de los esclavos , el número en la línea de cascada (CAS ,vea la fig.3.19), ya que éstas líneas conectarán todos los esclavos al master. Esta confirmación será necesaria puesto que muchos esclavos podrán requerir simultáneamente las interrupciones.
Durante el segundo INTA del ciclo de bus, el esclavo colocará el numero de interrupcion confirmando el dispositivo interrumpido en el bus de datos para que pueda ser leído por el 286.
El 286 entonces responderá por la invocación del tipo de interrupción apropiado, como en el caso de un único controlador de interrupciones.

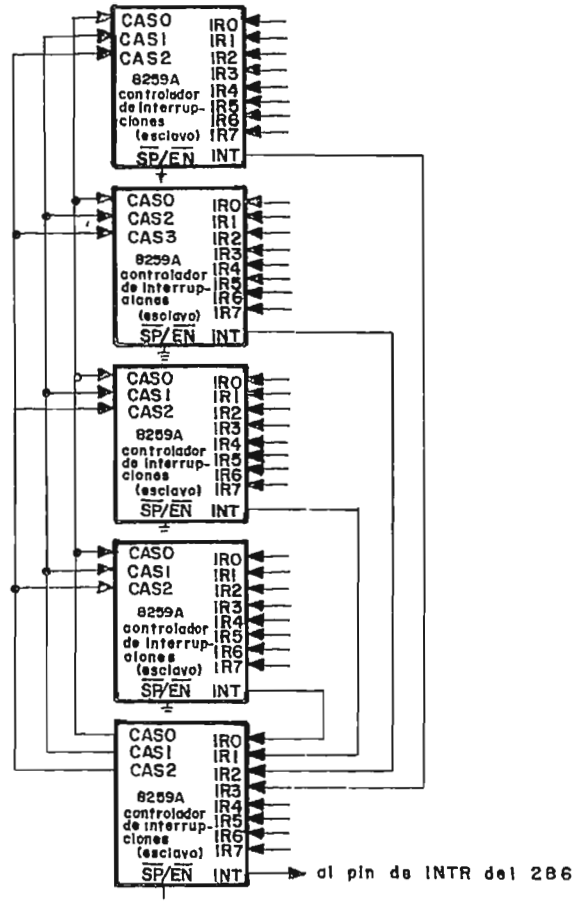


Figura 3 19 CONTROLADOR DE INTERRUPCIONES EN CASCADA

3.15 Desventajas del temporizador

El chip temporizador 8254 y el chip controlador de interrupciones 8259A fueron desarrollados antes de que el 286 fuera diseñado. Desafortunadamente, estos chip no pudieron sostenerse con un 286 de 8MHz (sistema de reloj de 16MHz). Por lo tanto, para obtener una operación segura, de los circuitos de las figuras 3.17 y 3.18 se deberá reducir la frecuencia del sistema de reloj a 10 MHz. Una solución sería, agregar al sistema un circuito extra para poder retener los 16 MHz: a través de la adición de estados de espera y retrazos, en aquellos ciclos del bus donde el 8254 o el 8259A fueran activados. Estos circuitos más complicados podrán ser encontrados en las referencias al final del capítulo.

3.16 Acceso directo a memoria (DMA)

La forma más rápida para transferir datos entre un dispositivo de I/O y la memoria vía el 286 será utilizando las interrupciones INS y OUTS. Por ejemplo, el siguiente programa leerá 2048 palabras desde un dispositivo de I/O (por ejemplo un disk drive) dentro de la memoria.

```
MOV DX, portnum; portnum es el número del dispositivo.
MOV CX, 2048    ; 2048 palabras= 4096 bytes
LES DI, memva   ; memva es la dirección virtual de destino
REP INSW        ; ésto hace el trabajo (w=word)
```

Cada palabra es primero leída dentro del 286 vía un ciclo de entrada del bus, luego escrita en memoria vía un ciclo de lectura del bus. Ya que los ciclos del bus requieren de dos procesos de reloj de longitud y dos ciclos de bus son requeridos para la transferencia, los datos para un 286 de 8 MHz se transferirán a :

$$\frac{(8 \text{ megaciclos/segundo})(2 \text{ bytes/palabra})}{4 \text{ ciclos/ palabra}} = 4\text{Mbyte/segundo}$$

Esto será cierto si se transfiere una palabra desde un dispositivo de I/O hacia memoria en único ciclo de bus, en lugar de dos ciclos. Entonces se podrá ejecutar una transferencia de datos de 8 billones de bytes por segundo lo cual sería la máxima velocidad del bus.
La mayor velocidad deberá ser particularmente estimable en sistemas con memoria virtual, al cual se transfieren largos segmentos desde el disco.
Por lo tanto es necesario un canal directo hacia memoria uno en el cual no se necesitará pasar por el 286.
A fin de conseguir éste canal (DMA) de memoria a acceso directo, el 286 posee un pin de entrada HOLD y un pin de salida HLDA. Así cuando un dispositivo accesa la línea HOLD, el 286 completará el ciclo de bus en ejecución, el control del bus y activará la señal HLDA para luego cederle el control al dispositivo solicitante.

El dispositivo que solicita la señal HOLD, completará el ciclo de bus, hasta que otro dispositivo accese la señal HOLD.
En ésta forma el 386 abandonará totalmente el bus; permitiendo así , al dispositivo que lo solicitó, la generación de señales del estado del bus, manejo de las líneas de direcciones y reconocimiento de estados de espera. Esto dará al dispositivo la habilidad para manejar el bus de datos durante los ciclos de escritura de memoria (transferencia directa a memoria) y monitoreo del bus de datos durante un ciclo de lectura de memoria (transferencia directa desde memoria). obviamente, ésto requerirá complejamente una pérdida de la circuitería para tomar toda la responsabilidad para el bus. Así en lugar de incorporar toda ésta circuitería dentro de cada dispositivo que deseará ejecutar el acceso directo a memoria, la lógica necesaria ha sido contenida en un solo chip, el controlador de acceso directo a memoria 82C59 DMA.
El 82C58 contiene 4 canales DMA, por lo tanto 4 dispositivos podrán ser habilitados para manejar el DMA. Así la función del canal DMA es producir un movimiento de referencias a memoria, por lo tanto la responsabilidad de los canales es ejecutar de manera correcta la referenciación a la localización adecuada (escritura o lectura de memoria). Así una capacidad de canal de acceso directo a memoria permitirá mayores velocidades de transferencia de datos, sin incluir a la CPU.
Los dispositivos de I/O conectados a cada canal, solo serán responsables de introducir los datos en el bus y de recibirlos cuando hayan sido enviados por el canal.
Un canal DMA producirá la demanda de movimiento de referencias a través de la ejecución de un programa de canal el cual obtendrá de memoria (ver fig 3.20) un programa de canal,el cual consistirá de una secuencia de bloques de comandos. Cada bloque especificará los diferentes tipos de operación de memoria (lectura, escritura) para luego ser llevados fuera por el canal DMA.
En la fig 3.20 el último bloc del programa de canal especifica el comando alto (STOP).
Cada bloque de comandos contiene espacio para la dirección física de origen o destino para la transferencia (típicamente solo una de ellas es utilizada); así como también el número de bytes que serán transferidos. Por lo tanto un único programa de canal podrá especificar una transferencia involucrando muchas áreas diferentes de memoria. Esta habilidad de dispersar la entrada de datos por toda la memoria y de recoger datos por las salidas es conocida como separación en la lectura (ó capacidad de una computadora para distribuir datos en varias zonas de memoria a medida que se producen en el sistema procedentes de disco o cinta magnética) y unión en la escritura (para una mejor explicación se deberá consultar la bibliografía de éste capitulo.)

El controlador DMA contiene un número de registros internos, para cada uno de los 4 canales. Para inicializar una transferencia por el DMA, el 286 utiliza un programa; el cual primero cargará el registro apuntador de comando, del canal

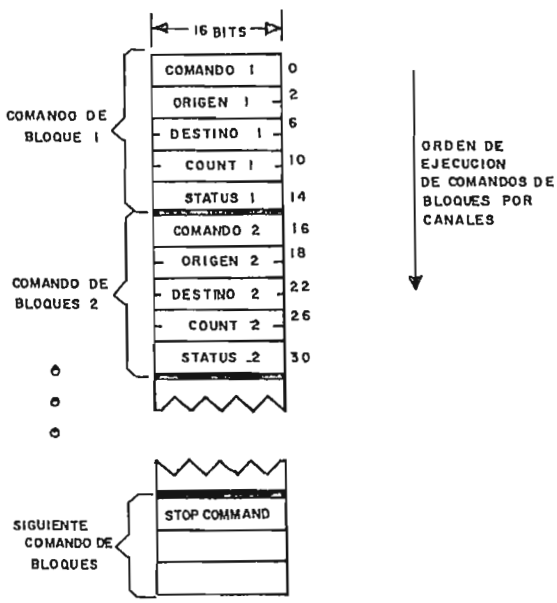


FIGURA 3.20 PROGRAMA DE CANAL

deseado; el cual a su vez inicializa el direccionamiento de un programa de canal.
El programa del 286 podrá entonces modificar el dispositivo del I/O conectado al canal para comenzar la transferencia. Después de dar ésta notificación, el 286 podrá activarse hacia otras tareas; así el controlador DMA interrumpirá al 286 cuando el programa haya sido completado.
La figura 3.22 describe el chip 82C50 controlador DMA (para mayor información consultar Microprocessor an Peripheral Handbook Vol 1, INTEL).
El chip posee los pines DREQ (requerimiento de DMA) DACK (reconocimiento DMA) y EOP (Fin de DMA) para cada canal. Cuando un dispositivo de I/O está listo para transferir una palabra, la señal es canalizada por todo el DREQ correspondiente. El canal entonces indicara que está listo para la transferencia por la señalización de retorno en la entrada DACK.
El dispositivo de I/O leera encontonces el bus de datos, si es una salida y manejará el bus de datos, si es una entrada.
Sucesivamente los DREQ accesaran palabras sucesivas de memoria, como han sido especificados en el programa de canal. Los dispositivos de I/O no necesitaran interesarse de donde la memoria esté trayendo o llevando datos, ya que todas las direcciones serán sostenidas por el controlador DMA, en el programa después de los comandos del canal. Cuando el programa del canal encuentra el final, el controlador DMA solicitará una interrupción a través del pin apropiado EOP.
La figura 3.23 muestra como conectar un controlador DMA 82258 a un sistema 286. Por simplicidad solo un canal ha sido utilizado (para mayor aplicación ver Microprocessor and Peripheral Handbook, VOL 1). El generador de reloj y el controlador de interrupciones han sido omitidos del diagrama para evitar confusiones.
Las líneas HOLD, HLDA, DREQ, DACK y EOP en la figura 3.22 fueron las discutidas en los puntos anteriores. La línea BHE será discutida en la próxima sección.
Note que la línea HLDA es utilizada para inhabilitar el transceptor del bus de datos cuando el 286 le confiere el control del bus. De esta forma el controlador DMA manejará las líneas de estado del bus S0,S1, M/IO y las líneas de direcciones A0-A23 una vez pase el control al bus. En la figura 3.22 se ha conèctado el controlador DMA de manera que los registros internos puedan ser accesados utilizando el sistema de I/S en mapeo de memoria. De esta manera los pines RD y WR del controlador DMA, tendrán que ser conectados a las líneas de comandos del bus MRDC y MWTC.
Los líneas de direcciones A0-A7 serán utilizadas para direccionar varios registros del DMA. El resto de líneas de direcciones serab decodificadas dentro del chip selector por el comparador programable.
El controlador de dispositivos periféricos (disco, cinta, etc..) se deberá de conectar a las líneas DREQ y DACK así como también, al bus de datos. Además las líneas MRDC y MWTC indicaran si la trasnferencia del DMA es una entrada (escritura de memoria) o

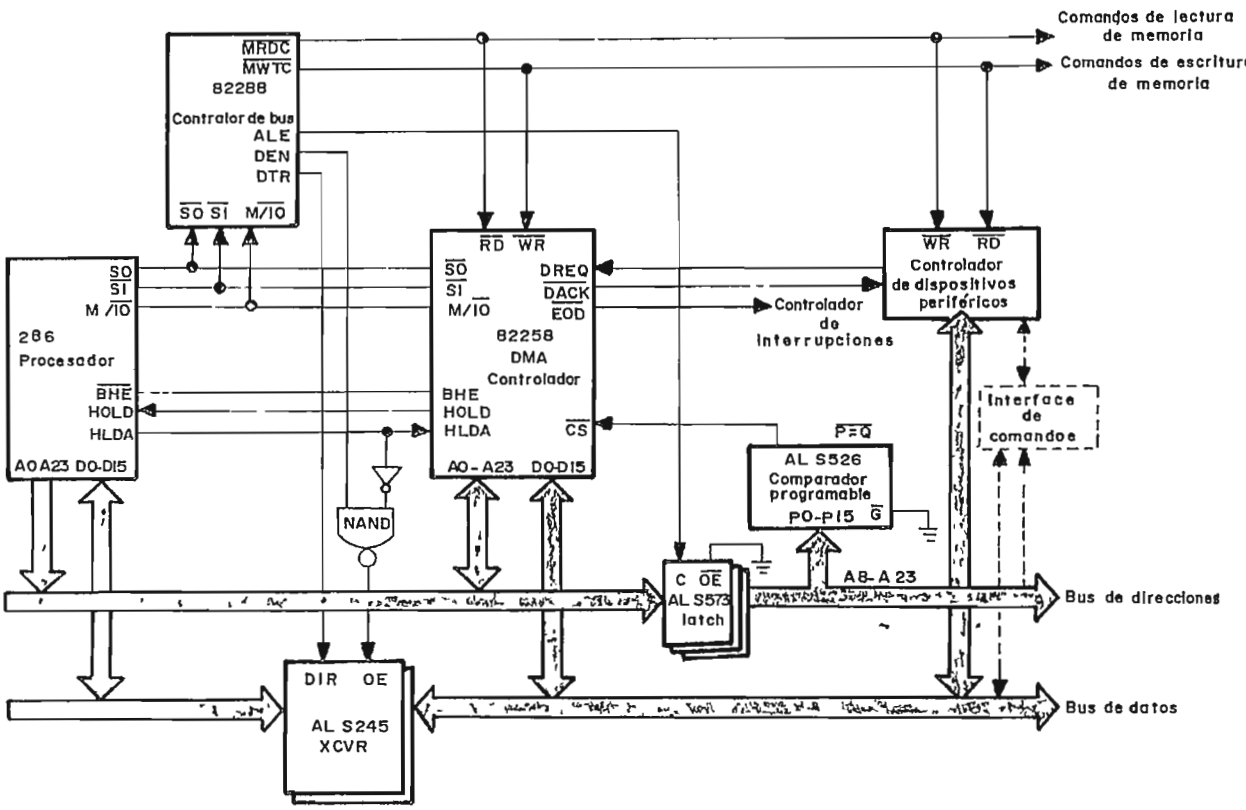


Fig 3 22 Conección de un Controlador DMA 82258

una salida (lectura de memoria).
Además muchos controladores de dispositivos que utilizan DMA probablemente también requieran interfase I/O convencional para comandos (buscar, , start I/O) y estado (fin de pagina, error de disco), de esta manera esta interfase es identificada por líneas discontinuas en la figura 3.22.

3.17 Memoria dinámica de acceso aleatorio

La memoria principal de escritura de un sistema de computación, consiste de un arreglo de chips de memorias de acceso aleatorio RAM. Existen dos tipos de memorias RAM: la RAM estática (SRAM) y la RAM dinámica (DRAM).
Cada celda de memoria de un chip RAM estática retendrá los datos indefinidamente, hasta que la fuente que la alimenta sea interrumpida. Por otro lado, cada celda de memoria de un chip de RAM dinamica podrá ser accesada una sola vez por un mínimo de cuatro milisegundos, sin pérdida de los datos. Esto es debido a que las celdas de las memorias DRAM están formadas por capacitores microscópicos, los cuales al cargarse alamcenarán un uno y un cero al descargarse. Además estos capacitores deberán ser recargados durante un proceso de lectura/escritura. Por lo tanto las RAM dinámicas requieren circuiteria de control e tra para acceso periódico o refrescamiento de cada celda de memoria. Además las RAM dinámicas son mas lentas que las RAM estáticas. Esto es inconveniente por el hecho de que el chip DRAM requiere un periodo de descanso, llamado tiempo de precarga, entre accesos sucesivos al chip de memoria,para recargar varios capacitores descargados durante el acceso. Pero apesar de todas estas desventajas muchos sistemas de computadoras utilizan el sistema de DRAMS en lugar de RAM estática para su memoria principal. Tambien, un simple chip DRAM puede almacenar muchos mas datos que un unico chip SRAM.

3.171 Escritura de bytes en memoria

Antes de mostrar como conectar una DRAM al 286, primero se proveerá algunos detalles concernientes al bus del 286. Muchas instrucciones del 286 leen o escriben un único byte en memoria, en lugar de una palabra completa. La lectura no causara ningún problema aunque el 286 tenga que buscar una palabra e ignorar la mitad de los bits. Pero esta ventaja no tiene ningún efecto en la escritura. La solución es organizar la memoria en dos bancos, un banco para todos los bytes de dirección impar y el otro para todas las direcciones de dirección par (ver figura 3.23).
De esta manera el 286 producirá una señal llamada activación alta del bus (BHE) siempre que se demande una transferencia de dirección de byte impar.Ahora se considerarán varios casos de transferencia de direcciones par e impar de bytes y palabras. Cuando el 286 transfiere una dirección de un byte impar el micro

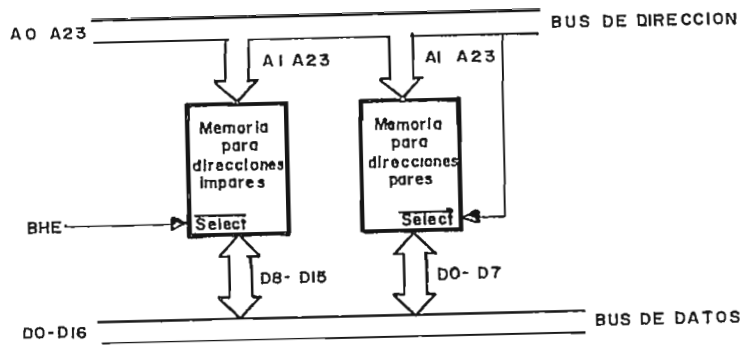


FIG 3 23 DIRECCIONAMIENTO DE BYTES EN UNA MEMORIA DEL 286

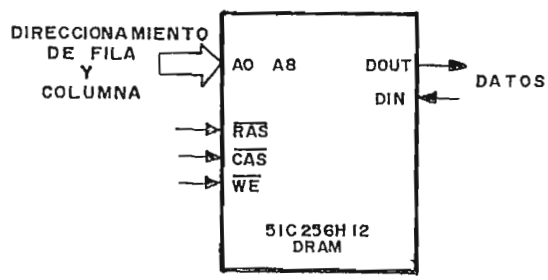


FIG 3 24 DIRECCIONAMIENTO DE FILA Y COLUMNA

fijo BHE colocando la dirección impar en el bus. En caso contrario cuando se transfiera una dirección de byte par esto desactiva BHE y colocara la dirección par en el bus. La concurrencia de esto es que el banco de dirección impar participa en un ciclo de bus sí y solo sí la dirección es impar por ejemplo, sí y solo sí el bit menos significante de el bus de direcciones (A0) es uno. La interpretación de BHE y A0 se resume en la tabla 3.3 .
Nota: de la figura 3.23 que todos los contenidos en las direcciones de bytes par, serán transferidas en las líneas D0 D7 del bus de datos, mientras que todos los contenidos en las direcciones de bytes impar serán transferidas en las líneas D8-D15 respectivamente. Las mismas reglas deberán de ser aplicadas a los puertos de I/O. Esto es debido a que en las secciones previas, se utilizó puertos de I/O pares y conexiones de los dispositivos periféricos en D0-D7.

3.172 Dentro del chip DRAM

La figura 3.24 muestra un chip DRAM 51C256 H-12 ,este es un chip DRAM de 256 K x 1 lo cual significa sostener 2 18= 262 144 bits de direcciones individuales de memoria. Las entradas del chip son de direcciones de 9 bits en dos partes. Los primeros 9 bits de la dirección serán locados en los pines A0-A8. Estos bits son llamados dirección de la fila y serán sostenidos dentro del chip a través de una señal de selección en el pin RAS (selección de la dirección de la fila). El resto de los 9 bits de la dirección son llamados dirección de la columna, y serán sostenidos dentro del chip atravéz de los pines A0-A8 por la activación de una señal de selección en el pin CAS (selección de la dirección de la columna). El pin WE (selección de escritura) indicará si el bit de la dirección será de escritura o lectura. De esta manera el bit a escribir será tomado desde DIN, y DOUT producirá el bit que será leído.Ahora se estudiara el chip DRAM para comprender como funciona. La figura 3.25 muestra que es lo que pasa cuando el chip efectua un proceso de lectura, así como también un proceso de escritura.
Las celdas de memoria del chip DRAM estan organizadas internamente en arreglos de 512 X 512 = 256. Las direcciones de las filas seleccionarán la fila (o palabra) en este arreglo, y la dirección de la columna seleccionará la columna. Además de este arreglo las celdas de memoria del chip también contienen un latch para sostener las direcciones de las filas y un latch para sostener las direcciones de las columnas; así como también un enorme latch de 512 bit para sostener el contenido de una fila completa.
El ciclo de memoria comenzará cuando la señal de RAS sea activada Esta señal a la vez seleccionará las direcciones de la fila dentro del latch. La dirección de la fila entonces será decodificada para seleccionar una de las 512 líneas de palabras. Así estas causarán que todos los 512 celdas de memoria de la

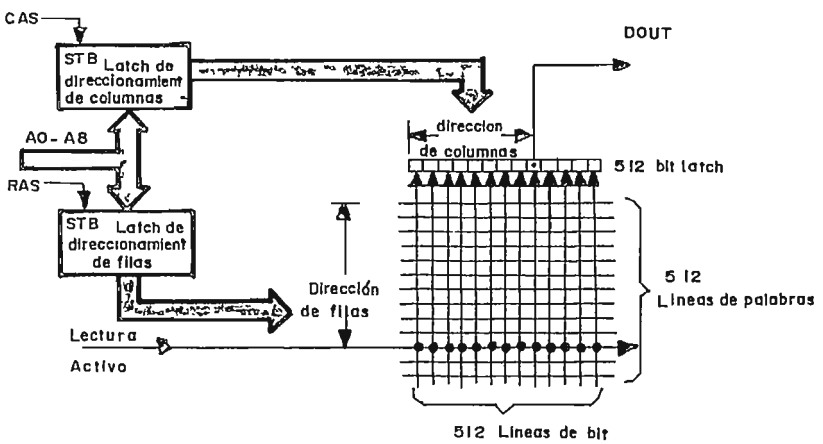


Fig 3 25 Interior de un chip DRAM durante una operación de lectura

filas sean sensadas simultaneamente a través de las líneas de bi y los latch. El sensado de las celdas de memoria destruirán s contenido, por lo tanto el chip DRAM restaurará las filas desc el latch.

Mientras todo esto sucede, CAS activará la dirección de l columna dentro del latch. La dirección de la columna es entonce decodificada para seleccionar uno de los bits de las filas de latch. Este bit será amplificado para ser manejado por la salid DOUT. El ciclo de memoria finalizará cuando la señal en RAM se hara es ahora cuando el período de descanso de precarg comenzará si WE (activación de escritura) cambia de bajo a alt en la mitad del ciclo de memoria.

3.18 El Controlador DRAM 8207

Como se puede observar de las secciones anteriores, existe un notable diferencia entre la interfase del chip DRAM y l del bus 286. Por lo tanto el chip controlador DRAM 82C soluciara esta diferencia. Además tomando cuidado de activar la direcciones de las filas y columnas además de un refresc periódico de la DRAM, el controlador tambien implementará un técnica llamada intercalación el cual efectivamente ocultará l DRAM en el tiempo de precarga.

La DRAM controlará la organización de la memoria en cuatr bancos. Por lo tanto el banco será determinado por los bits A1 A2 de la dirección física respectiva. De esta manera si se acces la secuencia de una palabra almacenada una después de otra e memoria, entonces el ciclo a través del banco será:
BANCO 0, BANCO 1, BANCO 2, BANCO 3, BANCO 0, BANCO 1, BANCO 2...
Así aunque el tiempo de precarga para el chip DRAM 51C256H fuera de 70 nanosegundos, el cual es menor que un ciclo de bus único se podría ignorar la precarga a menos de que el 286 accesará do veces el mismo banco en una fila. En este caso, el controlado DRAM automáticamente señalizaría al 286 a insertar estados d espera.
La figura 3.26 muestra como conectar el controlador DRAM 8207 chips DRAMS con un arreglo de 256k. Estos chip estan organizado dentro de 4 bancos, en el cual cada banco contiene 256k d palabras (512 bytes) de memoria, llegando a obtener un total d megabytes.

Cada banco es dividido por la mitad, con una mitad sosteniend todos los bytes de dirección impar y la otra mitad sosteniend los bytes de dirección par. Además los direccionamientos del bu del 286 fluirán dentro del controlador DRAN. Los bits A1 Y A conectarán al pin el selector de bancos BS0 Y BS1. Los bits A3 A11 llevarán la dirección de la fila del DRAM y los bits A12-A2 llevarán la dirección de la columna del DRAM.

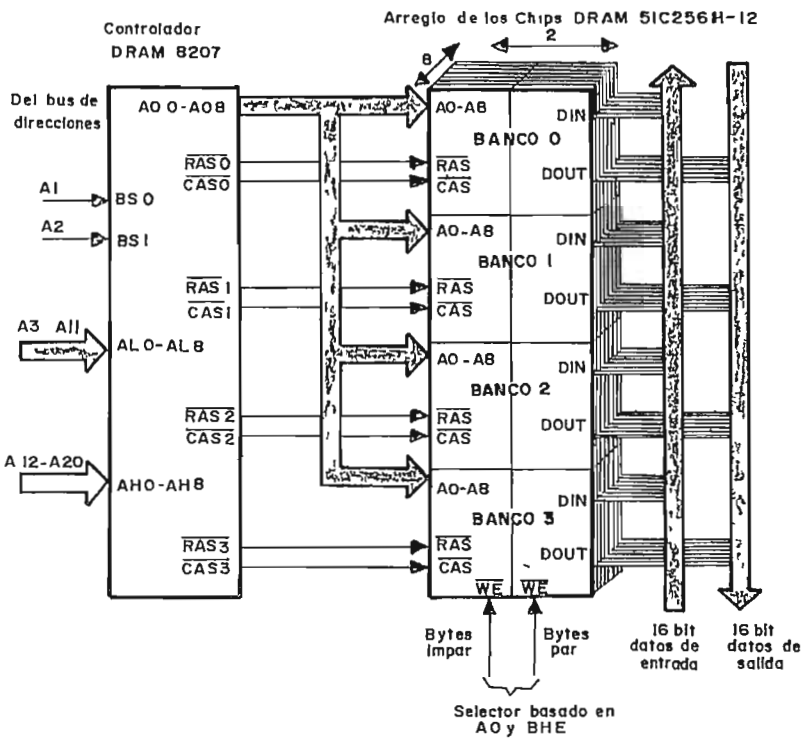


Fig 3 26 Conexion de un Controlador DRAM 8207 Un Arreglo de Chips de 256 K

3.161 Conexión del controlador del DRAM al bus del 286

Si se es cuidadoso, se podrá utilizar el subsistema de memoria mostrado en la fig. 3.26 con un 286 de 8MHZ sin ningún estado de espera, excepto cuando un banco sea accesado dos veces en una fila o una petición de memoria durante la operación de refresco. Así la figura 3.27 muestra como conectar el controlador DRAM 8207 al 286. La primera observación es que no posee el controlador del bus 82288. Por velocidad, el controlador DRAM decodificará el estado del bus directamente desde el 286. La segunda observación es que no existe el latch de direcciones. De nuevo por velocidad, el controlador DRAM contendrá su propio latch de direcciones. No obstante, las señales BHE y A0, utilizados para direccionamiento de bytes, no pasarán a través del controlador DRAM , porque se tendrán latches propios para ellos.

En la figura 3.27 se han utilizado latches de 2 bit, en lugar del latch usual de 8 bit. El latch será activado por la señal LEN similar a la señal ALE (activación de latch de direcciones) producida por el controlador del bus (no presentado).

El pin PE(activación de puerto) en el controlador DRAM 8207, será parecido al pin selector de chip, excepto que este controlará la entrada al subsistema de memoria controlado por el 8207.

El pin WE(activación de escritura) producirá una señal la cual se combinará con la dirección del byte del latch, para producir la señal de activación de escritura para los chip DRAM de dirección par o impar. El pin AACI (reconocimiento) proveerá una señal de ready si el controlador DRAM necesitara un estado de espera.

Note en la figura 3.27 la inclusión de las líneas de datos de entrada y salida separados. Entonces porqué no se deben de conectar los pines DIN y DOUT del chip DRAM juntos y tener una única ruta de las líneas de datos al 286? ésto no podrá ser hecho si lo que se necesita es explotar la estructura de bus segmentado del 286.

Recuerdese que el 286 se inicializa enviando una dirección antes de inicializar el ciclo del bus. Por lo tanto en una escritura de memoria, el chip controlador DRAM no enviará su señal de activación de escritura, sino hasta que se envíe la dirección de la fila y columna.

El DRAM comenzará su ejecución de la parte de lectura, solo si se hubo cancelada la activación de escritura. Esto causará que los datos temporalmente aparezcan en el pin DOUT. Por lo tanto si se une DOUT y DIN, entonces éstos datos interferirán con los datos que han sido escritos.

La solución será bloquear la salida del DRAM mientras se ejecuta la escritura. Por lo tanto se debería de utilizar un transceptor para éste fin, pero en su lugar se utilizara un buffer AS 240, el cual proveerá la amplificación y cambio de dos veces la velocidad del usual transceptor ALS 245. Por lo tanto un AS 240 bloqueara o amplificará cualquier entrada de 8 bits, dependiendo del pin OE(activación de salida).

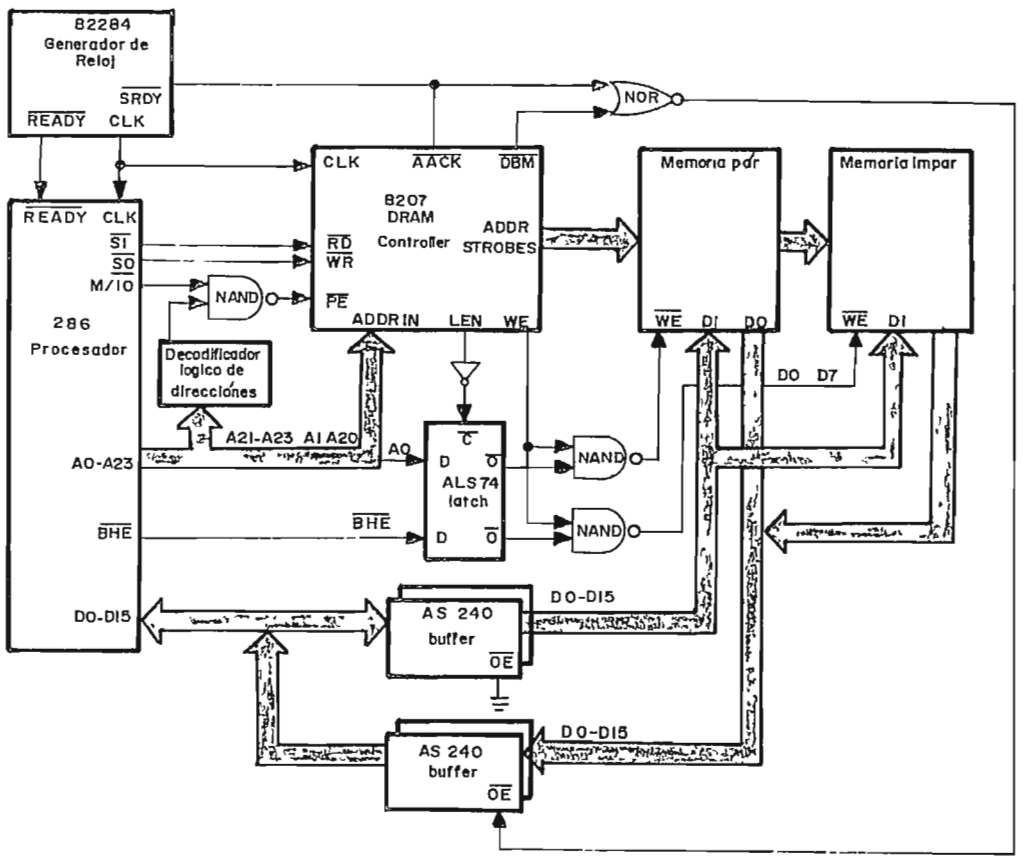


Fig 3 27 Conexion del 286 al DRAM

El controlador DRAM producirá una señal llamada DBM, la cual será similar a la señal de control de dirección del transceptor D1/R producida por el control del bus no presentado. Por lo tanto se podrá utilizar DBM para controlar la salida del buffer, después de combinarla con AACI, para corregir un posible estado de espera. Esto es exactamente lo que se ha hecho en el CI TO de la Fig. 3.27 .
Así la Fig. 3.27 describe un simple subsistema de 2 megabyte de memoria, pero se podrá conectar arriba de 8 subsistemas, cada uno con su propio controlador DRAM. Además nada hay que prevenir con el uso adicional de la conección del controlador de bus, latches , ti ansceptor para la lectura de memoria y dispositivos de I/O.

En conclusión, el DRAM es uno de los interfaces más complejos que se ha tocado en ésta sección. Además uno de los más importantes, puesto que es uno de las interfaces que más fuertemente es utilizado por el 286, el cual podrá ser consultado para mayor información en la bibliografía al final del capítulo.

3.19 Interfase del 286 y del 287

El chip procesador numérico 80287 (287 abreviado), será revisado en interacción con el 286.
El 286 y el 287 operan en paralelo,así cuando el 286 encuentra una instrucción del 287 (una instrucción ESC) primero se asegura que el 287 haya finalizado la ejecución de una instrucción ESC previa, sino, entonces el 286 esperará a que el 287 finalice la instrucción. El 286 entonces notificará al 287 que una instrucción ESC ha sido encontrada, solo entonces el 287 comenzará independiente a ejecutar la instrucción ESC; y el 286 se moverá a la siguiente instrucción.

3.191 Comunicación entre el 286 y el 287

El 286 y el 287 deberán intercambiar cierta información. Alguna de ésta información es transferida sobre el bus del 286, y alguna es transferida sobre una línea de señal especial conectada entre ambos procesadores. La siguiente lista es la información que un procesador deberá leer del otro, con una descripción de cómo ésta información es transmitida.

1.- El 286 deberá de ser capaz de reconocer cuando el 287 esta ejecutando una instrucción previa. De ésta manera el 287 posee un pin llamado BUSY, el cual es manejado por el bit B(BUSY) en el registro de estado del 287.
El bit B será 1 cuando el 287 esté ejecutando una instrucción o señalizando una interrupción, y 0 cuando el 287 está desocupado. Todo ésto es recibido por el 286 a través del pin BUSY.

2.- El 287 deberá tener alguna forma de reconocer cuando el 286 ha encontrado una instrucción ESC, y que variante de la

instrucción ESC ha reconocido.
El 287 conectará los puertos de I/O FA.FF (hexadecimal) al bus del 286. Estos serán utilizados para comunicación entre el 286 y el 287. Cuando el 286 ejecuta una instrucción ESC, éste notificará automáticamente la salida que será ejecutada en el bus, exclusivamente de el mismo modo que las operaciones de I O.

2. El 287 deberá de reconocer el direccionamiento de instrucciones ESC. Ya que este direccionamiento es necesario por el registro apuntador de excepciones.
El 286 enviará ésta información al 287 vía sus puertos como se describió en 2. Nótese que en el modo virtual el direccionamiento de instrucciones deberá ser direccionamiento virtual; en cambio en el modo real solo será suficiente la dirección física.

3. Si la instrucción ESC tiene un operando de memoria, entonces el 287 deberá de reconocer que tipo de dirección tiene el operando. Esto es debido a que el 287 necesitara la dirección del operando para almacenarlo en el registro apuntador de excepciones. Esto también será transmitido a través del 286 vía los puertos de 287.

4. Si una instrucción ESC tiene un operando de memoria, entonces el 287 de algún modo deberá de ser capaz de accesarlo. El acceso del operando, el cual puede alcanzar muchas palabras, requerirá traducción de direcciones y chequeo de protección . El 287 no accesará el mismo el operando, pero por el contrario, el 286 si lo hace. De ésta manera, el 286 podrá manejar la protección de chequeo y la dirección de traslación. Cuando el 286 ve un operando de memoria en una instrucción ESC, éste trasladá la dirección dentro del valor límite del direccionamiento físico, colocando ambos dentro de un registro interno, lo mismo que una bandera de dirección indicarían si los datos fluirán hacia o desde el 287.

El 286 y el 287 tienen el pin PEREQ (procesador e tensión request). Estos deberán de ser conectados juntos, así como también el pin PEACK (processor e tension acknowledge) en ambos procesadores. Cuando el 287 necesite leer o escribir una palabra de un operando, éste señaliza el 286 vía el pin PEREQ. El 286 entonces reconocerá la señal, a través del pin PEACK, y ejecutará un ciclo de bus de memoria y de I/O hacia los puertos del 287 para completar la transferencia deseada. El 286 entonces incrementa los registros internos, de éste modo, sucesivas señales PEREQ accesarán sucesivas palabras en memoria.
De ésta manera, el 287 podrá accesar operandos de multipalabras. El 286 limita el chequeo del registro interno para detectar segmentos completos.

5.- El 287 deberá de ser capaz de notificar al 286 que una excepción numérica no enmascarable ha sido ejecutada.

Ambos, el 286 y 827 poseerá un pin de ERROR, el cual deberá de ser conectado juntos. El pin de ERROR del 287 será manejado por el bit ES(error sumary) en el registro de estado. Este bit indicará la presencia de una escepción no enmascarable. El 286 chequeará el pin de ERROR siempre que se encuentra un ESC o una instrucción WAIT, y producirá una escepción de la unidad matemática si una señal de ERRUR está presente.

3.192 Conexión de un 287 al 286

La figura 4.20 muestra como conectar un chip 287 al 286. No hay muchos componentes, pero hay una parte de las líneas de conexiones que serán resumidas dentro de tres grupos:

- Conección del interprocesador 286-287
- Conección de los puertos I/O del 827
- Conección del reloj del 287.

La conección del interprocesador es única para el interfase 286 287, es decir FEREQ, FEACK, FRROR Y BUSY. los cuales ya fueron discutidos. El 287 adicionalmente monitoreará el bus de estado del 286 vía los pines S0,S1,CODI,INTA,RFADY y HLDA.

Los puertos de interfase de I/O ya fueron discutidos, pero se estudiarán otras propiedades. El 287 fué diseñado para ser conectado directamente al 286 antes de cualquier transceptor o sostenedor de bus. Esto es por la velocidad, y también porque algunos sistemas no pueden utilizar sostenedores para direccionamiento de bus o buffer del bus de datos. Por lo tanto es necesario un latch para la señal de selección de chip (npsl) y un bit de direccionamiento (CMD0 Y CMD1) para los puertos de I/O del 287.

Aquí la dirección completa no necesitará un latch, ya que solo serán necesarios los bit que utilize el 287. Por lo tanto en la fig.4.28 se ha sustituido el latch de 4 bit , el ALS175, por nuestro grupo usual de latchs de 8 bits, de el cual solo se necesitarán 3 bits. Por ésta razón los puertos de I/O del 287 están en el mismo lado del 286 de cualquier transceptor de bus en lugar de el lado con el resto de dispositivos de I/O de el 287. Esto es debido a que sería correcto usar la señal de dirección DT/R del transceptor cuando se está ejecutando una transferencia de I/O al 287.

Por lo tanto la señal selectora del chip del 287, deberá utilizarce para inhabilitar el transceptor del bus siempre que el 286 ejecute una operación de I/O a los puertos de 287.

De ésta manera se deberá ser cuidadoso, en todo caso cuando se utilice el selector de chip del 287 para éste propósito.

Además no se necesitará inhabilitar el transceptor durante un ciclo de bus de memoria o interrupción el cual coincidentalmente involucren direcciones o número de interrupciones idénticos al numero de puertos del 287. Por ésta razón, las señales M/IO y COD/INTA del 286 son incluídas también en el decodificador lógico de direcciones del 287, a fin de limitar la operación del selector de chip del 287, en un único ciclo de bus de I/O.

Finalmente, se considerara la forma de proveer la señal de reloj al 287.
El 287 obtiene la señal de reloj, CLK, del sistema visto anteriormente. El reloj del sistema también se deberá de conectar a la entrada CLK de el 287. De ésta manera la entrada CLK del 287 será utilizado para interaccionar con el 286, sin embargo el 287 no obtiene su propio tiempo interno de la entrada CLK 286. En lugar de esto, posee otro pin, CLK, el cual gobernará su velocidad. Esto permitirá a los dos procesadores correr bajo su propia velocidad.
El pin CLK del 287 se podría conectar a su propio generador de reloj o al reloj del sistema como el 286. Si el reloj del sistema es conectado al pin CLK del 287, entonces el pin CLM deberá permanecer conectado a tierra. El 287 entonces dividirá la frecuencia del reloj del sistema por tres.
Así, si el 287 trabajará con un 286 de 8 MHZ el cual tuviera un reloj de sistema de 16 MHZ, el 287 correría a 16/3=5.3 MHZ.
Por lo tanto para que el 287 funcione a una velocidad mayor se deberá de utilizar un generador separador e reloj. Por ejemplo si se conectara la señal de un reloj de 8MHZ al pin CLK de el 287, entonces éste funcionaría a 8 MHZ (lo cual sería la máxima frecuencia del 287) CLM deberá ser conectado a Vcc en éste caso para evitar la división de la frecuencia por tres.
Recuerdese que el generador de reloj 82284, cuando producía un reloj de sistema de 16 MHZ, también la señal PCLK de 8 MHZ.

3.20 Construcción y comprobación de un sistema basado en el 80286

En esta sección se presentarán los esquemas completos de un sistema basado en el 80286, para 8 MHz, con funcionamiento en estado de espera 0. Incluyendo EPROM, RAM y periféricos, solamente se necesitaran 40 circuitos integrados, de forma que, aunque es un diseño de altas prestaciones, capaz de ejecutar 1,5 millones de instrucciones por segundo (MIPS), es bastante simple de construir. Además de encontrar que este circuito esta hecho con componentes fácilmente disponibles.
El diseño es instructivo, puesto que contiene todos los circuitos que se necesitan típicamente en la mayor parte de los sistemas. Fabricando alrededor de la CPU y dos circutos integrados de soporte de Intel, estan los transceptores de datos, amplificadores de dirección, circuitos decodificadores de dirección, circuito READY (para controlar el número de estados de espera), EPROM, RAM, y tres dispositivos periféricos. Una vez fabricado, se tendrá un sistema completo preparado para hacer funcionar el software.

3.21 El diseño de la circuitería

Cada una de las próximas secciones es bastante corta. Se cubre el diseño sencillo, los esquemas del circuito, diagnóstico del

software y todo lo necesario.
El plan es presentar el diagrama de bloques y entonces explicar todos los bloques principales por turno. Como se ha mencionado, hay una pequeña cantidad de software diseñado para ayudar a comprobar la circuitería.
También hay alguna circuitería opcional de diagnóstico utilizando leds para proporcionar indicadores visuales de diagnóstico.

3.211 Diagrama de bloques del sistema

Conceptualmente, es mejor definir el núcleo del diseño alrededor de la CPU, algo que sea fácil de diseñar y que esté prácticamente garantizado que funcione con un esfuerzo mínimo. Al ver que funciona hará crecer su confianza. Y aunque si ud no planea actualmente construir un sistema, este núcleo le ilustra los principios de una configuración de mínima complejidad y altas prestaciones. Aparte de mostrar las principales funciones involucradas, la figura 3.29 muestra los caminos de interconexión cruciales, tales como el bus de datos, señales de estado, y señales de control y de mando. Estos buses y señales constituyen conexiones muy estandarizadas en los sistemas con el 80286.

Se examinará los componentes y conexiones. Como se muestra en el diagrama de bloques, el diseño del núcleo del 80286 consiste en la CPU 80286, el circuito integrado generador de reloj 82284, el circuito integrado controlador del bus 82288, dos circuitos EPROM, dos circuitos de RAM estática, tres circuitos periféricos, y algo de lógica de interconexiones.

La lógica de conexión incluye amplificadores de dirección, decodificadores de dirección, transmisores de datos, la lógica de READY.
Dentro de este núcleo se podrá opcionalmente añadir varios led como indicadores de diagnóstico de los circuitos mostrados en la figura 3.45 y 3.46 . Estos led serán ubicados en las señales críticas, indicando visualmente que el sistema esta funcionando por su encendido intermitente.

Aún más importante, si el sistema deja de funcionar inesperadamente, estos led indicarán el tipo de ciclo de bus que esta ejecutandose en el momento del fallo.
Esta ayuda señalará errores de la circuitería.

3.22 El núcleo: El procesador y sus componentes de soporte

La figura 3.30, 3.31 y 3.32 ilustran el trio de los componentes LSI que constituyen el corazón de muchos sistemas con el 80286. soportando al mismo 80286 estan otros dos componentes de Intel:

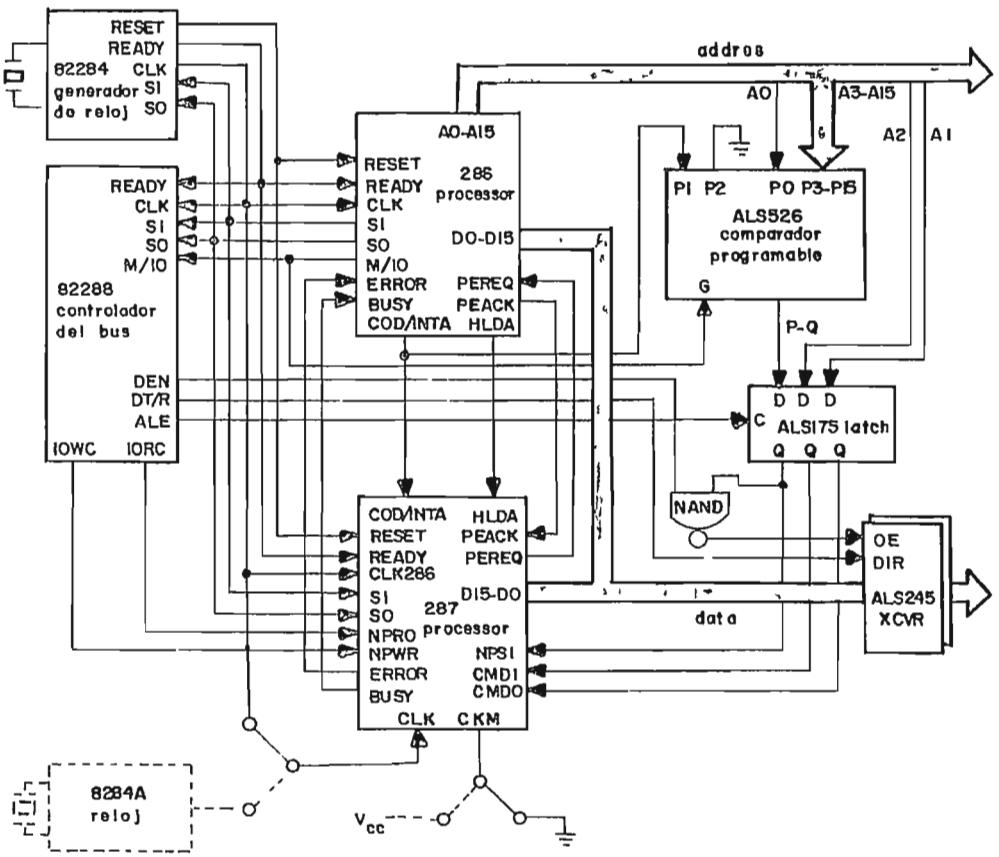


Fig 3 28 CONEXION DEL COPROCESADOR MATEMATICO 287 AL 286

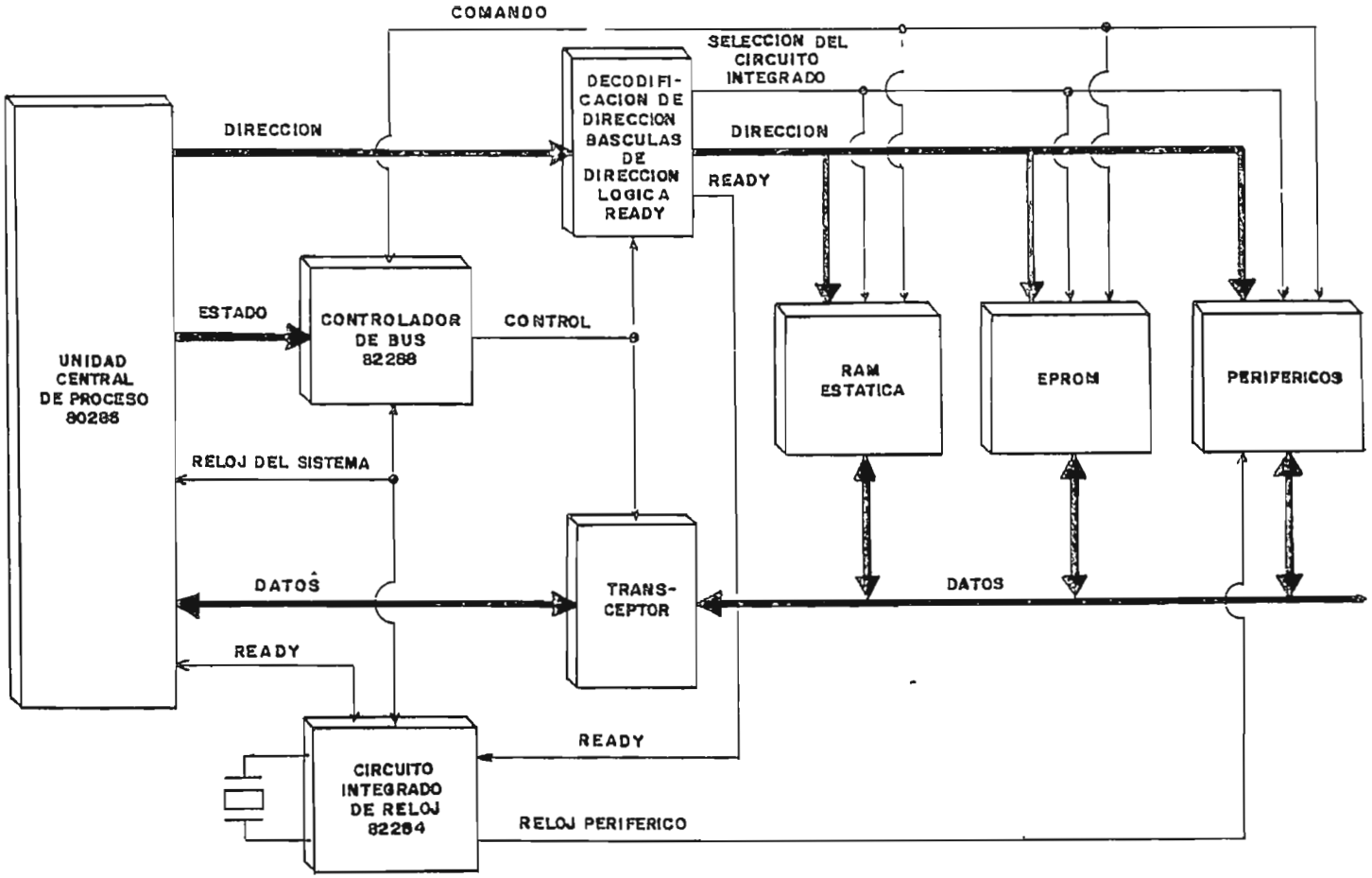


FIGURA 3 29 DIAGRAMA DE BLOQUES DEL NUCLEO DEL SISTEMA

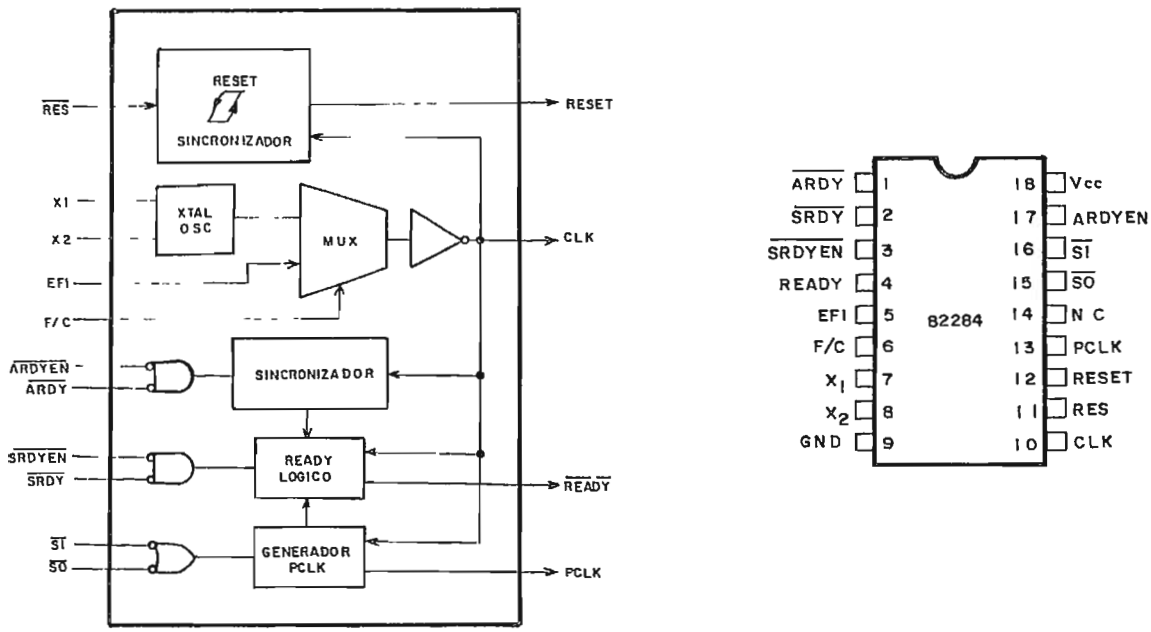


FIGURA 3 31 CIRCUITO INTEGRADO GENERADOR DE RELOJ 82284

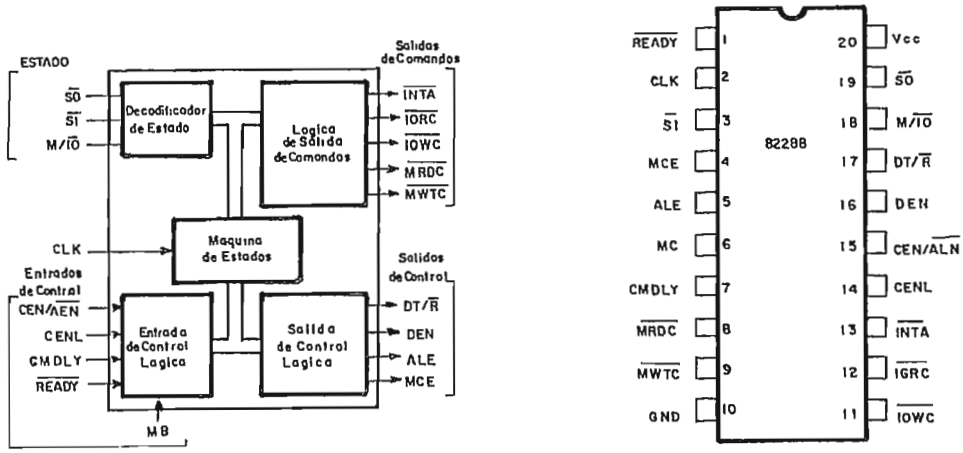
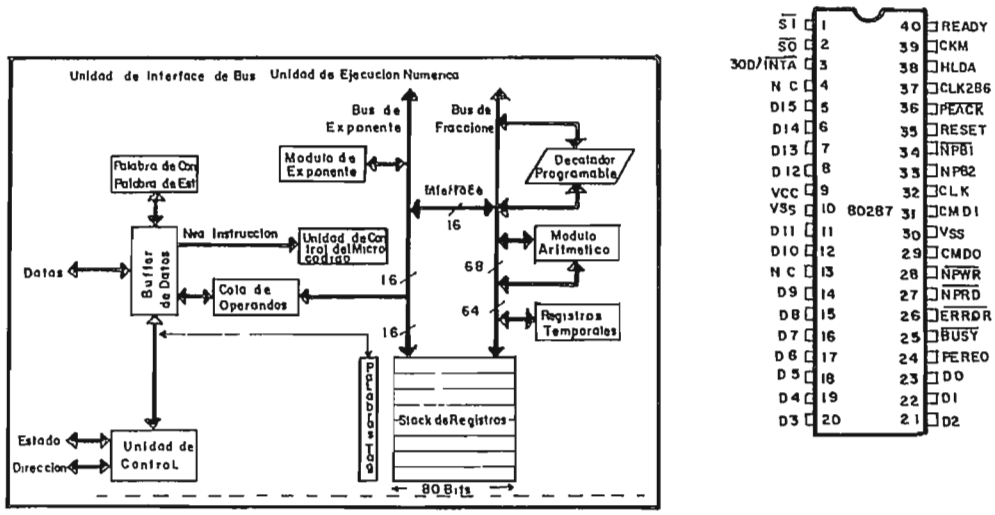


FIGURA 3 32 CONTROLADOR DE BUS 82288



NOTA

LAS PATILLAS N C NO DEBEN SER CONECTADAS

FIGURA 333 COPRECESADOR NUMERICO 80287

el generador de reloj 82284 y el controlador de bus 82288. Estos dispositivos se muestran esquemáticamente en la figura 3.38 .
El procesador numérico 80287, mostrado en la figura 3.33, también tiene una configuración normalizada de conexión cuando se incluye en un sistema.

3.221 El generador de reloj 82284

Acompañando a la CPU 80286 está el 82284, un circuito integrado generador del reloj que es comatible (véase la figura 11.10). Este reloj genera una señal CLK (reloj) estable de 16 MHz a partir de un cristal agregado de 16 MHz.
El CLK es muy parecido a un marcapaso marcando el ritmo de todo el sistema.
La CPU internamente divide el CLK por dos y, por tanto avanza a través de las microinstrucciones ó microcódigo, a una velocidad de 8 MHz. En conjunto, el sistema esta clasificado como un sistema de 8MHz.

3.222 La CPU 80286 y el controlador del bus 82288

A 8 MHz, la CPU 80286 es capaz de procesar una media de cerca de 1,5 millones de instrucciones por segundo. A 8 MHz, es una máquina 1.5 MIP, que es lo mismo que decir muy potente, aunque desprecíemos las capacidades únicas de la CPU de conmutación de tareas y de memoria virtual.
La CPU está contenida en un encapsulado de 68 patillas. Tiene unos buses de direcciones y de datos separados (no multiplexados), además de patillas para el estado, interrupciones,el interfaz del coprocesador 80287 y otras utilizaciones. Está disponible en dos variaciones del encapsulado ilustradas mediante la figura 3.30 El LCC (leadless chip carrier, portador de circuito sin patilla) estuvo disponible en primer lugar, y más recientemente los equipos de fabricación para el PGA (pin-grid array) han hecho al PGA también disponobles en cantidades masivas.
La CPU controla la circuntería del sistema iniciando los ciclos de bus. Cada ciclo de bus es una unidad de actividad que incluye a la CPU y a alguna parte del núcleo del sistema. Un ciclo de bus usualmente consta de operaciones de lectura o escritura.
Un ciclo de bus necesita un mínimo de solamente dos períodos de reloj de 8 MHz. la CPU realiza típicamente de 3 a 3.5 millones de ciclos de bus por segundo, cercano al máximo teórico de 4 millones.
Observe las formas de onda del ciclo de bus en la figura 3.35 Cada ciclo de bus empieza cuando la CPU activa por lo menos una línea de estado, o bien S0 o S1. Como mínimo, el ciclo necesita dos periodos del reloj del procesador, pero continúa indefinidamente hasta que el sistema activa la entrada READY de la CPU. Al activar la entrada READY, le dice a la CPU que puede terminar el ciclo de bus actual.
Un ciclo de bus de la CPU es siempre uno entre cinco tipos

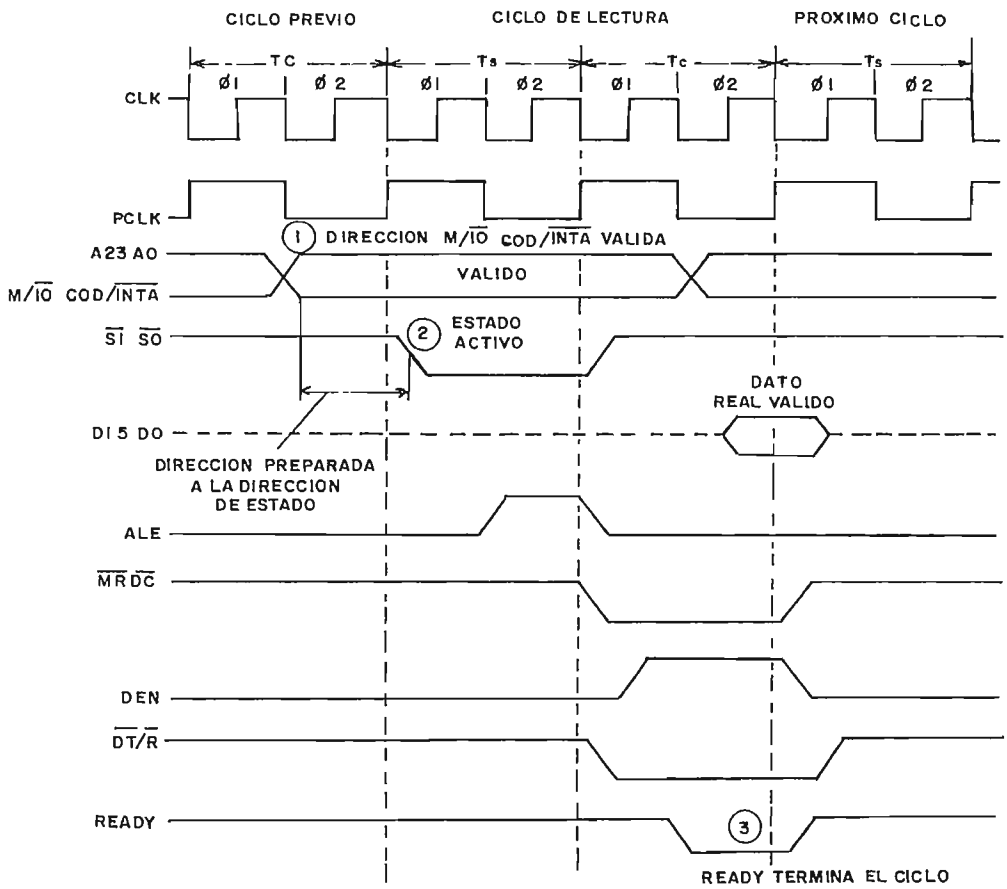


FIGURA 3.35a FORMAS DE ONDA DEL CICLO DE BUS

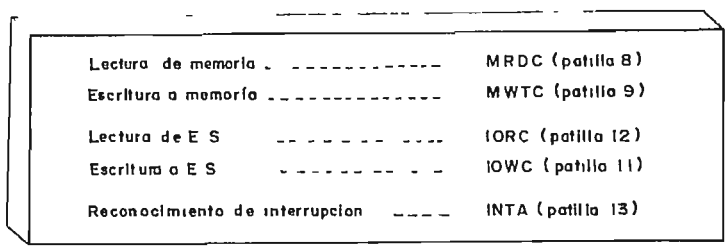


FIGURA TIPOS DE CICLOS DE BUS DEL 80286 Y SEÑALES DE COMANDO DEL 82288

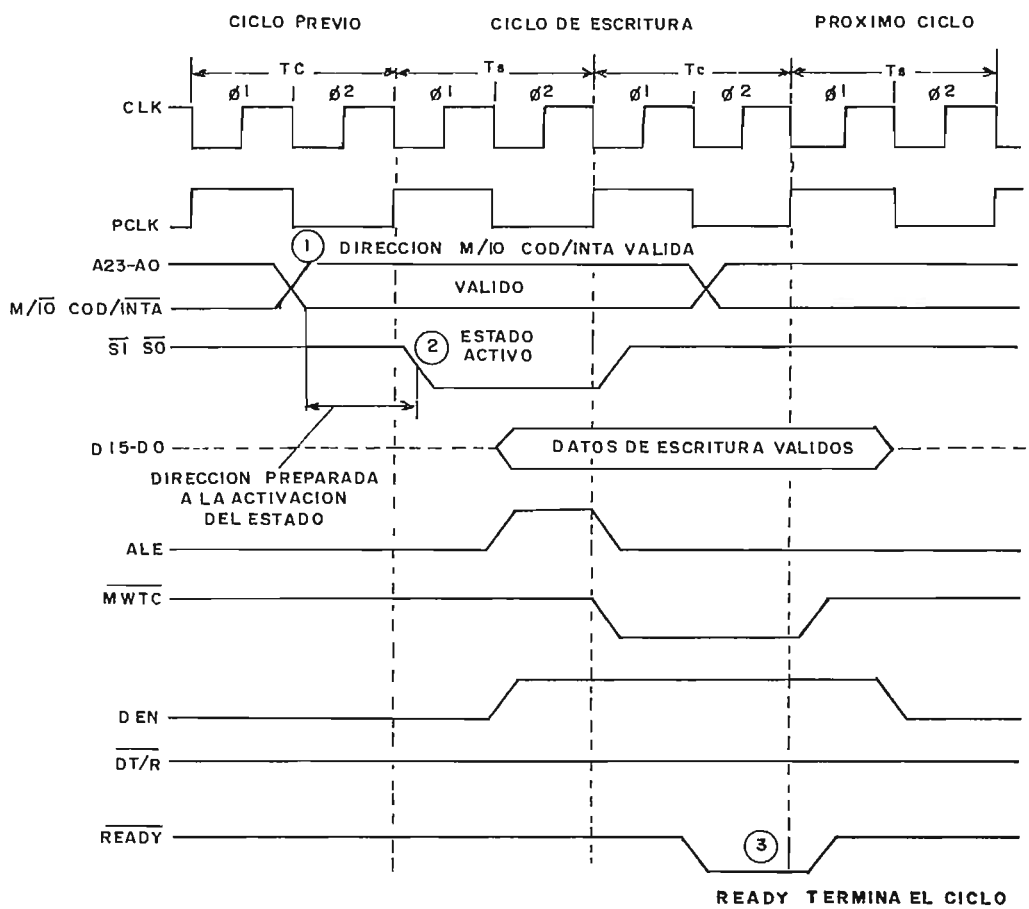


FIGURA 3 35b FORMAS DE ONDA DEL CICLO DEL BUS

Patillas del ESTADO DE CICLO DE BUS, SO y SI indican la iniciacion de un ciclo de bus cuando por lo menos uno de ellos pasa a nivel bajo ademas de M/IO y COC/INTA define el tipo de ciclo de bus El bus esta en un estado T siempre que uno o los dos SO y SI este a nivel bajo, SI y SO se activan a nivel bajo y quedan flotantes en 3 estados OFF durante el reconocimiento de una posesion del bus

Definicion del estado de ciclo de bus del 80286

COD / $\overline{INTA}$	M / $\overline{IO}$	$\overline{S\,I}$	$\overline{S\,O}$	Ciclo de bus iniciado
0 (LOW)	0	0	0	Reconocimiento de interrupcion
0	0	0	1	No ocurrira
0	0	1	0	No ocurrira
0	0	1	1	Ninguno, no un ciclo de estado
0	1	0	0	Si A I = 1 entonces para si no interrupcion
0	1	0	1	Lectura de dato de memoria
0	1	1	0	Escritura de dato en la memoria
0	1	1	1	Ninguno, no un ciclo de estado
1 (HIGH)	0	0	0	No se producira
1	0	0	1	Lectura E/S
1	0	1	0	Escritura E/S
1	0	1	1	Ninguno no es un ciclo de estado
1	1	0	0	No se producira
1	1	0	1	Lectura de instruccion en la memoria
1	1	1	0	No se producira
1	1	1	1	Ninguno no es un ciclo de estado

Figura 3 36 DEFINICION DEL ESTADO DE CICLO DE BUS

posibles:
- lectura de memoria
- escritura de memoria
- lectura de E/S
- escritura de E/S
- reconocimiento de interrupción.

Finalmente el ciclo de bus involucra al controlador de bus 82288, que activa en el correspondiente señal de comando. Note la correspondencia en la figura 11.6

Como muestra el esquema en la figura 3.39 el 82288 es colocado entre la CPU y el resto del sistema para ayudar a controlar el ciclo de bus. Traslada la información de estado pura (señales S0,S1,M/IO) desde la CPU a otros dos tipos de señales. Las señales de comando son de uno de los tipos que muestra la figura 3.34.Las señales de control del 80286 (ALE,DEN,DT/R) son el otro tipo, que activa el latch del bus de direcciones y el transmisor del bus de datos en los tiempos apropiados.

Al tener las funciones e actas de comando y de control incluidas en el 82288, hace que está sea una solución compacta y muy conveniente. Las señales que genera tienen tambien capacidades de comando muy potentes compatibles con la norma de bus Multibus (IEEE norma 769).

Las direcciones de la CPU sin almacenar van a almacenarse en unos latch, como ilustran los esquemas en la figura 3.40 . Los latch, a menudo controlados enteramente mediante el 82288, mantienen la dirección estable durante todo el ciclo de bus. Observe que en la figura 3.41 hay un circuito especial diseñado para abrir los latch tan pronto como READY indica que el ciclo previo está terminando. Este circuito abre los latch antes de lo que haría la señal ALE del 82288, proporcionando más tiempo de acceso a las direcciones (a e pensas del tiempo de mantenimiento de las direcciones después de la escritura). Ahora, aun en este diseño de altas prestaciones, la señal ALE tiene un objetivo (aquí se utiliza para cerrar los latch).

Las direcciones sin pasar por los latch tambien dirigen al decodificador, como se muestra en el esquema de la figura 3.41, para generar las señales de selección en circuitos integrados para la EPROM, RAM y circuitos periféricos. Este circuito decodificador se aprovecha del hecho de que los tiempos de direccionamiento de la CPU son de estructura segmentada (pipeline)(las direcciones aparecen antes de que la señal de estado indique el principio de un ciclo de bus).Esta estructura segmentado del 80286 es el mayor beneficio de la circuiteria.

El direccionamiento con estructura segmentada le da al decodificador un inicio anticipado en la generación de la señal de selección de circuitos para el próximo ciclo. Así la estructura segmentada permite la utilización de dispositivos de memoria más lentos sin necesidad de añadir ciclos de espera al ciclo del bus. Por su naturaleza, la estructura tubular necesita unos latch e ternos para mantener estable las direcciones y la señal de selección del circuito integrado PROM o RAM. Ahora bien, una CPU sin estructura segmentada necesita buffer de direcciones

e interior, de forma que la estructura segmentada no incremente el número de ciclos del sistema.

3.23 El ciclo de bus del 80286

La secuencia de la señal que se produce durante un ciclo de bus es muy directa, pero puede parecer confusa la primera vez. Recuerde que la figura 3.35 muestra estas señales. Cualquier ciclo de bus empieza con el paso uno, la aparición de una dirección en las patillas de dirección. Como se ha mencionado, la salida de la dirección segmentada es solapada con el final del ciclo de bus previo, tan pronto como el sistema activa la patilla READY al final del ciclo de bus previo, la CPU realiza el paso dos, la salida de la información de estado en S0, S1, M/IO y COD/INTA, como se detalla en la figura 3.35b Si el ciclo del bus es de escritura, la CPU también dirige su bus de datos en este momento.
El controlador de bus del 82288 traduce el estado de información de la CPU por las señales de comando apropiadas y señales de control. Como se ha explicado, la señal de comando activada corresponde al tipo de ciclo de bus que se está realizando. La señal de control ALE (Address Latch Enable, habilitación del latch de dirección) es siempre activada en el próximo flanco de CLK (inmediatamente después de que el 82288 ha reconocido que el estado de la CPU está activo). ALE provoca que los latch de direcciones mantengan las direcciones segmentadas y la mantengan estable para el resto del ciclo. Las otras señales de control DEN (Data Enable, habilitación de datos) y DT/R (Data Transmit/Receive, Transmitir/Recibir Datos), habilitan y controlan la dirección del flujo de datos.
La entrada READY en el 80286 permite "alargar" cualquiera de sus ciclos de bus tanto tiempo como sea necesario; así puede utilizarlo para proporcionar el tiempo extra para las memorias más lentas o para los periféricos. El ciclo de bus mínimo, sin alargar, dura sólo dos ciclos de reloj del procesador (dos periodos de 8 MHz, por ejemplo, un total de 250 mil millonésimas de segundo). El ciclo mínimo de bus puede alargarse en incrementos de ciclos del procesador (125 mil millonésimas de segundo) para acomodar a las memorias más "lentas" o a los periféricos. Los ciclos de reloj extras por encima de los dos mínimos son llamados ciclos de espera, y desde luego se alcanzan las más altas prestaciones del sistema cuando hay pocos ciclos de espera, si es que hay alguno.
Los requisitos de estados de espera son función del dispositivo que se está direccionando. Así, dependerá del decodificador de direcciones el determinar cuantos estados de espera, si hay alguno, se deberán de añadir en cada ciclo de bus particular. En este diseño, debido a que se está utilizando direccionamiento en estructura segmentada, no se necesitan ciclos de espera, aunque se utilicen EPROM de velocidad estándar (200ns) y RAM (150 ns). Los circuitos decodificadores (usualmente en unión de registros

de desplazamiento tal como el de la figura 3.42) deben generar una señal READY cuando ha llegado el tiempo de terminar el ciclo de bus. En caso contrario, si el sistema no genera una señal READY, su CPU " esperará " para siempre, esperando que termine el ciclo de bus actual. Puede verse que es crítico el generar una señal READY al final de cada ciclo de bus.

3.24 EPROM, RAM estática e interfaz de periféricos

El controlador de bus genera un conjunto de señales muy convenientes para interfaces como otros dispositivos. Dispositivos tales como EPROM, RAM y periféricos se unen muy directamente, como se muestra en los esquemas de las figuras 3.43 y 3.44 . Casi no se necesita lógica adicional.
Ahora bien, debido a la simplicidad del 8088 y a su naturaleza de propósito general, éste no realiza el más eficiente uso del tiempo de ciclo de la CPU. Así en este diseño de altas prestaciones, el circuito incluye dos pequeñas piezas de lógica extra para utilizar mejor el direccionamiento con estructura segmentada.

Los así llamados, "circuitos de altas prestaciones " que se encuentran en los esquemas de las figuras 3.40 y 3.42 , modifican la sincronización de la habilitación del latch de direcciones y de la señal de escritura, respectivamente. Para incrementar el tiempo de acceso a las direcciones (y mejorar así la posibilidad de cero estados de espera), el circuito de altas prestaciones abre los latch de direcciones más pronto de lo que lo hace la señal ALE estándar. Un efecto secundario es que la señal de escritura debe terminar así más pronto de lo usual, para proporcionar un tiempo de mantenimiento de la dirección después de enviar el pulso de la señal activa de escritura.
EPROM, RAM estática y los periféricos se unen mediante interfaces en gran parte de la misma forma. El interfaz de la EPROM es el más sencillo, puesto que las EPROM son de sólo lectura. Los interfaces de la RAM deben soportar ciclos de escritura de una amplitud de byte o de escritura de amplitud de 16 bits. Así, la habilitación de escritura de la RAM para los bytes de mayor peso o menor peso es controlada separadamente.
En este diseño de 8 MHz, los ciclos de lectura desde la EPROM de 200 ns y desde la RAM de 150 ns se producen sin estados de espera. Estos ciclos constituyen cerca del 75 por 100 de todos los ciclos de bus realizados de forma que los estados de espera medios son casi de valor cero. Sin embargo, los ciclos de escritura a la RAM necesitan un estado de espera. (note que el circuito que genera la señal READY en la figura 3.42 cualifica la decodificación de la dirección de RAM con las señales de comando MWTC y AMWTC para añadir un estado de espera solamente a las escrituras a la RAM, no para las lecturas de la RAM).
Los ciclos de bus a los circuitos periféricos provocan que el circuito de generación de la señal READY añada tres estados de

espera, puesto que los dispositivos periféricos necesita generalmente pulsos activos más largos en sus entradas. CS, RD WR, los tres estados de espera no afectan sustancialmente a la prestaciones del sistema, tal como el circuito que realiza ciclo de lectura y escritura de altas prestaciones con la EPROM y l RAM y los periféricos. Sin embargo, este diseño tiene solament unas pocas piezas , de forma que usted encontrará que es bastant fácil el hacerlo funcionar. Una vez que lo haya hecho encontrará que es un diseño muy instructivo, y al mismo tiempo práctico.

3.25 Diagrama y esquema de los circuitos

Un diagrama de disposición, como el esquema de la figura 3.37 es una posible forma de colocar los componentes necesarios para est sistema. Sin embargo, el diseño no es sensible a la disposición exacta. Asegurese solamente de tener una distribución limpia de la " alimentación " de 5 V y de la " tierra " 0 V en su tarjeta Así, coloque varios condensadores de desacoplo entre la tensión de + 5V y la tierra, porque esto ayudará a mantener limpia l distribución de la alimentación.
En las figuras 3.39 hasta 3.45 se podrá encontrar esquema detallado para este sistema 80286. Generalmente el flujo de la señales será de izquierda a derecha en cada página. Además, e título en cada página de esquemas ayuda a relacionarlo con e diagrama de bloques en la figura 3.29.

3.26 Indicadores de diagnóstico de la circuitería

Los indicadores de la circuitería opcionales que puede añadir a diseño del nucleo del sistema sirven como indicadores de la actividad de la circuitería. Sin embargo, funcionan de forma distinta de los dispositivos de salida típicos, debido a que estos indicadores de los circuitos no están bajo el control del software.
La circuitería de diagnóstico está hecha de lógica y dispositivos LED mandados por señales críticas del núcleo de los circuitos. Cuando se añaden al diseño central, los circuitos de diagnóstico le dan una indicación visual de si el sistema está funcionando o no. Encontrará que es una ayuda muy valiosa para la depuración del sistema o simplemente como herramienta para ayudarle a aprender el funcionamiento de la parte principal de la circuitería. En el diseño de un prototipo típico, los circuitos de diagnóstico comprenden varios circuitos integrados y dispositivos LED. En este ejemplo presentado, el esquema de los circuitos de diagnóstico aparecerán en la figura 3.45.
Los circuitos de diagnóstico que se utilizarán estan diseñados con dos claros objetivos: siendo estos:
1. Indicar si la CPU continúa realizando ciclos de bus.
2. Si la CPU ha dejado de realizar ciclos de bus,e indicar por qué.

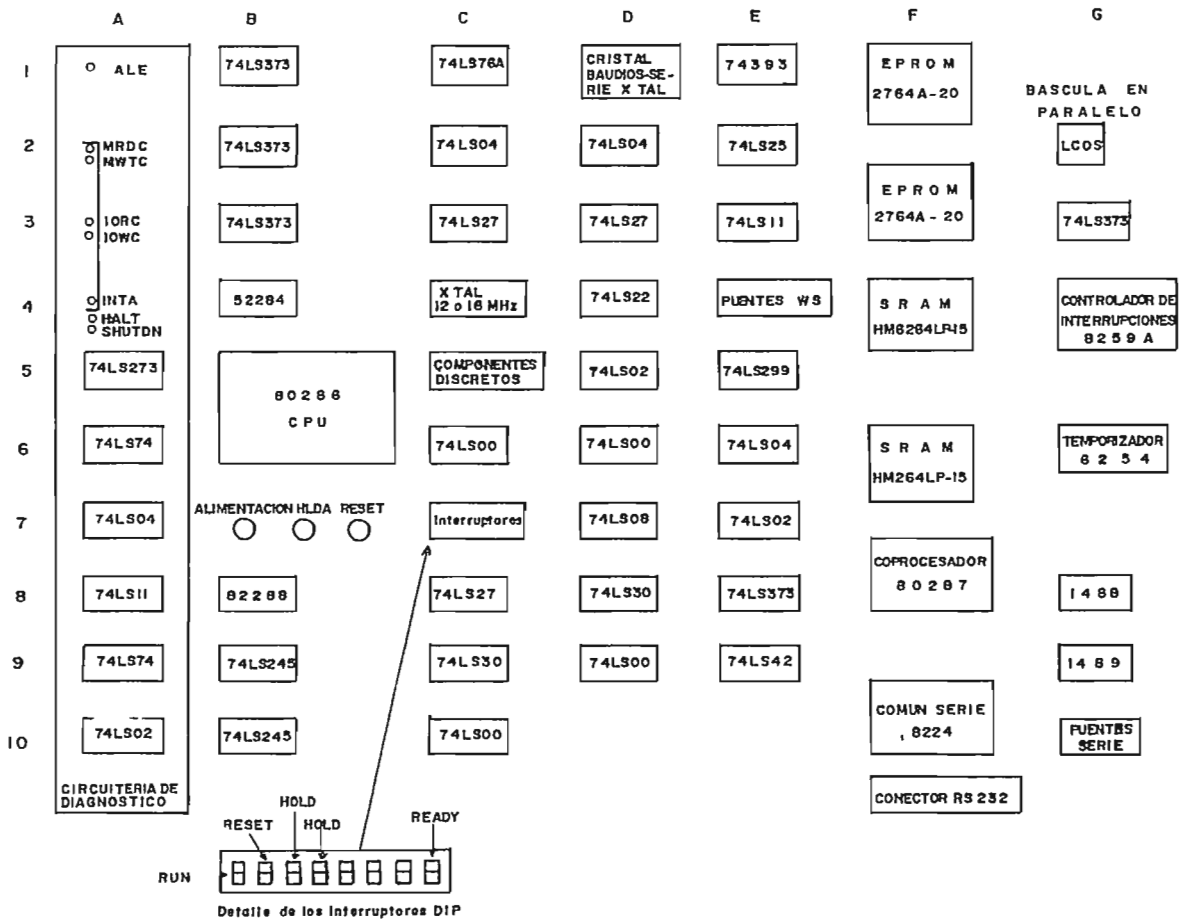


Figura 3.37 DIAGRAMA DE DISPOSICION DEL SISTEMA

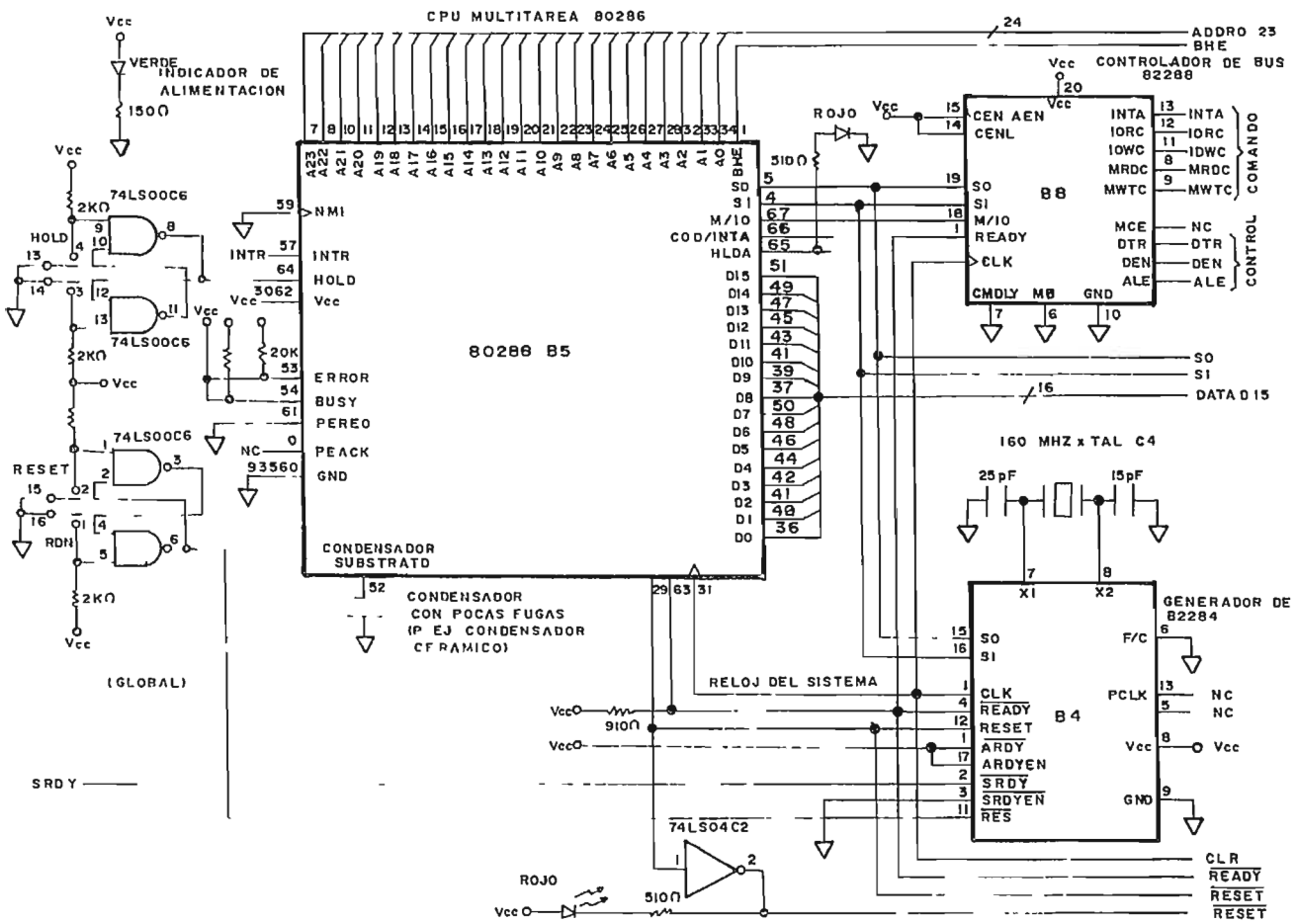


FIG 338 (80286 Y COMPONENTES DE SOPORTE)

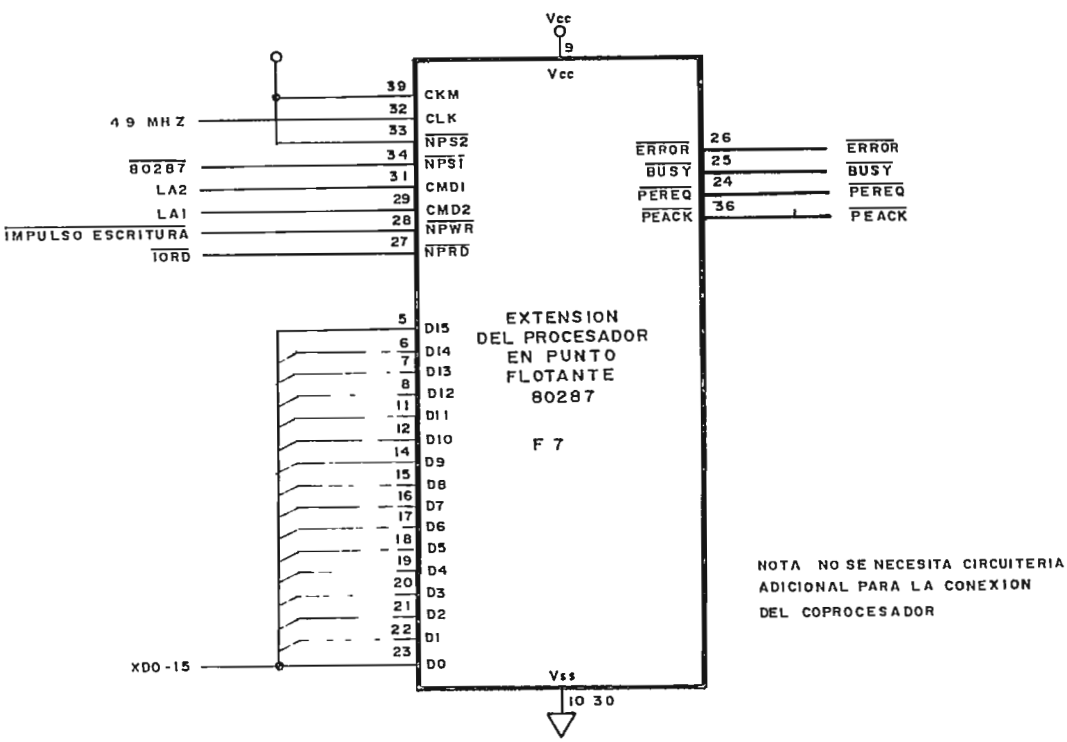


FIG. 339 COPROCESADOR NUMERICO 80287

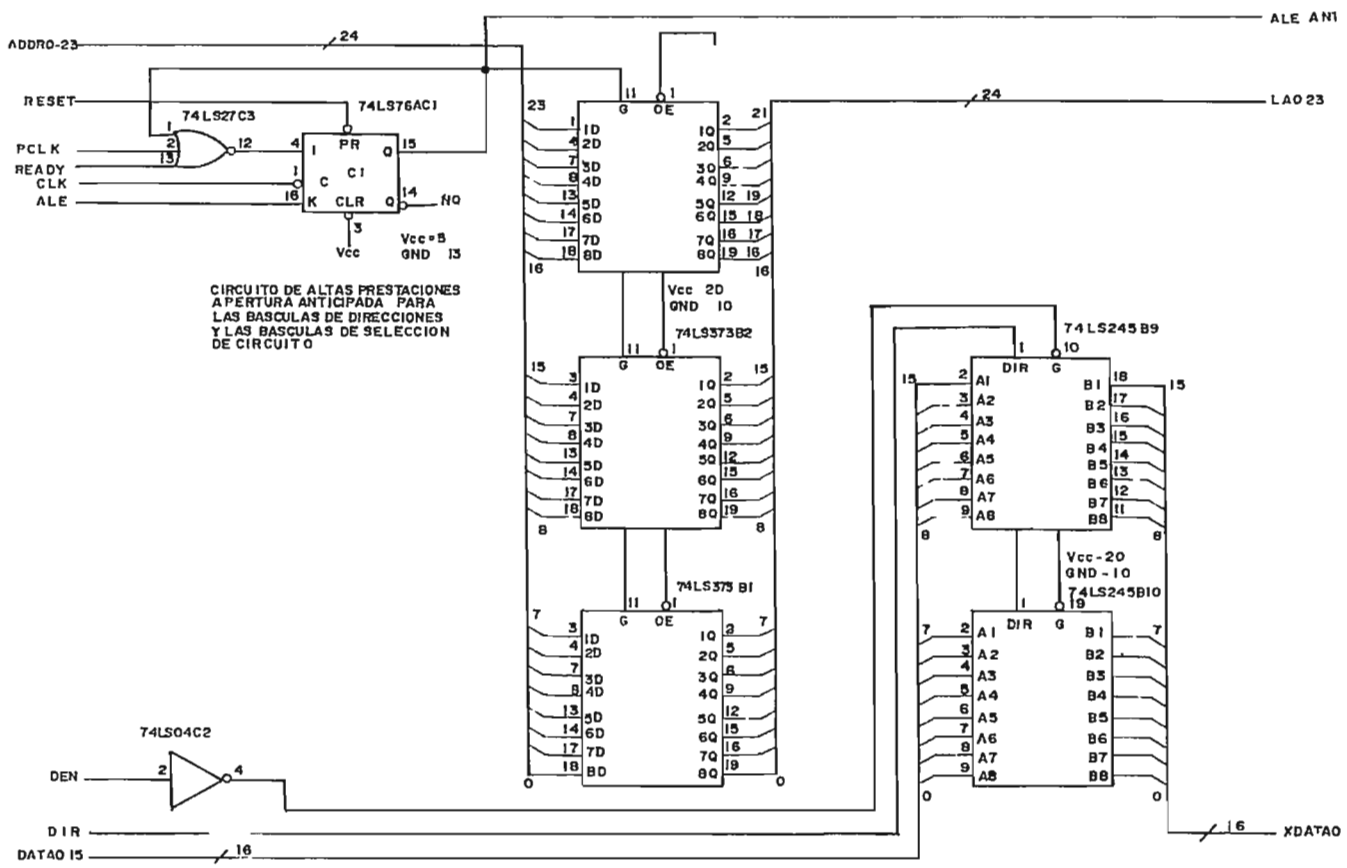


Figura 340 TRANSMISOR DE DATOS Y LATCH DE DIRECCION DE ALTAS PRESTACIONES

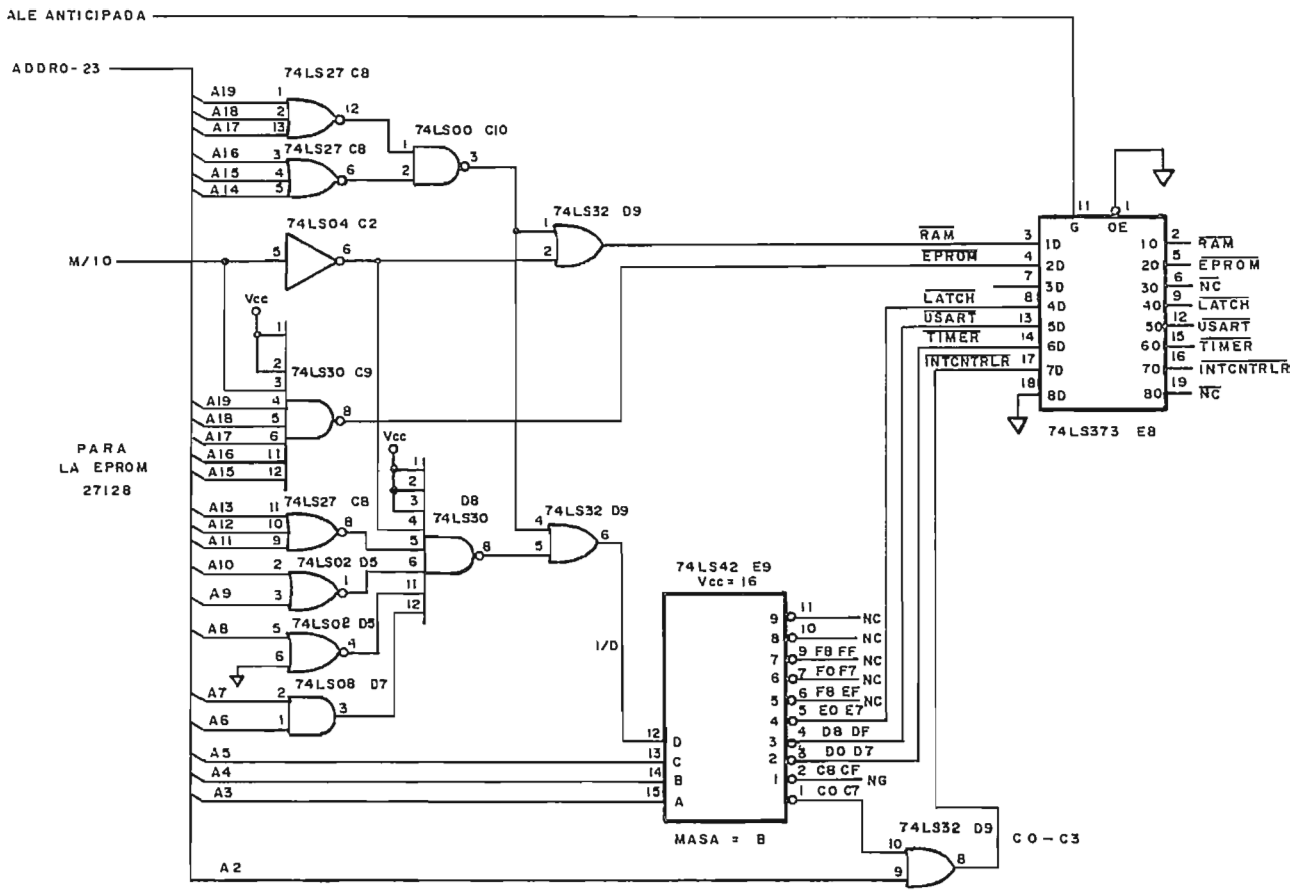


FIG 341 DECODIFICACION DE LA DIRECCION PARA GENERAR LA SELECCION DEL CIRCUITO DE ALTAS PRESTACIONES

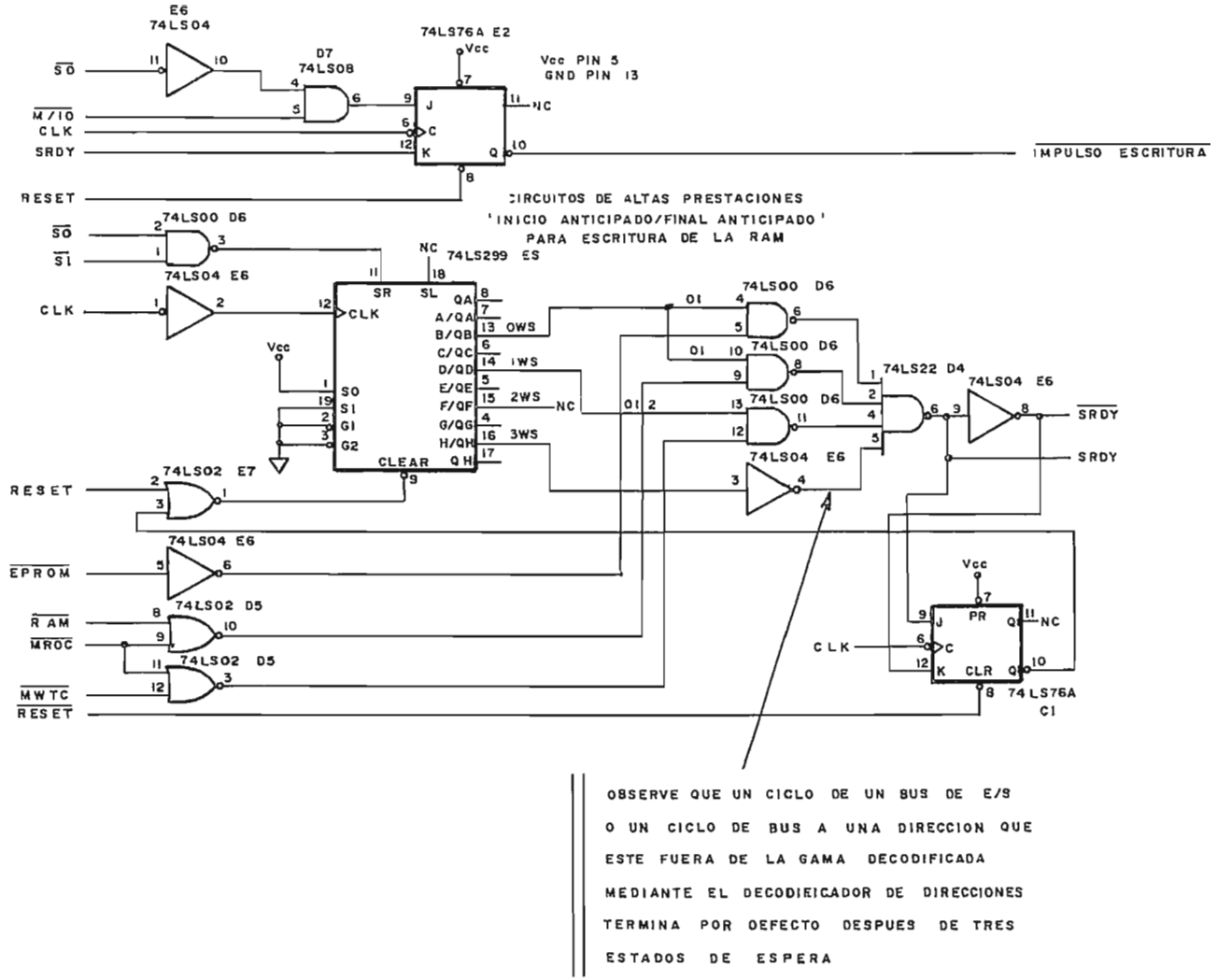


FIG. 3 42 GENERACION DE LA SINCRONIZACION DE ESCRITURA DE ALTAS PRESTACIONES Y GENERACION DE READY

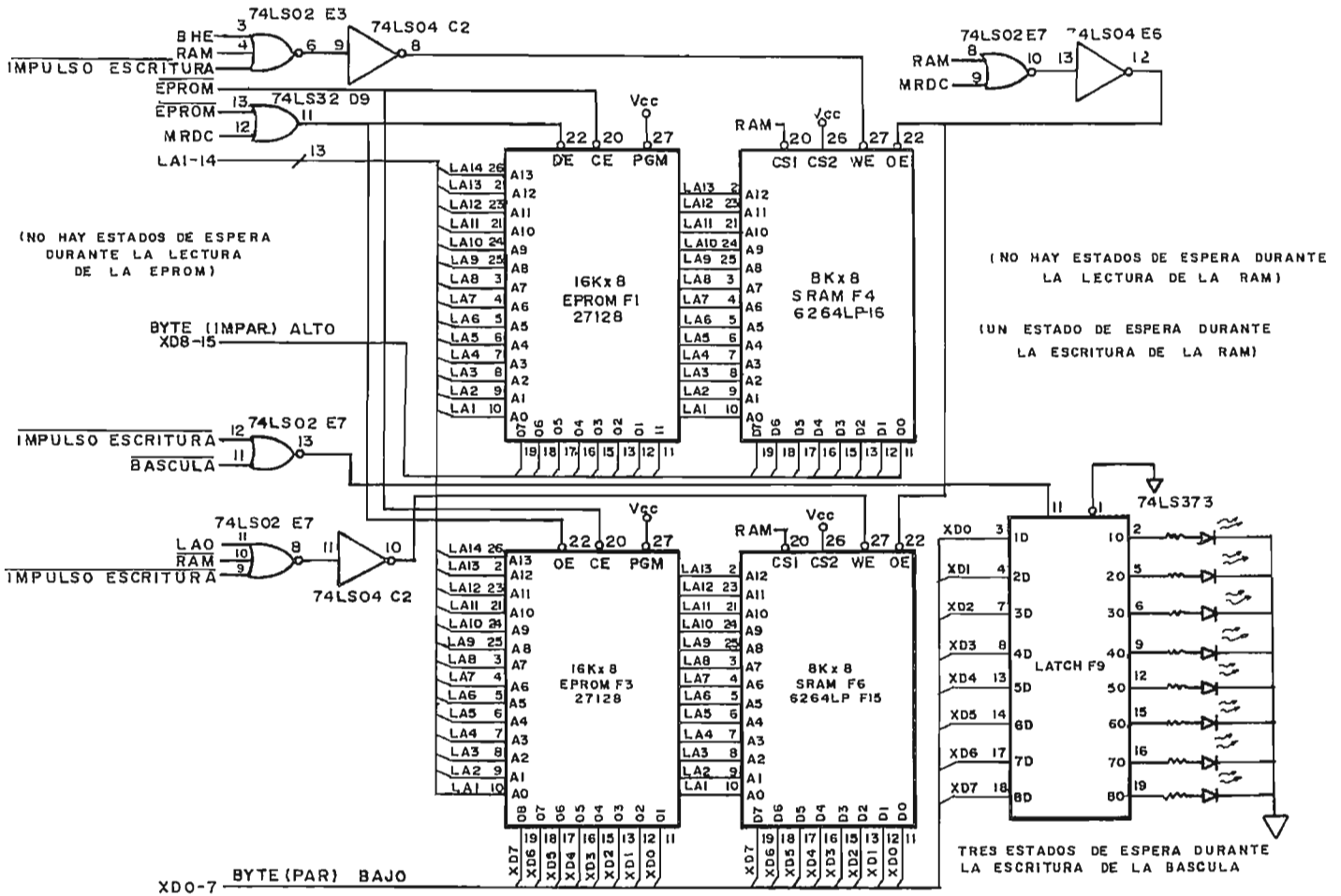


FIG 3 43 EPROM , RAM Y LATCH DE SALIDA A LAS QUE SE PUEDE ESCRIBIR

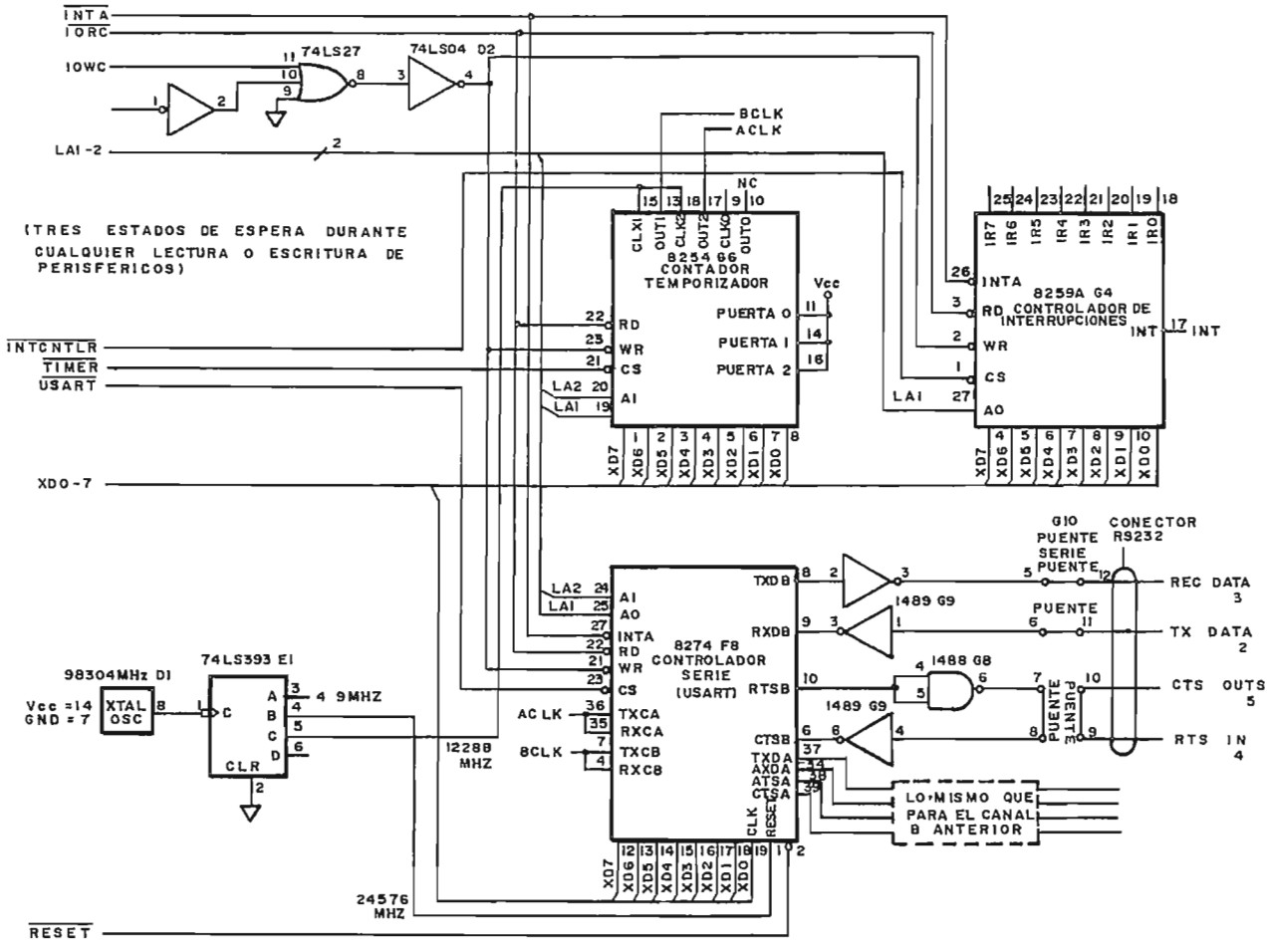


FIG 3 44 CONTROLADOR DE INTERRUPCIONES, TEMPORIZADOR Y PERIFERICOS DE LA USART

Si el circuito de diagnóstico cumple con estos dos objetivos, usted podrá observar los LED para determinar, con un alto grado de confianza, si su circuitería está trabajando correctamente.

3.27 Apariencia de los led de instrumentación

Los circuitos que se han elegido para este ejemplo del sistema con el 80,86 están basado en un cierto número de indicadores de señal, LED, colocados estratégicamente. Se observará como funciona este circuito de diagnóstico (figura 3.45).
En primer lugar, se comprobará la actividad de los ciclos del bus demandando un LED a la señal ALE. Si existe actividad, que es una condición normal, el controlador de bus 82288 dará un impulso de la señal ALE en cada ciclo de bus (aproximadamente a un cuarto de su intensidad total). Desde luego, el LED no se iluminará si la CPU se ha detenido en un ciclo determinado. Mientras observe que el LED ALE está encendido, estará seguro de que los circuitos por lo menos están realizando ciclos de bus.
Aparte del LED en la señal ALE, hay cinco LED de comando, conectados a las señales de comando del controlador de bus 82288. Cuando la CPU funciona, todos o parte de los LED de comando aparecerán encendidos, y tal vez parpadeando ligeramente, tal y como la CPU realiza ciclos de bus.
Por ejemplo, cuantas más veces la CPU realiza ciclos de escritura a la memoria, más brillante aparecerá el LED MWTC. Los cinco LED de comando serán indicadores directos de la actividad de las líneas de comando del 82288. Así, tal y como la CPU avanza en un programa, la intensidad del LED de comando podra fluctuar tal y como varíe la mezcla de tipos de ciclo de bus.
El esquema de la figura 3.45 incluye dos LED adicionales que proporcionan indicaciones especiales que conciernen al estado de la CPU si ésta se detiene actualmente. La única razón legítima por la que el 80286 se detiene, es si su software ejecuta la instrucción HALT (PARO), en cuyo caso la CPU se detiene con toda razón, hasta que se produce una interrupción NMI o una señal de reset (reinicio). Afortunadamente, al detenerse la CPU, genera una forma especial de estado que se ha destacado en la figura 3.16 , para lo cual se aprovechará uniendo algunos dispositivos lógicos y un LED a las líneas necesarias, indicadoras del estado del bus. El circuito que se ha elegido (fig. 3.45) detectará y guardará la condición de paro (HALT) y encenderá entonces el LED HALT.
Todas las otras causas de paro de la CPU indican un problema, y la razón básica puede ser determinada examinando todos sus indicadores de diagnóstico de los circuitos. Por ejemplo, o bien la CPU no ha recibido nunca la señal READY y esta todavía perdida en el ciclo actual de bus, o la CPU ha detectado un error triple masivo y se ha desconectado (deja de intentar la búsqueda de instrucciones). Cuando la CPU está colgada esperando la señal READY, uno de los LED de comando debe estar encendido fijamente, pero la luz ALE no estará encendida para nada. Sin embargo,

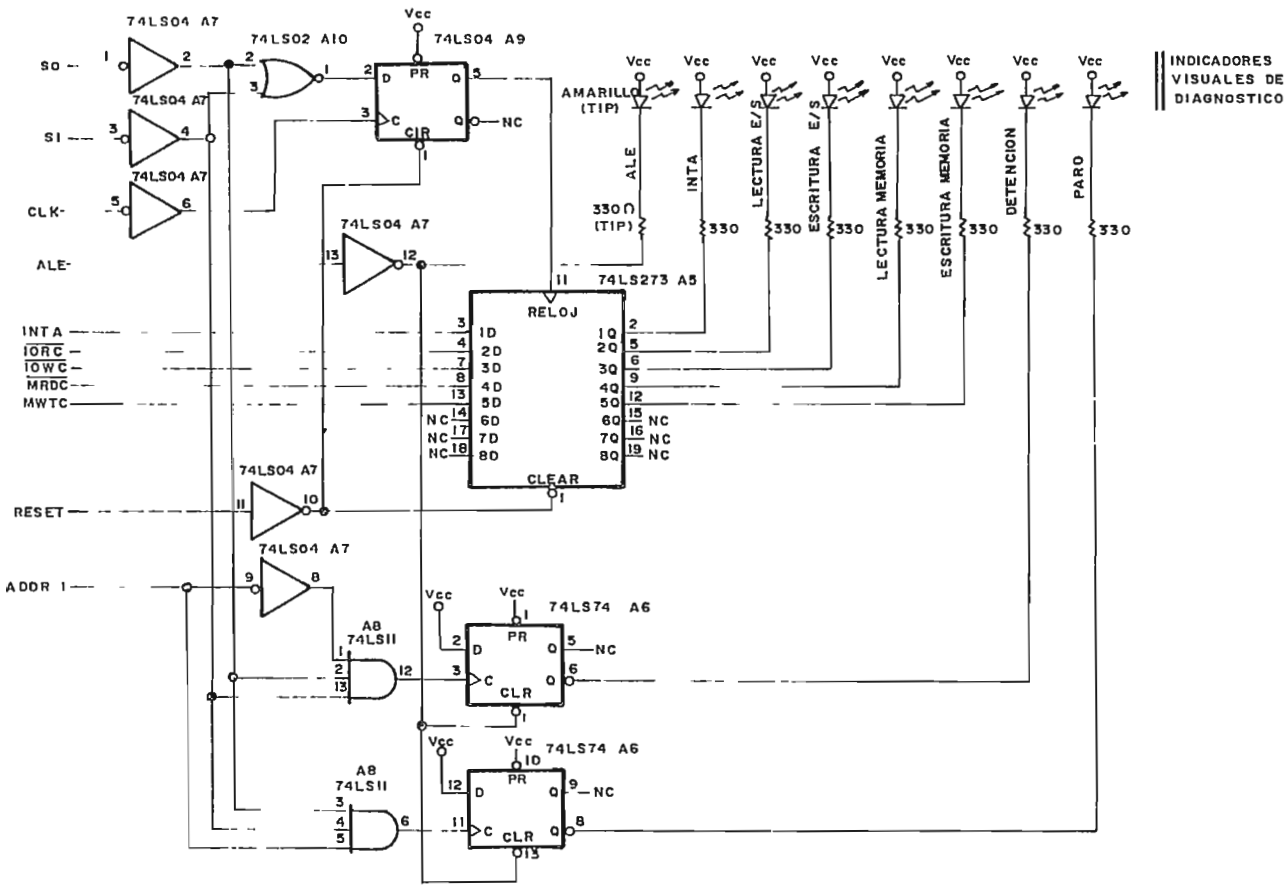


FIG 3.45 CIRCUITERIA DE DIAGNOSTICO (OPCIONAL)

cuando la CPU se ha detenido, el LED DETENCION estará encendido, y todos los otros indicadores estarán apagados.
Observe la figura 3.46 para saber la apariencia de los LED durante el funcionamiento normal (por ejemplo, funcionando o en reposo) y una operación anormal (por ejemplo, colgada o detenida).
Estos indicadores son generalmente tan fáciles de añadir que se pueden aplicar a cualquier nuevo prototipo de circuitos con el PC/104, aunque de tiempo en tiempo se utilicen emuladores sofisticados y analizadores lógicos, usted siempre podrá beneficiarse de la indicación continua y rápida dada por estos indicadores de diagnóstico.

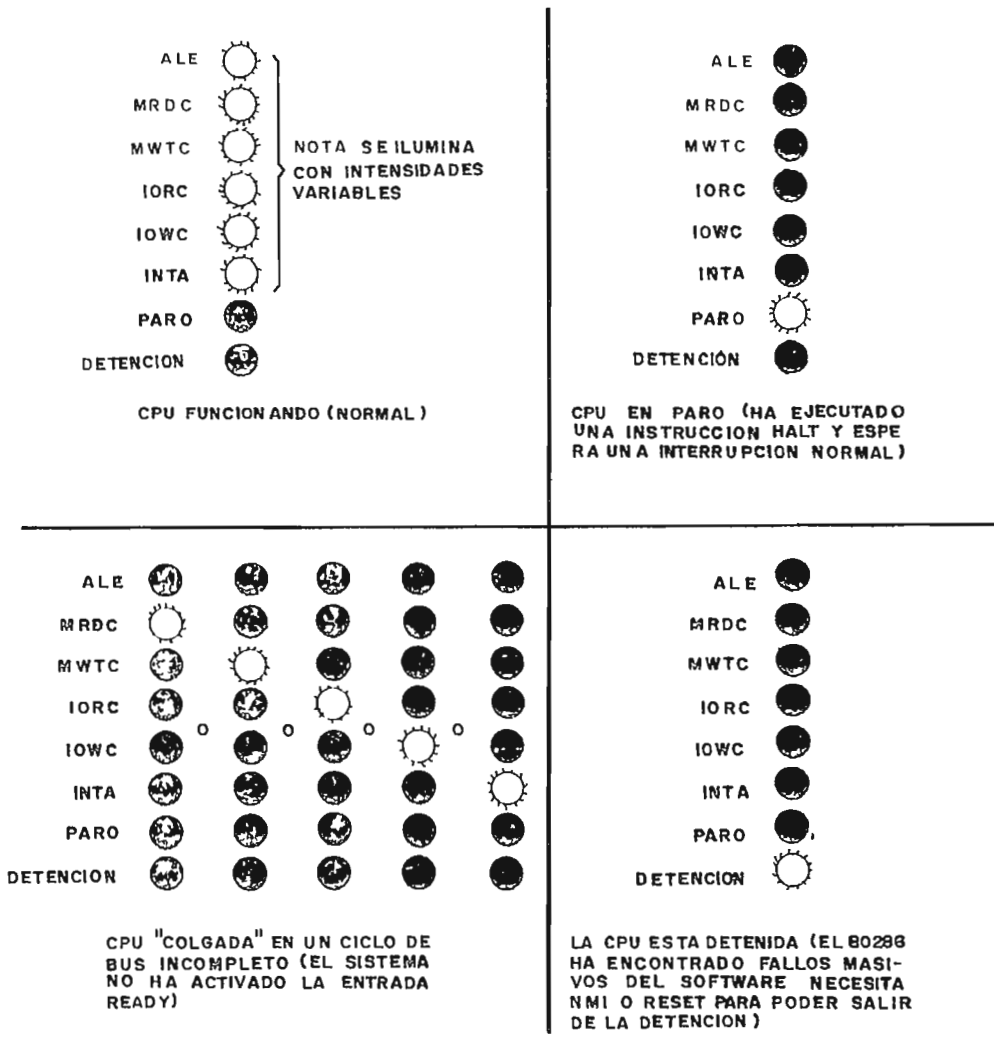
3.28 Software de diagnóstico básico

El software de diagnóstico que se describirá en esta sección está orientado a comprender el funcionamiento y relación de la EPROM, PAL y Latch de salida lo suficiente para que se confíe en el funcionamiento de los circuitos.
Intencionadamente se ha hecho que este ejemplo de software de diagnóstico sea lo más corto posible (de hecho, solamente 31 bytes), para hacerlo lo más comprensible posible. Una forma que se ha utilizado para hacerlo corto es hacer que funcione en un bucle infinito. Aunque es un programa muy pequeño, este bucle continuo hace posible que pueda observar la actividad del sistema utilizando un analizador lógico o también un osciloscopio estándar.
La figura 3.47 es la pieza típica de software de diagnóstico, escrito en el lenguaje ensamblador. Nótese que el propósito principal es ejercitar rápidamente los circuitos del sistema, de forma que se ha elegido el no comprobar extensivamente la CPU o habilitar el modo protegido.

```
;
;IGUALDADES
;
LATCH          EQU  0E0H
SENAL_BUENA    EQU  0AAH
SENAL_MALA     EQU  055H
;


;
;      CODIGO
;
        ASSUME CS:CODIGO_INICIAL
```

Figura 3.47 Código fuente del software de diagnóstico



FIGURÁ 3.46 APARIENCIA DE LOS LED DE INSTRUMENTACION

```
CODIGO_INICIAL      SEGMENT

PUNTO:      MOV     BX,0000H    ; Inicializar registro base con 0
            MOV     DS,BX       ; Inicializar registro DS con 0
            MOV     CX,5473H    ;
            MOV     [BX],CX     ; Escribir 5473 en posicion 0 de R
            JMP     LEER        ; Forzar a la CPU a interrumpir la
                                ;  busqueda, de forma que el valo
                                ;  del dato en la RAM no pued
                                ; quedarse en el bus de datos hast
                                ; que se produzca la lectura
LEER:       MOV     AX,[BX]     ; Leer el dato desde la posicion 0
                                ; de la RAM
            JNE     RAM_MAL     ; JMP si los datos de lectura y
                                ; escritura no son iguales
            MOV     AL,SENAL_BUENA
            OUT     LATCH,AL    ; Sacar AAH si el dato leido ha
                                  sido bueno
            JMP     BUCLE

RAM_MAL:    MOV     AL, SENAL_MAL
            OUT     LATCH,AL    ; Sacar FFH si el dato leido de la
                                  RAM fue malo

INICIO:     JMP     BUCLE       ; Se intenta que este sea el punto
                                ; de entrada despues de un reset de
                                ; la CPU. Por tanto debe estar en l
                                ; direccion fisica FFFFF0H

CODIGO_INICIAL  ENDS
                END
```

Figura 3.37 continuación

3.29 Como funciona el diagnóstico

El software de diagnóstico está diseñado en primer lugar para asegurar que la CPU y el sistema puedan realizar ciclos de bus lal y como la CPU busca el código desde EPROM, implícitament muestra que los ciclos de lectura de EPROM funcionan bien. E programa realizará explícitamente ciclos de bus para escribir leer la RAM. El diagnóstico también comparará el valor del dat leido desde RAM con el dato que ha escrito anteriormente Entonces realiza un ciclo de salida a los latch de salida de bits. Escribirá un AAH (hexadecimal) en los latch si el dato e la RAM se ha comparado satisfactoriamente, pero escribirá un 55 si el dato comparado fue malo. Y eso es todo. Entonces bifurca c nuevo (JMP) al principio y empieza de nuevo, en bucle infinito.

Cuando se ensamble el código fuente, se deberá de obtener e código objeto como el de la figura 3.48 .

```
                        ;
                        ; IGUALDADES
                        ;
00E0H           LATCH           EQU  0E0H
00AAH           SENAL_BUENA     EQU  00AAH
0055H           SENAL_MALA      EQU  055H

                        ;
                        ;       CODIGO
                        ;
                                ASSUME CS:CODIGO_INICIAL

0000            CODIGO_INICIAL  SEGMENT

                                ORG     000H

0000  BB 0000   BUCLE:          MOV  BX,0000H
0003  8E DB                    MOV  DS,BX
0005  B9 5473                  MOV  CX,5473H
0008  89 0F                    MOV  [BX],CX
000A  EB 01 90                  JMP  LEER

000D  8B 07     LEER:           MOV  AX,[BX]
000F  3B C1                    CMP  AX,CX
0011  75 06                    JNE  RAM_MAL
0013  B0 AA                    MOV  AL,SENAL_BUENA
0015  E6 E0                    OUT  LATCH,AL
0017  EB E7                    JMP  BUCLE

0019  B0 55     RAM_MAL:       MOV  AL,SENAL_MALA
001B  E6 E0                    OUT  LATCH,AL
001D  EB EA     INICIO:        JMP  BUCLE

001F            CODIGO_INICIAL ENDS

                                END
```

Figura 3.48 Código de diagnóstico ensamblado

```
Segment and Groups:
            Name                 Size   Align   Combine Class

CODIGO_INICIAL.............. 001F   PARA   NONE


Symbols:
                Name          Type     Value      Attr

FATAL.....................    Number   00E0
MUERE.....................    L NEAR   0000       CODIGO_INICIAL

INICIO....................    L NEAR   0011       CODIGO_INICIAL

LEER......................    L NEAR   000D       CODIGO_INICIAL

RAM_MAL...................    L NEAR   0019       CODIGO_INICIAL

SENAL_BUENA...............    Number   00AA
SENAL_MALA................    Number   0055

      38 Source Lines
      38 Total  Lines
      29 Symbols

47872  Bytes symbol space free

       0 Warning  Errors
       0 Severe   Errors
```

Figura 3.49 Código de diagnóstico ensamblado (continuación)

El paso final que debe dar es tener en cuenta el valor inicial de CS:IP de la CPU después del reinicio. La posición de su punto de entrada del software deberá de coincidir exactamente. La etiqueta INICIO se ha colocado para que sea su punto de entrada. Y después de reiniciar, el CS:IP inicial es F000:FFF0H.
El circuito integrado esta primeramente en el modo real y con este código de diagnóstico permanecerá en él. Recuerde que, en dicho modo, CS:IP genera directamente la dirección de búsqueda del código físico (es decir, sin ningún descriptor), añadiendo CS + IP de la siguiente forma:

```
(CS)     F000
(IP)   +  FFF0
       ------
       FFFF0
```

Inicialmente el 80286 también tiene cuatro líneas de dirección de mayor peso, todas a 1 durante los ciclos de bus relacionados con CS tales como la búsqueda de un código. Así que, hasta que

efectúa un JMP o CALL intersegmento, las cuatro líneas de dirección superiores son unos para los ciclos de bus relacionados con CS.
(La dirección física de la primera búsqueda de código es FFFFF0 y está exactamente en donde debe estar localizado el punto de entrada INICIO deseado.)
El código de diagnóstico es un modulo de código continuo. Despues de ensamblarlo, observará en la figura 3.48 que consiste en 29 bytes (decimal) antes de la etiqueta INICIO . Para colocar la etiqueta INICIO con un desplazamiento 0FFF0H, el modulo de código debe empezar con un desplazamiento de 0FFD3H (0FFF0H-1DH). Insertando la directiva apropiada ORG (origen) dentro del código fuente y ensamblandolo de nuevo, se podra comprobar el cálculo del desplazamiento correcto para colocar la etiqueta INICIO en el desplazamiento FFF0. La figura 3.49 demuestra que FFD3H es el desplazamiento correcto. La etiqueta INICIO esta correctamente en el desplazamiento 0FFF0H, el mismo que el valor inicial IP del circuito integrado.

```
1                               ;
2                               ;IGUALDADES
3                               ;
4= 00E0                         LATCH           EQU  0E0H
5= 00AA                         SENAL_BUENA     EQU  0AAH
6= 0055                         SENAL_MALA      EQU  055H
7
8                               ;
9                               ;     CODIGO
10                              ;
11                                   ASSUME CS:CODIGO_INICIAL
12
13 0000                         CODIGO_INICIAL SEGMENT
14
15 FFD3                                        ORG       0FFD3H
16
17 FFD3 BB 0000                 BUCLE:         MOV  BX,0000H
18 FFD6 8E DB                                  MOV  DS,BX
19 FFD8 B9 5473                                MOV  CX,5473H
20 FFDB 89 0F                                  MOV  [BX],CX
21 FFDD EB 01 90                               JMP  LEER
22
23 FFE0 8B 07                   LEER:          MOV  AX,[BX]
24 FFE2 3B C1                                  CMP  AX,CX
25 FFE4 75 06                                  JNE  RAM_MAL
26
27 FFE6 B0 AA                                  MOV  AL,SENAL_BUENA
28 FFE8 E6 E0                                  OUT  BASCULA,AL
```

Figura 3.49 Código de diagnóstico ensamblado después de dar el origen (ORG) al punto de entrada en FFFFF0H.

Figura 3.49 (continuación)

```
                                        JMP   BUCLE
                RAM_MAL:                MOV   AL,SENAL_MALA
                                        OUT   LATCH,AL
                INICIO:                 JMP   BUCLE

                CODIGO_INICIAL  ENDS
                                        END
```

```
Segment and Groups:
                        Name            Size    Align   Combine class

CODIGO_INICIAL..................        FFF2    PARA    NONE

Symbols:

                        Name            Type    Value   Attr

LATCH...........................        Number  00E0
BUCLE...........................        L NEAR  FFD3    CODIGO_INICIAL
INICIO..........................        L NEAR  FFF0    CODIGO_INICIAL
FFE0............................        L NEAR  FFE0    CODIGO_INICIAL

RAM_MAL.........................        L NEAR  FFEC    CODIGO_INICIAL
SENAL_BUENA.....................        Number  00AA
SENAL_MALA......................        Number  0055

                37 Source Lines
                37 Total  Lines
                29 Symbols

            49872  Bytes symbol space free
                0 Warning errors
                0 Severe errors
```

Figura 3.49 (cotinuación)

3.29 Depuración del núcleo del sistema

Cuando se ha completado el montaje de los circuitos, su CPU debe ser capaz de hacer funcionar el software. Ahora bien, si no lo hace la primera vez, sus herramientas de depuración están a mano. Es decir, éstos son los indicadores de diagnóstico de la circuitería añadidos al núcleo del sistema y el software de diagnóstico que acaba de crear. Pero, de una forma casi increíble, he aquí como puede empezar a encontrar los errores: la

primera vez que pruebe el sistema, no conecte el software de diagnóstico. Solamente pulse el botón de reset y deje que la CPU funcione.
Sin ningún software, la CPU funcionará leyendo cosas sin sentido en el bus y, al cabo de una docena de ciclos más o menos, responderá deteniéndose. Sin ninguna EPROM en el sistema, el LED indicador de paro deberá iluminarse inmediatamente después del reset. Todos los otros LED deberían estar apagados.
Después de instalar las EPROM, una técnica muy buena de depuración sería unir la línea READY a nivel alto (inactiva), de forma que la CPU empiece sus primeros ciclos de bus después del reset y que entonces añada ciclos de espera al primer ciclo de bus para siempre, esperando sin parar para que la entrada READY pase a nivel bajo (activo). Puede probar el circuito a su gusto, comprobando cuando los latch de dirección han almacenado la primera dirección, cuándo el decodificador de direcciones está aplicando una señal de selección del circuito de las EPROM, cuándo las EPROM están emitiendo el código de la primera operación que espera que se ejecute, y cuándo el código de operación se está propagando a través de los transceptores de datos hasta las patillas de datos de la CPU. Puede comprobar todo esto con un simple voltímetro, solamente comprobando los niveles altos o bajos en las patillas de los circuitos integrados.
Una vez corregidos estos errores, puede instalar las EPROM que contienen el código de diagnóstico de la figura 3.49 y conectar la entrada READY a la lógica que genera la señal READY de la figura 3.42 y comprobar los resultados.
Todavía más simple que el diagnóstico de la figura 3.49 puede grabar una EPROM con un código ultrasimple, un diagnóstico de una instrucción que solamente hace un JMP (salto) a sí mismo. Desde luego, todo esto comprueba la habilidad de buscar las instrucciones, pero ya es un inicio. El código ensamblado con un origen (ORG) es ilustrado en la figura 3.50.

```
                                ASSUME CS:CODIGO_SIMPLE
0000            CODIGO_SIMPLE   SEGMENT
FFF0                            ORG     0FFF0H
FFF0 EB FE      INICIO:         JMP     INICIO
FFF2            CODIGO_SIMPLE   ENDS
                                END
```

Figura 3.50 El programa de diagnóstico más sencillo posible: una instrucción de JMP que salta así misma

Cuando las EPROM programadas con el diagnóstico de una instrucción se conectan y la CPU está ejecutando el código, la lamparita ALE se iluminará debido al impulso ALE que se está generando en cada ciclo de bus por el controlador de bus 82288. Pruebe las EPROM que contienen una sola instrucción y compruebe si el sistema funciona (la lamparita ALE encendida). El LED MRDL también deberá estar encendido, puesto que todos los ciclos del bus son de lectura de memoria (para la búsqueda del código

).
Finalmente pruebe las EPROM que contienen el programa de diagnóstico más largo (figura 3.49). Debido a que también debe hacer un lazo sobre sí mismo y funcionar para siempre, la lamparita del LED también debe estar encendida continuamente.
El programa de diagnóstico más largo también escribe un valor a los latch de salida. El valor es una función del test de comparación de los datos en RAM: si todo va bien, el valor que se muestra (en binario en los LED de los latch) será 1010 1010 (AA en hexadecimal). La condición fallo en la comparación se supone que será indicada como 0101 1010 o 55H.

3.30 ¿ Que hacer si el sistema no funciona ?

El sistema de depuración que se ha aplicado es muy gradual: le permite depurar el sistema circuito por circuito. Si existen problemas que evitan que la CPU no pueda hacer funcionar ni el simple programa de una instrucción, los LED de instrumentación deberán de ser una ayuda. Por lo menos deben indicar que la CPU está intentando un ciclo de lectura de memoria (para buscar el código).
Si la CPU deja de funcionar, una de las luces de comando deberá de estar encendida. Esto indica que el ciclo de bus fue iniciado por la CPU y que el 82C88 generó el correspondiente comando, pero por alguna razón los circuitos principales no han activado la señal READY para completar el ciclo del bus.De forma que la CPU todavía está añadiendo estados de espera al ciclo, esperando a que el ciclo termine eventualmente con la señal READY. El que está lamparita en particular permanezca encendida nos dará una pista de qué parte del decodificador de direcciones y circuito de generación de READY se deberá de examinar.
Si la CPU deja de funcionar y la lamparita de paro esta encendida, deben de existir otros problemas. Si la CPU se ha detenido, indica que la información que se estaba leyendo durante los ciclos de adquisición está totalmente confundida por alguna razón. Compruebese el contenido de la EPROM, el cableado del bus de direcciones y del bus de datos y compruebese los transmisores de datos. Para descubrir los problemas de la circuitería que hacen que la CPU se detenga, investiguese el sistema mientras utiliza el programa de diagnóstico más sencillo de la figura 3.50 . Haga que funcione este programa de una sola instrucción antes de utilizar un software complejo.

CONCLUSIONES DEL CAPITULO III

En éste capítulo se ha finalizado analizando en una forma global algunos de los chips de soporte básicos que conforman un sistema alrededor del microprocesador 80286 de Intel, como son el generador de pulsos de reloj 82284, el latch ALS 573, el transceptor de datos ALS 245, el controlador de bus 82288, el chip de memoria PROM 285166, el temporizador programable 8254, el comparador programable ALS 526, el controlador de interrupciones 8259A, el controlador de acceso directo a memoria 82258, y el controlador de memoria dinámica aleatoria DRAM 8207. Así como también la forma como los chip 80286 y 80287 están organizados, además de su interconección . Además, se presentó algunas de sus características avanzadas, así como el manejo de memoria, también como conectar éstos dos chip con otros componentes para formar un sistema completo de microcomputadora.
Además se ha tomado el tiempo de construir un sistema con el 80286, probando dicho sistema atravez de programas sencillos escritos en lenguaje assembler.

BIBLIOGRAFIA

1 Edward Strauss Grupo Waite : 80286 Arquitectura y Sistemas Anaya: 1988. cap 11.

2 Stephen P. Morse : arquitectura del 80286: A Wiley Press Bool Nwe York. 1986. cap 6

3 Brenner, Robert C.: The IBM PC TROUBLESHOOTING AND REPAIR GUIDE Macrotend Inc.: U.S.A.: 1989. cap 4

4 Microprocessor and Perripheral Handbool : vol I: Intel 1987. cap 2 y 3.

5 Microprocessor and Perripheral Handbool : vol II: Intel 1987. cap 1 y 2.

6 Microprocessor Intel 1990. cap 3.

CAPITULO IV

SOFTWARE DE COMUNICACIONES POR MEDIO DEL 8088

Introducción.

Las comunicaciones entre microcomputadoras es un tema que cada día crece en importancia. En los negocios, el correo eléctronico ya esta reemplezando grandemente al telefono y al correo escrito, las bases de datos eléctronicas cada día se utilizan mas y mas, e tele un gran interes en la conexion de microcomputadoras a mainframes. En algunos paises, las redes de comunicaciones interactivas se esta convirtiendo en algo e tremadamente popular. Sin embargo, la información acerca de las tecnicas implicadas no se obtienen tan fácilmente. Los libros que tratan sobre comunicaciones con microcomputadoras tienden a concentrarse a como utilizar o como seleccionar determinado software de comunicaciones, en lugar de como escribir software de comunicaciones. Los libros tecnicos sobre el MS-DOS o las guias de programadores para computadoras personales dan muy poca información sobre comunicaciones.

En este capitulo se pretende documentar los conceptos más importantes sobre el tema de las comunicaciones por medio de microcomputadoras. En la primera parte de este capitulo se tratara sobre comunicaciones en general, incluyendo información acerca de las interfases de hardware, protocolos de software, y modems. La información cubierto se aplica a cualquier maquina que se pueda.

La parte dos de este capitulo tiene que ver con las IMB PC o compatibles, yendo desde el punto de vista del usuario, hasta la programación detallada a nivel del sistema, y por último, la parte 3 trata sobre ejemplos de programación en lenguaje BASIC y en lenguaje de ensamble, como ane os se incluye una versión en C de programas de comunicación

4.1 INTERFASES DE HARDWARE

En general para que dos dispositivos sean capoces de comunicarse, estos se deben de conectar de forma tal que las señales electricas trasmitidas por uno sean recividas por el otro.

Las comunicaciones se pueden llevar a cabo ya sea directamente, con alambres conectados a los dos dispositivos, o indirectamente, con un medio que intervenga. Este medio muy a menudo es el sistema de telefono publico, en tal caso se utilizan **modems** (MOdulador DEModulador) para convertir las señales en un e tremo en señales convenientes para la transmisión a traves de las lineas telefonicas-, y convertirla de regreso en el otro e tremo, para que puedan ser utilizadas. El modem se conecta a la computadora de la misma forma como se conecta cualquier dispositivo serie convensional.

En esta sección se discute la conexion directa entre dos

dispositivos, incluyendo los cables, sockets requeridos, y lo estandares comunmente utilizados para determinar que alambres se utilizan para determinados propositos.

4.1.1 CONECTORES Y SOCKETS

Hay varios tipos diferentes de conectores y sockets para conectar cables a dispositivos serie. Los conectores de 25 y 9 pines tipos D son los que mas comunmente se utilizan (algunas veces son referidos como DB-25 y DB-9) aunque hay otros tipos en uso, como los conectores circulares que se utilizan en algunas computadoras Apple.

Los conectores tipo D (son llamados asi porque la parte que rodea los pines tiene forma de letra D) contienen un cierto numero de pines o solets. Los que tienen pines se conocen como machos y los que tienen sockets como hembras. Cada pin o socket tiene un número impreso para poder describir las conecciones. Posteriormente se iran mostrando las conexiones necesarias para comunicar dos dispositivos por medio del puerto serie RS-232C

4.1.2 EL RS-232-C ESTANDAR

En general para hacer que dos equipos de diferentes manufacturas puedan comunicarse entre si, se han diseñado varios estandares. El mas ampliamente utilizado es el estandar RS-232-C, que fue originalmente diseñado para realizar conecciones entre terminales y modems. Se especifican las caracteristicas electricas de los circuitos entre los dos dispositivos y da nombres y números a los alambres necesarios para unirlos. Los nombres de los circuitos (conexiones) son un poco dificiles de recordar, y en la practica actual se han reemplazado por abreviaciones.

Como ejemplo, se verá la linea 2. Esta linea es conocida oficialmente como BA pero mas comunmente, como TXD (Trasmited Data). De acuerdo al estandar RS-232-C, la línea 2 lleva datos desde el terminal al modem. Por esta razón para operar correctamente, el terminal debe de producir salidas en la linea 2 y el modem recibirlas en la linea 2. Esto quiere decir que la línea 2 es una línea de transmisión para algunos dispositivos y de recepción para otros. Una conexion directa de pin 2 a pin 2 (y asi sucesivamente para los otros pines) se puede hacer solamente cuando un dispositivo transmite sobre la línea 2 y el otro recibe en la línea 2. Si esto no es asi, ambos dispositivos trataran de transmitir sobre la misma línea y la trasmisión no será posible.

DISPOSITIVOS DTE Y DCE

Para impedir que dos dispositivos intenten comunicarse cada uno con el otro por medio de los mismos alambres, los dispositivos se

dividen en dos tipos. Dispositivos tales como terminales las cuales usan el pin 2 como salida se conocen como DTE (Data Terminal Equipment). Dispositivos tales como modems los cuales utilizan el pin 2 como entrada son conocidos como DCE (Data Communications Equipment).

De acuerdo al RS-232-C, los dispositivos DTE deberan de tener conectores macho y los DCE conectores hembra. Sin embargo, los dispositivos no siempre cumplen esta regla y no es inmediatament obvio si un dispositivo es DTE o DCE.

Cuando se sabe que un dispositivo es DTE y el otro es DCE, se puede, al menos en teoria, conectarlos facilmente conectando el pin 7 a el pin 2 y asi sucecivamente para los otros pines. Esto se conoce como una conección **directa.** No todos los dispositivos cumplen con este estandar, desafortunadamente, y como resultado de esto ocurren varios problemas, los cuales se discutiran posteriormente, tambien se tratará con la situación cuando ambos dispositivos son DTE o DCE, pero por ahora se asumira que un dispositivo es DTE y el otro es DCE, y que cada uno suministra las señales requeridas por el otro en el pin correspondiente

4.1.4 COMUNICACION EN UN SENTIDO

Hay 3 conexiones principales que son utilizadas para comunicaciones: la linea 2 (datos desde DTE a DCE), la linea 3 (datos desde DCE a DTE), y la linea 7 (señal de tierra). La señal de tierra sirve como un punto de referencia comun para que la polaridad y el voltaje de las otras lineas pueda ser determinada. En el caso mas simple, donde solo un dispositivo trasmite y uno recibe, solo se necesitan dos alambres: la linea 2 o la linea 3 y la linea 7. La figura 4.1 muestra esta forma simple de comunicación.

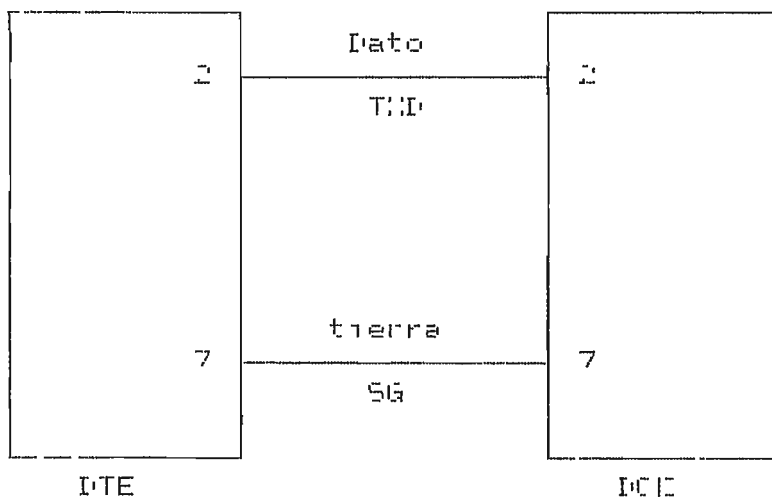


Figura 4.1 Comunicación en un sentido sin handshaking

4.1.5 HANDSHAKING POR MEDIO DE HARDWARE

En muchos casos, es necesario para el dispositivo que transmite saber si el dispositivo que recibe esta listo para recibir información. Por ejemplo, se podría enviar datos al printer, y la velocidad de comunicación puede ser mas rapida que la velocidad del printer. El printer tendrá que ser capaz de de detener al computador para que no envie mas caracteres hasta que se haya impreso el caracter que se ha recibido. Similarmente, se podría enviar datos de un computador a otro, y el segundo computador no pueda procesar los datos tan rapido como llegan.

En cualquiera de estos casos se debe de enviar información desde el dispositivo que recibe hacia el dispositivo que envía para indicar si esta listo o no. Esta información se conoce como **flujo de control o handshaking.**

hay dos tipos de handshaking: de hardware y de software. Ambos tipos implican señales que van desde el dispositivo que recibe hacia el dispositivo que transmite. Con el handshaking de hardware, el dispotivo receptor envía un voltage positivo a través de una conexion dedicada a handshaking siempre que este listo para recibir. Cuando el computador que transmite recibe un voltaje negativo, sabe que debe detener el envio de datos. Con el handshaking de software, descrito en la sección 4.3, la señal de handshaking consiste de caracteres especiales transmitidos a través del circuito de datos y no a través del circuito de handshaking.

Para incorporar handshaking de hardware, se necesita la conexion de al menos un alambre adicional para la señal de carry. Con esto el numero de alambres se eleva a 3: transmisión de datos, tierra y handshaking .

4.1.5.1 DTE a DCE

Cuando un dispositivo DTE esta transmitiendo a un dispositivo DCE, la linea 2 es utilizada para los datos y la linea 7 lleva la señal de tierra. Un dispositivo DCE normalmente controla la transmision del dispositivo DTE utilizando para handshaking la linea 6, conocida como DSR (Data Set Ready). Si el impresor es un DCE y el computador es un DTE, el pin 6 del impresor debe de ser conectado al pin 6 del computador, y el printer mantendrá un voltaje positivo en el pin 6 mientras sea capaz de recibir datos. Cuando se desea que el computador detenga el envio de datos, habrá una caida de voltaje en el pin 6 a un estado negativo.

A menudo se necesita una segunda linea de handshaking, la linea 5, CTS (Clear To Send) es tambien utilizada por un dispositivo DCE para controlar la transmisión de un dispositivo DTE. Cuando se utilizan dos lineas para handshaking, el dispositivo DTE debe de ser diseñado para transmitir solo cuando ambas líneas esten en alto, o positivas. Algunas veces las lineas tienen diferentes significado. Por ejemplo, para un impresor, una podría decirle al dispositivo transmisor que detenga la transmisión hasta que cierta cantidad de datos haya sido impresa, y la otra podría

indican que el printer esta sin papel. Sin embargo, estos significados no son estandarizados. Muchas computadoras son programadas para no transmitir a menos que ambas lineas esten en alto. Aun los impresores que no le dan especial significado a la segunda linea deberan al menos mantenerla en un voltage positivo: no todos lo hacen, algunas veces la segunda linea debe de ser falseada conectandola a la primera.

La figura 4.2 ilustra una comunicación en un sentido desde un dispositivo DTE a un DCE con dos alambres para handshaking.

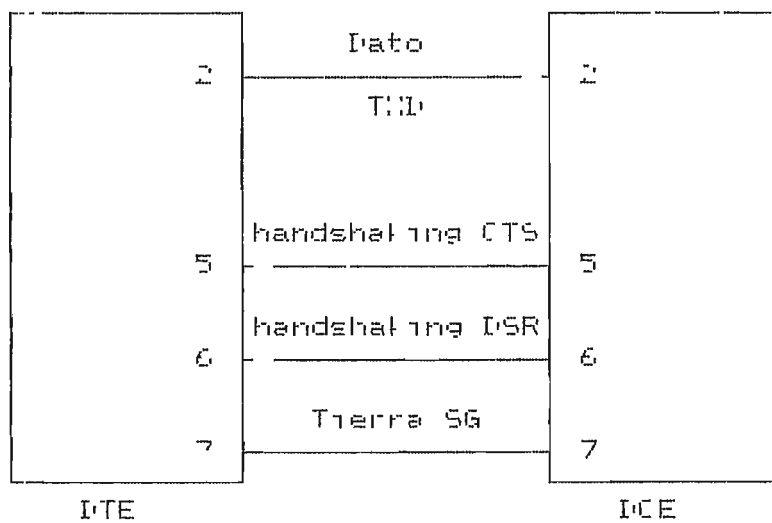


Figura 4.2 Comunicación en un sentido de DTE a DCE

4.1.5.2 DCE a DTE

En general para que un dispositivo DCE se comunique con un DTE, la linea 3 debe ser utilizada para la transmisión de datos, y si se requiere handshaking, la linea 20 se utiliza para enviar el handshaking desde el dispositivo DTE hacia el DCE. La linea 20 se conoce como DTR (Data Terminal Ready). La linea secundaria de handshaking es la linea 4, RQS (Request to Send). La figura 4.3 ilustra la comunicación desde un dispositivo DCE hacia un DTE con handshaking.

4.1.6 COMUNICACIONES EN DOS SENTIDOS

Los datos a menudo se transmiten en dos sentidos. Esto ocurre cuando dos computadoras se comunican una con otra, pero tambien ocurre en la comunicación entre dos dispositivos cuando se este utilizando handshaking por medio de software. El mínimo número de alambres utilizado en comunicaciones en dos sentidos es tres: las de datos transmitidos en cada una de las direcciones y la señal de tierra. Si se agrega una linea para handshaking en cada

dirección el total de líneas es 5, como se muestra en la figura 4.4

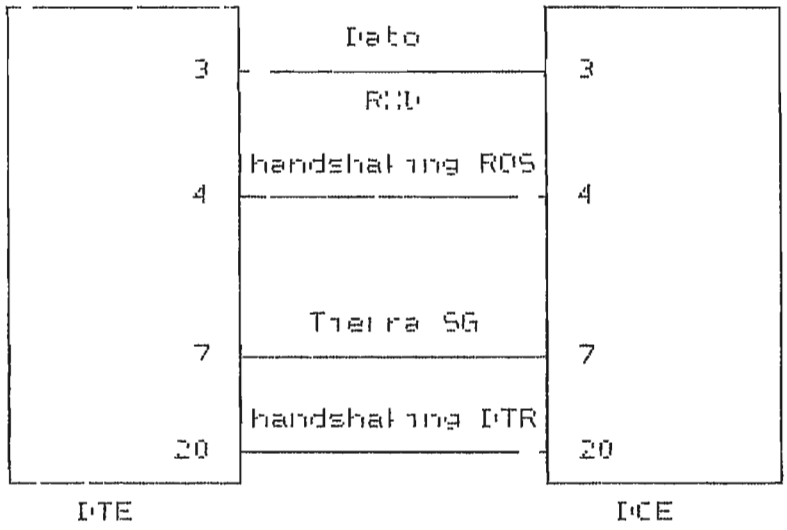


Figura 4.3 Comunicación en un sentido de DCE a DTE con handshaking

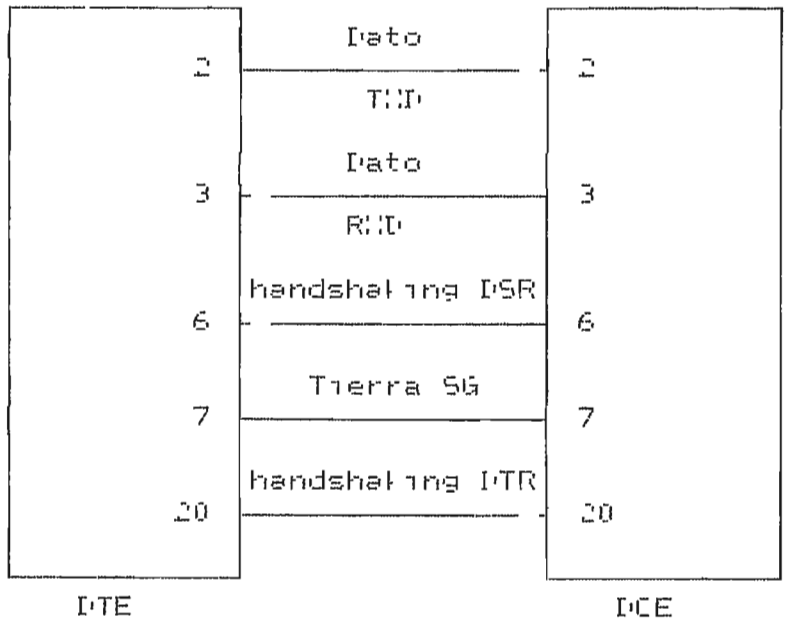


Figura 4.4 Comunicación serie en dos sentidos con handshaking principales

Cuando se utilizan líneas adicionales de handshaking en cada dirección, el total de líneas se eleva a 7. A menudo se utilizan dos líneas adicionales que habilitan la modem para dar mas información a la computadora. La línea CD (Carrier Detect) que se conecta al pin 8, y que se utilizará para indicar la presencia de la señal de carrier, que posteriormente se e plicará, tambien se utiliza la línea RI (Ring Indicator) que se conecta al pin 22, e indica que el modem esta siendo llamado por un dispositivo remoto. En estas condiciones el total de cone iones se eleva a 9, como se muestra el la figura 4.5.

Aunque e iste muchas otras cone iones definidas para el RS-232-C, esta forma con nueve cone iones es la mas comun y son las unicas que normalmente se conectan a un computador. Es por esto que los microcomputadores tienden a utilizar conectores de nueve pins en lugar de 25. en los ane os se muestra un pin out completo del puerto serie

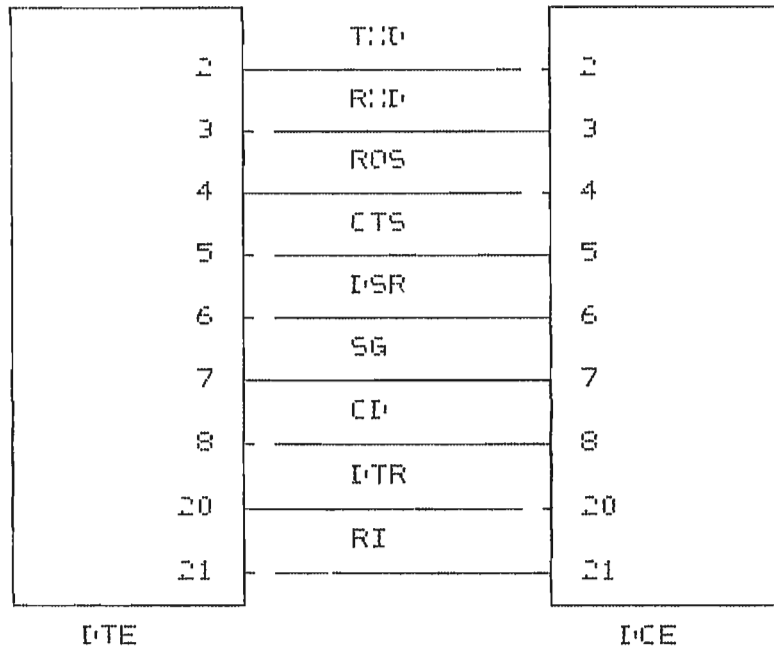


Figura 4.5 Las nueve conecciones del RS-232-C

4.1.7 MODEMS NULOS

Como se mencionó anteriormente, el RS-232-C fue originalmente diseñado para las comunicaciones entre una terminal, la cual es DTE y un modem, el cual es DCE; sin embargo, las aplicaciones se han e tendido a cone iones entre muchas otras clases de dispositivos que oficialmene no se consideran DTE o DCE, tales

como microcomputadoras e impresores.

Puesto que no hay estandares que indiquen si ciertos dispositivo deberan ser DTE o DCE, a menudo se tienen que conectar dos dispositivos DTE o DCE . En este caso se debe de conectar el pin 2 del primer dispositivo al pin 3 del segundo, y el pin 3 del primero al pin 2 del segundo. Las lineas de handshaking deberan ser cruzadas de la misma forma.

Las lineas se pueden conectar ya sea utilizando cables que conecten los puntos indicados de las lineas, o se puede comprar un conector especial que conecta ambos dispositivos y ejecute los cruces necesarios internamente. En cualquiera de estos dos casos, la intervención de cables o conectores, se llaman **modems nulos** ya que toman el lugar de dos modem el alambrado par un modem nulo se muestra en la figura 4.6

4.1.8 SEÑALES ELECTRICAS

El RS-232-C estandar pone las caracteristicas de las señales electricas utilizadas en conecciones serie. Se permiten solo dos estados: ESPACIO, que corresponde al binario 0, cuando un voltaje positivo existe, y MARCA que corresponde al binario 1, cuando existe un voltaje negativo.

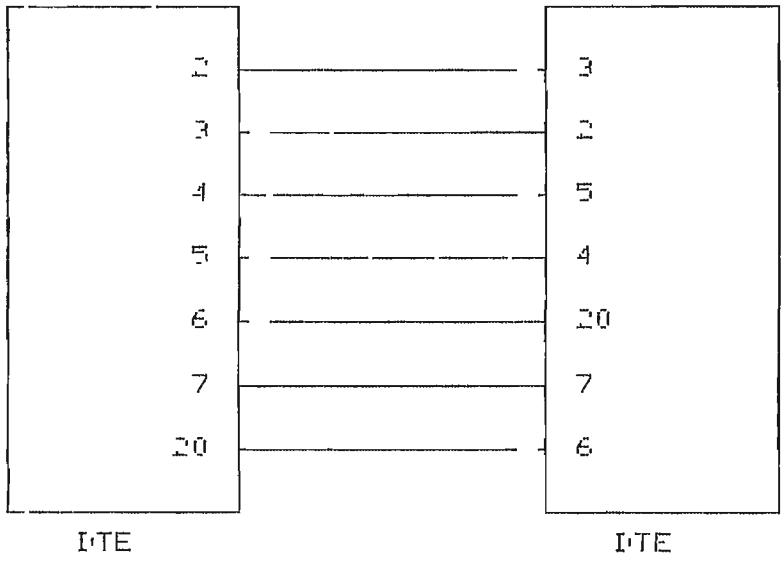


Figura 4.6 Conexiones de un modem nulo

En las lineas de datos (en las lineas 2 y 3), un voltaje positivo (ESPACIO) corresponde a un logico 0, y un voltaje negativo (MARCA) corresponde a un logico 1. En las lineas de handshaking (DTR y DSR), un voltaje positivo (ESPACIO) indica que la linea esta encendida (ON) significado " continue enviando", y un voltaje negativo (MARCA) significa "detenerse"

Los voltajes positivos (el estado ESPACIO) estan entre +5 y + 15 voltios para las salidas, y entre +3 y +15 voltios para las entradas. Estas diferencias son permitidas para tomar en cuenta las perdidas de voltage en las lineas. Los voltages negativos (el estado MARCA) estan especificados entre -5 y -15 voltios para salidas y entre -3 y -15 voltios para entradas.

Hay que notar que si la longitud del cable es demasiado grande, los niveles de voltage caen fuera de los limites permitidos. Inclusive, durante la transmisión se forman capacitancias que afectan la calidad de la señal, debido a las transiciones de niveles de negativo a positivo y viceversa. El RS-232-C no esta diseñado para utilizarse en grandes distancias. La maxima distancia que se considera segura es de 50 pies. Si los dispositivos que se desean comunicar estan colocados a una distancia mayor, se debera de utilizar un modem u otro medio de comunicación.

4.1.9 COMO DETECTAR ALGUNAS POSIBLES FALLAS

Las siguientes son algunas recomendaciones que seran utiles a la hora de resolver problemas que se puedan encontrar mientras se esta conectando dos dispositivos serie:

4.1.9.1 Cuando se necesita un moden nulo?

Como se menciona anteriormente, un modem nulo se necesita siempre que se esten conectando dos dispositivos que son ambos DTE o DCE. En estos casos no se puede transmitir con conexiones directas (2 con 2, 3 con 3, etc) si no que se debe de utilizar un modem nulo.

4.1.9.2 Problemas con el handshaking

Si un printer no responde al dispositivo que transmite, podría se que el printer requiera de dos lineas de handshaking esten altas, mientras que la computadora esta suministrando solo una. Esto es un caso comun con las IBM PC, las cuales son capaces de suministrar las dos lineas pero, a no ser que se hayan programados especialmente, suministra solo una. Este problema se puede resolver colocando un puente en el extremo que corresponde al printer de manera que un mismo alambre proporcione la señal requerida en las dos lineas de handshaking, esto se conoce como falsear un linea.

Si el computador no esta suministrando alguna o ninguna de las señales de handshaking, y el impresor insiste en recibir una o las dos, se puede falsear mas lineas, es decir cualquier linea que este en alto del lado del printer se puede conectar a otra(s) del mismo printer que necesiten una señal alta

Si el computador no esta enviando cuando lo debería de estar haciendo, es probable que este esperando la señal de handshaking la cual no se recibe. Si el printer esta enviando solo una señal de handshaking se puede falsear la otra señal en el extremo del computador.

4.1.9.3 Utilización de tester RS-232-C en linea.

Si se piensa contruir una interfase, se recomienda utilizar un tester RS 232-C en linea. Este pequeño y barato dispositivo (menos de 10 dolares), tiene dos conectores tipo D que pueden ser insertados entre los dispositivos que se desea comunicar. Se tiene una luz para cada pin que se enciende si hay señal en el respectivo alambre. El tester permite por lo tanto ver cuando los datos estan siendo transmitidos, y en que linea, y tambien cuales lineas de handshaking llevan voltages positivos.

4.1.9.4 Otros problemas

Si un dispositivo esta recibiendo basura, el problema esta probablemente es uno de los caracteres de transmision, en la sección 4.2 se dará mas información.

4.2 TRANSMISION DE CARACTERES

En esta sección se explicara como se codifican los caracteres individuales y se envian a traves de los cables. Los principios que se discuten aqui, son aplicables ya sea a las señales enviadas a traves de las lineas telefonicas entre modems, o señales enviadas a traves de cables entre computadoras.

4.2.1 CODIFICACION DEL TEXTO

Cuando se tiene almacenado texto (caracteres alfabeticos, marcas de puntuacion, etc) en la computadora, cada caracter diferente es representado por un número diferente. Estos números normalmente estan en el rango de 0 a 127 , o desde 0 a 255. Puesto que un byte puede tener un valor desde 0 a 255, es natural colocar un byte para cada letra o marca de puntuación en los datos de texto.

Hay dos diferentes convensiones para representar caracteres por medio de numeros: EBCDIC (Extended Binary Coded Decimal Interchange Code), el cual se utiliza por las computadoras IBM que no son de la serie IBM PC, y el codigo ASCII (American Standar Code for Information Interchange), el cual es utilizado por la mayoría de las otras computadoras. En este trabajo solo se tratara con el codigo ASCII, puesto que nuestro interes esta en la computadoras personales.

La tabla oficial ASCII da un número entre 32 y 126 a todos los

caracteres comunmente utilizados (letras mayusculas y minusculas), números, signos de puntuación y otros simbolos. los números del 0 al 31 y el 127 tienen significados especiales tales como retorno del carro, avance de linea, y otros caracteres no desplegables.
Por ejemplo, la letra mayuscula A esta almacenada como el decimal 65.
Puesto que el número 127 en binario utiliza siete bits, todos los caracteres que se representan con números desde 0 a 127 pueden ser almacenados en un byte, dejando un bit extra. Los bits en un byte se numeran del 0 al 7 (0 el menos significante) se puede ver que el codigo ASCII oficial utiliza solo los bits del 0 al 6. El bit 7 se ahorra.
La mayoria de las computadoras utilizan completamente los 8 bits de cada byte que se utiliza para codificar caracteres, dando un total de 256 combinaciones. Los primeros 128 bytes se utilizan para el codigo ASCII oficial, y los restantes son utilizados para caracteres extranjeros, simbolos matematicos, y caracteres graficos, etc. como el diseñador desee. Desafortunadamente no hay un estandar para esos caracteres extendidos, los cuales tienen diferentes significados en diferentes computadoras o en diferentes programas.

4.2.2 CARACTERES ASCII ESPECIALES

Los primeros 32 codigos ASCII no representan caracteres imprimibles, pero tienen significados especiales. Muchos de ellos fueron especialmente diseñados para comunicaciones. Algunos de ellos existen por razones historicas.
Los codigos del 1 al 26 tambien se conocen como Ctrl-A a Ctrl-Z, y normalmente pueden ser generados en un teclado de computadora sosteniendo la tecla Ctrl y presionando la letra apropiada al mismo tiempo. Algunos de los codigos tambien pueden ser generados presionando teclas especiales para esos codigos, tal como Tab para el codigo 9 o Enter para el codigo 13.

Los codigos ASCII especiales se listan a continuacion.

Numero	Caracter	Descripcion
0	NULL	Un metodo de causar deliberadamente un retardo. Cuando un printer es lento, es necesario enviar nulos despues de cada retorno del carro para permitirle al printer que el carro retorne al extremo izquierdo de la pagina.
1	SOH	Inicio de encabezado: indica que el siguiente texto es parte de un titulo.
2	STX	Inicio de texto: indica el inicio del texto actual o del mensaje.

3	ETX	Fin de texto.
4	EOT	Fin de transmision.
5	ENQ	Preguntar: Se utiliza como parte de de una secuencia de handshaking de software para preguntar al computador que recibe si reconoce la recepción de un mensaje.
6	ACK	Reconocimiento de recepción de un mensaje
7	BEL	Sonido del la campana del terminal
8	BS	Back Space (retroceder y borrar caracter caracter anterior
9	HT	Tab horizontal
10	LF	Avance de linea: causa un salto a la misma posición en la siguiente linea
11	VT	Vertical Tab
12	FF	Avance de pagina
13	CR	Retorno del carro: mueve al inicio de la linea, algunas veces tambien causa un avance de linea, pero esto varia.
14	SO	Cambio: Marca el inicio de una secuencia
Numero	Caracter	Descripcion
		de codigos de control especiales. ESC se utiliza a menudo actualmente (ver abajo)
15	SI	Cambio: Marca el fin de una secuencia de codigos de control iniciada con SO.
16	DLE	Escape de cadena de datos: similar a Esc
17	DC1	Dispositivos de control del 1 al 4: Son
18	DC2	cuatro codigos que se utiliza como se
19	DC3	desee, a menudo se utilizan en handshaking
20	DC4	de software.
21	NAK	Reconocimiento negativo: indica que la trasmision no fue recibida correctamente. Por ejemplo, un error de paridad puede haber sido detectado.

22	SYN	Sincronía no ocupado: similar a NULL pero utilizado en comunicaciones sincronas para mantener un dispositivo sincronisado durante la trasmision. Las comunicaciones sincronas seran descritas en las pro imas secciones.
23	ETB	Fin de bloque de trasmision: Utilizado cuando la trasmision esta dividida en bloque para detectar errores.
24	CAN	Cacelar: No tomar en cuenta el dato empleado
25	EM	Fin de medio: Indica que se apro ima el fin del papel o cinta.
26	SUB	Sustituto: corrige un caracter erroneo enviado. En la practica tambien se utiliza para indicar fin de transmision.
27	Esc	Escape: Indica el inicio de una secuencia de caracteres con significados especiales para el receptor.
28	FS	Separador de archivo — Marcan fronteras
29	GS	Separador de grupo — entre segmentos
30	RS	Separador de registro — de te to
40	US	Separador de unidad —

4.2.3 CODIFICACION DE MATERIAL NO TEXTUAL

No todo el material alamacenado en una computadora esta en forma de te to. Las instrucciones de programas, datos numericos, e imagenes graficas, por ejemplo, no se almacenan en forma ASCII.

Estos tipos de datos normalmente se codifican en forma tal que se utilicen los 256 posibles valores para un byte. Los numeros se almacenan en forma binaria y pueden ser representados por varios bytes. Las instrucciones de programas a menudo consisten en uno o dos bytes. En el conte to de las comunicaciones, a este tipo de material se le conoce como **datos binarios**, aun que los te tos tambien esta almacenados en forma binaria.

Cuando los bytes almacenan datos no te tuales, puede que algun valor corresponda a un número que tiene un significado especial en la tabla ASCII. Esto puede causar complicaciones si se estan transmitiendo datos y el dispositivo receptor interpreta un dato no te tual como por ejemplo, fin del mensaje. En este caso, los datos no pueden ser enviados en su forma original por que un byte en medio del mensaje podría accidentalmente corresponder a el simbolo par fin de mensaje y el dispositivo receptor terminaría con la comunicación.

Por este motivo se han desarrollado varios protocolos para evitar este problema: algunos de estos protocolos se cubrirán en las secciones posteriores.

4.2.4 CONVERSION A FORMA SERIE

Casi todas la computadoras almacenan y manipulan sus datos en forma paralela. Esto significa que cuando un byte es enviado de una parte de la computadora a otra este no se envía como un bit a la vez, si no que todos lo bits se envían al mismo tiempo por medio de varias líneas en paralelo.

El número de bits enviados al mismo tiempo varía de maquina a maquina pero normalmente es ocho o un multiplo de ocho, por lo tanto un computador puede trabajar con al menos un byte y a menudo con dos o mas byte al mismo tiempo.

Puesto que la comunicación de una computadora a otra se realiza en forma serial, esto significa que los datos son enviados un bit al mismo tiempo, a una interfase serie que debe tomar los datos recibidos en paralelo, y enviarlos fuera por medio de bits individuales separadamente.

Como se vio anteriormente, las líneas de datos en las comunicaciones serie pueden estar ya sea en la condición de MARCA o en ESPACIO, la cual en las condiciones directas es equivalente a voltage negativo o positivo, respectivamente. Cual quier dato trasmitido deberá primero ser trasladado a una secuencia de marcas y espacios. Para propositos de este traslado, una MARCA representa un uno, y un ESPACIO representa un cero.

4.2.5 COMUNICACIONES SINCRONAS Y ASINCRONAS

Una vez que los datos son convertidos a forma serial, hay dos formas en que pueden ser transmitidos: **Sincronicamente o Asincronicamente.**

Cuando los datos son trasmitidos por algo que se escribio en el teclado, estos son enviados y recibidos asincronamente. Una persona que escribe en el teclado no puede escribir en forma continua, a un solo paso; por lo tanto cuando el computador recibe las letras, hay distintos espacios entre cada caracter. Si las letras individuales estan siendo transmitidas serialmente a medida que son escritas, los espacios irregulares entre cada caracter hace imposible para el dispositivo receptor, despues de recibir un caracter, saber exactamente cuando el otro caracter va a llegar. A causa de esta falta de continuidad, es necesario colocar bits extras antes y despues de cada caracter para indicar al dispositivo receptor el inicio y el fin del caracter. Estos bits extra son conocidos como **bit de inicio (start bit) y bit de parada (stop bit).** Otro bit que se puede añadir es conocido como **bit de paridad,** que a menudo se añade para detectar errores en la trasmision (descritos mas adelante). Este metodo es conocido como **comunicaciones asincronas.**

Cuando los caracteres son enviados en un bloque a velocidad de máquina, estos pueden ser espaciados regularmente, y ya no será

necesario para cada caracter tener un bit de inicio y de parada, ya que una vez que el primer byte ha sido recibido el dispositivo receptor puede predecir exactamente cuando llegará el siguiente caracter. En otras palabras, se puede sincronizar el mismo con la computadora que transmite. Este metodo es conocido como **comunicación sincrona.**

La mayoria de los dispositivos con los que una computadora se comunica son por si mismos asincronos. Por lo tanto, en resto de esta sección se dedicará a las comunicaciones asincronas.

4.2.6 FRAMING (ESTRUCTURA)

En el caso de las comunicaciones serie asincronas, los bits que representan a un byte, los cuales son conocidos como **bits de datos**, son precedidos por el bit de inicio y seguidos por el bit de parada, y el bit de paridad, los cuales se describen completamente en esta seccion. Este proceso es conocido como **framing.**

El número de bits que representan a cada caracter varía de acuerdo a el protocolo de comunicación en uso. Este número se conoce como el número de bits de datos, o la **logitud de la palabra.** Normalmente es de 7 o de 8 bits. Cada caracter es enviado en un grupo formado por el bit de inicio, el caracter (los bits de datos), un bit opocional de paridad, y uno o mas bits de parada. Por razones de claridad, se hará referencia a cada grupo que consistirá de un caracter y sus bit asociados como una **extructura (frame)** esto es para evitar la confusión que se pueda dar cuando la palabra caracter se referirá algunas veces a los bits de datos y algunas veces a el grupo completo con los bits de inicio, parada y paridad.

4.2.6.1 BITS DE INICIO

Siempre se agrega un bit de inicio al principio de una estructura para alertar al dispositivo receptor que los datos estan llegando y para sincronizar el mecanismo que separa los bits individuales. Un bit de inicio es un ESPACIO, o binario 0.

Con una conección directa, un ESPACIO o 0 es transmitido como un voltage positivo. El voltage entre las estructuras es negativo. Por lo tanto en el inicio de cada estructura, el voltaje cambia de negativo a positivo.

4.2.6.2 BIST DE DATOS

Los estandares de comunicaciones series llamados **protocolos**, permiten la transmision de diferentes longitudes de caracteres, o palabras. El software de comunicación pregunta para seleccionar la longitud de la palabra, y pregunta si se desea enviar caracteres de 7 u 8 bits.

Los bits de datos son transmitidos del menos significante primero hasta el mas significante por ultimo.

4.2.6.3 BIT DE PARIDAD

El chequeo de paridad es un metodo para probar si la transmisión se ha recibido correctamente. El dispositivo emisor agrega un bit de paridad, el cual es calculado de acuerdo al contenido del bits de datos. El dispositivo receptor chequea que el bit de paridad indique si la relación entre los otros bits es correcta. Si no es asi, algo debe de haber ocurrido durante la transmision. La paridad puede ser establecida de cualquiera de las formas que se discute a continuacion.

Paridad Par: La paridad par significa que sumando los bit de datos y el bit de paridad resulta un número par. Por ejemplo, la letra A en binario es 01000001. Cuando se suman estos bit se obtiene 2, un número par. Puesto que el total de bits deve de ser par, el bit de paridad debe de ser 0.
Si la letra A se recibe con el bit de paridad en uno, eso indica que un error debe de haber ocurrido durante la transmision.

Paridad impar: La paridad impar significa que de la suma de los bit de datos y el bit de paridad debe de resultar un número impar. Asi, de nuevo utilizando la letra A, el bit de paridad deberá de ser puesto a 1, para que la suma total de los bit de datos sea 3, un numero impar.

No paridad: No siempre se utiliza un bit de paridad, y a menudo es ignorado por el dispositivo receptor aun cuando se utiliza. Todo depende de como se hayan programado los dos dispositivos. No paridad significa que no hay bit de paridad.

4.2.7 BITS DE PARADA

Al final de cada estructura, se envian bit de parada. Esto pueden ser uno, uno y medio, dos bits. **Uno y medio bits** significa que la longitud del bit es mayor que la normal. El bit de parada se emplea de forzar al mínimo los espacios entre estructura. Estos bits son enviados como binarios 1, lo cual en una conexión directa, es igual a un voltaje negativo.
Siempre hay al menos un bit de parada. Esto asegura que haya un voltaje negativo por al menos un periodo de tiempo entre dos estructuras de manera que la siguiente estructura sea reconocida por el bit de inicio positivo. Generalmente se utiliza mas de un bit de parada cuando el dispositivo receptor requiere tiempo extra antes de que pueda manejar en siguiente caracter que venga. Usualmente se utiliza dos bit a 110 baudios, la cual es la tasa de transmision mas baja para usos generales. Esto es para lograr consistencia con las antiguas terminales de teleproceso que necesitan tiempo extra para procesar caracteres.

4.2.8 TASA DE ENVIO (BAUD RATE)

La tasa de envio es la longitud de la señal mas corta dividida

un un segundo. BPS (bit por segundo) significa el número de digitos binarios transmitidos en un segundo.
Cuando cada nuevo caracter es recibido, el dispositivo receptor es resincronizado. Por lo tanto, cada bit de inicio se necesita para indicar el principio de una nueva estructura y disparar el mecanismo utilizado por el dispositivo receptor para leer e interpretar los bit que siguen.
Las tasas de Bps estan generalmente en series de 110, 150, 600, 1200, 2400, 4800, 9600, 19200 baudios. Las tasas mas comunes para comunicaciones con modems son 300 y 1200, 1200 es comun para comunicaciones entre la computadora y el impresor.

4.2.9 POSIBLES FALLAS

Cuando se desea comunicar dos dispositivos, estos deberan estar sincronizados con el misma tasa de envío, longitud de palabra, número de bit de parada, y paridad. Si se encuentra que no se esta recibiendo nada, es probable que exista un error en las conexiones fisicas: el dato esta siendo enviado a traves de la linea equivocada, hay una rotura en la linea, o la señal correcta de handshaking no esta siendo recibida. Si se esta recibiendo basura, estonces, el error probablemente este en las areas discutidas a continuacion.

4.2.9.1 Tasa de envio no acoplada

Si dos dispositivos estan puestos a diferentes tasas de envio, el dispositivo receptor puede tratar de interpretar los datos (a no ser que este programado para reportar error de paridad o de framing). Tipicamete, se verá que el número de caracteres recibidos difiere del número de caracteres enviados.

4.2.9.2 Longitud de palabra no acoplada

Si se estan enviando palabras de 8 bits y el dispositivo receptor esta esperando palabras de 7 bits, no se podran notar diferencias en la transmision del te to. Ya que a menudo solo los primeros 7 bit son significantes de cualquier manera. Esto es porque el bit 0 es enviado primero y el bit 7 no es utilizado en una demuna transmision ASCII. El dispositivo receptor puede tratar de interpretar el bit e tra como un bit de paridad y reportar un error de paridad. Por lo tanto un error de paridad significa necesariamente que los datos se hayan dañado durante la transmision, si no que puede indicar que la longitud de palabra no se acopla.
Si se envian palabras de 7 bits y se esperan palabras de 8 bit, el bit de paridad puede ser tratado como el bit 7 esperado. Puesto que el bit de paridad es 1 para la mitad de los caracteres y 0 para la otra mitad, a menudo se encontrara que el dispositivo receptor despliega caractes ASCII e tendidos, tales como caracteres gráficos.

4.2.9.2 Error de paridad

Un error de paridad, estrictamente hablando, indica que los datos han sido dañados durante la transmision. Sin embargo, puede significar que los dos dispositivos no se han puesto de acuerdo sobre la paridad (par, impar, o ninguna) o sobre la longitud de la palabra

4.2.9.3 Bits de parada

No deberia de haber problemas si dos bits son enviados y solo uno es esperado. El bit extra simplemente se mezcla con el espacio que es permitido entre caracteres. Sin embargo, enviar un bit de parada cuando se esperan dos podría causar problemas dependiendo de las caracteristicas del dispostivo receptor. Con los equipos modernos esto no es un problema

4.2.9.4 Error de estructura

Un error de estructura indica un desacople en el número de bits y se reporta usualmente cuando no se recibe un bit de parada esperado

4.3 HANDSHAKING Y BUFFERS

El handshaking se refiere a los metodos con los cuales el dispositivo receptor puede controlar el flujo de datos que vienen desde el dispositivo emisor. Algunas veces un impresor no puede imprimir caracteres tan rapido como los recibe. Debe de utilizar handshaking para que el computador suspenda la transmision. El handshaking tambien se utiliza cuando el printer se queda sin papel, o cuando un computador envía datos a otro computador y el computador receptor no puede procesar los datos tan rapido como los recibe.
Cuando se sepa que el dispositivo receptor puede procesar los datos mas rapido que la tasa de transmision, se puede dispensar el handshaking .

4.3.1 HANDSHAKING POR MEDIO DE HARDWARE

El handshaking por medio de hardware, como se discutió en la sección 4.1 consiste en utilizar cone iones especiales para controlar la transmision de datos. Para resumir: un disposito DCE normalmente utiliza DSR (Data Set Ready) como la linea de handshaking principal para decirle al dispositivo DTE que se encuentra listo para controlar la transmision que se esta recibiendo. Tambien se puede utilizar CTS (Clear To Send) como un handshaking auxiliar. Los equipos DTE, por otra parte, utilizan DTR (Data Terminal Ready) como linea principal de handshaking para decirle al dispositivo DCE que esta listo para recibir, y RUS (Request to Send) como linea auxiliar de handshaking . Por convension, estas lineas de handshaking llevan un voltaje

positivo cuando la transmision esta habilitada, y un voltage negativo cuando la transmision esta suspendida.

4.3.2 HANDSHAKING POR MEDIO DE SOFTWARE

Cuando las señales de handshaking se envian a traves de las linea de datos (TXD y RXD,lineas 2 y 3), en lugar de las lineas dedicadas para handshaking como sucede con el handshaking por medio de hardware, se esta utilizando lo que se llama **handshaking por medio de software.** Este metodo es el que se utiliza cuando se comunican dos computadores (ya sea directamente o vía modem) y cuano la comunicación en dos sentidos es posible.
Varios protocolos estandares se han establecido para governar el handshaking por medio de sofware. El mas comun de ellos es XON/XOFF.

4.3.2.1 XON/XOF

Bajo este protocolo, el dispositivo receptor envía un caracter ASCII DC3 (19 decimal, 13H) al dispositivo transmisor cuando desea detener la transmision. Si envía el ASCII DC1 (17 decimal 11H) se esta indicando que la transmisión puede continuar.
En la practica normal, se implementará un buffer. El DC3 será enviado al dispositivo transmisor cuando el buffer este casi lleno y el DC1 será enviado cuando el buffer este casi vacío. (los buffer se describiran mas tarde en esta sección)

4.3.2.2 ETX/ACK

En este metodo conocido como (end-of-transmission/acknowledge) Fin de trasmision/reconocimiento, los datos son enviados en paquetes de longitud fija. Despues de enviar cada paquete, el dispositivo transmisor envía un caracter ETX (fin de transmision), ASCII 3. El dispositivo receptor reconoce haber recibido la transmision enviando un caracter ACK, ASCII 6. Algunas veces se envía de regreso un caracter NAK (recocimiento negativo- negative acknowledge), ASCII 21, para indicar que fue detectado un error.

4.3.3 HANDSHAKING POR MEDIO DE HARDWARE Y SOFTWARE COMBINADO

Si se estan comunicando dos computadoras vía modem. Los modem probablemente utilizaran handshaking de hardware con las respectivas computadoras, pero las computadoras utilizaran handshaking de software entre si.
Por lo tanto las computadoras deben de ser programadas para comunicarse solamente cuando la linea DSR desde el modem este alta (y posiblemente solo si el carrier dectect, o CD este alto) y si no se ha recibido una señal de software de parada. Las computadoras se encargan automaticamente de el handshaking de hardware de manera que los programas deberan de preocuparse solamente del handshaking de software.

4.3.4 BUFFERS

Un buffer es un area de memoria en la cual se reciben los caracteres para ser transmitidos a algun lugar. El uso de un buffer reduce el número de señales de handshaking que deben de enviarse ya que los datos pueden ser transmitidos en paquetes lleno en lugar de caracter por caracter.

4.3.4.1 Buffers de entrada

Un buffer de entrada se utiliza cuando el dispositivo receptor esta recibiendo caracteres mas rápido de lo que puede manejarlos. Por ejemplo, un impresora podría recibir caracteres a 1200 baudios pero solo imprimirlos a un equivalente de 300 baudios. En lugar de tener que instruir al printer para que envie la computador una señal de parada despues de cada caracter hasta que sea impreso, los diseñadores de impresores a menudo colocan un area de memoria aparte dentro del printer, que es capaz de almacenar un cierto numero de caracteres.

Esta area de memoria es conocida como un **buffer del entrada.** Se puede pensar en este buffer como un tanque de agua. El tanque esta siendo llenado en el tope y vaciado en el fondo. Una señal de parada se envía cuando el buffer esta casi lleno. La señal de reinicio se envía cuando el buffer esta casi vacio. SI el impresor espera hasta que el buffer este completamente lleno antes de enviar la señal de parada, y dice que envien tan pronto como hay algun espacio en el buffer, el buffer podría ser anulado tan pronto como el buffer se haya llenado la primera vez. De aquí en adelante podría estar enviando señales de parada despues de que recibe cada caracter y enviar la señal de reinicio despues de que ha sido procesado, e actamente como si no hubiera buffer.

Otra razon para enviar la señal de parada antes de que el buffer esta completamente lleno es evitar perdida de caracteres que podrían recibirse simultaneamente con la señal de parada.

Si se esta utilizando handshaking de hardware, usualmente causa que el dispositivo emisor suspenda la transmision inmediatamete. Con handshaking de software, sin embargo, existe un tiempo de retardo antes de que la orden de parada tome efecto, ya que la orden de parada tiene que ser procesada por al maquina que esta enviando y se pueden estar mandando caracteres durante ese procesamiento.

4.3.4.2 Buffer de salida

Los buffer de salida se refieren al area en la cual los datos son colocados antes de ser transmitidos. Esto reduce inconvenientes para el operador. Por ejemplo, si se este escribiendo en el teclado, y los caracteres que se escriben estan siendo enviados directamente al printer u otro dispositivo. Cuando el prter ha recibido toda la información puede manipularla , enviar una señal de parada, en este caso se tendría que detener la escritura en el teclado. Con un buffer de salida, sin

embargo, se puede continuar escribiendo hasta que el buffer este lleno. En la practica la mayoría de computadores tienen un buffer de entrada de teclado, en el cual los caracteres se colocan a medida que se estan escribiendo. Los programas entonces toman sus entradas desde el buffer del teclado.

4.3.4.3 Buffer en linea

Es posible comprar dispositivos que se conectan entre la computadora y el impresor y que contienen un buffer muy grande (algunas veces almacenan hasta 64,000 caracteres). Estos dispositivos reciben los caracteres desde una computadora y los envian al printer; tambien reciben datos mucho mas rapido de lo que normalmente lo hace un printer, y los envía al printer a una velocidad apropiada para este.

Desde el punto de vista del computador, solo se esta enviando datos a un printer muy rápido. Desde el punto de vista del operador, la operación se completa tan pronto como los datos han sido enviados al buffer, luego se puede continuar creando un nuevo documento mientras el primero todavía esta siendo impreso.

Algunos buffer en linea sofisticados pueden realizar tareas a tras tales como conversion de serie a paralelo, seleccionar entre varios printer, imprimir copias multiples de un documento, y almacenar datos recibidos por medio de un modem para que sean procesados despues por el computador.

Uno de tales buffer, el Hayes Transet 1000, puede actuar como una maquina de respuesta para la computadora. Si se conecta a un modem se puede tener al modem respondiendo el telefono y almacenando los datos recibidos en el buffer aun si la computadora esta apagada. Tambien habilita dos computadoras para compartir el mismo printer o a una computadora para accesar dos printer y realiza otro número de trucos.

4.4 MODEMS

Que hace un modem? Un modem transforma los datos recibidos desde un computador en serie a una forma conveniente para su transmision por medio de las lineas telefonicas, y viceversa. Los modems pueden se utilizados para transferir datos entre dos computadoras en localizaciones remotas.

Ya se ha discutido como pasan los datos entre dos dispositivos conectados directamente. La interposición de dos modems y la linea telefonica no afecta la tasa de envio, el número de bits de datos, bit de paridad, bit de parada, o el handshaking de software. Lo que el modem hace es convertir los voltajes positivos y negativos que representan los bits individuales de cada caracter en señales apropiadas para las comunicaciones telfonicas. Inferentes modem trabajan de diferente forma; los tipos mas populares de modem se describen en las siguintes subsecciones

4.4.1 COMO TRABAJA UN MODEM

Conectar dos computadoras, vía modems, a la línea telefonica no gramfica la historia completa. En cuanto al usuario concierne, el mensaje desaparece en la linea telefonica en un extremo y reaparece en el otro. Entre tanto, la señal puede haber pasado por los cables de la compañía de telefonos local a una compañía de telefonos de larga distancia y esta a una compañía de satelites, que paso la comunicación a otra compañía local de telefonos. Afortunadamente se puede utilizar uno de los protocolos establecidos para comunicaciones con modems que se discutirá mas adelante, en la mayoría de los casos se pueden ovidar el resto. Existen otros protocolos, pero estos se utilizan comunmente solo en los Estados Unidos.

4.4.2 CONECTANDO EL MODEM A LA COMPUTADORA

En cuanto a la computadora concierne, un modem externo, es solo otro dispositivo serie. Se hacen conexiones normales de los alambres para trasmitir datos, recibir datos, tierra, y handshaking. Inclusive, se pued conectar tambien un detector de carry (Carry detect - CD), en la linea ocho, asi el modem puede permitir que la computadora sepa cuando esta presente una señal de carry, ademas, se puede conectar el indicador de campana (Ring Indicador - RI), en la linea 22, asi el modem podra indicar que el telefono esta sonando. Un voltage positivo en CD significa que una señal de carry esta presente. Un voltage positivo en RI significa que el telefono esta sonando. El que estas señales sean reconocidas por el computador depende del software de comunicación en uso. Algunos modems envian mensajes cuando el contrario se cuelga y cuando el telefono esta sonando. En el caso de un modem interno tal como el Hayes Smart Modem 1200B, el modem esta contenido en una tableta que se conecta directamente al computador. La tarjeta aparece ante el computador como un tarjeta serie de interface, de manera que el software no necesita saber si el modem que esta en uso es un interno o externo.
Hay que notar que la señales de handshaking de hardware (DTR, DSR) se utilizan para controlar al comunicación entre la computadora y el modem. Esta linea no pasan atraves de los alambres hasta el medem y la computadora remota. Si no que para habilitar el handshaking entre las dos computadoras se debe utilizar handshaking de software.

4.5 TRANSMISION DE ARCHIVOS

Cuando se trabaja con computadoras, a menudo se encuentra la necesidad de transferir archivos de una computadora a otra. Algunas veces la computadora que recibe esta realizando funciones de procesar datos de alguna manera. Por ejemplo, se puede haber escrito una carta en una computadora portatil y desear imprimirla en otra computadora que no lee el mismo tamaño de discos. Otras veces, la segunda computadora se utiliza unicamente como un

dispositivo de almacenamiento de datos.

La transferencia de archivos entre computadora puede servir para muchos propositos pero tambien posee muchos problemas. En esta sección se estudiaran las dificultades que pueden aparecer durante la transmision de un archivo. Se han diseñado varios metodos para resolver dichos problemas. En la siguiente sección se discutirá uno de ellos: el protocolo XMODEM, el cual originalmente fue diseñado para facilitar la trasferencia de datos entre microcomputadoras.

4.5.1 PORQUE SE NECESITAN LOS PROTOCOLOS

Casi todas las computadoras ofrecen la opción de manejar un puerto serie, y pueden ser programadas para enviar datos al puerto serie sin demasiada dificultad. Las computadoras grandes reciben datos en serie todo el tiempo ya que ellas estan controladas a traves de terminales que operan en serie.

Asi se podría pensar que para transmitir datos desde una microcomputadora a un computadora grande simplemente se debe de hacer que la microcomputadora piense que esta imprimiendo y que la computadora grande piense que esta conectada a una terminal. Sin embargo, no es tan simple como esto.

Los creadores de los metodos establecidos en comunicaciones de computadoras asumen que el material podría ser transferido desde un teclado, en forma de texto. La transmision de archivos, por otra parte, se da mucho mas rapido de lo que una persona puede escribir, y algunas veces incluye caracteres que no aparecen en el teclado. Los problemas que resultan se discutiran a continuacion, la mayoría de los cuales pueden ser resueltos por los protocolos de los cuales se discutirá uno.

4.5.1.1 LONGITUD DE PALABRA

Muchos de los datos contenidos en los archivos de una microcomputadora no contienen texto si no que programas, datos graficos, u otro material no ASCII. Estos datos, amenudo son referidos como **datos binarios**, que normalmente utilizan los ocho bits de cada byte.

Se debera de recordar que la tabla oficial ASCII, utiliza palabras de 7 bits. Muchas computadoras y canales de comunicación estan diseñados para entradas ASCII unicamente, y estan imposibilitadas para aceptar palabras de 8 bits. Estas insisten en palabras de 7 bits, el octavo bit se utiliza como bit de paridad. Por lo tanto, los datos binarios deben de ser convertidos de alguna forma a palabras de 7 bits antes de que muchas computadoras los acepten.

4.5.1.2 CONTROL DE CARACTERES

Otro problema que se puede encontrar durante la transmision de archivos tiene que ver con los caracteres de control. Los datos trasferidos amenudo contienen bytes con valores especiales que la

computadora receptora podría mal interpretar. Algunas computadoras, en realidad, podran manipular caracteres especiales en una forma particular que no pueden ser anuladas por medio de software. En general para resolver este problema, los datos deben de ser convertidos a caracteres que puedan ser aceptados por la computadora que recibe. El protocolo Kermit es un ejemplo de un protocolo que puede hacer esto.

4.5.1.3 LONGITUD DE BLOQUE

En muchas computadoras, la entrada es colocada en un buffer antes de ser procesada. El tamaño de este buffer es a menudo limitado intencionalmente a 128 o 256 bytes. Cuando este es el caso, no es posible enviar un archivo como una cadena continua de bytes. La computadora que envía debe de dividir el archivo y enviarlo como una serie de bloques, cada uno de los cuales es mas pequeño que el buffer de entrada de la maquina que recibe.

La division de datos en bloques tambien necesita chequeos de error mas sofisticados como se verá mas tarde.

4.5.1.4 CHEQUEO DE ERROR

El ruido en la lineas telefonicas, es algo que todo mundo ha experimenta algunas veces. Sin embargo, este ruido puede aparecer como datos en la computadora. Las transmisiones largas o una alta velocidad de transmision, aumentan la probabilidad de que los datos sean alterados. Por lo tanto, siempre es ventajoso, cuando se transfieren archivos, incorporar alguna forma de chequeo de errores en las comunicaciones.

Cuando los datos consisten de texto, generalmente es facil ver donde ha ocurrido el error ya que el texto se vera con basura. En cuanto a lo que concierne a datos binarios, no se puede decir lo que sucede y solamente se sabe, por ejemplo, que el programa no corre.

Como se vio en la sección 4.2 el chequeo de paridad que se incluye muy amenudo en comunicaciones serie, tiene solo un probabilidad de cincuenta-cincuenta de detectar un error en un solo byte, y un solo error puede ser fatal para un programa en lenguaje de maquina. Ademas, es fastidioso estar diciendo que hubo un error, despues que se ha transferido un archivo completo, y que se tiene que volver a transferirlo cuando solo una parte de el esta alterada. Es mucho mejor dividir el archivo en bloques, y chequear cada uno de estos bloques para ver si ha ocurrido algun error. De esta forma, la computadora tendrá que retransmitir unicamente los bloques en que se hayan detectado errores. Es por esto que la mayoría de los protocolos deviden un archivo en bloques.

4.5.2 PROTOCOLOS DE TRANMISION DE ARCHIVOS

Existen muchos metodos para tratar con los problemas antes mencionados. Algunos de ellos se discutiran brevemente a

continuación , se estudiará en detalle el protocolo utilizado para comunicaciones entre PC (XMODEM)

4.5.2.1 CONVERSION A HEXADECIMAL

Una forma simple de mandar datos binarios en caracteres ASCII transmisibles es mandar cualquier byte en su equivalente hexadecimal y transmitir los caracteres ASCII correspondientes a cada número (Es decir que el mensaje completo consistirá de caracteres del "0" al "9" y de la "A" a la "F"): dos caracteres para cada byte. El mensaje sera reconvertido por la computadora que recibe.

Este método resuelve el problema de enviar datos de 8 bits por canales de 7 bits. Sin embargo, esto no resuelve los problemas de handshaking y los problemas de chequeo de errores mencionados anteriormente, ademas, se necesita una longitud física para el archivo del doble y por lo tanto para la transmision de los datos.

4.5.2.2 XMODEM

El protocolo XMODEM se describirá detalladamente en la sección 4.6. Fue diseñado originalmente para transmisiones entre microcomputadoras, pero ha sido utilizado en comunicaciones de micro a mainframe. XMODEM ofrece chequeo de error y division de los datos en bloques. Sin embargo, no ofrece la capacidad de enviar datos de 8 bits por canales de 7 bits, o la conversión de caracteres de control en caracteres imprimibles.

4.5.2.3 KERMIT

Este protocolo se utiliza principalmente en las comunicaciones entre mainframes y microcomputadoras y es ampliamente aceptado, especialmente en en mundo academico. Resuelve todos los problemas planteados en esta capitulo.

4.5.3 OTRAS CONSIDERACIONES EN LA TRANSMISION DE ARCHIVOS

A continuacion se daran una serie de consideraciones que se deberan de mantener en mente cuando se transfieren archivos.

4.5.3.1 PAQUETES

Ambos protocolos XMODEM y KERMIT dividen los archivos que seran transferidos en bloques o paquetes. Cada paquete consiste en un encabezado y los datos mismos, un codigo de chequeo de error una marca de fin de bloque. La computadora receptora mandará una respuesta indicando si el paquete fue recibido correctamente. En le caso de XMODEM, esta respuesta consiste de un solo caracter ASCII devuelto al dispositivo emisor. En le caso de KERMIT, la respuesta misma esta en forma de paquete.

4.5.3.2 FORMATO DE LOS DATOS

El hecho que sea posible transferir datos binarios de una computadora a otra, no necesariamente significa que la segunda computadora sea capaz de leer los datos. Por ejemplo, los datos pueden consistir de programas que no correran en la computadora destino. Aun si las dos computadoras utilizan el mismo microprocesador, ya que podrían estar diseñadas de manera diferente, y si las computadoras son de la misma familia, el sistema operativo y los requisitos de memoria podrían ser diferentes.

Es por esto que a veces se es necesario convertir los datos a una forma que puedan ser utilizados por la computadora receptora.

4.6 XMODEM

En septiembre de 1977, Ward Christesen, (el autor de XMODEM) escribio un programa para CP/M-80, llamado MODEM. El tuvo incompatibilidad con los discos así que escribio un programa que le permitiera intercambiar programas con otros usuarios de CP/M. El proposito de este programa era el de ser usado por dos personas, una en cada computadora, en el extremo opuesto de la linea telefonica. Keith Petersen partiendo del programa MODEM hizo el XMODEM, en cual permite trasmitir hacia y desde computadoras que esten desatendidas.

El protocolo implicito en el programa original de Ward, el cual algunas veces es llamado "Protocolo de Christesen" ha llegado a ser conocido ampliamente como el protocolo XMODEM.

XMODEM es utilizado muy amenudo para descargar todo tipo de archivos binarios y de archivos ASCII desde computadoras caseras, tales como maquinas PC, CP/M, etc.

4.6.1 BLOQUES

Los datos trasferidos por XMODEM se dividen en bloques, cada bloque consiste de un encabezado de arranque (01H), un unico byte de numero de bloque, el complemento uno del número de bloque, 128 byte de datos, y un byte de chequeo. Este formato se ilustra en la tabla 4.1

Tabla 4.1 Formato de un bloque en XMODEM

Offset	Contenido
0	SOH (Star-of-header Character: ASCII 01)
1	Numero de bloque: Comenzando con uno, pero barriendo desde 0 hasta FF
2	Complemento uno del número de bloque (255 - número de bloque)
3 - 130	128 bytes de datos
131	Byte de chequeo: Suma de los bytes de datos unicamente el carry es ignorado

El número de bloque comienza en uno, pero es calculado en modulo 256, es decir, que despues de 255 (FFH) regresa a cero. El complemento uno puede se calculado restando el número de bloque de 255 o complementando todos los bits del número (uno convertirlos en cero y viceversa). El byte de chequeo es un solo byte que se calcula por la suma total de los 128 byte y se ignora el carry

4.6.2 PROTOCOLO DE CINCO NIVELES

Antes que la computadora que envía pueda mandar sus datos, tiene que recibir un caracter NAK (Reconocimiento negativo) desde el dispositivo que recibe. El programa receptor debe de enviar un caracter NAK (15H) como un **timeout (tiempo sin recibir)** cada diez segundos sin recibir datos. Es el primero de tales NAK que dispara el inicio de la trasmision.

Una vez que el programa comienza a recibir un bloque, reporta un error siempre que un ocurra un espacio de un segundo o mas entre caracteres en el bloque, incluyendo en byte de chequeo. Sin embargo, debe de esperar que la linea este desocupada antes de enviar un NAK para indicar un error.

Hay que notar que un segundo de timeout no es suficiente para muchas comunicaciones de larga distancia, y amenudo se utiliza un tiempo de espera mas grande en lugar del establecido oficialmente.

El dispositivo receptor, chequea el número de bloque y reporta un error si esta fuera de secuencia. Si el número de bloque es el igual al ultimo que se envio, indica una retransmisión que no debería de ser considerada un error. Despues de recibido cada bloque, el dispositivo receptor envía un caracter ACK (06H) si el bloque se ha recibido correctamente, o NAK si no es asi. En el último caso, el transmisor reenvia el bloque. Despues de que el bloque es reconocido, se transmite el siguiente bloque.

Al final de la transmicion, en transmisor envía un EOT (04H) y espera por un ACK, reenviando un EOT si no recibe uno.

4.6.3 LA OPCION CRC

El byte de chequeo no es suficiente para detectar todos los errores. Por lo tanto, una extension de XMODEM, conocida como opción CRC, ha sido incorporada, la cual consiste de dos bytes. esta extension es llamada un chequeo de redundancia ciclica (CRC-16 - Cyclical redundancy check) y dectecta errores al menos en un 99% de la veces. El CRC será aplicado posteriormente.

Por convencion, el dispositivo receptor debe indicar al dispositivo emisor que la opción CRC será utilizada, enviando un caracter C en lugar de NAK para solicitar el inicio de la transmision. Puesto que no todas la versiones de XMODEM incorporan la opción CRC, el dispositivo receptor deberá de volver a enviar un NAK si, despues de varios intentos, no recibe

respuesto. El formato de bloque utilizado con la opción CRC se muestra en la tabla 4.2 .

Tabla 4.2 Formato de un bloque en XMODEM con opción CRC

Offset	Contenido
0	SOH (Start-of-header Character: ASCII 01)
1	Numero de bloque: Comenzando con uno, pero barriendo desde 0 hasta FF
2	Complemento uno del número de bloque (255 - número de bloque)
3 - 130	128 bytes de datos
131	Byte alto de CRC
132	Byte bajo de CRC

4.6.4 VENTAJAS Y DESVENTAJAS DE XMODEM

Las principales ventajas de este protocolo son su simplicidad y su universalidad. Pero tiene varias desventajas, sin embargo, esto es porque a menudo se utiliza en formas para la cual no fue originalmente diseñado.

No se lleva a cabo ninguna transforamción especial para permitir que datos de 8 bits sean enviados por canales de 7 bits. Por lo tanto, si se estan enviando datos binarios, se debe hacer por medio de una comunicación de 8 bits. El protocolo especifica 8 bits de datos, no paridad, y un bit de parada, aun si solamente datos textuales estan siendo transmitidos.

Tambien, no hay protección en contra de datos binarios que se puedan interpretar como señales de control. Si el dispositivo de control, por ejemplo, siempre trata el ASCII 4 como fin de transmision, solamente se podran enviar 3 bloques puesto que el bloque cuatro tendra el ASCII 4 como número de bloque. Esto significa que aun si los datos estan siendo enviados no seran recibidos.

4.7 TOPICOS DE PROGRAMACION

Existen algunos aspectos de programación de comunicaciones que son de aplicaciones generales y no estan limitadas a una máquina. Se discutirá algo de eso en esta sección. Uno de estos temas generales es el manejo de interrupciones de I/O, puesto que los principios son claramente de aplicaciones generales, y porque la comprension es necesaria para secciones mas avanzadas. Tambien se incluye una sección para el UART, una nota sobre Chequeo de redundancia Ciclica (CRC), y una nota sobre buffer circulares.

4.7.1 UART

El UART, o Universal Asyncdronous Receiver and Transmiter, es un chip (8250) especialmente diseñado para manipular comunicaciones

asincronas. Un USART, o Universal Synchronous Asynchronou Receiver and Transmitter, este diseñado para manipular ambos tipo de comunicaciones sincronas y asincronas. La siguiente información se aplica igualmente al UART y al USART

4.7.1.1 EL TRABAJO DEL UART

El UART se encarga de 4 tareas principales

1. Convertir las señales en paralelo que vienen desde el CPU a señales serie para la transmision fuera del computador, y convertir las señales serie que vienen al computador en paralelo para ser procesadas por la computadora.
2. Arrega los bit necesarios de arranque, parada, y paridad para cada caracter que sera transmitido, y los separa estos bits de los caracteres que recibe
3. Asegura que los bits individuales sean enviados fuera a la tasa de envio apropiada, calcula el bit de paridad de los caracteres transmitidos y recibidos, y reporta cualquier error detectado
4. Establece las señales apropiadas de handshaking de hardware y reporta el estado de la linea de handshaking de la computadora que envia.

4.7.1.2 CONEXIONES AL UART

Las conexiones al UART tipicamente consisten de las siguientes:

° Ocho pines para transferir datos en paralelo

° Dos pines para transmitir y recibir datos

° Señal de reloj desde la cual se calcula la tasa de envio

° Circuito de control a traves del cual el UART puede recibir instrucciones y reportar el estado

° Linea(s) de interrupción a traves de la cual el UART puede alertar al CPU de un cambio de estado

° Un pin selector que le indica al UART cuando actuar

El UART y su circuiteria asociada amenudo esta incorporada en una tarjeta conocida como tarjeta interface serie, la cual casi siempre es un accesorio opcional para una computadora. En otros casos, el UART viene construido dentro de la computadora. Los printer, modems y otros equipos de comunicaciones frecuentemente traen incorporado al UART

4.7.1.3 REGISTROS PRINCIPALES DEL UART

El UART tiene varios registros, los cuales son localizaciones internas de memoria. Estas localizaciones típicamente contienen el último caracter recibido; el siguiente caracter a ser transmitido; información de estado concerniente a las señales de handshaking que se estan detectando; e información tal como si el chip esta listo para recibir otro caracter para transmision y si un caracter esta listo para ser enviado.

Cuando el software de comunicación esta recibiendo datos a través del UART lo primero que se hace es leer el registro de estado y averiguar si se ha recibido un caracter. Si es así, el caracter es leído. Esta lectura normalmente limpia el registro de estado de manera que indique que el caracter ya no esta disponible. Cuando otro caracter ha sido recibido, el registro de estado lo indicara, y el nuevo caracter sera leído.

La transmision es similar. El registor de estado indica si el UART esta listo para recibir un caracter para transmision. Cuando cada caracter es enviado, el registro de estado vuelve a su estado previo hasta que un nuevo caracter es cargado dentro del UART para su transmisión.

4.7.2 POLLIN VERSUS INTERRUPCIONES

El ciclo de examinar el estado, leer, examinar estado y así sucesivamente se conoce como **pollin.** La mayoría de UART ofrecen una alternativa para este método cíclico. El UART puede ser programado para enviar una señal especial conocida como **una interrupcion** cuando un evento de comunicación ocurre. La computadora debe de haber sido programada para reconocer la interrupcion y reaccionar de acuerdo a ella.

Las interrupciones típicamente son generadas cuando un caracter ha sido recibido o transmitido o cuando una señal de handshaking cambia. Pueden haber diferentes interrupciones para diferentes eventos o solo una interrupcion. En el último caso es necesario que el computador examine un registro especial para averiguar que causo la última interrupcion. Algunas veces el UART puede ser programado para generar interrupciones para ciertos eventos y no para otros.

4.7.2.1 POLLIN (EXCRUTINIO)

La utilización de este método tiene dos desventajas. La primera es similar a un telefono sin campana. Se tendría que estar levantandolo cada cierto tiempo para ver si alguien esta llamando. Si se tienen varios telefonos en un escritorio, de debera de estar levantando cada uno de ellos uno a la vez. Esto es lo que sucede con el metodo de Pollin.

El primer problema con este método es la energía de procesamiento que se gasta con una computadora que continuamente tiene que estar explorando el byte de estado. Si la computadora esta dedicada solo a manejar tareas específicamente de

comunicación, entonces, esto no es un problema. Sin embargo, las computadoras a menudo estan trabajando en varias tareas al mismo tiempo. Esto se aplica a minicomputadoras, y a un número cada vez mayor de microcomputadoras con la introducción de programas co residentes. En este caso es necesario utilizar un método que evite el chequeo innecesario.

El segundo problema que aparece con este método, es que se puede perder un nuevo caracter mientras el primero esta siendo procesado. Esto depende de dos factores: la velocidad a la cual los caracteres estan siendo recibidos y la velocidad a la cual estan siendo procesados. Tipicamente, los caracteres que llegan se estan desplegando en la pantalla, almacenado en un archivo, e interpretandolos. Algunos de ellos necesitan ser interpretados, como en el caso de secuencias de escape.

Este proceso lleva tiempo, cuando una microcomputadora tipica utiliza este método de chequeo, los caracteres que llegan facilmente se pueden perder, si la tasa de envio excede a 1200 bauldios, aunque no se requiera un acceso al disco.

Podria ser posible utilizar control de handshaking, ya sea por software o por hardware, para detener el flujo de caracteres que llegan despues que cada caracter ha sido recibido y reiniciarlo despues que ha sido procesado, pero este metodo es goseramente ineficiente y podría aun no trabajar por que el retardo de tiempo que hay entre el momento que se envía la señal de parada y el momento en que se detiene el dispositivo remoto.

A pesar de estos inconvenientes, el método de chequeo es muy comunmente utilizado en comunicaciones serie, y si se estan utilizando estrictamente comandos del DOS, (como se vera mas adelante) el polling es el único método disponible para la IBM PC. Es por estos inconvenientes que se han mencionado anteriormente que son pocos los programas de comunicación serios que se han escrito para IMP PC que estan adheridas a los comandos del DOS.

4.7.2.2 INTERRUPCIONES DE HARDWARE

La utilización de una interrupción es equivalente a utilizar un telefono con una campana. Ya no se tiene que permanecer levantando el telefono para ver si alguien esta llamando. Solmente hay que esperar a que el telefono suene, y mientras tanto es posible concentrarse en otras tareas hasta que ocurra una llamada. En otras palabras, un UART programado para enviar interrupciones envía una señal cuando un evento relevante ocurre. Esto es la forma que las computadoras saben cuando algo ha sucedido y no tienen que gastar tiempo realizando chequeos.

Las computadoras tienen mas de una linea de interrupcion, pero solo una se asigna a un chip en particular. Esto significa que aunque el UART puede ser programado para generar una interrupción cuando ocurra uno de varios eventos diferentes, el CPU solo que el UART ha generado la interrupcion, y no cual evento dentro de UART la causó.

Aunque el uso de interrupciones de entrada/salida (I/O) puede

durar cierto tiempo, no esta excento de problemas. Es importante recordar que una interrupción puede ser recibida en cualquier tiempo durante un ciclo de procesamiento. El computador y el software que esta corriendo en el debe de ser diseñado de manera que cualquier tarea puedo ser suspendida cuando una interrupción ocurra y volver a la tarea cuando el evento que causo la interrupción haya sido tratado.

Hay que recordar que un computador puede manipular interrupciones desde dos tarjetas serie, el teclado, el reloj, un mouse, y un disco duro, todas estas interrupciones pueden llegar al mismo tiempo. Como se puede ver, es importante ser cuidadoso cuando se esta implementando un programa que maneje interrupciones.

4.7.3 BUFFER CIRCULARES

Otro topico de programación que puede ser discutido en terminos generales son los buffer circulares. Como se discutio anteriormente un buffer es una región de memoria utilizada como un lugar de almacenamiento temporal para datos que esperan ser trasmitidos, o datos recibidos que esperan ser procesados. Un tipo comun de buffer es el buffer circular. Los caracteres son devueltos desde el buffer de una manera LIFO (primero en entrar/ primero en salir). Si el buffer llega a estar lleno, los nuevos caracteres seran escritos sobre los caractares antiguos en el buffer; sin embargo, se deberá de implementar un handshaking apropiado para impedir que se envien caracteres al buffer cuando este está lleno.

El mismo método puede ser usado para buffer de entrada y para buffer de salida. En el caso de un buffer de entrada bajo un manejador de interrupciones de comunicaciones, los caracteres seran colocados en el buffer por la rutina manejadora de interrupciones. Luego estos caracteres seran devueltos desde el buffer por una rutina del programa principal. Un método general de programación de un buffer circular se discute a continuacion.

4.7.3.1 CREACION DE BUFFER CIRCULARES

Lo primero, una area de memoria es colocada y una variable que recuerda su tamaño (SIZE) es creada. Luego variables que registren la cuento (COUNT) de caracteres, y el offset (OFFSET) desde el inicio (STARTPOS) seran inicializadas a cero.

4.7.3.2 AÑADIENDO CARACTERES

El offset para un nuevo caracter es calculado como sigue:

OFFSET = COUNT + STARTPOS

Si OFFSET = SIZE, OFFSET = 0

Los caracteres están colocados a un offset OFFSET dentro del buffer. Recordar que la primera localización está en un offset de cero. Así si el buffer es de 128 bytes el offset mas alto será 127 , no 128. Si COUNT es igual a SIZE, se tienen mas caracteres de los que caben en el buffer, y se incrementa STARTPOS. Si STARTPOS, da como resultado, que apunta mas allá del fin del buffer, es retornado al inicio. Si COUNT es menor que SIZE, se incrementa COUNT. La figura 4.7 muestra el proceso de llenar un buffer circular.

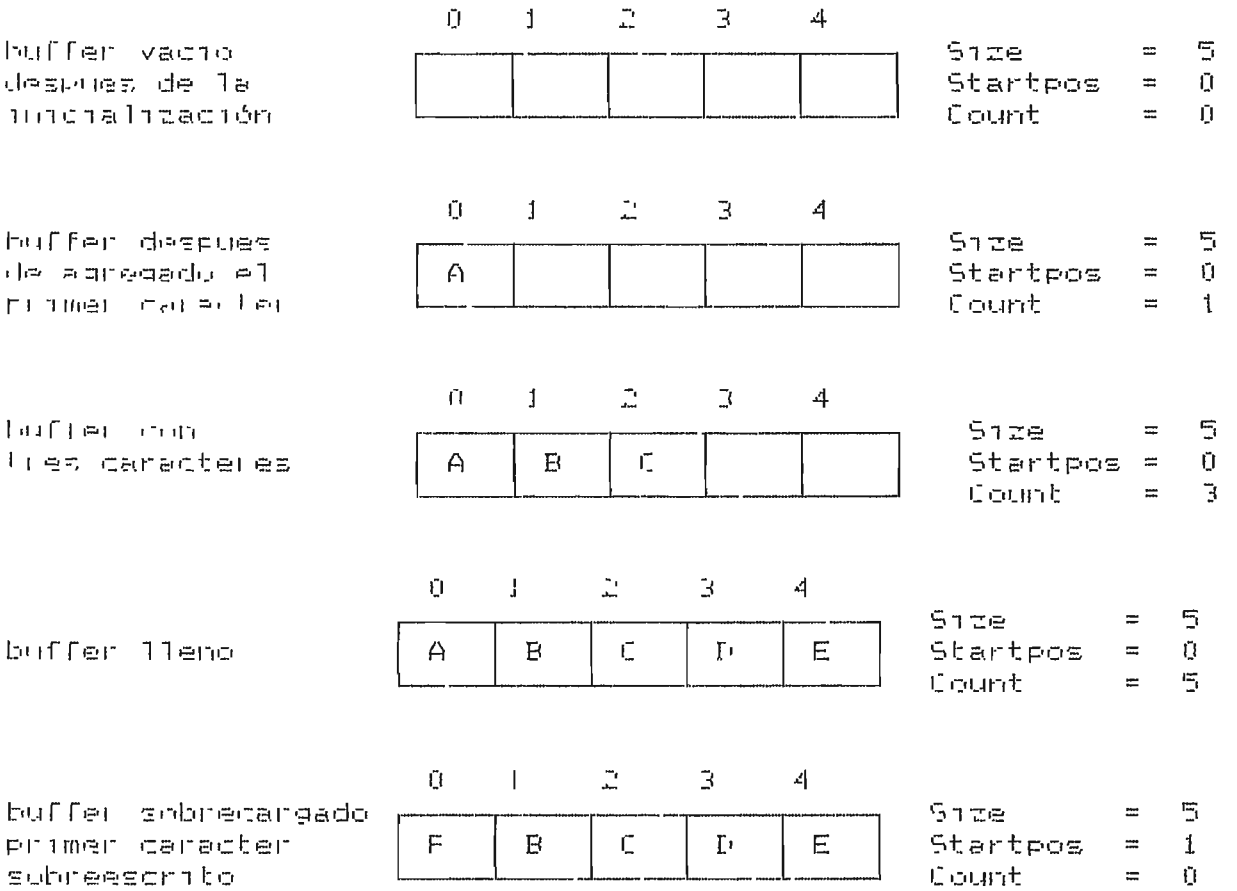


figura 4.7 Llenando un buffer circular

4.7.3.3 DEVOLVIENDO CARACTERES

Si COUNT es cero, no hay caracteres disponibles, así la función retorna con un resultado FALSO. El primer caracter disponible es removido de un offset STARTPOS en el buffer. Habiendo tomado el caracter, la función incrementa STARTPOS (retornando a cero cuando STARTPOS sea igual a SIZE) y decrementa COUNT.
La figura 4.8 muestra el proceso de vaciar un buffer circular.

```
 0    1    2    3    4
 A    B    C    D    E    Size     =  5
                           Startpos =  0
                           Count    =  5

 buffer lleno


 0    1    2    3    4
[A]   D    C    D    E    Size     =  5
                           Startpos =  1
                           Count    =  4

 Un caracter devuelto
 El primer caracter todavía esta ahi
 pero sera sobre escrito por un
 nuevo caracter
```

figura 4.8 Vaciando el buffer

4.7.4 CHEQUEO DE REDUNDANCIA CICLICO

Los chequeos de redundancia ciclica (CRC) son una forma de chequeo de error. El proposito de un codigo de error es calcular un numero que este relacionado matematicamente a la cadena de datos que corresponde al mensaje que sera transmitido. Esto es hecho por el dispositivo transmisor. El dispositivo receptor tambien calcula el número utilizando la misma fórmula basado en los datos recibidos. Se espera que cualquier error en los datos afectara el número calculado, de manera que los errores puedan ser detectados.

La forma mas simple de calcular un número basado en una cadena de texto es simplemente ir sumando el valor ASCII de todos los caracteres. Esto es conocido simplemente como chequeo.

El calculo CRC tiene dos ventajas sobre el chequeo. Primero, revela una proporción de error mas alta. Segundo, esta orientada a los bits y no a los caracteres, lo que permite trabajar con protocolos que producen una cadena de bits y no una cadena de bytes.

La teoria del calculo de CRC implica tecnicas matematicas avanzadas, las cuales no se discutiran aqui por considerarse fuera de lugar, si no, que trataré de explicar su calculo de una manera muy simple.

Hay varios diferentes calculos para el CRC. En cada caso, la cadena de bits que forman el mensaje es tratado como un numero binario enorme. Primero, un número (n) de ceros es agregado al final del enorme numero, para multiplicarlo por 2^n. Luego, el número (ahora aun mas grande) es dividido por un número (d) que varía dependiendo de cual estandar de CRC esta siendo utilizado.

Solamente el residuo es retenido. Este número es transmitido al dispositivo receptor, el cual ejecuta el mismo calculo para asegurarse que los dos números coinciden y que no ha habido perdida o daño en los datos.

Diferentes programas de comunicación utilizan diferentes versiones de CRC XMODEM, con la opción de CRC, utiliza CRC-CCITT. Cada CRC ejecuta los calculos de una manera ligeramente diferente. En el caso de CRC-16

En el caso de CRC-CCITT

n 16

d = 11000000000000101

Esta descripción explica como el CRC es calculado. Sin embargo, aprender como programar en equipo de manera que ejecute el chequeo de CRC es un asunto complicado. Algunos avanzados equipos de comunicación ya incorporan diseños especializados de hardware para realizar estos calculos. Para mas informacion, leer **Technical Aspects of Data Comunications** por John E McNamara. En la parte 3 de este libro se dan algunos ejemplos individuales.

4.8 COMUNICACION A NIVEL DEL USUARIO EN LA IBM PC

Esta sección describe la operación de las comunicaciones serie desde el punto de vista del usuario. La sección 4.9 describe la programación para comunicaciones serie utilizando las funciones disponibles a través del DOS e el BIOS. La sección 4.10 describe la arquitectura de una IBM PC, enforada a operaciones de entrada/salida serie, y programación a nivel del sistema. Este capitulo finalizara con una aplicación de programación muy util realizada en BASIC como una prueba de lo que se dijo al principio de este trabajo de que no siempre lo mas complicado es lo mas eficiente.

4.8.1 HARDWARE

Hay varias maquinas de la serie IBM PC. Tambien hay un gran numero de computadoras PC compatibles hechas por compañias distintas a IBM pero diseñadas para correr el software escrito para las IBM PC. Desafortunadamente, las no compatibilidades se muestran mas a menudo en el area de comunicaciones que en cualquier otra. Si una maquina no es verdaderamente compatible, una computadora que corre todo el software escrito para las IBM

PC, aun no garantiza que correra el software de comunicación escrito para estas maquinas, se utilizara el termino PC para referirse a todas las maquina de la serie IBM PC y para las que verdaderamente son compatibles.

4.8.1.2 LA ESTRUCTURA DE LA COMPUTADORA

La serie IBM PC esta diseñada de una forma modular, permitiendole al usuario configurar un sistema para sus propios requerimientos.

La IBM PC misma no viene con una interface serie construida, aunque algunas PC compatibles si la tienen.

Las PC tienen un número de expansores dentro, en los cuales se pueden conectar varios dispositivos. Es simple añadir una targeta de expansion: simplemente se tiene que remover la tapa de la computadora e insertar la nueva targeta en un expansor libre. Algunas veces se tienen que poner unos pequeños interruptores en la tarjeta o en la PC; esto se explica en las instrucciones que vienen con la targeta. Las targetas de expansion realizan funciones como interface para el printer, control de un monitor, o entradas salidas serie.

4.8.1.3 UN VISTAZO AL DOS

Para trabajar con una PC se necesita cierto conocimiento en lo que respecta al sistema operativo. Un sistema operativo consiste de uno o mas programas diseñados par govenar las operaciones basicas de la computadora. El PC DOS es el sistema operativo mas comunmente utilizado en las IBM PC, aunque otros tales como CP/M y Xenix estan disponibles. El PC DOS se puede utilizar en algunas computadoras no IMB que sean verdaderamente compatibles. La mayoría de lo que se diga en esta y en la siguiente sección sera relevante para el PC DOS y el MS-DOS. Aunque el DOS significa sistema operativo de disco, se vera que cubre mas que solo operaciones de disco.

No se intentara aqui dar una completa descripción del DOS. Hay varios excelentes libros para ese tema, asi como el manual que viene con el DOS, y el manual de referencia technica DOS. Esta sección sera enfocada a los aspectos del DOS que son relevantes para las comunicaciones serie a nivel del usuario.

4.8.1.4 CARGANDO EL DOS

El DOS esta contenido en el archivo COMMAND.COM y dos archivo ocultos que estan en el disco pero no se pueden ver en el directorio del disco. En el caso de IBM PC, estos dos archivos son BIO.COM y IMBIOS.COM.

Antes de que la computadora pueda ser utilizada, el sistema operativo debe de ser cargado en la memoria. Si el computador tiene un disco duro, automaticamente carga el sistema operativo cuando se enciende la maquina. Si solo tiene unidades de disco

file rules, se tiene que colocar el disco del DOS en la unidad A.

4.8.1.5 QUE HACE EL DOS

El DOS puede ser dividido en dos partes: Comandos del DOS y funciones del DOS.

Comandos del DOS

Los comandos del DOS se encargan de ciertas tareas domesticas tales como copiar discos, leer directorios para ver que es lo que tienen, poner fecha y hora, etc. Para que DOS realice estas tareas el usuario debe entrar el comando respectivo desde el teclado.

Funciones del DOS

La segunda parte del DOS consiste en funciones tales como abrir y cerrar archivo, e escribir caracteres en la pantalla que no pueden ser llamados desde el teclado pero que solo pueden ser llamados a traves de programas. Estas son conocidas como funciones del DOS. Estas funciones hacen mas facil la escritura de programas. Ya que estas funciones son utilizadas frecuentemente, es conveniente tenerlas en memoria todo el tiempo de manera que diferentes programas puedan utilizarlas.

4.8.1.6 NOMBRES DE LOS DISPOSITIVOS DEL DOS

El DOS tiene un conjunto de nombres de dispositivos construidos dentro de la computadora. Los dispositivos son listados a continuacion:

AUX:	El primer puerto adaptador serie/paralelo
COM1:	El primer puerto adaptador serie/paralelo
COM2:	EL segundo puerto adaptador serie/paralelo
CON:	La consola (teclado y pantalla)
PRN:	El primer printer paralelo
LPT1:	El primer printer paralelo
LPT2:	El segundo printer paralelo
LPT3:	El tercer printer paralelo
NUL:	No existente dispositivo fingido

Los dos puntos despues del nombre del dispositivo son opcionales. Se pueden agregar otros dispositivos añadiendo un manejador de dispositivos discutidos en esta misma sección.

El DOS esta configurado de manera que permite hasta dos adaptadores de interface serie, o dos dispositivos diferentes al mismo tiempo, y estos son referidos en la documentación y por el sistema operativo como COM1 y COM2. Hay dos conjuntos de direcciones reservadas para estos dispositivos como se aplicara posteriormente.

4.8.1.7 MANEJADORES DE DISPOSITIVOS

Se puede extender y modificar el DOS para utilizar archivos adicionales conocidos como **manejadores de dispositivos** que normalmente pueden ser reconocidos por la extension .SYS en el nombre de los archivos. Estos manejadores de dispositivos contienen extensiones al sistema operativo. Son cargados en memoria al mismo tiempo que el DOS y efectivamente forman parte de el.

Si en el directorio raiz del disco del DOS existe un archivo con el nombre CONFIG.SYS, el DOS lee el archivo cuando es cargado y ejecuta los comandos contenidos en el. Estos comandos consisten en instrucciones de la siguiente forma:

DEVICE = DEV.SYS

Donde DEV es el nombre de un manejador de dispositivo particular.

4.8.2 EL COMANDO MODE

El comando MODE tiene dos propositos:

1. Asigna un nombre de dispositivo y le dice al DOS si el impresor estandar esta en uno de los puertos paralelo o uno de los puertos serie.

2. Configura los paramentros del puerto serie.

4.8.2.1 ASIGNACION DE DISPOSITIVOS

LPT1 (Line printer 1) es el nombre de dispositivo utilizado para el impresor estandar. Se asume que LPT1 se refiere al printer conectado al puerto paralelo. Si se desea que se refiera al printer conectado al puerto serie (es decir, asignar LPT1 a COM1 o COM2), se deberá de escribir

```
MODE LPT1:=COM1:

o

MODE LPT1:=COM2:
```

De esta forma se puede cambiar la salida de un programa sin tener que reescribir el programa. Por ejemplo, si un programa le dice al DOS que envie algo a LPT1, la salida se dará en el dispositivo que se le dijo al DOS que era LPT1. El programa no necesita saber que lo que se tiene conectado a LPT1 es, por ejemplo, un modem establecido como COM1 y un printer serie como COM2. Simpre que el programa le diga al DOS que envie algo a COM1, se enviaran ahi

con tomar en cuenta lo que se ha asignado a COM1 con el comando MODE.

Un rival uno puede tambien ignorar el DOS completamente y accesar al dispositivo directamente, suministrando lo que se conoce como dirección del dispositivo y la forma correcta de manipularlo, en el programa que aparece al final del capitulo se dará un ejemplo de esto.

4.9.2.2 ESTABLECIENDO LAS OPCIONES DE COMUNICACION

El comando MODE tambien es utilizado para configurar las siguientes caracteristicas de el puerto serie:

- Tasa de envio (Baud Rate)
- Paridad
- Bits de datos
- Bits de parada
- Handshading del printer

Esta facilidad es utilizada principalmente para configurar al puerto serie para utilizar un printer serie. Con este comando se logra que el software le diga al DOS que imprima algo sin tener que preocuparse por indicar paridad, bit de parada, o aun si el printer es serio o paralelo.

Para utilizar el comado MODE para establecer los parametros de el puerto serie COM1, a 9600 baud, no paridad, 8 bits de datos, y un bit de parada, se escribira lo siguiente:

```
MODE COM1:9600,N,8,1
```

Note que los parametros estan especificados sin abreviaciones, y estan separados por comas.

Baud Rate
Las tasas de envio validas son 110, 150, 300, 600, 1200, 2400, 4800, y 9600. Solo los primeros dos caracteres de cada número son requeridos: asi, podria poner una tasa de envio de 9600 simplemente entrando 96

Paridad
La paridad es puesta entrando N para NO, O para impar, o E para par

Bits de datos
El numero de bits de datos puede ser ya sea de 7 o de 8

Bits de parada
Estos pueden ser uno o dos

Hhandshaking del printer

Como se discutió anteriormente, la IBM PC generalmente esta configurada como un dispositivo DTE, espera recibir señales de handshaking en el pin 6 y en el pin 8. En realidad, ambas lineas deben de estar en alto antes de transmitir. Suponiendo que se desea transmitir algo, pero las lineas de handshaking no estan en alto. Si el dispositivo al cual se esta transmitiendo es un printer, algo podrian estar equivocadas. El printer podria estar apagado, fuera de linea, sin papel, pero lo mas probable es que desconecte las lineas de handshaking es que el buffer este lleno. El DOS deberia mantenerse intentando, y eventualmente el buffer se limpiara y mas datos podran ser enviados.

El parametro final que se puede entrar al final de la lista toma cuidado de esta situación. Es una p opcional. Esta es utilizada cuando el dispositivo en cuestion es un impresor. Si hay una p al final de los parametros de la lista, El DOS se mantendrá intentando para enviar caracteres hasta que se presione Crtl_Break en lugar de reportar un error cuando las lineas de handshaking no esten en alto. Esto es porque los impresores tipicamente estan en estado de ocupado, y por lo tanto tratan de suprimir la comunicación, por mas tiempo que otros dispositivos serie.

Si no se coloca el parametro P, el DOS reportará un error de escritura cuando se le indique que envie un caracter y no este presente la linea de handshaking, luego preguntaará si se desea abortar, reintentar, o ignorar el error.

Se podria preguntar que resulta si se le dice al DOS que ignore completamente el handshaking. La respuesta es que esto es imposible. NO se puede enviar nada a el puerto serie por medio del DOS a menos que las lineas de handshaking esten en alto. En las secciones anteriores se describieron varios métodos de comunicaciones, algunos de ellos utilizan handshaking de hardware, otros handshaking de software, y posiblemente otros ninguno de ellos. En general para utilizar estos metodos de comunicación alternativos, y todavía trabajar con el DOS, se deberá de engañar al DOS haciendolo "pensar" que las señales de handshaking estan ahi. Si el dispositivo suministra la señal en un linea pero no en la otra, su pueden unir las dos. Si no hay ninguna señal de handshaking, se puede realimentar a la computadora conectando la linea 20 de regreso a las lineas 6 y 8.

4.8.3 ENTRADA/SALIDA ESTANDAR

Otra caracteristica del DOS permite que los programas tomen sus entradas desde la entrada estandar y envien su salida a la salida estandar. La entrada y salida estandar a menudo se refieren al teclado y la pantalla de video respectivamente pero pueden tambien referirse a otros dispositivos de entrada y salida. No todos los programas utilizán estas caracteristicas: algunos toman sus entradas directamente desde el teclado y la envian directamente a la pantalla.

Para comprender mejor este concepto, suponga que se desea escribir una carta al presidente de IMB, se podría rotular la carta ya sea como: "Para el presidente de IBM Corporation" o, si se conoce el nombre, "Mr J Smith, IBM Corporation" Suponga ahora que se dio un cambio en la mesa directiva de IMB en el tiempo que la carta tardo en llegar. La forma en que se rotulo la carta podría determinar si la carta llegaría al nuevo presidente o al antiguo presidente. De manera similar, un programa puede enviar sus salidas al dispositivo de salida estandar, siempre que pueda estar a tiempo, o a un dispositivo de salida particular.

En el caso de programas que utilicen la entradas y salidas estandar, se puede utilizar al DOS para cambiar los dispositivos de entrada y salida estandar cuando se invoque el programa. Esto se hace utilizando los simbolos , , y .

Un buen ejemplo es comando DOS SORT. Suponga que se desea clasificar un archivo llamado NOMBRES.TXT. Para hacer eso, se escribe el siguiente comado:

```
SORT  NOMBRES.TXT
```

SORT tomara su entrada desde NOMBRES.TXT y envía el resultado al dispositivo de salida estandar, es decir, a la pantalla. Si se desea enviar la salida a el impresor establecido como LPT1, se escribe el siguiente comado:

```
SORT  NOMBRES.TXT  LPT1:
```

Para clasificar el archivo y guardar el archivo clasificado en un nuevo archivo de datos (NUEVO.TXT), se escribe el siguiene comando:

```
SORT  NOMBRES.TXT  NUEVO.TXT
```

Para clasificar los datos y agregar los datos clasificados al final de un archivo existente (VIEJO.TXT), se escribe:

```
SORT  NOMBRES.TXT    VIEJO.TXT
```

Como se puede ver, la entrada y salida estandar pueden ser ya sea un dispositivo fisico o un archivo. La salida estandar de un programa puede ser redirigida para convertirse en la entrada de otro programa. Para hacer esto se utiliza el simbolo |. Para producir un directorio clasificado, se escribe:

```
DIR|SORT
```

El directorio clasificado aparecera en la pantalla. El concepto de pasar la salida de un programa a otro es llamado **piping (canalizacion.** Un programa que toma su entrada de la entrada estandar, la modifica, y envía los resultados a la salida estandar es llamado un **filtro.**

Si se combinan los dos conceptos se puede imprimir un directorio

clasificado escribiendo:

```
DIR|SORT  LPT1:
```

4.8.4 EL COMANDO COPY

Todo usuario del DOS esta familiarizado con el comando COPY para copiar archivos. Con este comando, se puede copiar un archivo llamado NOMBRES.TXT del disco en la unidad A a el disco en la unidad B, escribiendo:

```
COPY A:NOMBRES.TXT B:
```

Ademas de copiar archivos en discos, se puede tambien mandarlos a los dispositivos. Para desplegar un archivo en la pantalla, se escribe:

```
COPY NOMBRES.TXT CON:
```

Para imprimir un archivo en el impresor establecido como LPT1:, se da el siguiente comando:

```
COPY NOMBRES.TXT LPT1:
```

Cuando se ha finalizado se presiona Ctl-Z, o se presiona F6, para cerrar el archivo.

Se puede escribir directamente a el printer escribiendo el comando:

```
COPY CON: LPT1:
```

COPY puede , en un grado limitado, ser utilizado para accesar el puerto serie. Se puede copiar un archivo llamado ARCHI a COM1, escribiendo:

```
COPY ARCHI COM1:
```

Seria muy util si se pudiera utilizar el cumando COPY para trasuferir archivos desde el puerto serie: en otruas palabras, descargar un archivo. Con el sistema operativo CP/M, sobre el cual se baso el DOS, puede hacer esto. Sin embargo esta caracteristica no fue implementada en el DOS y su ausencia a dado lugar a cierta confusión, ya que el manual del DOS es inconsistente con su contenido. Es posible que los autores del DOS pensaban implementar esta capacidad de descargar archivos, pero cambiaron de idea. Esto se puede ver en el manual del PC-DOS 3.1 en la pagina 7-61 que dice "Usted puede tambien utilizar el comando COPY para transferir archivos entre algunos dispositivos del sistema". Pero en la pagina 7-62 dice "No se puede utilizar

el comando COPY para transferir archivos utilizando el COM o el puerto auxiliar serie AUX". Aun esta indicación es incorrecta, puesto que se puede utilizar el comando COPY para transferir archivos al puerto serie, pero no desde el.
Otro error en el manual del DOS esta en la pagina 7-66. Da varios ejemplos utilizando COPY, incluyendo "copy aux con". Esto es consistente con las instrucciones primitivas que podian transferir datos entre dispositivos, y tambien consistente con el CP/M, pero no es cierto para el DOS

4.8.5 EL COMANDO CTTY

Este es otro comando que se utiliza en comunicaciones serie. Con el se puede controlar la IBM PC desde una terminal externa. Primero, se conecta la terminal a el puerto serie de la IBM PC. Luego se utiliza el comando MODE par establecer los parametros de comunicación. Finalmente se escribe:

```
CTTY COM1:
```

De ahi en adelante, la PC no tomará en cuenta nada que se escriba en el teclado, y tomará sus entradas solamente de la terminal remota. Las salidas seran enviadas tambien a la terminal.
Para retornar la PC al control local, se escribe desde la terminal remota el siguiente comando:

```
CTTY CON:
```

Es posible tener acceso remoto a una PC dejandola encendida y con en el modo CTTY, y conectada a un modem que ha sido programado para responder el telefono. Se puede marcar en la PC desde la computadora remota, y dar comandos para que la PC, corra programas, etc.

4.9 COMUNICACIONES EN IMB PC A NIVEL DEL BIOS Y DEL DOS

El DOS y el BIOS suministran un número de funciones construidas que pueden ser llamadas desde programas y varias de estas funciones estan relacionadas con comunicaciones serie. Esta sección describe las funciones del DOS relacionadas con comunicaciones serie, y las ventajas y desventagas de utilizarlas.
Los programas no tienen que utilizar obligatoriamente las funciones del DOS y del BIOS. Es posible, y a menudo mas rapido, lograr los mismos resultados de forma diferente. Por ejemplo, relativamente pocos programas comerciales utlizan las funciones del DOS cuando escriben programas para una pantalla monocromatica, es porque es mucho mas fácil mover el texto directamente a un area de memoria conocida como buffer de pantalla. Ni aun BASIC, es cual es suministrado con las IBM PC,

utiliza las funciones del DOS para manipular la pantalla. Con programas de comunicación serie a menudo es posible obviar el DOS y controlar el puerto serie directamente, esto se logra enviando la salida directamnete a una determinada dirección, esto se verá en el programa de ejemplo. Sin embargo, es importante estar informado de como se puede utilizar las funciones del DOS y del BIOS en programación de comunicaciones ya que a menudo se encuentra que presenta mayor compatibilidad y menos esfuerzo de programación a bajo nivel.

4.9.2 INTERRUPCIONES DE SOFTWARE

las interrupciones de software significa instruir al microprocesadorr para que salte a determinada localización de manera que las instrucciones almacenadas ahi se ejecuten. Cuando un interrupción de software es leida por la unidad de control, el microprocesador lee un area de memoria conocida como **tabla de vectores de interrupcion.** Esta area contiene direcciones para cada posible interrupción. Cada dirección es de cuatro bytes de longitud. Asi, para ejecutar la interrupción 21h, for ejemplo, el procesador ve el offset de (4 x 21H) en la tabla de vectores de interrupción, y luego ejecuta la función almacenada en esa dirección, la cual termina con IRET o instrucción de retorno de interrupción. El procesador luego continua con la siguiente instrucción en el programa original.

La manera de accesar las funciones del DOS y del BIOS ya se ha discutido en detalle en el capitulo 2

4.9.3 LAS FUNCIONES DEL DOS

Las funciones del DOS que se encargan de las comunicaciones serie son accesadas invocando la interrupción 21H. Estas funciones se aplican al dispositivo AUX (standar auxiliary divice) como es llamado en el manual del DOS. Es muy dificil precisar exactamente a cual dispositivo auxiliar se refiere. El manual de referencia tecnica del DOS indica "Axiliary (AUX, COM1, COM2). Esto parece querer decir que AUX puede ser asignado como COM1 o COM2, pero no hay nada en el manual del DOS o en el manual de referencia tecnica del DOS concerniente a esto. En realidad la unica referencia a AUX en el manual del DOS es que es una palabra reservada.

La primera función de comunicaciones serie es llevada a cabo colocando 4 en el registro AH, colocando el caracter a ser enviado en el registro DL e invocando la interrupción 21H. el caracter será enviado a COM1.

No hay forma de establecer los parametros de comunicación por medio de las funciones del DOS, en su lugar el programador lo hace por medio de funciones del BIOS o el usuario lo hace por medio del comando MODE.

Ni las entradas ni las salidas serie del DOS retornan ninguna información de error. Estas funciones son tan primitivas que no

son muy utilizadas. IBM admite en el manual de referencia tecnica del DOS "para mayor control, es recomendable que se utilice las rutinas del BIOS (interrupción 14h)

4.9.4 LAS FUNCIONES DEL BIOS

Para utilizar las cuatro funciones del BIOS relacionadas con comunicaciones serie se debe de accesarlas a traves de la interrupoción 14H, Se coloca un número de 0 a 3 en AH indicando cual de las cuatro funciones se requiere. Luego se coloca el número de puerto en DX, el número de puerto es cero par COM1 y 1 par COM2.

F4.9.4.1 FUNCION DE ESTABLECER PARAMETROS

La primera función del BIOS, función 0, es utilizada para establecer los parametros de comunicación. Es accesada colocando 0 en AH y el número del puerto en DX (0), el byte que representa los parametros se coloca en AL, y luego se invoca la interrupción 14H. los bit cero y uno del valor colocado en AL definen la longitud de la palabra. Para palabras de 8 bits, ambos bits estan en uno. Para palabras de 7 bits, el bit 1 esta en 1 y el bit 0 esta en 0. El bit 2 indica el número de bit de parada. Un valor de 0 representa un bit de parada, y 1 representa dos bit de parada. el bit 3 y 4 representan la paridad como se muestra en la tabla 4.3. los bit 5 y 7 indican la tasa de envio como se muestra en la tabla 4.4.

Tabla 4.3 INT 14H establecer paridad

Bit 4	Bit 3	Paridad
0 o 1	0	Ninguna
0	1	Impar
1	1	Par

Tabla 4.4 INT 14h establecer tasa de envio

Bit 7	Bit 6	Bit 5	Baud Rate
0	0	0	110
0	0	1	150
0	1	0	300
0	1	1	600
1	0	0	1200
1	0	1	2400
1	1	0	4800
1	1	1	9600

4.9.4.2 FUNCION DE TRANSMITIR CARACTER

La siguiente función del BIOS es la función de transmitir caracter, es utilizada para transmitir caracteres. Es accesada colocando 1 en AH y en DX el número del puerto, en caracter a ser enviado se coloca en AL, y se invoca la interrupción 14H. Hay que notar que el caracter no será enviado hasta que las linea de handshaking del dispositivo receptor esten el alto. Aun con el bajo nivel del BIOS el handshaking de hardware no puede ser ignorado.

En la practica normal de programación, se podría llamar la función de obtener estado el puerto (ver abajo) y enviar el caracter solamente cuando las linea de handshaking esten en alto.

El registro AH indicará cualquier condición de error. Si el bit 7 de AH es 0, se podrá dar la comunicación. Si el bit 7 de AH es uno, entonces los restantes bits de AH indicaran el tipo de error ocurrido, utilizando el mismo codigo de error que se obtiene en la función de obtener estado del puerto que se discutirá en breve

4.9.4.3 FUNCION DE RECIBIR CARACTERES

La función de recibir caracteres es la función 2, es accesada colocando 2 en AH, en el número del puerto en DX, luego se invoca la interrupción 14. El BIOS espera hasta que un caracter es recibido desde el puerto serie o el tiempo de espera (timeout) se agota. Cuando el caracter es recibido, es colocado en AL, y si ocurre un error, este se reporta en AH.

Si AH es cero, entonces no ha ocurrido ningun error. Si no es cero, entonces los bits del 0 al 7 indican el tipo de error. Sin embargo, si el bit 7 esta puesto indicando un error de timeout, los restantes bits tendran valores impredecibles.

En la practica normal de programación no es la función de recibir caracter a la que se llama repetidamente, si no que, a la función de obtener el estado el puerto, la función de recibir caracter es llamada solamente cuando se sabe que un caracter esta disponible. Esto da mucho control, puesto que el programador escoge su propio tiempo de espera (timeout) y puede estar haciendo algo util entre la recepción de caracteres, en lugar de solo estar esperando a que venga el siguiente caracter.

Este metodo tambien evita problemas que resultan de un error en el BIOS de las computadoras IBM PC primitivas que reportan un error de tiempo de espera como un error de paridad.

4.9.4.4 FUNCION DE OBTENER ESTADO DEL PUERTO

Para obtener el estado del puerto, se utiliza la función 3, que es accesada colocando 3 en AH, en número de puerto en DX, e invocando la interrupción 14H. Esta función suministra mucha información acerca del estado actual del puerto serie por medio del registro AX. La tabla 4.3 muestra el significado de los bit del registro AX.

Delta significa que la señal relevante ha cambiado desde la ultima vez que el estado del puerto fue leido. Por ejemplo, el bit 5 del byte retornado en AL indica si la señal de handshaking DSR recibido esta en alto o no. El bit 1 indica si el estado de la señal DSR ha cambiado desde la ultima vez que el estado del puerto fue leido. La información delta generalmente es utilizada en conección con el manejador de interrupciones I/O, y puesto que no es accesible a traves del DOS ni del BIOS la inforamación delta no es utilizada aqui.

4.9.5 HANDSHAKING BAJO EL BIOS

El BIOS se comporta muy estrañamente ignorando las señales de handshaking. Las función que establece los parametros de comunicación no establece ninguna señal de handshaking. La función que recibe el caracter enciende la señal DTR y apaga la señal RTS y luego espera por el caracter a ser recibido, retornando un error de timeout si no recibe ninguno en cierto periodo de tiempo. La función que transmite caracter pone en la DTR y RTS, y espera a que ambas DSR y CTS sean puestas en uno por el otro dispositivo. Si no estan puestas, la función retorna despues de un timeout.

Tabla 4.5 Codigos de estado de AH

Bits de AH	Significado si estan en 1
7	Timeout Error
6	Transmitter shift register empty
5	Transmitter holding register empty
4	Break Detect
3	Framing error
2	Parity error
1	Overrun error
0	Data Ready
Bits de AL	
7	Received line detect
6	Ring indicator
5	Data set ready
4	Clear to send
3	Delta receive line singa detect
2	Trailing edge ring indicator
1	Delta data set ready
0	Delta clear to send

Lo malo es que mientras el bios insiste en recibir dos señales de handshaking solamente suministra una al otro dispositivo. Y no solo eso, si no que tambien apaga ROS cuando esta esperando recibir, y si el dispositivo con el cual se esta comunicando es otra PC que insiste en recibir dos señales de handshaking se tendra que falsear la señal esperada puntando ROS a DTR en el dispositivo remoto y desconectando ROS en la PC que transmite.

Hay que recordar que hasta que se ha llamado la función de enviar o recibir caracter, la señal de handshaking saliente esta baja, o apagada. Si el otro dispositivo espera una señal de handshaking, se debera llamar primero la funcion de enviar o recibir caracter. Luego al menos DTR permanecera alto.

Sin embargo, siguiendo el procedimiento normal de llamar la función de obtener el estado del puerto para ver si se ha recibido algun caracter y luego llamar la función de recibir caracter si hay uno, puede causar problemas. Podría que nunca se reciba alguno ya que no se llamará la función de recibir hasta que haya algo, y no habrá nada porque las lineas de handshaking se han dejado bajas porque aun no se ha llamado la función de recibir caracter.
Lo que se recomienda es que como parte de la secuencia de inicialización, despues de establecer los parametros llamar la función que recibe caracter una vez aun sabiendo que no hay nada en el puerto. De manera que se pueda encender DTR. Y no ser engañado por la función que transmite caracter poniendo ROS y pensando que no se han unido los alambres, ya que la función que recibe caracter apagara ROS de nuevo e actamente cuado se desea.

Tambien hay que notar que no hay forma de apagar las señales de handshaking una vez finalizada la comunicación. De manera que no hay que pensar que el modem, si es que se ocupa uno, esta desconectado cuando se cierren los archivos, o se podría tener una cuenta de telefono muy grande.

4.9.6 EL DOS Y EL BIOS VERSUS CONTROL DIRECTO DE HARDWARE

Cuando se trabaja con comunicaciones serie se deberá ponderar los pro y los contra de utilizar las funciones del BIOS y del DOS o programar directamente el hardware.

4.9.6.1 LAS VENTAJAS DEL DOS Y DEL BIOS

1. Primero, pueden ahorrar esfuerzo en la programación y espacio en el archivo que almacena el programa. Si los problemas ya han sido resueltos, ¿Por que reinventar la rueda? Y si las funciones ya estan en memoria esperando a ser llamadas, ¿por que duplicarlas?

2. Segundo, se puede lograr mas compatibilidad con otros maquinas

que trabajen con el DOS. Algunas maquinas que trabajen con el DOS son considerablemente diferentes de las IMB PC en su arquitectura. Ellas pueden tener diferentes chips en diferentes localizaciones. Y si se quiere escribir directamente al hardware se podria fracasar, ya que las localizaciones no son las mismas. Sin embargo, las funciones del DOS casi siempre son las mismas en todas las maquinas que trabajen con el DOS

4. Finalmente, los programas que utilicen funciones exclusivamente del DOS deberian ser compatibles con futuras versiones de las PC

4.9.6.2 LAS DESVENTAJAS DEL DOS Y DEL BIOS

Desafortunadamente, en el caso de las comunicaciones hay limitaciones sobre cuanto se puede llevar a cabo por medio del DOS y del BIOS. Nada que se pueda hacer a nivel del sistema puede ser llevado a cabo manteniendose con el DOS y el BIOS

4.9.6.2.1 MANEJADOR DE INTERRUPCIONES I/O

El chip 8250 UART es utilizado en las tarjetas serie compatibles con IBM PC y puede ser programado para generar interrupciones cuando ocurran ciertos eventos, tales como el recibimieto de un caracter. La ventaja de este metodo, y las tecnicas implicadas, seran discutidas en siguientes secciones. Por su puesto que este chip se deberá de programar directamente; las funciones del DOS no pueden ayudar. Este metodo es casi escencial para diseñar software de comunicación sofisticado.

4.9.6.2.2 HANDSHAKING

Todas las funciones del DOS requieren de dos lineas de handshaking que se encuentren en alto antes de que se pueda transmitir. En algunos casos, tales como una conección directa a una gran computadora, las señales de handshaking no estan presentes. Sin embargo, usualmente es posible falsear las lineas salientes de handshaking a las lineas entrantes. Solo una linea saliente es suministrada. La programación del UART directamente, permite controlar las lineas de handshaking o ignorarlas si se desea.

4.10 EL UART

Anteriormente se discutio en forma general el UART. En esta seccion se discutiran los detalles de este chip. el UART casi siempre es utilizado en las IBM PC para comunicaciones asincronas en serie. Si se planea hacer programas de comunicaciones a nivel del sistema (es decir, trabajar con los puertos en donde estan localizados los registros del UART) se debe de tener

como parte de esta sección.

4.10.1 LOS REGISTROS DEL UART

De manera similar al CPU, el UART contiene registros, los cuales son de tres tipos.

1. Registros de control, los cuales reciben los comandos desde el microprocesador

2. Registros de estado, los cuales son utilizados para informar al microprocesador de que esta sucediendo en el UART

3. Registros de buffer, los cuales sostienen caracteres pendientes de transmisión o procesamiento

La forma en que los registros son accesados depende de la arquitectura de la computadora en que el UART este instalado. En le caso de una IBM PC, los valores para ser colocados en los registros son enviados a una dirección apropiada I/O por medio de la instrucción OUT de lenguaje de ensamble. Los registros son leidos por medio de la instrucción IN acompañada de la apropiada dirección. Las direcciones de estos registros seran dadas en la siguiente sección.

4.10.1.2 REGISTROS DE CONTROL

Los cuatro registros de control que se utilzan para recibir comandos desde el microprocesador son:

Registro de control de linea

El registro de control de linea es utilizado para establecer los parametros de comunicación. EL significado de cada bit en el registro se muestra en la tabla 4.6 y una descripción mas detallada de cada a continuación.

Tabla 4.6 Bits de registro de control

Bit	Significado
0	Longitud de palabra
1	Longitud de palabra
2	Bits de parada
3	Habilitador de paridad
4	Seleccionador de paridad
5	Paridad uno
6	Ruptura
7	DLAB

El bit 0 y el bit uno registran la longitud de palabra el significado se muestra en las tabla 4.7

Tabla 4.7 Longitud de palabra

Bit 0	Bit 1	Longitud de palabra
0	0	5
0	1	6
1	0	7
1	1	8

El bit 2 registra el número de bits de parada. Si este bit es 0, se utiliza un bit de parada. Si es 1 se utilizan dos bit

El bit 3 habilita la paridad: si es 0, no se trasmite ni se espera bit de paridad. Si es 1, un bit de paridad es enviado y esperado

El bit 4 selecciona paridad par si este en 1 y paridad impar si esta en 0. Este bit es ingnorado a no ser que el bit 3 este puesto.

El bit 5 hace el bit de paridad de logica 0 si esta puesto. Si el bit 5 esta en 0, la paridad será de logica 1.

El bit 6 se utiliza para enviar un comando de ruptura. Y fuerza a salir de la condición de espacio (lgica 0) hasta que el bit 6 este puesto a 0

El bit 7 es llamado Divisor latch acces bit (DLAB). Si esta puesto en 1, una operación de lectura o escritura accesa a los latchs del generador de baud rate (ver abajo). Si esta puesto a 0, operaciones de leer o escribir accesan al buffer receptor o transmisor y al registro habilitador de interrupciones.

Registro de control del modem

El registro de control del modem controla las señales de handshaking enviadas hacia afuera desde el UART. Cada bit de este registro es listado en la tabla 4.8 y explicado despues.

Tabla 4.8 Bits del registro de control del modem

Bit	Abreviación	Nombre
0	DTR	Data terminal ready
1	RTS	Request to send
2	Out1	User defined output 1
3	Out2	User defined output 2
4	Loop	Test mode loop_back

El bit 0 es utilizado para habilitar al dispositivo remoto a transmitir. Si el bit 0 esta en 0, DTR esta en logico 1, es decir, solicita al dispositivo remoto que no envie, si el bit 0 esta en 1 habilita al dispositivo remoto para transmitir

El bit 1 es utlizado exactamente de la misma forma para controlar la salida de RQS

El bit 2 y 3 controlan las salidas auxiliares conocidas como OUT1 y OUT2. Estas no tiene nada que ver con las tarjetas serie. Pueden ser utilizadas para controlar las señales de detector del carry e indicador de campana, pero generalmente parecen no estar alambradas asi, en una IBM PC el bit 3 debe de estar en 1, debido a una anomalía en el diseño

El bit 4 habilita el modo de diagnostico

Los bit del 5 la 7 estan permanentemente a cero.

Registro habilitador de interrupciones

Utilizando el manejador de interrupciones I/O, es posible instruir al 8250 (PIC) que genere una señal de interrupción siempre que ciertos eventos ocurran. El registro habilitador de interrupciones es utilzado para decirle al 8250 cuales eventos deberan causar interrupción.
Si el programa utiliza el metodo de polling (escrutinio) no se habilitaran las interrupciones. En lugar de eso, el programa examinara el estado de los registros para ver si ha sucedido algo. Los bits correspondiente a cada interrupción se muestran en la tabla 4.9

Tabla 4.9 Bits del registro habilitador de interrupciones

Bit	Interrupción puesta
0	Data available
1	Transmitter holding register empty
2	Receiver line status
3	Modem estatus
4-7	(siempre en cero)

Latches de divisor de baud rate

La tasa de envio es puesta colocando en dos registros un número con número por el cual la entrada de reloj (1.8432 MHz) debe de ser dividida. Resultando una frecuencia que es 16 veces la tasa de envio. Estos dos registros son el byte menos significante

del latch divisor (DLL) y el byte menos significante del latch divisor (DLM). Los divisores utilizados para generar diferentes tasas de envios se muestran en la tabla 4.10

Tabla 4.10 Divisores de tasa de envio

Baud rate	Decimal	Hex	LSB	MSB
300	384	180	1	80
1200	96	60	0	60
2400	48	30	0	30
4800	24	18	0	18
9600	12	0C	0	0C

Nota que no se esta limitado a los rangos convencionales de tasas de envio. Valores intermedios tambien pueden ser generados seleccionando un divisor apropiado. Esto puede ser util en aplicaciones tales como control de procesos

4.10.1.3 REGISTROS DE ESTADO

Los tres registros de estado reportan al microprocesador que esta sucediendo en las diferentes partes del UART.

Registro de estado de linea

El registro de estado de linea es utilizado para obetener información concerniente a la recepción y transmisión de datos. El significado de los bits individuales se muestra en la tabla 4.11 y son explicados despues.

- **Data ready** significa que un caracter ha sido recibido desde fuera. Este bit permance puesto hasta que el caracter ha sido leido del buffer del registro receptor.
- **Overrun Error** significa que un caracter fue recibido antes que el caracter previo halla sido leido. Esto indica que los caracteres estan siendo recibidos mas rapido de lo que estan siendo procesados.
- **Framing Error** significa que un bit de parada valido no fue detectado despues del ultimo caracter recibido.
- **Break Interrupt** significa que se ha recibido una señal de ruptura

- **Transmiter Holding Register Empty** significa que el UART esta listo para recibir un caracter para su transmisión.
  Transmiter Shift Register Empty significa que el UART no esta en el proceso de transmitir un caracter. Este registro es utilizado en el paralelo para el proceso de conversión serie, y su estado no es de interes para el software de comunicación

Tabla 4.11 Bits del registro de estado de linea

Bit			
0	DR	Data Ready	Un caracter que esta viniendo ha sido recibido y colocado en el buffer receptor
1	OE	Overrun Error	Un caracter ha sido recibido antes de que el anterior fuera removido para su procesamiento
2	PE	Parity Error	La paridad del caracter que esta viniendo esta equivocada
3	FE	Framing Error	Un caracter recibido no tiene un bit de parada valido
4	BI	Break Interrupt	Una ruptura se esta recibiendo
5	THRE	Transmitter holding register empty	El UART esta listo para recibir nuevos caracteres para transmitirlos
6	TSRE	Transmitter shift register empty	TSR esta esperando por un caracter desde el THR
7	[SPARE]		Este bit esta permanentemente puesto a cero

Registro de estado del modem

El registro de estado del modem da información acerca del estado de las lineas de handshaking, y no necesariamente se debe tener un modem. El significado de cada bits individual se muestra en la tabla 4.12

Los bits 1,2 y 4 son conocidos como bit delta. Estos indican un cambio desde la ultima vez que el registro fue leido. Esto se utiliza en programación con manejo de interrupciones.

Tabla 4.12 Bits del registro de estado del modem

Bit	Nombre	Significado si esta puesto
0	Delta CTS	Limpiar para enviar ha cambiado
1	Delta DSR	Linea de dato listo ha cambiado
2	TERI	Ring indicador RI ha cambiado
3	Delta RLSD	Detector de señal de linea recibida ha cambiado
4	CTS	Limpiar para enviar esta alta (OK)
5	DSR	Dato listo esta alto (OK)
6	RI	Ring indicador esta alto
7	RSLD	Detector de señal de linea recibida esta en alto

Registro identificador de interrupciones

El registro identificado de interrupciones suministra información acerca del estado de las interrupciones pendientes. El bit 0 esta puesto en uno si no hay interrupciones pendientes. Si esta en cero, el bit uno y dos indican cual interrupción esta pendiente de acuerdo con lo que se indica en la tabla 4.13

Tabla 4.13 Bits del registro identificador de interrupciones

Bit 2	Bit 1	Interrupción pendiente
1	1	Estado de linea
1	0	Dato recibido disponible
0	1	Regitro transmisor vacio
0	0	Estado del modem

4.10.1.4 REGISTROS DE BUFFER

la tercera categoría de registros del UART son los registros de buffer. Hay dos registros de buffer: Receptores y trasmisores

Registros de buffer receptores

El registro de buffer receptor sostiene el ultimo caracter recibido. Una ves ha sido leido, el registro de estado de linea indica que el buffer receptor esta vacio hasta que otro caracter sea recibido. Si el segundo caracter es recibo antes que el primero caracter has sido leido, un error de overrun será reportado.

4.11 DIRECCIONES DE ENTRADA SALIDA

Ademas de los 640K de memeria principal que es accesada en

terminos de segmentos y complementos, tambien es posible accesar memoria conocida como memoria de entrada/salida (I/O). 768 direcciones I/O, o puertos como a menudo se llaman, estan disponibles, y un dispositivo tipico utiliza varios de estos.

Se puede accesar estas direcciones de entrada salida dando instrucciones IN y OUT al Microprocesador. Por ejemplo para enviar el contenido del registro AL al puerto 3F8H, la instrucción en lenguaje de ensamble es:

```
OUT 3F8H,AL
```

Para leer desde el puerto, cuya dirección esta en el registro DX, y poner el resultado en AL, la intrucción es:

```
IN AL,DX
```

4.11.1 DIRECCIONES DEL ADAPTADOR SERIE I/O

Los puertos I/O utilizados para comunicaciones serie en la IBM PC consisten en una serie de direcciones comenzando en 3F8H para el primer adaptador y 2F8H para el adaptador secundario. Para direccionar un registro particular dentro del UART, de debe de agregar un complemento a estas dirección base que corresponde al registro requerimiento.

Cuando la lógica de reconocimiento de direcciones detecta un número en el rango de 3F8H a 3FFH, o 2F8H a 2FFH, se manda un voltaje al selector del chip del UART. El UART entonces toma los bits 0,1,2 del bus de direcciones, a los cuales esta conectado, y a las lineas de control de bus de lectura o escritura de I/O, a los cuales tambien esta conectado, para decidir que acción tomar. Las direcciones base y complemento para COM1 y COM2 se muestran en la tabla 4.14.

Note que algunos de estos registros comparten la misma dirección. Sin embargo no habrá confución entre el buffer transmisor y receptor ya que la instrucción OUT claramente esta orientada al buffer trasmisor, y cualquier instrucción IN lo estará al buffer receptor

Los latches de división tambien comparten direcciones con otros buffer. Estos son seleccionados por medio del bit de acceso al latch divisor (DLAB) en el registro de control de linea. Cuando DLAB esta en 1, el latch divisor será direccionado. Cuando DLAB esta en cero, el otro registro en la misma dirección será seleccionado.

Tabla 4.14 Direcciones I/O Para COM1 y COM2 para IMB PC

Offset	COM1	COM2	Registro Selecionado
0	3F8	2F8	TX Buffer
0	3F8	2F8	RX Buffer
0	3F8	2F8	Divisor Latch LSB
1	3F9	2F9	Divisor Latch MSB
1	3F9	2F9	Interrup enable register
2	3FA	2FA	Interrup identifatión Register
3	3FB	2FB	Line control register
4	3FC	2FC	Modem control register
5	3FD	2FD	Linea status register
6	3FE	2FE	Modem status retister

4.11.2 INTERRUPCIONES

Ya se ha discutido las diferencias entre interrupciones de hardware e interrupciones de sofware. Debido a que en esta sección se discute la programación a nivel del sistema unicamente se trataran aqui las interrupciones de hardware.

En una IBM PC existe un numero de lines de interrupción. Las interrupciones son generadas alcanzando un cierto nivel de voltaje en esas lineas, manteniendo el nivel hasta que la interrupción hava sido **reconocida**

Hay 8 lineas de interrupción (IR0) en las IBM PC y 16 en las IBM PC-AT. Estas se muestran en las tablas 4.15 y 4.16 respectivamente.

Las lines de IR0 no van directamente al CPU, si no que a un chip dedicado especialmente para manejar interrupciones. Este chip es el 8259A Programable Interrupt Controller (PIC). La IBM PC-AT tiene dos de estos controladores. El PIC establece prioridades entre las interrupciones para impedir el caos que se pueda dar si las interrupciones llegan al mismo tiempo.

Las prioridades en la IBM PC estan asignadas con las mas alta a IR00 y la mas baja a IR07.

El PIC tiene un registro en donde las interrupciones de varios dispositivos son habilitadas. Se asume que IR03 e IR04 no estan habilitadas. Por lo tanto, si se desea realizar un manejador de interrupciones se tiene que instruir al PIC para que habilite las lineas de interrupción apropiadas. Esto se hace leyendo el registro con una instrucción IN al puerto 21H, colocando el bit apropiado a cero (bit 4 para IR04, bit 3 para IR03) y escribiendo este valor de regreso con la intrucción OUT al puerto 21H. Esto es aparte de habilitar las interrupciones en el UART como se mencionó anteriormente.

Cuando el PIC recibe una interrupción, se asume que la interrupción apropiada ha sido habilitada y no esta pendiente otra interrupción, inserta un voltaje positivo en la CPU. La CPU reconoce entonces la interrupción por medio de una señal a el

PIC, y solicita al PIC que le indique cual interrupción ha ocurrido. El PIC envía un numero (vía bus de datos) al CPU (este numero es 8 mas el numero de IR0). En otras palabras, para IR04 (COM1) envía el numero 12 (0CH) y para IR03 (COM2) envía el numero 11 (0BH). La CPU ejecuta entonces una sección apropiada de codigo salvando las direcciones del programa que se esta ejecutando en el stack y ejecutando una llamada lejana a la localización de memoria apuntada por el vector de interrupciones para esa interrupción. Los que se encuentra en dicha localización de memoria es lo que se llama un manejadore de interrupciones. Para mas detalles ver el primer libro de la bibliografía de este capitulo.

Tabla 4.15 Lineas de IR0 para una IMB PC

IR0 Linea	Dispositivo
0	Timer
1	Teclado
2	Reservado
3	Puerto serie 2
4	Puerto serie 1
5	Disco fijo
6	Disco flexible
7	Puerto paralelo 1

Tabla 4.16 Lienes de interrupción para una IBM PC-AT

0	Timer
1	Teclado
2	Compuerta de controloador 1 a 2
3	Puerto serie 2
4	Puerto serie 1
5	Puerto paralelo 2
6	Disco flexible
7	Puerto paralelo 1
8	Clock
9	Redirección desde IR0 2
10	Reservado
11	Reservado
12	Reservado
13	Coprocesador
14	Disco fijo
15	Reservado

4.12 COMUNICACIONES EN BASIC

Esta sección utiliza el interprete de BASIC avanzado versión 2.0 o 3.0 para la IBM PC escrito por Microsoft:

¿Por que BASIC?

En el capitulo uno, se dijo que uno de los errores mas comunes en la gente que cree conocer de computadoras, es pensar que entre mas complicado es un compilador, paquete o interprete, es mas efectivo, independientemente de las necesidades particulares. Esto no es ni debe de ser asi. Es sorprendente, para un usuario avanzado del lenguaje C, descubrir que las facilidades disponibles en el buffer que BASIC maneja son aun mas poderosas que aquellas bajo el lenguaje C. No se puede escribir un manipulador de interrupciones puramente en C ya que hace falta una instrucción IRET. Pero esto si se puede hacer en BASIC

BASIC Automaticamente establece un buffer de salida de 128 bytes, y un buffer de entrada cuya longitud puede ser puesta por el usuario. Los buffer son vaciados y llenados a un nivel de interrupciones. Que se suministran cuando se abre un stream I/O (ver abajo), la recepción de un caracter provoca su colocación en le buffer de entrada automaticamente sin que el programa tenga que hacer algo, y al lograrse la transmisión de un caracter automaticamente se coloca en el buffer de salida el siguiente caracter ha ser enviado (si hay uno). Todo lo que programa tiene que hacer es solicitar a BASIC que envie caracteres, y estos seran colocados en el buffer de salida para su transmisión a traves de un proceso de interrupciones. Similarmente el BASIC puede solicitar la entrada de caracteres, y estos seran removidos desde el buffer de entrada y pasados al programa.

4.12.1 I/O STREAMS

A causa de las similitudes entre escribir a dispositivos y a archivos en disco, muchos de los mismos comandos que se utilizan para accesar archivos en disco se usan para el acceso serie I/O. Aunque BASIC no utiliza este termino, se utilizará la expresión **I/O stream** para referirse a un canal de I/O tratado comm un archivo. Esto es para aclarar que se se esta refiriendo a archivos en disco. Un I/O stream puede ser abierto, leeido, escrito, y cerrado similarmente como un archivo en disco.

4.12.1.1 EL COMANDO OPEN

Un I/O stream se abre enviando a BASIC un comando de inicio con las palabra OPEN "COM". El resto del comando depende de los parametros requeridos. Los posibles parametros se discuten a continuación.

Puerto de comunicacion

Si se desea utilizar COM1, se comienza el comando con

OPEN "COM1

Si se desea utilizar COM2, se comienza con:

OPEN "COM2

En los siguientes ejemplos se utilizará COM1.

Baud Rate

Se puede especificar cualquier baud rate valido (75, 110, 150, 300, 600, 1200, 2400, 4800, o 9600) Despues de la instrucción de abrir. Ejemplo:

Suponiendo que se desea un baud rate de 1200, la instrucción sera:

```
OPEN "COM1,1200
```

Paridad

El siguiente parametro es una sola letra mayuscula que representa la opción de paridad. Y puede ser una de las siguientes:

- S Space
- O Impar
- M Mark
- E Par
- N Ninguna

Si no se desea paridad el comando se verá como sigue:

```
OPEN "COM1, 1200, N
```

Si se utiliza 8 bits de datos, la paridad debe de ser N, si se utiliza cuatro bits de datos, la paridad no puede ser N.

Bits de datos

Se pueden especificar de cuatro a 8 bits de datos agregando el numero apropiado al comando. Si se desean 8 bit de datos, el comando seria:

```
OPEN "COM1, 1200, N, 8
```

Bits de parada

Se pueden especificar ya sea uno o dos bits de parada agregando otro numero a la linea de comando.

Otras opciones de comunicaciones

Se pueden especificar varias opciones de handshaking y otras opciones agregando mas parametros al comando. Estas se listan a continuacion:

RS Suprimir la señal RTS dehandshaking al dispositivo remoto

CSn Indicar cuantos milisegundos esperar para que llegue la señal CTS antes de retornar un error de timeout. El nuemro n puede estar entre 0 y 65535. Si es cero u omitido, entonces CTS es ignorada.

DS Trabaja de la misma forma que CS, excepto que esta relacionada con DSR

CD También trabaja de la misma forma para la linea de CD

LF Causa que se envie un avance de linea (LF) despues de cada retorno del carro.

PE Solicita que la paridad sea chequeada. Si se encuentra un Basic genera un error 57

Numero de archivo

Se debe de agregar un numero de archivo en la linea de comando; este numero será utilizado despues en las intrucciones que se refieran al I/O stream. Por ejemplo, para usar el numero de archivo 1 se agregaria:

AS #1

al comando

tamaño del buffer

Se puede especificar el tamaño del buffer de entrada en bytes con una expresión de la forma:

LEN =

Mas un numero. Si no se desea especificar un tamaño particular, el buffer será de 128 bytes. El tamaño no puede exceder 256 bytes a no ser que se especifique una longitud mayor utilizando la opción /C: cuando se cargue el BASIC. Y no puede exceder 128 bytes a no ser que se especifique un buffer de longitud mas grande utilizando la opción /S: cuando se cargue el BASIC. Por ejemplo, para seleccionar un buffer de 1024 se escribe:

BASICA /S:1024/C:1024

Seguido por supuesto, por otros opciones requeridas en la linea de comando.

Ejemplos:

El siguiente ejemplo especifica COM1,1200 baud, no paridad, ocho bits de datos, un bit de parada, no espera para señales de handshaking, número de stream 1, y un buffer de 128 bytes

```
OPEN "COM1:1200,N,8,1,CD0,DR0" AS #1 LEN=128
```

El siguiente ejemplo especifica COM2, 600 baud, paridad par, siete bits de datos, una espera de 500 milisegundos para CTS, no espera para DSR, número de stream 2, y un buffer de 256 bytes

```
OPEN "COM2:600,E,7,1,CS500,DS0" AS #2 LEN=256
```

4.12.1.2 LECTURA DESDE UN I/O STREAM

La instrucción BASIC INPUT # trabaja de la misma forma con I/O stream como lo hace con archivos. Lee suficientes datos para constituir una variable. Por ejemplo

```
INPUT #1,P$
```

Entra un string y lo asigna a la variable P$

LINE INPUT # y INPUT# también trabajan de la misma forma que con los archivos. LINE INPUT# entra una línea y la coloca en una variable string. INPUT# lee un número fijo de caracteres. Para leer cinco caracteres desde el stream designado como #1, se entraría lo siguiente

```
P$ = INPUT$(5,1)
```

Si no estuvieran presente cinco caracteres en el buffer en el ejemplo anterior, se generaría un error. Para evitar esto, es una buena idea para saber el número de caracteres en el buffer utilizar LOC (ver abajo) y luego entrar ese número de caracteres.
El comando BASIC GET también puede ser utilizado con stream. Se especifica el número de bytes que se leerán desde el buffer y se colocarán en un registro.
LOC(archivo#) retorna el número de caracteres que se encuentran actualmente en el buffer de entrada de un stream. Si el número mayor que 255 BASIC retorna 255. Para leer los caracteres del buffer en la variable P$ se escribe:

```
P$ = INPUT$(LOC(1),1)
```

LOF(archivo#) retorna la cantidad de espacio libre en el buffer, es decir, el tamaño del buffer menos el número de caracteres que

se encuentran en el buffer. EOF(archivo#) indica si el buffer esta vacio, retornando -1 si esta vacio y 0 si no lo esta.

4.12.1.3 ESCRIBIENDO A UN I/O STREAM

PRINT #, y WRITE # trabajan en comunicaciones en la misma forma que trabajan con los archivos. PRINT # envia una variable al stream. WRITE # envia una linea delimitada por comillas. PUT trabaja de manera similar a las forma que GET: se especifican el numero de bytes a escribir:

4.12.1.4 CERRANDO UN I/O STREAM

Para cerrar un stream se utiliza el comando CLOSE, el cual cierra todos los archivos, o se especifica el número del stream a cerrar con el comando CLOSE # seguido del número del stream.

4.12.2 PROGRAMACION DEL UART POR MEDIO DEL BASIC

A pesar de la potencia de las funciones de comunicación en BASIC, hay veces que se necesita control directo sobre el hardware. Esto se podria dar cuando se desea incluir un break, o cuando se desea controlar directamente las lineas de handshaking.

Como se vio en secciones anteriores el UART contiene varios registros. Cada registro contiene un byte de inforamación, y cada bit dentro del byte tiene un significado diferente. En general para saber el estado del UART, es necesario leer los registros y examinar los bits individual dentro de ellos. En general para cambiar el estado de UART, es necesario cambiar los bits individuales dentro de los registros. Por lo tanto, se necesita ejecutar las siguientes funciones en BASIC:

1. Leer un byte desde un registro

2. Examinar los bits dentro de un byte

3. Cambiar los bits dentro de un byte

4. Escribir un byte a un registro

4.12.2.1 LECTURA DE UN BYTE DESDE UN PUERTO

El comando BASIC INP lee un byte desde un puerto. Si ve la tabla 4.14, se vera que el registro de estado del modem par COM1 es 3FEH. Por lo tanto, para leer el registro de estado del modem, se escribe:

```
MODEMSTA = INP(&H3FE)
```

4.12.2.2 ESCRITURA DE UN BYTE HACIA UN PUERTO

El comando OUT de BASIC envia un byte a el puerto. El registro que almacena el byte a trasmitir desde COM1 esta en 3F8H. Para enviar un caracter CH para que sera transmitido, se escribe:

OUT &H3F8,CH

Para ver las condiciones de error consultar manual de BASIC

4.13 UN PROGRAMA DE MUESTRA

El programa que se muestra en el listado 4.1, esta diseñado para descargar archivos a una computadora IBM PC desde otra computadora. La otra computadora debe de ser capaz de imprimir archivos a un printer serie. Y debera de establecerse para imprimir a un baud rate de 1200, ocho bits de datos, un bit de parada, y no paridad. Si se puede escoger que se transmita un avance de linea despues de cada retorno del carro, se debera de seleccionar esta opción.

El programa se ha escrito pensando en la calridad en lugar de la velocidad. Sin embargo, Se ha probado a un baud rate de 1200 y trabaja bien, si se escribe en un disco flexible o en un disco duro.
Para generar un buffer suficientemente largo, se debera invocar al BASIC escribiendo:

BASICA /S:1024/C:1024

Puesto que la IMB PC normalmente esta configurada como un dispositivo DTE, como la mayoria de los printer, se puede utilizar el mismo cable que se utilza para conectar la computadora al impresor: simplemente se reemplaza el printer de la otra computadora por la PC.

Se debera de correr el programa, en la computadora que recibira el archivo,el cual preguntara el nombre del archivo que se desea descargar. Luego se debera de instruir a la otra computadora que envie el archivo con el comando apropiado de impresión.
Los caracteres recibidos seran mostrados en la pantalla a medida que se vayan recibiendo y se salven en disco. Cuando se vea en la pantalla que el archivo ha sido recibido, se presiona ESC para cerrar los archivos y salir del programa.

Con algunas modificaciones este programa se puede utilizar como un convertidor serie a paralelo. Suponiedo que se tiene una computadora PC y un printer paralelo y una computadora CP/M con una interface serie pero no en paralelo. Se pueden imprimir archivos desde la CP/M. Se conecta la CP/M a la PC, y se utiliza el programa, eliminando las lineas 100,110, y 1040, y cambiando

la linea "0" a "00 LPRINT LN$

Tambien es comun para algunos ingenieros trabajar con computadoras portatiles que siempre utilizan discos de 3.5 ", para recoger datos, estos datos son posteriormente almacenados en una computadora mas grande, que no siempre tiene unidas de disco de 3.5, haciendo necesario un convertidor que no es tan barato. El programa anterior puede ser utilizado para lograr estas transferencias. Pero por supuesto, que para el desarrollo de software a nivel comercial, habra que realizar mejoras.

Listado 4.1 Programa para descargar archivos entre dos PC

```
10   '┌──────────────────────────────────────────────────────────────┐
20   '│                        DESCARGA.BAS                          │
30   '│                                                              │
40   '│                  UNIVERSIDAD DE EL SALVADOR                  │
50   '│                  FACULTAD DE INGENIERIA                      │
60   '│                  ESCUELA DE INGENIERIA ELECTRICA             │
70   '│                                                              │
80   '└──────────────────────────────────────────────────────────────┘
100  GOSUB 200:'  PREGUNTAR POR EL NOMBRE DEL ARCHIVO
110  GOSUB 300:'  ABRIR ARCHIVO
120  GOSUB 400:'  ABRIR COMUNICACIONES
130  GOSUB 500:'  COMENZAR MENSAJE
140  GOSUB 600:'  CAPTURAR
150  GOSUB 700:'  CERRAR ARCHIVOS
160  GOSUB 1100:' FIN DE MENSAJE
170  STOP
200 '-----------------------------------------------------------------------
210 '                PREGUNTAR POR EL NOMBRE DEL ARCHIVO
220  CLS
230  INPUT "ENTRE EL NOMBRE DEL ARCHIVO A CAPTURAR ";F$
240  RETURN
300 '-----------------------------------------------------------------------
310 '                ABRIR ARCHIVO
320 '
330  OPEN F$ FOR OUTPUT AS 2
340  RETURN
400 '-----------------------------------------------------------------------
410 '                ABRIR COMUNICACIONES
420 '
430  OPEN "COM1:1200,N,8,1" AS 1 LEN = 10124
440  RETURN
500 '-----------------------------------------------------------------------
510 '                COMENZAR MENSAJE
520 '
530  PRINT
540  PRINT "PUEDE INSTRUIR A LA OTRA COMPUTADORA QUE ENVIE ";F$
550  PRINT "PRESIONE ESC CUANDO FINALICE LA CAPTURA"
560  RETURN
```

Listado 4.1 (continuación).

```
600 '        --- ------------------------------------------------------------
610 '                    CAPTURA
620 '
630 PAUSA = 0: C$=""
640 WHILE C$     CHR$(27)
650 IF EOF(1) THEN 710          :'NADA RECIBIDO
660 B=LOC(1)                    :'NUMERO DE CARACTERES EN EL BUFFER
665 IF B  671 THEN 500          :'BUFFER LLENO EN 2/3 PARAR ENTRADA
666 ESTBUF=(B 337 AND PAUSA=0)
670 IF ESTBUF THEN 900          :'BUFFER EN 2/3 REANUDAR ENTRADA
680 LN$ = INPUT$(LOC(1),#1)     :'LEER BUFFER EN LN$
690 PRINT LN$                   :'DESPLEGARLO EN LA PANTALLA
700 PRINT #2, LN$               :'GRABARLO EN EL ARCHIVO
710 C$=INKEY$
720 WEND
730 RETURN
800 '-----------------------------------------------------------------------
810 '                    DESHABILITAR HANDSHAKING
820 '
830 IF PAUSA=1 THEN 870         :'YA ESTA EN MODO DE PAUSA
840 OUT &HF3FC,0                :'APAGAR DTR Y DSR
850                             :'EN REGISTRO DE CONTROL DE MODEM
860 PAUSA = 1
870 RETURN
900 '-----------------------------------------------------------------------
910 '                    HABILITAR HANDSHAKING
920 '
930 OUT &HF3FC,3                :'BIT 1 Y 2, DTR Y DSR ENCENDIDOS
940                             :'EN REGISTRO DE CONTROL DE MODEM
950 PAUSA = 0
960 RETURN
1000 '-------------  ------- ---------------------------------------------
1010 '                    CERRAR
1020 '
1030 CLOSE #1                   :'CERRAR EL COM1
1040 CLOSE #2                   :'CERRAR ARCHIVO DISCO
1050 RETURN
1060 '-----------------------------------------------------------------------
1070 '                    FIN DE MENSAJE
1080 '
1090 PRINT
1100 PRINT "CAPTURA FINALIZADA"
1110 RETURN
1120 '-----------------------------------------------------------------------
```

CONCLUSIONES

-Ni en las funciones de comunicación del DOS ni del BIOS son muy adecuadas, y a ellas se les ha dado claramente baja prioridad por los diseñadores de sofware de comunicaciones. Quizas se pueda esperar mejores soportes en futuras versiones del DOS, pero en este momento sera necesario utilizar programación al mas bajo nivel para lograr un completo aprovechamiento de las capacidades de comunicación de la IBM PC.

Como advertencia acerca de las AT, es necesario decir que por ser mucho mas rapidas que las PC, existe el riesgo que puedan ser demasiadas rapidas para el 8250. En otras palabras, puede enviar instrucciones al 8250 mas rapido de lo que este puede procesarlas. Por esta razon, no debera de evitar codigos que envien sucesivamente al 8250; Se deberan de colocar retardos entre codigos que envien datos sucesivamente al 8250. Esto solo se aplica cuando se esta programando en lenguaje de ensamble.

De los dos métodos que existen para diseñar software de comunicaciones, el de escrutinio (polling) y el de interrupciones, se puede decir que su selección dependerá del medio en que se utilizara el software. Si se estan utilizando comandos del DOS el unico medio disponible es el de escrutinio. Este método a pesar de que requiere que la computadora este dedicada unicamente a la comunicación, es ampliamente utilizado si se desea que la computadora realice trabajo util mientras espera comunicarse entonces el método deberá de ser el de las interrupciones, claro esta que es mucho mas complicado.

-Aunque el UART es un dispositivo diseñado para comunicaciones, tambien puede ser utilizado para el control de procesos, es necesario investigar por lo tanto mas acerca de este chip

BIBLIOGRAFIA

1 GW-BASIC Version 3.22
User's Guide
User's Reference
Tacard Bell

2 Dettmann, DOS Programmer's Reference
QUE

3 W. Sutton, Peter, Mastering Serial Communications
Sybe

CAPITULO V

EL SOFTWARE EN ROM

Introducción

Se utiliza el software para hacer funcionar un ordenador. Conseguir que funcione y se mantenga funcionando es mucho más fácil si parte del software está implementando permanentemente en el interior del ordenador. Esta es la razón de la existencia de los programas en la ROM.(ROM son las iniciales de Read Only Memory, que se traduce por memoria de sólo lectura.)Esta memoria está registrada permanentemente en la circuitería de los chips del PC y no puede ser alterada, borrada o perdida. Los PC vienen con una cantidad sustancial de ROM, que contiene los programas y los datos que son necesarios para activar y hacer funcionar al ordenador y sus dispositivos periféricos. La ventaja de tener los programas fundamentales del ordenador almacenados en la ROM es que están allí- implementados en el interior del ordenador- y no hay necesidad de cargarlos en la memoria desde el disco, de la misma forma en que se carga el DOS. Debido a que están siempre residentes, los programas ROM son muy a menudo los cimientos sobre los que se construye el resto de los programas (incluyendo el DOS).

Hay cuatro elementos diferentes en las ROM de la familia PC de la IBM: los programas de arranque, que hacen el trabajo de poner en marcha el ordenador; la ROM-BIOS, que es un acrónimo para indicar el sistema de entrada/salidas básico y que está formada por una colección de rutinas en lenguaje máquina, que proporciona los servicios de soporte para las operaciones del ordenador; la ROM BASIC, que proporciona el núcleo del lenguaje de programación BASIC , y las extensiones ROM, que son programas que se añaden a la ROM principal cuando se conectan al ordenador ciertos equipos especiales. Se examinarán cada uno de éstos cuatro elementos principales en lo que falta de este capítulo.

El bloque situado en la zona más alta de la memoria se reserva para almacenar los programas en ROM, comenzando en el segmento F000 hexadecimal. Los diferentes modelos de la familia PC usan diferentes cantidades de este espacio de 64K, dependiendo de la complejidad de sus necesidades. Por ejemplo, el modelo original del PC, con su hardware relativamente simple, utiliza sólo 40K de los 64K del bloque F para los programas en ROM, mientras que el PCjr y el AT, con su hardware mucho más complejo, usan completamente el espacio de 64K.

5.1 LA ROM de arranque

El primer trabajo de los programas en ROM tiene que ser supervisar la puesta en marcha del ordenador. A diferencia de

otros aspectos de la ROM, las rutinas de arranque están poco relacionadas con la programación de la familia PC, pero vale la pena comprender lo que hacen.
Son varias las tareas realizadas por las rutinas de puesta en marcha. Por ejemplo, ejecutan una comprobación rápida de la fiabilidad del ordenador (y de los programas de la ROM) para estar seguros de que todo está trabajando en orden: inicializan los chips y vectores de interrupción, comprueban qué equipo opcional está conectado y, si está conectado un controlador de disco, terminan cargando el sistema operativo situado en ese disco.
El test de la fiabilidad, parte de un proceso conocido como POST (Power on Self test), es un primer paso importante que permite estar seguros de que el ordenador se encuentra listo. Todas las rutinas del POST son bastante breves, en tiempo, excepto las que se dedican a realizar los test de memoria, que pueden ser fastidiosamente largos, cuando el ordenador contiene una gran cantidad de memoria.
El proceso de inicialización es ligeramente más complejo. Una rutina pone los valores por defecto de los vectores de interrupción. Estos valores por defecto apuntan hacia las rutinas dedicadas al tratamiento de interrupciones estándar, que se encuentran localizadas en el interior de la ROM-BIOS, o apuntan hacia rutinas que no hacen nada, y que los programas suplirán después. Otra rutina de inicialización determina qué equipo está conectado al ordenador, y entonces sitúa un registro de él en localizaciones estándar de la parte baja de la memoria.(Se discutirá esta lista con más detalle en el capítulo). La forma de adquirir esta información varía de modelo a modelo; por ejemplo, en el PC se recoge en su mayor parte de las posiciones de dos bancos de microinterruptores que tiene la placa del sistema ordenador; en el PCjr se determina por inspección lógica y la comprobación (de hecho, el programa de inicialización sondea a todas las posibles opciones y espera respuesta); la información se lee de un área de memoria no volátil especial (que puede ser fijada por los programas de diagnóstico).
Con cualquier método que se utilice, la información del estado queda registrada y almacenada de la misma forma en que todos los modelos, de manera que los programas pueden monitorizarla. Las rutinas de inicialización comprueban también los nuevos equipos y las extensiones de la ROM. Si encuentran algo, dan momentáneamente el control a las extensiones de la ROM, de forma que puedan inicializarse a sí mismas. Posteriormente, las rutinas de inicialización continúan ejecutando las rutinas de puesta en marcha que quedan (veremos más sobre ésto después).
La parte final de procedimiento de puesta en marcha, después de los test del POST, el proceso de inicialización y la incorporación de las extensiones de la ROM, se denomina cargador bootstrap, y es una rutina corta que se emplea para cargar un programa almacenado en el disco. En esencia, el cargador bootstrap intenta leer un registro, llamado registro de arranque, que está localizado en un disco, y si es satisfactorio el proceso,

pasa el control del ordenador al programa que está incluido en aquel registro. Este programa tiene la tarea de cargar el resto del programa del disco. Normalmente, este programa es el sistema operativo del disco DOS, pero podría ser un programa diferente y autocargable, como, por ejemplo, el simulador de vuelo de Microsoft. Si el cargador boot-strap no puede leer el registro cargador del disco, activa simplemente el "cassette" BASIC implementando en la ROM. (Para los equipos que no son miembros de la familia ampliada de PC, se visualiza el mensaje de error non-boot). Tan pronto como ocurre uno de estos dos procesos, el procedimiento de puesta en marcha del sistema termina, y los otros programas se encuentran listos para funcionar.

5.2 La ROM-BIOS

La ROM-BIOS es la parte de la ROM que está en actividad durante todo el tiempo que está trabajando el ordenador. El papel de la ROM-BIOS es proporcionar los servicios fundamentales que se necesitan para que se puedan realizar todas las operaciones del ordenador. En la mayor parte de las ocasiones el BIOS controla los dispositivos periféricos del ordenador, tales como la pantalla, el teclado y los controladores del disco. Cuando se utiliza el término BIOS en su sentido más riguroso, se hace referencia a los programas encargados de realizar el control de dispositivos. Estos programas transforman un simple comando, tal como "leer algo de un disco", en todos los pasos necesarios que permitan realizar de forma efectiva dicha acción, incluyendo la detección de errores y su corrección. En el sentido más amplio, BIOS no solo hace referencia a las rutinas que son necesarias para controlar los dispositivos del PC, sino también hace referencia a las rutinas que contienen información, o realizan tareas, que son fundamentales en otros aspectos del funcionamiento del ordenador, como, por ejemplo, guardar la hora y el día.

Puede imaginarse que los programas del BIOS están entre los programas personales (incluyendo el DOS) y el hardware. En efecto, esto quiere decir que, por un lado, recibe las solicitudes efectuadas por los programas para realizar los servicios de entrada/salida. Estos servicios se llaman desde los programas mediante combinación de dos números : uno , el de la interrupción (que indica el sujeto que ha solicitado un servicio, como, por ejemplo, la impresora), y el otro, un número de servicio (que indica el servicio específico que se debe realizar). Por otro lado, el BIOS se comunica con los dispositivos hardware del ordenador (pantalla, controladores de disco, etc.), usando los códigos de control específico que requiere cada dispositivo. En esta actividad, el BIOS también maneja algunas interrupciones hardware producidas por algún dispositivo para atraer la atención del procesador. Por ejemplo, siempre que se presiona una tecla, el teclado genera una interrupción que reconoce el BIOS.

De todo el software de la ROM, los servicios del BIOS son probablemente los más interesantes y útiles para los programadores. Dado que se tratará a fondo más adelante, evitaremos sobre discusiones específicas de qué es lo que hacen los servicios del BIOS y focalizaremos la atención sobre cómo trata el BIOS los procesos de entradas y salidas del ordenador.

5.3 Vectores de interrupción

La familia IBM PC, como todos los otros ordenadores basados en la familia 8086 de microprocesadores de Intel, se controla a través del uso de interrupciones, que pueden generarse o bien por hardware o bien por software. Las rutinas de servicio del BIOS no son una excepción: a cada una se le asigna un número de interrupción al que hay que llamar cuandob se desea utilizar el servicio.

Cuando ocurre una interrupción, el control del ordenador pasa a la subrutina de tratamiento de la interrupción, que a menudo está almacenada en la ROM del sistema (una rutina de servicio del BIOS no es nada más que un programa de gestión de interrupciones). El programa de gestión de la interrupción se llama cargando su dirección segmentada en el registro que controla el flujo del programa el registro CS (segmento de código) y el registro IP (puntero de instrucciones), pareja de registros que se conoce como el par CS:IP.
Los vectores de interrupción se colocan, previamente, durante el proceso de puesta en marcha del sistema para apuntar hacia las rutinas dedicadas al tratamiento de las interrupciones en rom. Se almacenan en una tabla en memoria RAM como si fueran pares de palabras, con la porción de la dirección relativa, en el primer lugaar, y la porción del segmento, en segundo lugaar. Los vectores de interrupción se pueden cambiar para apuntar a una nueva rutina de tratamiento de la interrupción, simplemente localizando el vector y cambiando su valor.
Como regla general, las interrupciones de la familia PC se pueden dividir en tres categorías: microprocesador, hardware, software. (DOS, BASIC, direccionamiento y uso en general.
Las interrupciones de microprocesador, llamadas a menudo interrupciones básicas, están integradas en el propio microprocesador. Cuatro de ellas(las interrupciones 0,1, 3 y 4) se generan por el propio microprocesador, y otra (la interrupción 2, que es la interrupción no enmascarable) se activa mediante una señal generada por uno de los dispositivos externos.

Las interrupciones del hardware están implementadas en el hardware del PC. Ocho de estas interrupciones están cableadas en el interior del microprocesador, o en la placa principal del sistema, y no pueden ser cambiadas. Todas las interrupciones del hardware son supervisadas por el chip 8259A PIC. Los códigos reservados son 2, 8, 9 y del 11 al 15.
Las interrupciones del software están incorporados dentro del

diseño del PC y son parte de los programas de la ROM-BIOS. Las rutinas del BIOS activadas mediante estas interrupciones no pueden cambiarse, pero los vectores que apuntan a estas rutinas sí que pueden ser cambiadas para apuntar hacia rutinas diferentes. Los códigos reservados son 5,16 al28 y 72.

3.4 Las competencias del DOS

Después de que el cargador boot-strap pasa el control al registro de arranque del disco, éste chequea para ver si el DOS está almacenado en el disco, buscando dos archivos de programa ocultos, llamados IBMBIO.COM e IBMDOS.COM. Si los encuentra, los carga en la memoria junto con el intérprete de comandos del DOS, COMMAND.COM. Durante este proceso, las partes opcionales del DOS tales como los controladores de los dispositivos instalables, también pueden ser cargados.
El archivo IBMBIO.COM contiene ampliaciones a la ROM-BIOS.
Estas ampliaciones pueden ser cambios, o adiciones, de las operaciones básicas de E/S, que incluyen a menudo correcciones de la ROM-BIOS existente, o nuevas rutinas para nuevos equipos, o cambios personalizados de las rutinas estándar ROM-BIOS. Ya que son parte del software del disco, las rutinas del IBMBIO.COM proporcionan una forma conveniente para modificar la ROM-BIOS. Todo lo que se necesita, junto con las nuevas rutinas, es que los vectores de interrupción de las rutinas previas se cambien para apuntar hacia las nuevas localizaciones de memoria, donde encuentran las nuevas. Siempre que se añada algún dispositivo al ordenador, sus programas de soporte se pueden incluir en el archivo IBMBIO.COM, o también como un controlador del dispositivo instalable, eliminando la necesidad de reemplazar los chips de ROM. Puede imaginarse que las rutinas de la ROM-BIOS son el software del sistema del más bajo nivel disponible, que permiten realizar las operaciones de entrada y salida más fundamentales, y que se pueden todavía considerar primitivas. Las rutinas del IBMBIO.COM, que son extensiones de la ROM-BIOS, están esencialmente al mismo bajo nivel, proporcionando también funciones básicas. Por comparación, las rutinas del IBMDOS.COM son más sofisticadas,y se pueden pensar en ellas como las que ocupan el nivel siguiente a los lenguajes de programación, que son de nivel superior.
El archivo IBMDOS.COM contiene las rutinas de servicio del DOS. Los servicios del DOS, como los servicios del BIOS, se pueden llamar desde los programas a través de un conjunto de interrupciones, cuyos vectores están situados en la tabla de vectores de interrupción que se encuentran en las posiciones bajas de la memoria. Una de las interrupciones del DOS, la interrupción 33 (21 hex), es particularmente importante, porque, cuando se llama, da acceso a un gran grupo de rutinas secundarias, llamadas funciones del DOS. Las funciones del DOS proporcionan un control sobre las operaciones con los archivos del disco. Todos los procesos del disco estándar -formateo de diskettes, lectura y escritura de datos, abrir,cerrar y borrar

archivos, efectuar búsquedas en el directorio- están incluidos en las funciones del DOS y proporcionan la base a otros programas del DOS de nivel más alto, tales como FORMAT, COPY y DIR. Los programas pueden utilizar los servicios del DOS cuando es necesario un control mayor de las operaciones de E/S que el permitido por los lenguajes de programación normales y cuando se es reacio a profundizar en el nivel del BIOS.
El archivo COMMAND.COM es la tercera de las partes más importantes del DOS, al menos desde un punto de vista utilitario. Este archivo contiene las rutinas que interpretan lo que se escribe a través del teclado, cuando se está en modo comando del DOS. Compara las palabras tecleadas con una tabla de nombre de comandos.
COMMAND.COM puede diferenciar entre comandos internos que son parte del archivo COMMAND.COM, tales como RENAME o ERASE, y comandos externos, tales como los programas de utilidad del DOS(como DEBUG, o alguno de nuestros propios programas). Actúa sobre las entradas, ejecutando las rutinas requeridas por los comandos internos, o buscando en el disco los programas requeridos y cargándolos en la memoria. La información completa sobre el archivo COMMAND.COM, y de cómo trabaja, es interesante, y por su importancia merece ser investigada, de la misma forma que los programas del DOS. Le recomendamos que lea el manual de "Referencia Técnica del DOS" o el IBM PC a fondo para encontrar la información adicional.

Las interrupciones del DOS se encuentran siempre disponibles cuando el DOS está en uso.Algunos programas y lenguajes de programación utilizan estos servicios proporcionados, a través de las interrupciones del DOS,para manejar sus operaciones básicas especialmente las E/S del disco. Los códigos reservados van del 32 al 255 (del 32 al 96 son los que se utilizan, los otros no se aconseja tocarlos).

Las interrupciones del BASIC son asignadas por el propio BASIC y están siempre disponibles cuando se utiliza el BASIC. Los códigos reservados van del 128 al 240.
Las interrupciones de direccionamiento son una parte de la tabla de los vectores de interrupción y se utilizan para almacenar direcciones segmentadas. No hay interrupciones, o subrutinas de tratamiento, asociadas con estas interrupciones. En particular, están asociadas con tres tablas muy importantes: la de inicialización del video, la tabla base del disco y la de caracteres gráficos. Estas tablas contienen parámetros que utiliza la ROM-BIOS en procedimientos de puesta en marcha y para la generación de caracteres gráficos. Los códigos reservados van del 29 al 31, el 68 y el 73 (JR 68 y 73 se utilizan solo con el PCjr.
Las interrupciones de uso general están establecidas para los programas para su uso temporal. Los códigos reservados van del 96 al 103.
Los vectores de interrupción están almacenados en las localizaciones más bajas de la memoria; la primera localización

de memoria contiene el vector para la interrupción número 0, y así sucesivamente. Como cada vector tiene una longitud de dos palabras, se localiza cualquier interrupción en la memoria multiplicando su número de interrupción por cuatro. Por ejemplo, el vector para la interrupción 5, que es la interrupción de servicio de escritura en pantalla, tendría el byte de dirección relativa de 20 (5*4 =20). Se pueden examinar los vectores de interrupción representando este número decimal en notación hexadecimal y utilizando DEBUG (que sólo acepta valores hexadecimales). En el caso de la interrupción 5, la localización 20 se representa mediante el valor hexadecimal 14, y los siguientes comandos:

```
DEBUG
D 0000:0014 L 4
```

que mostrarán cuatro bytes, en hexadecimal, como éstos:

```
54 FF 00 F0
```

Convirtiéndolos en una dirección segmentada y teniendo en cuenta el almacenamiento invertido, se puede ver que el vector de interrupción para el punto de entrada en la ROM de la rutina de servicio de escritura en pantalla(interrupción 5) es F000:FF54. La misma instrucción DEBUG se puede utilizar para encontrar otros vectores de interrupción de la misma manera.
La tabla 5-1 es un listado de las interrupciones principales y de las localizaciones de sus vectores. Estas son las interrupciones que los programadores suelen encontrar más relacionadas con la mayoría de estas interrupciones. Las interrupciones que no están mencionadas en esta lista están, en su mayor parte, reservadas para los desarrollos futuros del IBM.

5.5 Cambio de los vectores de interrupción

El interés principal de la programación con vectores de interrupción no es leerlos, sino cambiarlos, de forma que apunten a una nueva rutina de tratamiento de la interrupción. Para hacer esto, se debe de escribir una rutina que realice una función diferente a la que realizan las interrupciones estándar de la ROM-BIOS,o del DOS, almacenar la rutina en RAM, y entonces asignar una nueva dirección al vector de la interrupción ya existente en la tabla.
Un vector puede ser alterado byte por byte utilizando lenguaje ensamblador , o utilizando una instrucción del lenguaje de programación como el POKE del BASIC. En algunos casos puede haber peligro de que ocurra alguna instrucción en la mitad de la realización del cambio del vector. Si no está interesado en esta casuística, puede utilizar el POKE. Si está interesado en evitarla, hay dos formas distintas para cambiar un vector, mientras se toman precauciones contra su utilización cuando se

cpld en la mitad del proceso del cambio.
El primer método es cambiar el vector suspendiendo las interrupciones mientras se realiza el cambio: para ello se utiliza una instrucción de inhibición de interrupciones (CLI). El CLI suspende todas las interrupciones, excepto la interrupción no enmascarable.

Tabla 5.1 Interrupciones principales utilizadas en la familia de los ordenadores personales de IBM

INTERRUPCION DECIMAL	INTERRUPCION HEXADECIMAL	DIRECCION	USO
0	0	0000	Generada por la CPU cuando se realiza una division por cero
1	1	0004	Usada para ejecutar programas paso a paso (como con DEBUG)
2	2	0008	Interrupción no enmascarable.
3	3	000C	Usada para poner puntos de ruptura en los prog.
4	4	0010	Usada cuando un resultado aritmetico se desbord.
5	5	0014	Invoca a la rutina de servicio de escritura en pantalla del BIOS.
8	8	0020	Tic de reloj generado por hardware
9	9	0024	En la mayoria de los modelos, generada por la acción del teclado, simulada por el PC Jr para compatibilizar el modelo.
13	D	0034	Generada durante el retorno vertical del haz en el CTR,para el control del video.
14	E	0038	Señales de atención del diskette (por ejemplo señalizar finalización)
15	F	003C	Usada para el control de la impresora
16	10	0040	Invoca al servicio de visualización del video del BIOS.
17	11	0044	Invoca al servicio del listado de equipo del BIOS.
18	12	0048	Invoca al servicio de

Tabla 5.1 (continuación)

			tamaño memoria del BIOS
19	13	004C	Invoca a los servicios de diskette del BIOS.
20	14	0050	Invoca el servicio de comunicaciones del BIOS
21	15	0054	Invoca a los servicios de cassete de cinta mag nética del BIOS
22	16	0058	Invoca al servicio de te teclado estándar del BIO
23	17	005C	Invoca el servicio de impresora del BIOS
24	18	0060	Activa o desactiva el lenguaje ROM-BASIC
25	19	0064	Invoca a la rutina de puesta en marcha de BIOS
26	1A	0068	Invoca el servicio de hora y fecha del BIOS
27	1B	006C	Interrupción generada en el teclado. actúa bajo el BIOS
28	1C	0070	Interrupción generada en cada tic de reloj,se activa una rutina si esta creada.
29	1D	0074	Apunta a la tabla de parámetros de control de video.
30	1E	0078	Apunta a la tabla base del disco.
31	1F	007C	Apunta a la parte alta de los caracteres gráficos del video.
32	20	0080	Invoca al " servicio de programa terminado " del DOS.
33	21	0084	Invoca a todos los servicios de tipo función del DOS.
34	22	0088	Si está creada. se invoca una rutina de interrupción producida por teclado bajo el DOS
35	23	008C	Si está creada. se invoca una rutina de interrupción producida por teclado bajo el DOS
36	24	0090	Si está creada.se invoca una rutina deinterupción cuando se produce error crítico bajo el DOS

Continuación Tabla 5.1

37	25	0094	Invoca el servicio de lectura de diskette del DOS
38	26	0098	Invoca el servicio de escritura del diskette del DOS
39	27	009C	Termina el programa, pero lo mantiene en memoria bajo el DOS
68	44	0110	Apunta a la parte baja de los caracteres gráficos del video; solo en el PCJr
72	48	0120	Invoca el programa para transformar el teclado PCJr en teclado del PC
73	49	0124	Apunta a la tabla de transformación para los dispositivos suplementarios del teclado

(NMI). Se supone que la NMI sólo será usada para señalar una situación verdaderamente urgente, como puede ser una situación de tipo "incendio-en-el-ordenador", pero desafortunadamente se ha venido utilizando también para situaciones muy ordinarias, tales como la señalización de una acción en el teclado del PCjr. Como consecuencia, aun cuando el enmascaramiento de las interrupciones con CLI ofrece una seguridad razonable contra la interrupción en la mitad del cambio de un vector de interrupción, no es perfecta. Le mostraremos dos ejemplos de este primer método- que permiten poner un vector de interrupción con las interrupciones -inmendidas-. El primer ejemplo pone el vector con dos instrucciones MOV, que mueven las dos palabras del vector a su lugar:

```
XOR   AX,AX                 ; acumulador a cero
MOV   ES, AX                ; registro segmento a cero
CLI                         ;suspende las interrupciones
MOV   WORD PTR ES:36 XX     ;mueve la parte de dirección relativa al
                             vector
MOV   WORD PTR ES:38,YY     ;mueve la parte de segmento del vector
STI                         ;activa las interrupciones.
```

Aunque esta técnica es correcta, se corre el pequeño riesgo de que venga una NMI entre las instrucciones MOV (aunque es un riesgo muy pequeño). El riesgo se puede reducir combinando los dos MOV en una simple instrucción de movimiento (MOVS). El uso de la instrucción de movimiento de cadena es mucho más pesado.

porque requiere inicializar varios registros. : primero se ajustan los registros para el movimiento requerido.

```
XOR   DI,DI     : obtiene una palabra cero
MOV   ES,DI     : ajusta el párrafo a (=0)
MOV   DI,36     : ajusta la dirección relativa a número
MOV   SI,XXXX   : ajusta la dirección relativa de
MOV   CX,2      : número de palabras
CLD             : ajusta el sentido hacia adelante

: ahora hagamos el movimiento con las interrupciones suspendidas

CLI             : suspende las interrupciones
REP MOVSW       : movimiento repetido de palabras
STI             : reactiva las interrupciones
```

Se han mostrado dos formas diferentes para cambiar un vector de interrupción, siguiendo el método de "hágalo usted mismo". El otro método es permitir que lo haga el DOS, empleando la rutina de servicio del DOS número 37, que fue diseñada con este propósito. Hay dos ventajas muy importantes al permitir que el DOS varíe las interrupciones en lugar de hacerlo por nosotros mismos. Una ventaja es que el DOS realiza la tarea de poner el vector en su lugar de la forma más segura posible. La otra ventaja es de más alcance. Con la aparición del chip procesador 80286 en el modelo AT, la familia PC está empezando a pasar al campo en que elementos tan familiares como los vectores de interrupción y los registros de segmento no son lo que solían ser. Usar un servicio del DOS, para poner un vector de interrupción en lugar de ponerlo nosotros mismos, es una de las formas con la que podemos reducir el riesgo de que nuestros programas sean incompatibles con las nuevas máquinas o los nuevos sistemas operativos.
Aquí hay un ejemplo de cómo usar el servicio de 37 del DOS para poner un vector de interrupción:

```
MOV   DX,XX     : carga la parte de dirección relativa del vector
MOV   DS,YY     : carga la parte de segmento del vector
MOV   AH,37     : pide la función de ajuste de interrupción
MOV   AL,?      : cambia la interrupción número ?
INT   33        : interrupción de llamada a función DOS
```

Este ejemplo muestra, de la forma más simple posible, cómo se puede utilizar el servicio del DOS. Sin embargo, ilustra una dificultad importante y sútil: se tienen que cargar una de las direcciones en el registro DS (segmento de datos) - lo que bloquea el acceso normal a los datos a través del registro DS. Para evitar este problema de una manera útil, he aquí una forma de hacerlo.

```
PUSH  DS                        : salva el segmento de datos en curso
```

```
MOV   DX,OFFSET PGROUP:XX  ; obtiene la dirección relativa del
                             vector
PUSH  CS                   ; mueve nuestro propio segmento de
                             código
POP   DS                   ;...al segmento de datos
MOV   AH,37                ; pide la función de ajuste de
                             interrupción
MOV   AL,9                 ; cambia la interrupción número 9
INT   21H                  ; interrupción de llamada a función
                             DOS
POP   DS                   ; realmacena nuestro segmento de
                             datos original
```

5.6 Referencia de las direcciones bajas de memoria

Muchas de las operaciones del PC están controladas por datos que se encuentran en las posiciones bajas de la memoria, particularmente en las dos áreas adyacentes de 256 bytes, que empiezan en 400 y 500 hex.
Los datos se almacenan en esas áreas desde el BIOS durante el proceso de puesta en marcha. Aunque el control de estos datos se supone que está reservado al BIOS, los programas pueden inspeccionarlos, o cambiarlos. Aunque no desee usar la información del área de control del BIOS, es importante su estudio, porque revela gran cantidad de detalles sobre el funcionamiento interno de la familia PC.
Para evitar confusión sobre estas direcciones bajas de la memoria, tenga en cuenta que la dirección de memoria 400 puede también estar expresada en formato segmentado, como 0040:0000 o 0000:0400. Las tres notaciones se refieren exactamente a la misma posición.

5.6.1 El área de información de control

Algunas de las posiciones de memoria de las áreas 400 y 500 hex son particularmente interesantes. La mayoría de ellas contienen datos vitales para la operación de varias rutinas de servicio del BIOS y del DOS. En algunos casos, los programas pueden restituir la información almacenada en esas localizaciones, invocando una interrupción del BIOS; en todos los casos se puede acceder a la información directamente. Para chequear fácilmente los valores de estas localizaciones en su propio ordenado, use el DEBUG o el BASIC.

Para usar el DEBUG, introduzca estos comandos:

```
DEBUG
D XXXX:0000  L 1
```

En este ejemplo, XXXX representa la dirección en hexadecimal que

quiere examinar. El L 1 le dice al DEBUG que visualice un byte. Para ver más o más bytes, introduzca después de la instrucción L el número de bytes (en hexadecimal) que desee ver.
Para visualizar datos con el BASIC, se puede utilizar el programa que se muestra abajo, haciendo las sustituciones necesarias para dirección en hex y número de bytes en decimal:

```
10 DEF SEG=0
30 FOR I= 0 TO  numero.de.bytes.en.decimal -1
40  VALOR = PEK (&Hdirección.en.hex.  + I)
49  IF VALOR <16 THEN PRINT "0";es necesario para encabezar con 0
50  PRINT HEX$ (VALOR);  " ";
60 NEXT I
```

Se han listado las direcciones más útiles en las siguientes páginas. Todas ellas están dadas en hexadecimal.
410(una palabra de 2 bytes). Esta palabra almacena los datos de la lista de equipos,que es recogida por el servicio de listado de equipos, interrupción 17 (11 hex). El formato de esta palabra aparece mostrada en la tabla 5-2 establecida para el PC y el XT, y ciertas partes pueden aparecer con un formato diferente en los modelos posteriores, incluyendo el PCjr.

tabla 5.2 Codificación de la palabra asociada a la lista del equipamiento situada en la dirección 410 hex.

BIT F E D E B A 9 8	7 6 5 4 3 2 1 0	SIGNIFICADO
X X	. . . .	Número de impresoras
. . X	. . . .	1 si está instalada la impresora serie
. . . X	. . . .	1 si está instalado el Joystick
. . . . X X X .	. . . .	Número de RS-232
. . . . X	. . . .	0 sin está instalado el chip de DMA
. . . .	X X	Número de unidades de diskettes-1: 00=1 unidad;01=2 unidades 10=3 " " ;11=4 " "
. . . .	. . X X	Modo video inicial:01=color a 40 columnas en el PCJr 10=color a 80;columnas;11= 80 columnas monocromo para el resto de ls modelos
. . . .	. . . . X X . .	Placa RAM del sistema:00= 16k;01=32k;11=64k
. . . .	 X	1 si está presente alguna unidad de disco

412(un byte). Este byte es utilizado únicamente en el PCjr para contar el número de errores detectados en la conexión del teclado por infrarrojos. Otros modelos utilizan este byte solo durante la inicialización. Un byte interesante, pero que no tiene ningun significado útil desde el punto de vista de programación.
413(una palabra de dos bytes). Contiene el tamaño de memoria utilizable en Kb. En el PCjr devuelve la cantidad de memoria que queda libre después de reservar la memoria de pantalla. En otos modelos esta palabra tiene el mismo propósito: le dice cuánta memoria se puede utilizar. El servicio de interrupción del BIOS número 18 (12 hex) escribe el valor en esta palabra.
417(dos bytes de bits de estado del teclado). Estos bytes son utilizados para controlar la interpretación de las acciones del teclado por las rutinas de la ROM-BIOS. Cambiando estos bytes, se cambia el significado de las pulsaciones de las teclas. Puede cambiar libremente el primer byte, en la localización 417, pero no es una buena idea cambiar el segundo byte. 419(un byte). Este byte está reservado para controlar las entradas alternativas al teclado. Está destinada para usos futuros.
41A(una palabra de dos bytes). Esta palabra apunta a la cabecera del buffer del teclado del BIOS situado en la dirección 41E. Sitio donde se almacenan las acciones de las teclas hasta que son utilizadas.
41C(una palabra de dos bytes). Esta palabra apunta a la cola del buffer.
41E(32 bytes, usados como 16 elementos de dos bytes). El buffer del teclado es utilizado para almacenar las 16 últimas acciones realizadas en el teclado, hasta que son leídas por los servicios del BIOS a través de la interrupción 22 (16 hex). Este es un buffer de cola de espera circular: ésta es la razón por la cual hay dos punteros para indicar la cabecera y la cola (en 41A y 41C). No es prudente enredar con estos datos.

43E(un byte). Este byte indica si los diskettes necesitan ser calibrados antes de buscar una pista. Los bits 0 al 3 corresponden a las unidades 0 a la 3. Si un bit vale 0, es necesaria la recalibración. Generalmente encontrará que está puesto a 0 si ha habido algún problema durante la utilización más reciente de la unidad del disco. Por ejemplo, el bit de recalibración será 0 si trata de solicitar el directorio (DIR) de una unidad sin diskette, y pulsa C en respuesta a la pregunta:

ERROR: Unidad no preparada, Leyendo la unidad 0
Cancelar, Reintentar, Ignorar?

43F(un byte). Este byte devuelve el estado del motor del diskette. Los bits 0 al 3 corresponden a las unidades 0 a la 3. Si el bit es 1, el motor del diskette está funcionando.
440(un byte). Este byte almacena la cuenta atrás hasta que el motor del diskette se detiene. La cuenta se pone a 37 (aproximadamente 2 segundos) al principio de cada operación del diskette. Con cada señal de reloj, la cuenta es decrementada. El motor del diskette se detiene cuando la cuenta llega a cero.

441(un byte). Este byte indica el estado del diskette, cada bit representa un tipo de error particular (vease la tabla 5-2). Un bit de valor 1 señala que ha ocurrido un error; un valor de 0 indica que no ha ocurrido ningún error.

Tabla 5.3 Byte de codificación del modo video, situado en la dirección 449 hex.

BIT 7 6 5 4 3 2 1 0	SIGNIFICADO
X	Diskette fuera de tiempo: falló la respuesta en tiempo
. X	Fallo en la busqueda de la pista
. . X	Fallo del chip controlador del diskette
. . . X	Comprobación de la redundancia cíclica (CRC) : error en los datos
. . . . X . . .	Error en el DMA del diskette
. . . . . X . .	Sector no encontrado: diskette dañado o no formateado
. . . . . . X .	No encontrada la marca de dirección en el diskette
. . . . X	El comando del diskette solicitado es inválido

442(siete bytes). Estos siete bytes almacenan la información del estado del controlador del diskette.

En la dirección 449 hex comienza un área de 30 bytes, utilizando para el control del video. El BIOS utiliza esta área para guardar la información crítica del video. Los programas pueden inspeccionar cualquiera de estos datos, sin problemas, pero en la mayoría de los casos es arriesgado modificarla. El cambio de alguno de estos datos puede causar interferencias, produciendo irregularidades en la operación del ordenador. Los únicos bytes que parecen ser seguros y útiles de cambiar son los campos de localización del cursos .

449(un byte). Un valor de 0 a 10, o de 13 a 15 en este byte especifica el modo de video activo(vease la tabla 5-4). Este es el mismo código del modo de video utilizado en los servicios de video del BIOS.

Los programas en BASIC pueden leer este byte para detectar el modo de video con estas instrucciones.

```
DEF SEG=0                    , ajusta a cero el registro DS
MODO.VIDEO= PEEK(&H449)      , mira la posición hexadecimal 449
```

Tabla 5.4 Byte de codificación del modo de video en la direcc. 449 hex

CODIGO	SIGNIFICADO
0	Texto a 40 columnas, sin color
1	Texto a 40 columnas, 16 colores
2	Texto a 80 columnas sin color
3	Texto a 80 columnas, 16 colores
4	Gráficos de resolución media, 4 colores
5	Gráficos de resolución media, sin color (4 grados de gris)
6	gráfico de alta resolución, 2 colores
7	Modo de adaptador monocromo
8	Gráficos de baja resolución, 16 colores
9	Gráficos de resolución media, 16 colores
10	gráficos de alta resolución, 4 colores
13	Gráficos de resolución media, 16 colores
14	Gráficos de alta resolución, 16 colores

44A(una palabra de dos bytes). Esta palabra almacena la anchura de la pantalla en columnas de texto. La anchura de las columnas se almacena en el equivalente hexadecimal de 20,40 u 80 columnas (el modo de video 8, gráficos de baja resolución, tiene una anchura de texto de 20).
44C(una palabra de dos bytes). Longitud de regeneración de la pantalla. Este es el número de bytes utilizado por una página de pantalla. Varía con el modo.
44E(una palabra de dos bytes). Dirección relativa del comienzo de la memoria de pantalla. De hecho, esta dirección indica la página que se está usando, dando su dirección relativa.
450(ocho palabras de 2 bytes). Estas palabras dan la posición del cursor para las ocho pantallas, comenzando con la página 0. El primer byte de cada palabra da la columna (0 a 19, 39 ó 79) y el segundo byte da la fila (0 a 24). La posición del cursor se puede controlar modificando esta información. Para los lenguajes de programación que no implementan el control del cursor, ésta puede ser una manera de controlar el cursor sin crear un interfaz en lenguaje ensamblador con las rutinas del BIOS.
Cuando cambie el dato de este byte, observe que el cambio no tiene efecto inmediatamente, sino que espera a la siguiente salida de pantalla. Para comprobar esto, entre en el debug, e introduzca este comando:
F 0:450 L 2 8 8
El cursor salta a la fila 8, columna 8 después de que pulse Intro. Hay que decir, necesariamente, que no es una buena técnica de programación, pero debe conocer cómo funciona.
460(una palabra de dos bytes). Estos dos bytes almacenan el tamaño del cursor, basado en el número de líneas que tiene el cursor. El primer byte de la línea de exploración final; el segundo byte, la línea de exploración de comienzo. A diferencia

de los campos de posición del cursor, el cambio de estos valores no altera el cursor automáticamente.

462(un byte). Este byte almacena el número de página que se está visualizando en ese momento.

463(una palabra de dos bytes). Esta palabra contiene la dirección del puerto del chip controlador del video 6845. Normalmente contiene la dirección 3D4 hex.

465 (un byte). Este byte contiene el modo de CRT.

466 (un byte). Este byte contiene la máscara de bits de la paleta de colores .

467(cinco bytes). Estos bytes son utilizados por el control del cassette.

46C(cuatro bytes almacenados como dos palabras de dos bytes, pero tratados mediante un número de cuatro bytes). Esta área se utiliza como contador de un reloj maestro, que se incrementa con cada señal de reloj. Empieza a contar a las 0 horas. Cuando el contador alcanza el equivalente a 24 horas, se vuelve a poner a cero, y el byte de la dirección 470 hex se pone uno a uno. El DOS o el BASIC, calcula la hora a partir de este valor y selecciona el tiempo, poniendo en este campo la cantidad apropiada. La rutina 26 (1A hex) del BIOS proporciona este valor.

470(un byte). Este byte indica que ha ocurrido un paso por cero de reloj. Cuando el contador del reloj pasa a medianoche (y se pone a 0), este byte se pone a 1, lo que significa que la fecha debe de ser incrementada. EL valor es puesto a 1 por la rutina de señal de reloj para indicar que ha pasado la medianoche. Se vuelve a poner a cero, cuando se lee el reloj, y se utiliza la interrupción 26 (1A hex). Esta puesta a cero automática se basa en la suposición de que cualquier programa que lea el reloj incrementará la fecha cuando lea esta señal.

NOTA

Este byte se pone a uno a medianoche y no es incrementado nunca. No hay indicación acerca de si pasan dos medianoches antes de que se lea el reloj.

471 (un byte). Este byte se utiliza para indicar una acción de interrupción desde teclado dentro del BIOS. Si el bit es 1, se ha presionado la tecla break.

472 (una palabra de dos bytes). Estos bytes se utilizan para señalar que está en curso una reinicialización producida desde el teclado. Cuando el sistema se reinicializa desde el teclado con la combinación de teclas Ctrl-Alt-Del, esta palabra contiene el valor 1234 hex mientras dura el proceso de la inicialización. Este hecho es una verdadera curiosidad.

473(un byte). Reservado para IBM. Este byte se pone a 24 hex cuando IBM declara reparto de dividendos.

474(cuatro bytes). Esta área se utiliza sólo con el PCjr para el control especial del diskette.

478(8 bytes en dos campos de 4 bytes). Estos bytes se

utilizan sólo en el PCjr para controlar las señales de temporización (para la impresora paralelo y la puerta serie (o impresora serie)).

485(un byte). Este bit almacena el carácter que se debe repetir si pulsa la tecla repeat. Unicamente en el PCjr.

486(un byte). Este byte se utiliza para temporizar la espera inicial antes de que empiece la acción de la tecla repeat. Unicamente en el PCjr.

487(un byte). Este byte se utiliza para almacenar el código de la función Fn. Unicamente en el PCjr.

488(un byte). Este byte es el tercer byte de estado del teclado que sólo se aplica al PCjr. (Los otros dos bytes de estado del teclado situados en las localizaciones 417 y 418 hex. y se utilizan también en el resto de los modelos, incluyendo el PCjr.).

500 (un byte). Este byte es utilizado por el DOS y el BASIC para controlar la operación de escritura en pantalla. Hay tres valores hexadecimales posibles almacenados en esta localización:

00 Indica el estado OK
01 Indica que se está efectuando una operación de impresión de pantalla
FF Indica que un error ha ocurrido durante una operación de impresión de pantalla.

504(un byte). Este byte es utilizado por el DOS cuando un sistema de diskette único, tal como un XT o un PCjr, imita a un sistema de dos diskettes. El valor indica si la unidad real está actuando como unidad A o unidad B. Los valores usados son:

00 Actuando como unidad A
01 Actuando como unidad B

510 (una palabra de dos bytes). Esta área es utilizada por el BASIC para almacenar el valor del segmento de datos (DS) por defecto. Este es el puntero de segmento de datos por defecto del BASIC.

El BASIC nos permite usar nuestro propio valor de segmento de datos con la instrucción "DEF SEG = valor". (la dirección relativa del segmento se especifica mediante las funciones PEEK o POKE). También se puede devolver el segmento de datos a su posición por defecto usando la instrucción DEG SEG sin = valor. Aunque el defecto almacenado en esta localización, se puede obtener utilizando esta pequeña rutina:

```
DEF SEG = 0
SEGMENTO.DATOS = PEEK (&H511) + 256 + PEEK(&H510)
```

NOTA: El BASIC administra sus propios datos internos basándose en el segmento de datos por el defecto. Intentar cambiarlo es como sabotear la operación del BASIC.

512(cuatro bytes). Esta área la utiliza el BASIC como un vector de interrupción, que apunta a la rutina de servicio de la interrupción del tic de reloj del BASIC.

MOIA : Para funcionar mejor, el BASIC hace correr al reloj del sistema cuatro veces más rápido que su velocidad estándar: para ello, el BASIC debe reemplazar la rutina de interrupción del BIOS estándar es llamada por el BASIC con la cadencia normal: esto es, una vez por cada cuatro señales rápidas.

516 (cuatro bytes). Esta área es utilizada por el BASIC como un vector de interrupción que apunta a la rutina de manejo de la tecla break del BASIC.

51A(cuatro bytes).Esta área es utilizada por el BASIC como un vector de interrupción que apunta a la rutina de manejo de los errores de diskette del BASIC.

5.6.2 Area de comunicaciones entre aplicaciones

Aunque el control de la información del BIOS abarca la parte más grande y más importante del área del bloque 400, el área de comunicaciones entre aplicaciones, o ICA, también está localizada en esta zona. El ICA es un área reservada de 16 bytes, comprendida entre la posición 4F0 y la 4FF, y se utiliza para almacenar datos que pueden ser compartidos por diferentes programas. Es particularmente útil para los programas del DOS que se ejecuten por separdo, pero que tienen que dejar información para otros programas posteriores.
El ICA no se utiliza demasiado. Entre los pocos programas, que se conoce, que lo utilizan están algunas versiones de comunicaciones asíncronas de IBM, Lifetree, Volkswriter, y TimeMark. Debido a que cualquier programa puede almacenar datos en el ICA, en un momento determinado puede haber información de varios programas.

AVISO: El ICA está localizado en los 16 bytes de la localizacion 4F0 a la 4FF. Un error tipográfico en algunas ediciones del manual de "Referencias técnicas del IBM" los sitúan entre la 500 a la 5FF. Esto es incorrecto.

5.6.3 Los identificadores de la versión de la ROM y de la máquina

Al estar los programas del BIOS en memoria fija, no pueden cambiarse fácilmente cuando se necesita hacer ampliaciones o correcciones. Esto significa que los programas en ROM deben haber sido comprobados muy cuidadosamente antes de congelarlos en los chips de memoria. Aunque hay una alta probabilidad de que existan serios errores en los programas contenidos en la ROM de un sistema, IBM ha seguido una trayectoria estupenda: hasta ahora, en los programas contenidos en ROM de la familia PC sólo se han

encontrados unos pocos, y relativamente poco importantes. errores.
Las diferentes versiones del software en ROM podrían presentar un pequeño desafío a los programadores que describen las diferencias que afectan a las características operativas de sus programas. Pero un desafío mayor para los programadores es que algunos miembros de la familia (el PCjr y el AT, en particular) utilizan un conjunto ligeramente diferente de rutinas para la ROM-BIOS de las que vienen con el IBM PC estándar.
Para asegurar que los programas están trabajando con los programas de la ROM apropiados y con el ordenador correcto, IBM ha proporcionado dos marcas de identificación, que están permanentemente disponibles al final de la memoria en la ROM del sistema. Una marca identifica la fecha de salida de la ROM, y puede usarse para identificar la versión BIOS, y la otra da el modelo de la máquina. Estas marcas están siempre presentes en las propias máquinas de IBM, y también las de los fabricantes de unos pocos miembros de la familia PC.
La fecha de salida de la ROM se puede encontrar en un área de 8 bytes, comprendida entre F0000:FFF5 y F0000:FFFC (dos bytes antes del byte de ID de la máquina). Está formada por caracteres ASCII, en el formato americano de la fecha: por ejemplo, 06/01/83 representa el 1 de junio de 1983. Esta marca de salida es una característica común de los ordenadores personales de IBM, pero está presente solamente en unos pocos compatibles del IBM. Por ejemplo los ordenadores Compaq no lo tienen.
El único propósito de las fechas en la marca de salida es para identificar las diferentes versiones de la ROM.
Puede mirar la fecha de salida con el DEBUG, utilizando los siguientes comandos:

```
DEBUG
D F000:FFF5 L 8
```

O DESDE BASIC UTILIZANDO ESTA TECNICA:

```
10 DEF SEG = &HF000
20  FOR I = 0 TO 7
30    PRINT CHR$(PEEK(&HFFF5 + I));
40 NEXT
50 END
```

Aquí tiene un ejemplo de lo que puede encontrar: Yo tengo tres PC y cada uno viene con diferente ROM. Uno tiene la versión 04/24/81, otro la versión 10/19/81 y el último la versión 10/27/82.
La versión de adaptación del BIOS está disponible bajo ciertas circunstancias; por ejemplo, la unidad de mejora de PC que transforma un PC a las especificaciones de XT viene con la adaptación 10/27/82. Ocasionalmente, la adaptación del BIOS está también disponible por separado.
El ID de la máquina es un byte localizado en F000:FFFE. La tabla 5-6 lista los valores de ID publicados para los cinco modelos de

IBM PC. Podemos esperar que, probablemente, esta secuencia continue en los futuros modelos.
Tenga cuidado, ya que hay algunas inconsistencias en la forma de asignar los ID de las máquinas.FE fué el valor anunciado originalmente como identificador para el XT y, después pE wl PC portátil; todavia algunos XT tienen actualmente la firma del PC: "FF".En general, no se puede contar con que estas asignaciones de firma sean sólidas; IBM ha trastocado la definiciónu un poco, o bien en la publicación de las firmas, o bien debido a que hayan sido cambiadas realmente. En aquellas situaciones en las que las diferencias entre los modelos sean lo suficiente significativas como para requerir que un programa necesite identificar la máquina sin equivocarse, se puede considerar que las firmas son sólidas: por ejemplo es el caso entre el PC Jr y el AT, en los cuales cada uno tiene sus propias caractersiticas especiales. Pero cuando las variaciones entre los madelos de máquinas son menores, como, por ejemplo, entre el PC original, el PC estándar, el PC-2 (que acepta 256k de memoria en su placa del sistema), EL XT y el PC portatil, entonces la firma pueden variar. Para todos los propósitos prácticos se puede considerar que ambas asignaturas, la FF y FE, son la identificación de la máquina: mas o menos PC standar

Tabla 5.6 Los ID de maquinas de los cinco modelos IBM PC

ID DEC	HEX	MAQUINA
255	FF	PC (el ordenador personal de IBM original)
254	FE	XT y PC portátil
253	FD	PCJr
252	FC	AT

5.64 La ROM BASIC

Ahora se examinará el tercer elemento de la ROM: la ROM BASIC.
La ROM BASIC actua de dos formas. Primero, proporciona el núcleo del lenguaje BASIC, incluye la mayor parte de los comando y el cimiento fundamental, como, por ejemplo, la gestión de la memoria.
Las versiones del disco del BASIC que se ven en los archivos de proarama BASIC.COM y BASICA.COM, son esencialmente suplementos de la ROM BASIC , y confían en que la ROM BASIC hagan la mayor parte del trabajo. El segundo papel de la ROM BASIC es proporcionar lo que la IBM llama cassete BASIC que es el BASIC que se activa cuando se pone en marcha el ordenador sin disco.
Cuando se usa algunos de los interpretes BASIC, tal como el BASIC de cassette, el cartucho de BASIC del PC Jr o de los BASIC de

discos (BASIC o BASICA), tambien se utiliza los programas de la ROM BASIC aunque no hay nada que permita decir que es así por otra parte, los programas BASIC compilados no hacen uso de la ROM BASIC.
Esta ROM BASIC es caracteristica solo de la familia PC de IBM. Ninguno de los miembros de la familia PC extendida, tal como los ordenadores compac tiene la ROM BASIC: por ello, las partes del equivalentes del BASIC estan incluidas en los programas BASIC basados en disco.

5.6.5 Las extensiones de la ROM

El cuarto elemento de la ROM tiene mas que ver con el diseño de la PC, que con el contenido de su memoria. El PC fue diseñado para permitir dos tipos de extensiones del software interno en la ROM: uno, para extensiones permanentes al software de la ROM-BIOS , y el otro, para extensiones previstas para los cartuchos de software removibles. Para cada uno de ellos estan reservadas areas especiales de memorias. Las extensiones permanentes de la ROM-BIOS son programas que operan con la ROM-BIOS incorporada, pero añade caracteristicas no soportada por la ROM-BIOS básica. Usualmente, estos son programas de soporte para nuevos dispositivos periféricos. El mejor ejemplo de este tipo de extensiones de la ROM es el soporte ROM-BIOS para el disco duro de IBM, que fue introducido con el XT. Otras se encuentra en la Enhanced Graphic Adapter (EGA). Puesto que la ROM-BIOS original no podia anticipar la aparición de programas de soporte para el hardware futuro, las extensiones a la ROM eran, obviamente, una adición necesaria y última. Las extensiones permanentes de la ROM-BIOS utiliza dos areas de memoria. Una es la parte no utilizada del bloque de memoria F,que, desafortunadamente, puede variar de modelo a modelo. En la mayor parte de los modelos, el area de 24k comprendida desde el segmento F000 hasta la F600 esta disponible. La otra area de memoria para las extensiones de la ROM es el bloque C de memoria, cuyo párrafo de segmento va desde la C000 a la CFFE. La ROM-BIOS del disco duro del IBM XT se conecta en esta area, el parrafo de segmento C800, y el Enhanced Graphic Adapter de IBM se conecta en el párrafo C000. Aunque las extenciones de la ROM permanentes proporcionadas por la IBM tiene localizaciones predecibles, hay siempre alguna posibilidad de conflicto entre extensiones de BIOS proporcionadas por otros fabricantes. Normalmente las extensiones de la ROM permanentes estan instaladas en el ordenador, conectadas con parte de una tarjeta de expansion, o conectadas en un zocalo de la ROM, que hay disponible en la tarjeta de sistema del ordenador.

5.7 Estructura física del disco

Las unidades de disco y el sistema operativo del ordenador determinan la **capacidad** de los discos utilizados, pero la **estructura** del disco es escencialmente la misma,

independientemente de la organización. Los datos estan siempre registrados en la superficie del disco en una serie de círculos concéntricos, llamados **pistas.** Cada pista se divide posteriormente en segmentos, llamados **sectores.** La cantidad de datos que puede ser almacenada en cada cara de un disco depende del número de pistas (su densidad) y el tamaño de sus sectores. La densidad del disco puede variar considerablemente según la unidad utilizada.

Para los diskettes de 5 1/4 pulgadas estándar del PC la localización de cada pista y el número de caras utilizables están impuestos por las características **hardware** de los discos y de las unidades de discos, y por lo tanto, son fijas e unmodificables.Sin embargo, la localización, tamaño y el número de sectores por pista están bajocontrol del **software.**

5.8 Formatos estándar del DOS

Se verán en primer lugar los cuatro formatos más comunes del PC, los utilizados por IBM como formatos estándar para los diskettes de 5 1/4 pulgadas.Los cuatro formatos estan derivados del número de caras y del número de sectores en cada pista: simple o doble cara, y ocho o nueve sectores (ver tabla 5.7)

TABLA 5.7 Formatos Estándar de DOS

NOTACION	CARAS	SECTORES	PISTAS	TAMANO NOM.
S-8	1	8	40	160K
D-8	2	8	40	320K
S-9	1	9	40	180K
D-9	2	9	40	360K

Aunque hay varios formatos, sólo dos son utilizados habitualmente :el S-8 y el D-9. El S-8 constituye el denominador común más bajo, por lo que ha sido utilizado tradicionalmente en programas comerciales, puesto que la utiliación del S-8 garantiza que un diskette puede ser leído por cualquier versión del DOS. El formato D-9 es el de mayor capacidad que puede utilizar la mayoría de las unidades de 5 1/4 pulgadas, por lo que la mayor parte de la gente lo utiliza para sus diskettes de trabajo.

5.9 Estructura lógica del disco

Sea cual sea el disco que se utilice, los discos del DOS están todos formateados lógicamente de la misma forma: **las caras, las pistas y los sectores** están identificados numéricamente utilizando la misma notación, y ciertos sectores están siempre reservados para programas e índices especiales que utilizán el DOS para gestionar las operaciones del disco. El BIOS localiza los sectores en un disco por un sistema de coordenadas de tres dimensiones, compuesto por un número de pista (también conocido como cilindro), un número de cara (también llamado cabeza) y un número de sector. El Dos por otra parte, localiza la información por el número de sector y numera los sectores secuencialmente de fuera a dentro. La secuencia empieza con el primer sector del disco: sector 1 de la cara 0 y pista 0, seguido por los sectores que quedan en la misma cara y pista. Para los diskettes de doble cara, el noveno sector de la cara 0 y pista 0, está seguido por el primer sector de la cara 1 y pista 0. El orden prosigue a través de todos los sectores de una cara y pista, después a través de la siguiente cara, en la misma pista (así todas las caras de una pista se encuentran antes de la próxima localización de pista).

5.10 Distribución del espacio del diskette

El proceso de formateado divide los sectores de un disco en cuatro secciones para cuatro usos diferentes. Las secciones, en el orden en que están almcenadas , son **el registro de puesta en marcha, la tabla de localización de ficheros (FAT), el directorio y el espacio de datos.**

El registro de puesta en marcha (boot) es siempre un sector único situado en el sector 0 de la pista 0, cara 0. El registro de puesta en marcha contiene, entre otras cosas, un corto programa para comenzar el proceso de puesta en marcha, aunque no tengan el sistema operativo. A parte del programa de puesta en marcha , el contenido exacto del registro varía de un formato a otro.

Tabla de localización de ficheros, o FAT, va a continuación del registro de puesta en marcha, comenzando normalmente en el sector 1 de la pista 0, cara 0. La FAT contiene el **registro oficial del formato del disco** y **los mapas de la localización de los sectores utilizados por los archivos.** El DOS utiliza la FAT para guardar un registro de la utilización del espaciode datos. Cada entrada de la tabla contiene un código específico para indicar el espacio que está inutilizado (debido a defectos del disco). Al utilizarse la FAT para controlar todo el área utilizable de almacenamiento de datos, se conservan dos copias idénticas de ella, en previsión de que alguna se dañe. Ambas copiasm de la FAT pueden ocupar tantos sectores como se necesiten: 2 ó 4 en discos flexibles.

El directorio de archivos, es el siguiente elemento del disco. Se utiliza como una tabla de contenidos, identificando cada archivo del disco con un elemento de directorio que contiene cierta cantidad de información, como el nombre y tamaño de los archivos. Una parte de la entrada es un número que apunta al primer grupo de sectores utilizados por el archivo (este número es también la primera entrada de este archivo en la FAT). El tamaño del directorio varía según el formato del disco. Ocupa **4** sectores en los diskettes de simple cara y **7** en los de doble cara.

El espacio de datos, que ocupa la mayor parte del diskette (desde el directorio hasta el último sector), se utiliza para almacenar datos realmente, mientras que las otras tres secciones se utilizan para organizar el espacio de datos. Los sectores del espacio de datos están organizados en unidades conocidas como **clusters**. El tamaño de un clusters varía según el formato. En diskettes de simple cara, los clusters ocupan un sector y en los de doble cara ocupan un par de sectores adyacentes.

5.11 La estructura lógica en detalle

Conviene ahora estudiar un poco más profundamente cada una de las cuatro secciones del disco: el registro de puesta en marcha, el directorio , el espacio de datos y la tabla de localización de archivos.

5.11.1 El registro de arranque (boot)

El registro de arranque consiste, primariamente, en un corto programa, en lenguaje de máquina, que activa el proceso de carga del DOS en la memoria. Para realizar esta tarea, el programa comprueba primero si el disco está formateado por el sistema (si contiene los archivos IBMBIO.COM y IBMDOS.COM) y entonces procede en consecuencia.

En todos los formatos de disco, excepto el S-8 y el D-8, se encontrarán en el registro de arranque algunos parámetros clave, que comienzan con el cuarto byte (ver tabla 2). Estos parámetros son parte del bloque de parámetros el BIOS utilizados por el DOS para controlar cualquier dispositivo tipo disco. El resto del programa de arranque empieza en los primeros tres bytes (bytes 0, 1, 2) y continúa en los bytes siguientes del bloque de parámetros del BIOS.

Al final de los registros de arranque de las versiones DOS 2.00 y siguientes hay una signatura de 2 bytes, 55 aa hex.

Tabla 5.8. Parámetros del registro de arranque

OFFSET	LONGITUD	DESCRIPCION
3	8 bytes	ID del sistema (p. ej. IBM 3.1)
11	1 palabra	Número de bytes por sector (p.ej. 512. 0200h)
13	1 byte	Número de sectores por cluster (p.ej. 01 ó 02)
14	1 palabra	Número de sectores reservados al principio
16	1 byte	Número de copias de la FAT: 2 para diskettes
17	1 palabra	Número de elementos del directorio raiz (ej. 64 o 112)
19	1 palabra	Número total de sectores en el disco (ej.720 para el D-9)
21	1 byte	De formato (ej. FF,FE,FD o FC)
22	1 palabra	Número de sectores por FAT (ej. 1 ó 2)
24	1 palabra	Número de sectores por pista (ej. 8 ó 9)
26	1 palabra	Número de caras (cabezas) (1 ó 2)
28	1 palabra	Número de sectores especiales reservados

5.11.2 El directorio

Los directorios de los discos se utilizan para almacenar la mayor parte de la información básica sobre los archivos contenidos en el disco, incluyendo el nombre de los archivos su tamaño, el comienzo de elemento de la FAT,la hora y fecha en que fueron creados y unas pocas caracterosticas especiales del disco (ver figura 5.9) la única información que no contiene el directorio es la localización exacta de los cluster individuales que componen el archivo; estos están alamcenados en la FAT

tabla 5.9. Las ocho partes de un elemento de directorio

CAMPO	OFFSET	DESCRIPCION	TAMAÑO	FORMATO
1	0	Nombre del archivo	8	caracter ASCII
2	8	Extension del arch.	3	caracter ASCII
3	11	Atributo	1	Bit codificador
4	12	Reservado	10	No utilizado, cero
5	22	Hora	2	Palabra,codificada
6	24	Fecha	2	Palabra,codificada
7	26	Comienzo de entrada de la FAT.	2	palabra
8	28	Tamaño del archivo	4	Entero

5.11.3 Campo 1: El nombre del archivo

Los primeros 8 bits de cada elemento de directorio contienen el nombre del archivo, almacenado en formato ASCII. Si el nombre de archivo tiene menos de los 8 caracteres se completa por la derecha con espacios en blanco (CHR$(32)). Las letras deberán ser mayusculas puesto que las minusculas no son reconocidas correctamente.
Tambien pueden aparecer en el primer bits del campo del nombre de archivo 3 códigos que se utilizán para indicar situaciones especiales. El primer bits de los directorios que no se utilizán toma el valor de 00 hex . Esto hace posible que el DOS sepa cuando no hay ningun directorio activo mas, sin tenr que buscar el final del directorio.
El hecho de que el primer byte del campo dell nombre del archivo sea E5 hex , indica normalmente que el archivo ha sido borrado. **Cuando se borra un archivo, solo quedan afectados dos caosas del disco. Al primer bit del nombre del archivo se le asigna el valor E5 hex y la cadena de utilización de espacio de archivo se borra en la FAT;** esto se vera con mas detalle en la sección dedicada a la FAT.
El resto de la información del directorio acerca del archivo se mantiene, como, por ejemplo, el resto de su nombre su tamaño e incluso su numero de cluster de comienzo. a información perdida puede ser recuperada mediante métodos adecuados, siempre que el elemento de directorio no se haya utilizado por otro archivo. Se advierte que cuando es necesario crear un nuevo elemento de directorio, el DOS utiliza el primero disponible, reciclando rápidamente la entrada correspondiente a un archivo ya borrado, haciendo imposible la recuperación.

5.11.4 Campo 2: extención del nombre del archivo

Inmediatamente a continuación del nombre del archivo se encuentra la extención estandar del nombre del archivo, en formato ASCII. Son 3 bits y, como el nombre de archivo, se completa con espacio en blanco, si consta de menos de 3 caracteres, hasta alcanzar en ese numero. Mientras un nombre de archivo debe tener al menos un caracter ordinario, la extención puede consistir en tres espacios en blanco.

5.11.5 Campo 3: atributos del archivo

El tercer campo de la entrada directorio consta de 1 bits, y cada uno de sus bits se utiliza para caracterizar el elemento de directorio. Los bits del bits de atributo se codifican por separado como bits del 0 al 7.

5.11.6 Campo 4: reservado

Esta área de 10 bits esta reservada para posible usos futuros. Los diez bits contienen normalmente el valor 00 hex.

5.11.7 Campo 5: la hora

El campo 5 contiene un valor de dos bits, que señala la hora en que fue creado, o sufrio su ultimo cambio en el archivo.

5.11.8 Campo 6: la fecha

El campo 6 contiene un valor de dos bits que indica la fecha en que se creo el archivo, o en que se modofoco por última vez.

5.11.9 Campo 7: número de cluster de comienzo

El séptimo campo consta de dos bits que indicaa el número de cluster de comienzo del espacio de datos del archivo. **Actua como el punto de entrada en la cadena de localización del archivo en la FAT.** Para archivos sin espacio localizado y etiquetas de volumen, el número de cluster de comienzo es 0, en lugar del valor FFF hex que se utiliza en la FAT para indicar el final del archivo.

5.11.10 Campo 8: tamaño del archivo

El último campo del elemento de directorio indica el tamaño del archivo en bits. Esta codificado como un entero sin signo de 4 bits, lo que permite disponer de archivos de buen tamaño: de hecho, tanto como permita la capacidad del disco utilizado. Es importante resaltar que cuando el DOS esta leyendo un archivo, establece el fianl de archivo cuando lee todo el archivo, segun su tamaño, o cuando llega al final de la cadena de localización del la FAT (denotado por FFF hex), sea cual sea el que aparezca primero.

5.12 El espacio de datos

Todos los archivos de datos y subdirectorios (que actuan como archivos de datos) estan alamcenados en el espacio que ocupa la parte última y mas grande del disco. En la mayoria de las ocasiones, cada archivo se almacena en un bloque de espacio contiguo. Sin embargo, un archivo puede dividirse en varios bloques no continuos, especialmente cuando se añade información a un archivo existente, o cuando se alamcena un nuevo archivo en el espacio dejado por un archivo borrado. No s inusual que un

archivo de datos este disperso por todo el disco .
Esta especie de fragmentación del archivo dificulta de alguna forma el acceso a los datos del archivo. Además es mucho más difícil de recuperar un archivo que se ha borrado involuntariamente si esta fragmentado, simplemente porque hay que realizar muchas operaciones de busqueda en cada uno de los sectores individuales que constituye el archivo.
Haya o no haya archivos fragmentados, será útil comprender como utilizida el DOS la tabal de localización de archivos (FAT) para distribuir el espacio del disco y como la FAT forma una cadena de localización para conectar los cluster que constuyen un archivo.

5.13 La tabla de localización de archivos (FAT)

La tabla de localización o asignación de archivos (File allocation table) almacena registros, que muestra cómo se está asignando el espacio del disco. Hay que distinguir entre cómo está organizado la FAT, que es relativamente simple e inmediato, y cómo está almacenada en el disco, lo cual es más complejo.
Como ya se ha señalado, los formatos de disco estándar alamcenan dos copias de la FAT, aunque pueden ser más de dos, o incluso sólo una copia. Cada copia de la FAT ocupa un sector en los diskettes de 8 sectores y dos sectores en los de 9 sectores. En los formatos de disketbes de alta capacidad designados por QD-15, la FAT utiliza 7 sectores.
En la mayor parte de los formatos de disco, el DOS realiza dos copias de la FAT, pr si una de ellas se estropea o queda ilegible. El programa CHKDSK, que detecta la mayor parte de los errores que pueden producirse en la FAT y el directorio, no detecta si las dos FAT son diferentes.
Hay dos formatos de FAT: uno de 12 bits y otro de 16 bits. El formato de FAT de 12 bits es más común, y el mas complicado de los dos. La FAT de 16 bits se utiliza sólo con discos que exceden la capacidad de 13 bits, como ocurre con el disco duro de 20 megabytes del AT. Se verá primeramente la FAT de 12 bits estudiar, y después se explicará en qué se diferencia del FAT de 16 bits.
La FAT esta organizada como una tabla de hasta 4096 números, que van desde el 0 al 4095 (0 al FFF hex), con un elemento para cada cluster en el espacio de datos. El número de cada elemento indica el estado y uso del cluster correspondiente.Adviértase que el rango de los números guardados en la FAT están definidos de forma que no excedan de tres dígitos hexadecimales. Esto es clave para comprender cómo está almacenada la FAT de 12 bits, como se verá brevemente.
Si el elemento de la FAT es 0, indica que el cluster está libre y disponible para su uso. Si el elemento de la FAT es 4087 (FF7 hex), el cluster está declarado como inutilizable por un error de formateo: esto se llama también marca de pista mala (bad-track). Los valores de la FAT del 4081 al 4086 (FF1 al FF6 hex) se reservan también para señalar la imposibilidad de utilizar un

determinado cluster, pero no se utilizan.
Los clusters están numerados por orden del 2 a un número que es superior en una unidad al número de cluster del disco. Una entrada de la FAT de 12 bits que contenga cualquier número entre 2 y 4080 (02 y FF0 hex) indica que el clusters correspondiente está siendo utilizado por un archivo. Un valor de la FAT de 4095 (FFF hex) indica que el correspondiente cluster contiene la última parte de los datos del archivo. Los valores entre 4088 y 4094 (FF8 al FFE hex) tendrían el mismo significado, pero no se utilizan.
Con todo esto en mente, se puede ver que las entradas de la FAT forman **una cadena de localización**: el elemento del directorio del archivo contiene el número de cluster de comienzo, y las entradas de la FAT designan los demás clusters utilizados y el final del archivo (ver Tabla 5.10). Cuando un archivo es **borrado**, todos los elementos de la FAT que determinan su cadena de localización de espacio son marcados como disponibles (puestos a 0); pero los datos del archivo real en el espacio de datos no sufren modificación alguna, la mayor parte de la información del elemento del directorio del archivo se conserva.

Tabla 5.10 Cadena de atribución de espacio de un archivo en la FAT

ELEMENTO DE LA FAT	VALOR DEC	VALOR HEX	SIGNIFICADO
0	253	FD	El disco es de doble cara,doble densidad
1	4094	EFE	Entrada no utilizada, disponible
2	3	003	El siguiente cluster del archivo es el cluster 3
3	5	005	El siguiente cluster del archivo es el cluster 5
4	4087	FF7	El cluster no es utilizable, pista mala
5	6	006	El siguiente cluster del archivo, es el cluster 6
6	4095	FFF	Ultimo cluster del archivo y final de esta cadena de atribución de espacio
7	0	0	Entrada no utilizada

Aunque la FAT está organizada como una simple tabla de valores numéricos, está almacenada de una forma bastante compleja, con objeto de hacer la tabla tan compacta como sea posible. Para conseguirlo, hace uso de algunos trucos del formato de datos del 8088, específicamente del alamcenamiento inverso. La FAT sacrifica la simplicidad en aras de la eficacia.
El rango de número de clusters está definido de forma que los elementos de la FAT sean 4095 (FFF hex), o menos. Esto hace posible almacenar cada elemento de 3 dígitos hexadecimales en 12 bits, o 1 1/2 bytes. Los elemenos de la FAT se agrupan por pares,

ocupando cada uno 3 bytes (0 y 1 ocupan los primeros 3 bytes , 2 y 3 los siguientes 3 bytes, y así sucesivamente). Los tres bytes se decodifican de la siguiente forma: si un par de elementos de la FAT está formado por 123 y 456 hex, los tres bytes que contienen serían, en hexadecimal, 23, 61, y 45. En sentido inverso, si los tres bytes son AB, CD, EF, los dos valores de la FAT son **DAB** y **EFC**. Este sistema parece curioso cuando se utiliza en términos ordinarios, pero es rápido y eficiente cuando trabaja con instrucciones en lenguaje de máquina. Dado cualquier número e cluster, se puede encontrar el valor de la FAT multiplicando el número de cluster por 3, dividiendo por 2 y utilizando el número completo del resultado como un desplazamiento de la FAT. Cogiendo una palabra de esa ubicación, se tendrán los tres dígitos hexadecimales del elemento de la FAT, más un dígito hexadecimal extraño, que se puede hacer desaparecer con una de las rápidas instrucciones del lenguaje de máquina. Si el número de cluster es impar, se desecha el dígito de mayor orden; si es par, el dígito de menor orden. El valor obtenido de esta manera es el número siguiente del cluster del archivo, a menos que sea FFF, que indica el último cluster del archivo.

Este complejo esquema fue diseñado originalmente para los formatos de diskettes de 8 sectores. No resulta tan eficaz cuando se utiliza para otros formatos, como los formatos de 9 sectores, donde la FAT llega a ser ligeramente mayor que un sector. En conjunto es un método muy preciso y eficiente.

Los detalles mencionados hasta ahora son válidos para las FAT de 12 bits, que pueden alojar hasta 4080 clusters. Si un formato de disco tiene un número superior de clusters, es necesario utilizar la FAT de 16 bits.

La FAT de 16 bits trabaja de la misma forma que la de 12 bits, pero es más simple. Las entradas en la FAT de 16 bits son, obviamente, 4 bits mas largas, lo que permite un mayor rango de números de cluster. Puesto que 16 bits son exactamente dos bytes, una palabra, la FAT de 16 bits no necesita la disposición de almacenamiento comprimido utilizado con la FAT de 12 bits. En cambio, la FAT de 16 bits es una simple tabla de valores de 2 bytes, almacenados a continuación de otro.

Los valores especiales para la FAT de 16 bits (para señalar, por ejemplo,las pistas malas) son una extension lógica de las utilizadas en las FAT de 12 bits. Únicamente se ha añadido el valor F en la posición de mayor orden. Por ejemplo, el valor de final de archivo es el FFFF hex (en lugar del FFF), y el valor de cluster malo es FFF7 hex (en lugar de FF7).

CONCLUSIONES

En este capítulo se ha tratado de hacer enfasis en aquellos aspectos mas relevantes de el software en ROM, como por ejemplo los cuatro elementos diferentes en las ROM de la familia PC de IBM: los programas de arranque, que hacen el trabajo de poner en marcha el ordenador; la ROM-BIOS, que es un acrónimo para indicar el sistema de entrada/salida básico y que está formada por una colección de rutinas en lenguaje de máquina, que proporciona los servicios de soporte para las operaciones del ordenador,; la ROM-BASIC, que proporciona el núcleo del lenguaje de programación BASIC, y las extenciones ROM, que son programas que se añaden a la ROM principal cuando se conectan al ordenador ciertos equipos especiales.
De esta manera se ha analizado la función de la ROM de arranque en una computadoea personal, a la vez que se ha mencionado la ROM-BIOS , los vectores de interrupción, referencias de las direcciones bajas de memoria debido a que muchas de las operaciones del PC están controladas por datos que se encuentran en dichas direcciones.
Aunque no se desee usar la información del área de control del BIOS, es importante su estudio, porque revela gran cantidad de detalles sobre el funcionamiento interno de la familia PC.

REFERENCIAS BIBLIOGRAFICAS

Norton Peter: Título de la obra original: The Peter Norton Programmer's Guide to the IBM PC
Traducción: José M. Gutierrez,y Pedro Muro
MICROSOFT- ANAYA MULTIMEDIA; 1987; Colombia.

CONCLUSIONES GENERALES

La realización de este trabajo nos lleva a las siguientes conclusiones:

1. Que es posible y necesario,la realización de software propio a nivel competitivo, en nuestro medio. Es completamente falso que no se pueda ni deba desarrollar nuestro propio software. Si bien es cierto que hasta el momento se ha utilizado software de una manera relativamente barata, esto es debido a que se ha hecho en forma ilegal, en caso de que en nuestro país se regule legalmente la industria del software, como lo esta en otros paises, nos veríamos en dificultades para obtenerlo. Por otra parte, no hay forma de obtener software para aplicaciones especificas, en donde se tiene que recurrir al diseño propio. No todas las necesidades de software se pueden satisfacer con procesadores de textos, bases de datos, u hojas electronicas. Aplicaciones tales como el control de procesos industriales por medio de microprocesadores, se deberan de diseñar de acuerdo a las necesidades propias y casi siempre diferentes de cada situación.

2. Que no se debe de pensar que el diseño de software es una actividad para gente academicamente privilegiada, es cierto, que este tipo de diseño no es tan facil como nos gustaría que fuera, pero más cierto es que no es tan dificil como mucha gente cree. El temor a cometer errores debe de hacerse a un lado, ya que como se detalla en el capitulo 4, el mismo mundialmente famoso sistema operativo DOS, tiene algunas inconsistencias, y no por eso ha dejado de ser de gran ayuda.

3. Que no todo se puede aprender en los libros. Se sabe que todas las disciplinas cientificas no descuidan la investigación, esto tambien es valido para el area del software, algunas tecnicas de proteccion contra copia de discos, estan basadas en caracteristicas no editadas del DOS. Cuando se tiene dudas sobre cual es la funcion de determinados procedimientos, instrucciones, etc., la mejor manera de aprender es probando.

4. Que en caso de descuidar la importancia del diseño de software propio, se estará continuando con la dependencia patologica, en casi todos los sentidos, de la cual adolecen los paises latinoamericanos. Por otra parte, el desarollo de la industria del software, redundara en oportunidades de trabajo, crecimiento economico, y en algun grado, un poco de independencia tecnologica.

5. Que es cierto, que lo que es valido para el software, no es valido para el Hardaware, es decir, que aunque se puede lograr la creacion de programas altamente eficientes, no es posible en paises como el nuestro ni siquiera pensar en competir con paises industrializados, en la creción de hardware. Pero eso no quiere decir que no sea factible y rentable el ensamblaje de

computadoras, además para un mantenimiento eficiente de computadoras, es muy conveniente tener conocimientos de hardware.

6. Que es de esperar que tarde o temprano, en nuestro país se implementara las redes de comunicacion entre computadores personales, algo muy comun en otros paises, por lo tanto, es necesario comenzar a compreder y aplicar la teoria sobre las comunicaciones entre PC. Por otra parte, a nivel comercial, especialmente en los bancos, la comunicación de computadoras es una actividad diaria e indispensable.

7 Que una de las principales causas del poco avance en el desarrollo de tecnologia propia, es el independencia de los esfuerzos realizados, combinada, con el desconocimiento de información pertinente al problema que se desea resolver. Esto en un futuro tendra que enfrentarse por medio de computadoras en comunicación, esto es otra razón para comenzar a desarrollar el software de comunicaciones entre PC.

RECOMENDACIONES GENERALES

1. Diseñar planes de estudio que satisfagan en alguna medida las necesidades aqui planteadas. Comenzando por crear un sistema efectivo de transmitir la informacion de modo que dos o mas estudiantes no tengan que recorrer el mismo camino innecesariamente, es decir, que si a un estudiante le toma seis meses en investigar algo, a los demas no les debe llevar lo mismo, ya que no es lo mismo buscar la informacion que asimilarla. Esto se puede realizar creando listados de los problemas ya resueltos que indiquen en donde esta la información pertinente y en forma resumida y si se quiere en forma algoritmica.

2. Con los recursos que cuenta la Escuela de ingenieria electrica, se puede continuar este trabajo. Entre los adelantos que vale la pena realizar podemos mencionar:
 a) Modificar el programa del capitulo dos para que sea mas eficiente en la traduccion al español de programas útiles.
 b) Modificar el programa del cápitulo dos para que cuente con un modo mas de operación, ya que actualmente trabaja con sectores, el cual se podria modificar para que trabaje con archivos y al presionar las teclas de avance de sector se muestre el siguiente sector del archivo que como se sabe no siempre son contiguos.

3. Crear una materia que trate exclusivamente sobre las comunicaciones entre PC, y lo que son las redes de comunicación entre PC

4. Construir el circuito mostrado en el capitulo tres, y utilizarlo con fines pedagógicos en el estudio de microprocesadores.

ANEXO 1

MANDATOS DEL DEBUG

MANDATOS DEL DEBUG

Notación del formato

[ ] Los elementos encerrados entre corchetes son opcionales. Para incluir elementos opcionales, escriba solamente la información entre los corchetes. No se deben de escribir los corchetes

MAYUSCULAS Las palabras mostradas en mayusculas se denominan palabras claves. Se deben de escribir las palabras claves; sin embargo al hacerlo se puede utilizar cualquier combinacion de mayusculas o minusculas.

minusculas se deben de sustituir los elementos mostrados en minusculas. Si se encuentran entre corchetos son opcionales. Por ejemplo:

nombre_arch

Indica que se debe de escribir el nombre del archivo en lugar de la palabra nombre_arch.

| Una barra vertical indica opcion. Hay que escoger uno de los elementos separados y escribirlo como parte del mandato. Por ejemplo:

ON|OFF

Significa que se puede escribir ON u OFF pero no ambos a la vez.

Se deben de incluir todos los signos de puntuacion tales como comas, signos de igualdad, interrogacion, dos puntos, barras y barras invertidas. Los signos de puntuacion entre corchetes son opcionales.

COMANDO	FORMATO	PROPOSITO
A	A[Direccion]	Permite escribir en nemonico a partir de dirección
C	C rango dirección	Compara memoria
D	D[Direccion] o D[Rango]	Visualiza memoria en hexa y ASCII
E	E direccion [lista]	Sustituye el contenido de la memoria
F	F rango lista	Cambia bloques de memoria
G	G[=direccion] [direccion[..]]	Ejecuta instrucciones con punto de ruptura opcionales
H	H valor1 valor2	Suma y resta valor1 y valor 2

COMANDO	FORMATO	PROPOSITO
L	L[Direccion lunidad sector # de sectores]]	Carga archivos o sectores desde el disco.
M	M rango direccion	Mueve bloques de memoria
N	[D:] [path] nombre_arch	Define archivo y parametros
P	P[=direccion][valor]	Ejecuta todas las instrucciones que abarca una instruccion CALL
Q	Q	Salir de debug
R	R[nombre reg]	Visualiza registros (reg)/ banderas, permite cambiar registros
S	S rango lista	Busca caracteres
T	T[=direccion][valor]	Ejecuta instrucciones y visualiza el estado final de los registros.
U	U[direccion] o U[rango]	Desemsambla instrucciones
W	W[direccion lunidad sector sector]]	Graba archivos o sectores en discos

ANEXO 2

LISTADO FUENTE DEL PROGRAMA DSK.COM

En este anexo se muestran los listados completos los nueve archivos que componen el programa DSK. Los cuales se describen brevemente a continuacion.

1. DSK.ASM
   Este archivo contiene el procedimiento principal de DSK. Simplemente llama a otros procedemientos para que realicen el trabajo.

2. EDITOR.ASM
   Este archivo contiene los procedimientos para aceptar cambios en la pantalla y en la memoria

3. KBD.ASM
   Este archivo contiene todos los procedimientos que aceptan datos desde el teclado

4. CONTROL.ASM
   Este archivo contiene el procedimiento que trabaja como control central que acepta ordenes desde el teclado y llama al procedimiento correspondiente para ejecutar la orden

5. DESP_SEC.ASM
   Este archivo contiene los procedimientos para desplegar el sector selecionado, las ventanas ASCII y Hexa. y las indicaciones.

6. DISK_IO.ASM
   Este archivo contiene los procedimientos que manejan todo lo relacionado con la lectura y escritura en discos

7. VIDEO_IO.ASM
   Este archivo contiene los procedimientos que se encargan de desplegar caracteres, lineas, patrones de lineas, etc.

8. FANTASMA.ASM
   Este archivo contiene los procedimientos que se encargar de crear y desplazar el cursor fantasma

9. CURSOR.ASM
   Este archivo contiene los procedimientos que se encargan de limpiar la pantalla, mover el cursor y avanzar linea.

TECLAS UTILIZADAS PARA MANIPULAR ESTE PROGRAMA

Cuando se corre este programa, automaticamente se muestra en la pantalla la mitad del sector 0. A partir de este punto, se pueden utilizar las teclas que se muestran a continuación con sus respectivas funciones.

TECLAS:	FUNCION
F1	Muestra el siguiente sector
F2	Muestra el sector previo
F3	Permite seleccionar el sector que se desea ver
Shift-F5	Graba el sector modificado
F10	Salir al DOS

Si se presionan otras teclas se realizarán modificaciones en la posición determinada por el cursor fantasma, de la siguiente manera:

Si se escribe unicamente un caracter y luego se presiona la tecla ENTER, el DSK sabe las modificaciones se harán en la ventana ASCII, automaticamente se actualiza la ventana HEXA

Si se escriben dos caracteres y luego se presiona la tecla ENTER, el DSK sabe que las modificaciones se harán en la ventana Hexa, y automaticamente se actualiza la ventana ASCII

NOTA: Las modificaciones que se realicen en un sector, no serán grabadas hasta que se presionen las teclas Shift y F5 simultaneamente. Si no se graban las modificaciones antes de leer otro sector, estas no seran registradas en el disco

```
C:\TASM\ DSK.ASM

CGROUP GROUP CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP
CODE_SEG    SEGMENT  PUBLIC
        ORG     100h
        EXTRN   LIMPIA_PANTALLA:NEAR, LEE_SECTOR:NEAR
        EXTRN   INI_DESP_SEC:NEAR, ESCRIBE_ENCABEZADO:NEAR
        EXTRN   ESCRIBE_LINEA_INDICADOR:NEAR, MASTER:NEAR
DISK    PROC    NEAR
        CALL    LIMPIA_PANTALLA
        CALL    ESCRIBE_ENCABEZADO
        CALL    LEE_SECTOR
        CALL    INI_DESP_SEC
        LEA     DX,INDICADOR_EDITOR
        CALL    ESCRIBE_LINEA_INDICADOR
        CALL    MASTER
        INT     20h
DISK    ENDP
CODE_SEG        ENDS
DATA_SEG        SEGMENT PUBLIC
        PUBLIC  SECTOR_OFFSET
;-------------------------------------------------------------------;
; SECTOR_OFFSET es el complemento de el medio sector desplegado     ;
; debe de ser un multiplo de 16, y no mayor de 256                  ;
;-------------------------------------------------------------------;
SECTOR_OFFSET   DW      0
        PUBLIC  SECTOR_ACTUAL_NO, DISK_DRIVE_NO
SECTOR_ACTUAL_NO        DW      0       ;Inicializa Sector 0
DISK_DRIVE_NO           DW      0       ;Inicializa Driver A
        PUBLIC  LINEAS_DESPUES_SECTOR, LINEA_ENCABEZADO_NO
        PUBLIC  ENCABEZADO_PARTE_1, ENCABEZADO_PARTE_2
;-------------------------------------------------------------------;
; LINEAS_DESPUES_SECTOR es el numero de lineas en el tope de la     ;
; pantalla antes del despliegue del medio sector                    ;
;-------------------------------------------------------------------;
LINEAS_DESPUES_SECTOR   DB      2
LINEA_ENCABEZADO_NO     DB      0
ENCABEZADO_PARTE_1      DB      'Disco ',0
ENCABEZADO_PARTE_2      DB      '          Sector ',0
        PUBLIC LINEA_INDICADOR_NO, INDICADOR_EDITOR,ENTRAR_NO_SECTOR
LINEA_INDICADOR_NO      DB      21
INDICADOR_EDITOR        DB      'Presione tecla de funcion, o ENTER '
                        DB      'Caracter o byte Hexa: ',0
ENTRAR_NO_SECTOR        DB      'Que sector desea ver: ',0
        PUBLIC SECTOR
;-------------------------------------------------------------------;
;El sector entero (hasta 512 Bytes) esta almacenado en esta area    ;
;de memoria                                                         ;
;-------------------------------------------------------------------;
SECTOR  DB      512 DUP(0)
DATA_SEG        ENDS
        END     DISK
```

```
;_____________________________________________________________________;
; Este procedimiento cambia un byte en la memoria y en la             ;
; pantalla                                                            ;
;                                                                     ;
; AL    Byte a escribir en sector y cambiar en la pantalla            ;
;                                                                     ;
; Utilize: SALVAR_CURSOR_REAL, RECUPERAR_CURSOR_REAL                  ;
;           SALVAR_CURSOR_REAL, RECUPERAR_CURSOR_REAL                 ;
;          MOVER_A_POSICION_HEXA, MOVER_A_POSICION_ASCII              ;
;          POSICION_CURSOR, ESCRIBE_HEX, ESCRIBE_CAR                  ;
;          ESCRIBE_LINEA_INDICADOR, ESCRIBE_MEMORIA                   ;
;                                                                     ;
; Lee:     INDICADOR_EDITOR                                           ;
;_____________________________________________________________________;
EDITAR_BYTE  PROC NEAR
          PUSH     DX
          CALL     SALVAR_CURSOR_REAL
          CALL     MOVER_A_POSICION_HEXA
          CALL     POSICION_CURSOR
          CALL     ESCRIBE_HEX
          CALL     MOVER_A_POSICION_ASCII
          CALL     ESCRIBE_CAR
          CALL     RECUPERA_CURSOR_REAL
          CALL     ESCRIBE_FANTASMA
          CALL     ESCRIBE_A_MEMORIA
          LEA      DX,INDICADOR_EDITOR
          CALL     ESCRIBE_LINEA_INDICADOR
          POP      DX
          RET
EDITAR_BYTE      ENDP

CODE_SEG         ENDS
          END
```

```
;FADO  EDITOR.ASM

CGROUP  GROUP CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT  PUBLIC

DATA_SEG        SEGMENT  PUBLIC
        EXTRN   SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   CURSOR_FANTASMA_X:BYTE
        EXTRN   CURSOR_FANTASMA_Y:BYTE
DATA_SEG        ENDS

;_____________________________________________________________________;
; Este procedimiento escribe un byte a SECTOR en la localizacion     ;
; de memoria apuntada por el cursor fantasma                          ;
;                                                                     ;
; DL    Byte a escribir en sector                                     ;
;                                                                     ;
; El complemento es calculado por:                                    ;
; OFFET = SECTOR_OFFSET + 16*CURSOR_FANTASMA_Y+CURSOR_FANTASMA_X      ;
;                                                                     ;
; Lee:    CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y, SECTOR_OFFSET         ;
; Escribe: SECTOR                                                     ;
;_____________________________________________________________________;
ESCRIBE_A_MEMORIA  PROC  NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,SECTOR_OFFSET
        MOV     AL,CURSOR_FANTASMA_Y
        XOR     AH,AH
        MOV     CL,4                        ;CURSOR_FANTASMA_Y*16
        SHL     AX,CL
        ADD     BX,AX                       ;BX = SECTOR_OFFET +16*Y
        MOV     AL,CURSOR_FANTASMA_X
        XOR     AH,AH
        ADD     BX,AX                       ;Esta es la direccion
        MOV     SECTOR[BX],DL               ;Ahora almacenar el byte
        POP     CX
        POP     BX
        POP     AX
        RET
ESCRIBE_A_MEMORIA ENDP

        PUBLIC  EDITAR_BYTE
        EXTRN   SALVAR_CURSOR_REAL:NEAR, RECUPERA_CURSOR_REAL:NEAR
        EXTRN   MOVER_A_POSICION_HEXA:NEAR, MOVER_A_POSICION_ASCII:NEAR
        EXTRN   ESCRIBE_FANTASMA:NEAR, ESCRIBE_LINEA_INDICADOR:NEAR
        EXTRN   POSICION_CURSOR:NEAR, ESCRIBE_HEX:NEAR
        EXTRN   ESCRIBE_CAR:NEAR

DATA_SEG        SEGMENT PUBLIC
        EXTRN   INDICADOR_EDITOR:BYTE
DATA_SEG        ENDS
```

```
; LADO BIO_IO.ASM

CGROUP GROUP CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

BS      EQU     8
CR      EQU     13
ESC     EQU     27

CODE_SEG        SEGMENT PUBLIC
        PUBLIC  CADENA_A_MAYUSCULA
;-------------------------------------------------------------------;
;Este procedimiento convierte una cadena a letras mayusculas       ;
;                                                                  ;
;  DS:DX  direccion de la cadena en el buffer                      ;
;-------------------------------------------------------------------;
CADENA_A_MAYUSCULA  PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX                  ;Apunta a contador de caracteres
        MOV     CL,[BX]             ;contador de caracteres esta en el
                                    ;segundo byte de buffer
        XOR     CH,CH               ;Limpia byte superior de contador

LAZO_MAYUSCULA:
        INC     BX                  ;Apuntar al siguiente caracter en buffe
        MOV     AL,[BX]
        CMP     AL,'a'              ;Ver si es letra minuscula
        JB      NO_MINUSCULA
        CMP     AL,'z'
        JA      NO_MINUSCULA
        ADD     AL,'A'-'a'          ;Convertir a mayuscula
        MOV     [BX],AL
NO_MINUSCULA:
        LOOP    LAZO_MAYUSCULA
        POP     CX
        POP     BX
        POP     AX
        RET
CADENA_A_MAYUSCULA  ENDP

        PUBLIC  CONVERTIR_DIGITO_HEX
;-------------------------------------------------------------------;
;Este procedimiento convierte un caracter de ASCII (hexa) a un     ;
;nibble (4bits)                                                    ;
;                                                                  ;
;            AL Caracter a convertir                               ;
;  Retorna:  AL Nibble                                             ;
;            CF en 1 si hay error, 0 si no lo hay                  ;
;-------------------------------------------------------------------;
CONVERTIR_DIGITO_HEX    PROC NEAR
        CMP     AL,'0'              ;Es un digito legal?
        JB      MAL_DIGITO          ;No lo es
        CMP     AL,'9'              ;Todavia no hay seguridad
        JA      PRUEBA_HEXA         ;Podria ser hexa
```

```
        SUB     AL,'0'          ;Es decimal, convertirlo a nibbel
        CLC                     ;Carry = 0 no hay error
        RET
PRUEBA_HEXA:
        CMP     AL,'A'          ;Todavia no hay seguridad
        JB      MAL_DIGITO      ;no es hexa
        CMP     AL,'F'          ;aun no se esta seguro
        JA      MAL_DIGITO      ;no es hexa
        SUB     AL,'A'-10       ;es hexa, convertirlo a numero
        CLC
        RET
MAL_DIGITO:
        STC                     ;Carry = 1 hay error
        RET
CONVERTIR_DIGITO_HEX    ENDP

        PUBLIC HEXA_A_BYTE
;-------------------------------------------------------------------------;
;Este procedimiento convierte los dos caracteres de DS:DX                 ;
;de hexa a un byte                                                        ;
;                                                                         ;
; DS:DX  direccion de los dos caracteres para numero hexa                 ;
;                                                                         ;
; Retorna: AL byte                                                        ;
;          CF 1 si hay erro, 0 si no lo hay                               ;
;Utiliza:  CONVERTIR_DIGITO_HEX                                           ;
;-------------------------------------------------------------------------;
HEXA_A_BYTE  PROC NEAR
        PUSH    BX
        PUSH    CX
        MOV     BX,DX           ;Poner direccion en BX para
                                ;direccionamiento indirecto
        MOV     AL,[BX]         ;obtener primer digito
        CALL    CONVERTIR_DIGITO_HEX
        JC      MAL_HEXA        ;carry en uno hay error
        MOV     CX,4            ;desplazar 4 posiciones a la izq.
        SHL     AL,CL
        MOV     AH,AL           ;Retener una copia
        INC     BX              ;Obtener segundo digito
        MOV     AL,[BX]
        CALL    CONVERTIR_DIGITO_HEX
        JC      MAL_HEXA
        OR      AL,AH           ;Combinar dos nibbles
        CLC                     ;Limpiar carry
HECHO_HEXA:
        POP     CX
        POP     BX
        RET
MAL_HEXA:
        STC
        JMP     HECHO_HEXA

HEXA_A_BYTE     ENDP

        PUBLIC  LEE_CADENA
```

```
        EXTRN   ESCRIBE_CAR:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento ejecuta una funcion muy similar a la funcion         ;
;0Ah del DOS, pero esta funcion retorna un caracter especial si          ;
;una tecla de funcion es presionada. ESC borrara la entrada y            ;
;comenzara de nuevo                                                      ;
;                                                                        ;
; DS:DX Direccion para el buffer. El primer byte debe de contener        ;
;       el maximo numero de caracteres a leer (mas uno para              ;
;       ENTER). El segundo byte sera utilizado por este proce-           ;
;       dimiento para retornar el numero de caracteres que se            ;
;       han leido asi:                                                   ;
;               0   No se han leido caracteres                           ;
;              -1   Se ha leido un caracter especial                     ;
;           otro    Se han leido numeros                                 ;
;                                                                        ;
; Utiliza:  BACK_SPACE, ESCRIBE_CAR                                      ;
;-------------------------------------------------------------------------;
LEE_CADENA      PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    SI
        MOV     SI,DX                   ;Utiliza SI como registro indice
                                        ;BX como complemento
COMENZAR:
        MOV     BX,2
        MOV     AH,7                    ;Llamada a entrada sin eco
        INT     21h                     ;y chequeo de contro-break
        OR      AL,AL                   ;Es un caracter ASCII extendido?
        JZ      EXTENDIDO               ;Si, lee en caracter extendido
NO_EXTENDIDO:
        CMP     AL,CR                   ;Es un retorno de carro?
        JE      FIN_ENTRADA             ;Si, la entrada esta echa
        CMP     AL,BS                   ;Es un back_space?
        JNE     NO_BS                   ;No lo es
        CALL    BACK_SPACE              ;Si, elimine el caracter anterior
        CMP     BL,2                    ;Este el buffer vacio?
        JE      COMENZAR                ;Si ahora se puede leer ASCII extendido
                                        ;Nuevo
        JMP     SHORT LEE_SIG_CAR       ;No, contiene lectura normal
NO_BS:
        CMP     AL,ESC                  ;Es un ESC, purgar buffer
        JE      PURGAR_BUFFER           ;Si, eliminar el buffer
        CMP     BL,[SI]                 ;Ver si el buffer esta lleno
        JA      BUFFER_LLENO            ;Buffer es lleno
        MOV     [SI+BX],AL              ;Si no, salvar caracter en el buffer
        INC     BX                      ;Apuntar siguiente byte en buffer
        PUSH    DX
        MOV     DL,AL                   ;Mostrar caracter en la pantalla
        CALL    ESCRIBE_CAR
        POP     DX
LEE_SIG_CAR:
        MOV     AH,7
        INT     21h
        OR      AL,AL                   ;Un caracter ASCII extendido no es valid
```

```
                                        ;cuando el buffer no esta vacio
        JNE     NO_EXTENDIDO            ;caracter es valido
        MOV     AH,7
        INT     21h                     ;Desechar el caracter extendido
;-------------------------------------------------------------------------;
;Señala una condicion de error enviando un beep                           ;
;-------------------------------------------------------------------------;
ERROR:
        PUSH    DX
        MOV     DL,7                    ;Codigo ASCII de BEEP
        MOV     AH,2
        INT     21h
        POP     DX
        JMP     SHORT LEE_SIG_CAR

;-------------------------------------------------------------------------;
; Vacia la cadena del buffer y borra todos los caracteres                 ;
; desplegados en la pantalla                                              ;
;-------------------------------------------------------------------------;
PURGAR_BUFFER:
        PUSH    CX
        MOV     CL,[SI]                 ;Colocarse en el extremo superior
        XOR     CH,CH                   ;del buffer
LAZO_PURGA:
        CALL    BACK_SPACE              ;Back_space movera el cursor de regres
        LOOP    LAZO_PURGA
        POP     CX
        JMP     COMENZAR                ;Ya se puede leer extendido ASCII
                                        ;puesto que el buffer esta vacio
;-------------------------------------------------------------------------;
;El buffer esta lleno, de manera que no se puede leer otro               ;
;caracter. Enviar un beep para alertar al usuario de esta                ;
;codicion                                                                ;
;-------------------------------------------------------------------------;
BUFFER_LLENO:
        JMP     SHORT   ERROR           ;El buffer esta lleno sume beep

;-------------------------------------------------------------------------;
;Lee el codigo ASCII extendido y lo coloca en el buffer como el          ;
;unico caracter, luego retorna un -1 como numero de caracteres           ;
;leidos                                                                  ;
;-------------------------------------------------------------------------;
EXTENDIDO:
        MOV     AH,7                    ;Leer el codigo ASCII extendido
        INT     21h
        MOV     [SI+2],AL               ;Colocar solo este caracter en buffer
        MOV     BL,0FFh                 ;Caracteres leidos = -1
        JMP     SHORT FIN_CADENA

;-------------------------------------------------------------------------;
;Salvar el contador de el numero de caracteres leidos y retornar;
;-------------------------------------------------------------------------;
FIN_ENTRADA:
        SUB     BL,2                    ;eco y ENTER
FIN_CADENA:
```

```
        MOV     [SI+1],BL
        POP     SI
        POP     BX
        POP     AX
        RET
LEE_CADENA      ENDP

PUBLIC  LEE_BYTE
;-------------------------------------------------------------------------;
;Este procedimiento lee ya sea un solo caracter ASCII o un nume-;
;ro hexadecimal de dos digitos, esta es solo una version de pru-;
;ba de LEE_BYTE                                                           ;
;                                                                         ;
; Retorna byte en  AL  codigo de caracter (a no ser AH=0)                 ;
;                  AH  1 Si se lee caracter ASCII                         ;
;                      0 Si no se lee caracter                            ;
;                     -1 Si se lee una tecla especial                     ;
;                                                                         ;
; Utiliza: HEXA_A_BYTE, CADENA_A_MAYUSCULA, LEE_CADENA                    ;
; lee    : ENTRADA_TECLADO, ETC...                                        ;
; Escribe: ENTRADA_TECLADO, ETC...                                        ;
;-------------------------------------------------------------------------;
LEE_BYTE  PROC NEAR
        PUSH    DX
        MOV     LIMITE_NUM_CAR,3        ;Permite solo dos caracteres
                                        ; y ENTER
        LEA     DX,ENTRADA_TECLADO
        CALL    LEE_CADENA
        CMP     NUM_CAR_LEIDOS,1        ;Ver cuantos caracteres hay
        JE      ENTRADA_ASCII           ;Solo uno tratelo como ASCII
        JB      NO_CARACTERES           ;Solo se toco enter
        CMP     BYTE PTR NUM_CAR_LEIDOS,0FFh
        JE      TECLA_ESPECIAL
        CALL    CADENA_A_MAYUSCULA
        LEA     DX,CARACTERES           ;Direccion de la cadena a conver
        CALL    HEXA_A_BYTE             ;Convertir cadena de hexa a byte
        JC      NO_CARACTERES           ;Error no cambiar caracteres
        MOV     AH,1
LECTURA_HECHA:
        POP     DX
        RET
NO_CARACTERES:
        XOR     AH,AH                   ;Indicar no lectura de caracter
        JMP     LECTURA_HECHA
ENTRADA_ASCII:
        MOV     AL,CARACTERES
        MOV     AH,1
        JMP     LECTURA_HECHA
TECLA_ESPECIAL:
        MOV     AL,CARACTERES[0]        ;Retorna codigo scan
        MOV     AH,0FFh
        JMP     LECTURA_HECHA

LEE_BYTE        ENDP
        PUBLIC  BACK_SPACE
```

```
        EXTRN   ESCRIBE_CAR:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento elimina caracteres, uno a la vez, de el               ;
;buffer y de la pantalla cuando el buffer no esta vacio. Si el            ;
;buffer esta vacio este procedimiento simplemente no hace nada            ;
;                                                                         ;
; ENTRADA: DI = BX  Caracter mas reciente todavia en buffer               ;
;                                                                         ;
; Utiliza: ESCRIBE_CAR                                                    ;
;-------------------------------------------------------------------------;
BACK_SPACE      PROC NEAR
        PUSH    AX
        PUSH    DX
        CMP     BX,2            ;Esta el buffer vacio?
        JE      FIN_BS          ;Si, lea el siguiente caracter
        DEC     BX              ;Remover un  caracter del buffer
        MOV     AH,2            ;Remover un caracter de la pantalla
        MOV     DL,BS
        INT     21h
        MOV     DL,20h          ;Escribir un espacio ahi
        CALL    ESCRIBE_CAR
        MOV     DL,BS           ;Regresar de nuevo
        INT     21h
FIN_BS:
        POP     DX
        POP     AX
        RET
BACK_SPACE      ENDP

        PUBLIC  LEE_DECIMAL
;-------------------------------------------------------------------------;
; Este procedimiento toma la salida del buffer de LEE_CADENA y            ;
; lo convierte de digito decimal a palabra                                ;
;                                                                         ;
;   AX  palabra convertida de decimal                                     ;
;   CF  1 Si hay error. 0 si no hay                                       ;
;                                                                         ;
; Utiliza: LEE_CADENA                                                     ;
; Lee    : ENTRADA_TECLADO, etc                                           ;
; Escribe: ENTRADA_TECLADO, etc                                           ;
;-------------------------------------------------------------------------;
LEE_DECIMAL     PROC NEAR
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     LIMITE_NUM_CAR,6
        LEA     DX,ENTRADA_TECLADO
        CALL    LEE_CADENA
        MOV     CL,NUM_CAR_LEIDOS
        XOR     CH,CH
        CMP     CL,0
        JLE     MAL_DIGITO_DEC
        XOR     AX,AX
        XOR     BX,BX
CONVERTIR_DIGITO:
```

```
            MOV     DX,10
            MUL     DX
            JC      MAL_DIGITO_DEC
            MOV     DL,CARACTERES[BX]
            SUB     DL,'0'
            JS      MAL_DIGITO_DEC
            CMP     DL,9
            JA      MAL_DIGITO_DEC
            ADD     AX,DX
            INC     BX
            LOOP    CONVERTIR_DIGITO
DECIMAL_ECHO:
            POP     DX
            POP     CX
            POP     BX
            RET
MAL_DIGITO_DEC:
            STC
            JMP     DECIMAL_ECHO
LEE_DECIMAL ENDP

CODE_SEG    ENDS

DATA_SEG    SEGMENT PUBLIC
            PUBLIC ENTRADA_TECLADO
ENTRADA_TECLADO LABEL   BYTE
LIMITE_NUM_CAR  DB      0
NUM_CAR_LEIDOS  DB      0
CARACTERES      DB      80 DUP (0)
DATA_SEG    ENDS
            END
```

```
TITLE   CONTROL.ASM

CGROUP GROUP CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        PUBLIC  MASTER
        EXTRN   LEE_BYTE:NEAR, EDITAR_BYTE:NEAR
        EXTRN   ESCRIBE_LINEA_INDICADOR:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   INDICADOR_EDITOR:BYTE
DATA_SEG        ENDS

;-------------------------------------------------------------------------;
;Este es el control central. Durante la edicion normal y mien-           ;
;tras se observan los sectores, este procedimiento lee caracte-          ;
;res desde el teclado y si el caracter es una tecla de comando           ;
;tales como las teclas de cursor, MASTER llama los procedimien-          ;
;tos que realizaran el trabajo indicado. Este programa esta he_          ;
;cho para las teclas especiales listadas en la TABLA_MASTER ,            ;
;donde el procedimiento invocado esta almacenado directamente            ;
;despues del nombre de la tecla.                                         ;
;Si el caracter no es una tecla especial, entonces se debera co-         ;
;locar en el buffer de edicion, es decir, en el modo de edicion          ;
;                                                                        ;
; Utiliza: LEE_BYTE, EDITAR_BYTE, ESCRIBE_LINEA_INDICADOR               ;
;     Lee : INDICADOR_EDITOR                                             ;
;-------------------------------------------------------------------------;
MASTER  PROC NEAR
        PUSH    AX
        PUSH    DX
LAZO_CONTROL:
        CALL    LEE_BYTE            ;Lee caracter en AX
        OR      AH,AH               ;AX=0 si no se ha leido caracter
                                    ;-1 para el codigo extendido
        JZ      NO_CARACTER_LEIDO

        JS      TECLA_ESPECIAL      ;Lee codigo extendido
        MOV     DL,AL
        CALL    EDITAR_BYTE
        JMP     LAZO_CONTROL        ;Leer otro caracter
TECLA_ESPECIAL:
        CMP     AL,68               ;F10 salir
        JE      FIN_CONTROL         ;Si, salga

        LEA     BX,TABLA_MASTER
LAZO_ESPECIAL:
        CMP     BYTE PTR [BX],0 ;Fin de tabla
        JE      NO_EN_TABLA         ;Si, tecla no esta en tabla
        CMP     AL,[BX]             ;Esta entrada en tabla?
        JE      CONTROL             ;Si, ir a control
        ADD     BX,3                ;No, pruebe con la siguiente entrada
        JMP     LAZO_ESPECIAL       ;Revisar la siguiente entrada

CONTROL:
        INC     BX                  ;Apuntar a direccion de procedimiento
        CALL    WORD PTR [BX]       ;Llamar procedimiento
```

```
        JMP     LAZO_CONTROL    ;Esperar por otra tecla
NO_EN_TABLA:
        JMP     LAZO_CONTROL
NO_CARACTER_LEIDO:
        LEA     DX,INDICADOR_EDITOR
        CALL    ESCRIBE_LINEA_INDICADOR  ;Borra caracteres invalidos
        JMP     LAZO_CONTROL             ;Retornar de nuevo
FIN_CONTROL:
        POP     DX
        POP     BX
        POP     AX
        RET
MASTER  ENDP
CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
CODE_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR_PROXIMO:NEAR   ;En disk_io.asm
        EXTRN   SECTOR_PREVIO:NEAR    ;En disk_io.asm
        EXTRN   FANTASMA_ARR:NEAR, FANTASMA_ABA:NEAR
        EXTRN   FANTASMA_IZQ:NEAR, FANTASMA_DER:NEAR
        EXTRN   ESCRIBE_SECTOR:NEAR   ;En disk_io
        EXTRN   SECTOR_N:NEAR
CODE_SEG        ENDS
;-------------------------------------------------------------------;
;Esta tabla contiene las teclas permitidas en codigo ASCII y las;
;direcciones de los procedimientos que se llamaran cuando cada   ;
;tecla es presionada.                                            ;
;El formato de la tabla es:                                      ;
;    DB  72              ;Codigo para cursor hacia arriba       ;
;    DW  OFFSET CGROUP:FANTASMA_ARR                             ;
;-------------------------------------------------------------------;
TABLA_MASTER  LABEL BYTE
        DB      59                              ;F1
        DW      OFFSET CGROUP:SECTOR_PREVIO
        DB      60                              ;F2
        DW      OFFSET CGROUP:SECTOR_PROXIMO
        DB      72                              ;Cursor arriba
        DW      OFFSET CGROUP:FANTASMA_ARR
        DB      80                              ;Cursor abajo
        DW      OFFSET CGROUP:FANTASMA_ABA
        DB      75                              ;Cursor izq
        DW      OFFSET CGROUP:FANTASMA_IZQ
        DB      77                              ;Cursor derecha
        DW      OFFSET CGROUP:FANTASMA_DER
        DB      98                              ;Shift F5
        DW      OFFSET CGROUP:ESCRIBE_SECTOR
        DB      61                              ;Entrar numero de secto
        DW      OFFSET CGROUP:SECTOR_N
        DB      0                               ;Fin tabla
DATA_SEG        ENDS
        END
```

```
LISTADO  DESP_SEC.ASM

CGROUP    GROUP CODE_SEG, DATA_SEG
          ASSUME CS:CGROUP, DS:CGROUP

VERTICAL_BAR     EQU      0BAh
HORIZONTAL_BAR   EQU      0CDh
SUPERIOR_IZQ     EQU      0C9h
SUPERIOR_DER     EQU      0BBh
INFERIOR_IZQ     EQU      0C8h
INFERIOR_DER     EQU      0BCh
TOPE_T           EQU      0CBh
FONDO_T          EQU      0CAh
TOPE_SEP         EQU      0D1h
FONDO_SEP        EQU      0CFh


CODE_SEG         SEGMENT  PUBLIC

          PUBLIC  DESP_MEDIO_SECTOR
          EXTRN   ENVIA_CRLF:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento despliega medio sector (256 bytes)                    ;
;                                                                         ;
;    DS:DX   Complemento de sector, en bytes multiplicar * 16             ;
;                                                                         ;
;Utiliza:         DESP_LINEA, ENVIA_CRLF                                  ;
;-------------------------------------------------------------------------;
DESP_MEDIO_SECTOR        PROC     NEAR
          PUSH    CX
          PUSH    DX
          MOV     CX,16             ;Despliega 16 lineas
MEDIO_SECTOR:
          CALL    DESP_LINEA
          CALL    ENVIA_CRLF
          ADD     DX,16
          LOOP    MEDIO_SECTOR
          POP     DX
          POP     CX
          RET
DESP_MEDIO_SECTOR        ENDP

          PUBLIC  DESP_LINEA
          EXTRN   ESCRIBE_HEX:NEAR
          EXTRN   ESCRIBE_CAR:NEAR
          EXTRN   ESCRIBE_CAR_N_VECES:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento despliega una linea de datos, o 16 bytes              ;
;Primero en hexa, y luego en ASCII                                        ;
;                                                                         ;
;        DS:DX Complemento a partir de sector en bytes                    ;
;                                                                         ;
; Utiliza: ESCRIBE_CAR, ESCRIBE_HEX, ESCRIBE_CAR_N_VECES                  ;
; Lee    : SECTOR                                                         ;
;-------------------------------------------------------------------------;
DESP_LINEA       PROC     NEAR
          PUSH    BX
```

```
        PUSH    CX
        PUSH    DX
        MOV     BX,DX           ;Complemento utilizado en BX
        MOV     DL,' '
        MOV     CX,3            ;Escribe 3 espacios
        CALL    ESCRIBE_CAR_N_VECES
        CMP     BX,100h
        JE      ESCRIBE_UNO
        MOV     DL,'1'
ESCRIBE_UNO:
        CALL    ESCRIBE_CAR
        MOV     DL,BL
        CALL    ESCRIBE_HEX

        MOV     DL,' '          ;Escribe separador
        CALL    ESCRIBE_CAR
        MOV     DL,VERTICAL_BAR ;Trazar lado izquierdo del cuadro
        CALL    ESCRIBE_CAR
        MOV     DL,' '
        CALL    ESCRIBE_CAR

        MOV     CX,16           ;Muestra 16 bytes
        PUSH    BX              ;Salva complemento para lazo ASCII
LAZO_HEX:
        MOV     DL,SECTOR[BX]   ;Obtener un byte
        CALL    ESCRIBE_HEX
        MOV     DL,' '          ;Escribe espacio entre numeros
        CALL    ESCRIBE_CAR
        INC     BX
        LOOP    LAZO_HEX

        MOV     DL,VERTICAL_BAR ;Escribe lado derecho del cuadro
        CALL    ESCRIBE_CAR
        MOV     DL,' '
        CALL    ESCRIBE_CAR
        MOV     CX,16
        POP     BX
LAZO_ASCII:
        MOV     DL,SECTOR[BX]
        CALL    ESCRIBE_CAR
        INC     BX
        LOOP    LAZO_ASCII
        MOV     DL,' '
        CALL    ESCRIBE_CAR
        MOV     DL,VERTICAL_BAR
        CALL    ESCRIBE_CAR

        POP     DX
        POP     CX
        POP     BX
        RET
DESP_LINEA      ENDP

        PUBLIC  INI_DESP_SEC
        EXTRN   ESCRIBE_PATRON:NEAR, ENVIA_CRLF:NEAR
```

```
        EXTRN   GOTO_XY:NEAR, ESCRIBE_FANTASMA:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   LINEAS_DESPUES_SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
DATA_SEG        ENDS
;-------------------------------------------------------------------------;
;Este procedimiento inicia el despliege de medio sector                   ;
;Utiliza:  ESCRIBE_PATRON, ENVIA_CRLF,DESP_MEDIO_SECTOR                   ;
;          ESCRIBE_FILA_NUM, GOTO_XYB                                     ;
;          LINEAS_DESPUES_SECTOR                                          ;
; Lee    :  PATRON_LINEA_SUP, PATRON_LINEA_INF                            ;
;Escribe:  SECTOR_OFFSET                                                  ;
;-------------------------------------------------------------------------;
INI_DESP_SEC    PROC NEAR
        PUSH    DX
        XOR     DL,DL                           ;Mover el cursor a posicion
        MOV     DH,LINEAS_DESPUES_SECTOR
        CALL    GOTO_XY
        CALL    ESCRIBE_FILA_NUM
        LEA     DX,PATRON_LINEA_SUP
        CALL    ESCRIBE_PATRON
        CALL    ENVIA_CRLF
        XOR     DX,DX
        MOV     SECTOR_OFFSET,DX  ;Pone complemento de sector a 0
        CALL    DESP_MEDIO_SECTOR
        LEA     DX,PATRON_LINEA_INF
        CALL    ESCRIBE_PATRON
        CALL    ESCRIBE_FANTASMA
        POP     DX
        RET
INI_DESP_SEC    ENDP

        EXTRN   ESCRIBE_CAR_N_VECES:NEAR, ESCRIBE_HEX:NEAR, ESCRIBE_CAR:
        EXTRN   ESCRIBE_DIGITO_HEX:NEAR, ENVIA_CRLF:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento escribe numeros indice (0 a F) en la parte  ;
;Superior del despliegue del medio sector                                 ;
;Utiliza: ESCRIBE_CAR_N_VECES, ESCRIBE_HEX, ESCRIBE_CAR                   ;
;          ESCRIBE_DIGITO_HEX, ENVIA_CRLF                                 ;
;-------------------------------------------------------------------------;
ESCRIBE_FILA_NUM  PROC NEAR
        PUSH    CX
        PUSH    DX
        MOV     DL,' '
        MOV     CX,9
        CALL    ESCRIBE_CAR_N_VECES
        SUB     DH,DH             ;Comenzar con cero
LAZO_NUMERO_HEX:
        MOV     AL,DH
        CALL    ESCRIBE_HEX
        MOV     DL,' '
        CALL    ESCRIBE_CAR
        INC     DH
        CMP     DH,10h
        JB      LAZO_NUMERO_HEX
```

```
        MOV     DL,' '                  ;Escribir numeros hexa sobre ve
                                        ;HEXA
        MOV     CX,2
        CALL    ESCRIBE_CAR_N_VECES
        XOR     DL,DL
LAZO_DIGITO_HEX:                        ;Escribe numeros hexa sobre ven
        CALL    ESCRIBE_DIGITO_HEX      ;ASCII
        INC     DL
        CMP     DL,10h
        JB      LAZO_DIGITO_HEX
        CALL    ENVIA_CRLF
        POP     CX
        POP     CX
        RET
ESCRIBE_FILA_NUM        ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR:BYTE
PATRON_LINEA_SUP LABEL BYTE
        DB      ' ',7
        DB      SUPERIOR_IZQ,1
        DB      HORIZONTAL_BAR,12
        DB      TOPE_SEP,1
        DB      HORIZONTAL_BAR,11
        DB      TOPE_SEP,1
        DB      HORIZONTAL_BAR,11
        DB      TOPE_SEP,1
        DB      HORIZONTAL_BAR,12
        DB      TOPE_T,1
        DB      HORIZONTAL_BAR,18
        DB      SUPERIOR_DER,1
        DB      0
PATRON_LINEA_INF  LABEL BYTE
        DB      ' ',7
        DB      INFERIOR_IZQ,1
        DB      HORIZONTAL_BAR,12
        DB      FONDO_SEP,1
        DB      HORIZONTAL_BAR,11
        DB      FONDO_SEP,1
        DB      HORIZONTAL_BAR,11
        DB      FONDO_SEP,1
        DB      HORIZONTAL_BAR,12
        DB      FONDO_T,1
        DB      HORIZONTAL_BAR,18
        DB      INFERIOR_DER,1
        DB      0
DATA_SEG        ENDS

        END
```

```
;;;;  DISK_IO.ASM

CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        PUBLIC  LEE_SECTOR

DATA_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR:BYTE
        EXTRN   DISK_DRIVE_NO:BYTE
        EXTRN   SECTOR_ACTUAL_NO:WORD
DATA_SEG        ENDS

;-------------------------------------------------------------------------;
;Este procedimiento lee el primer sector del disco A y muestra  ;
;la primera mitad del sector                                    ;
;-------------------------------------------------------------------------;
LEE_SECTOR      PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,DISK_DRIVE_NO        ;Numero de driver

        MOV     CX,1                    ;Leer un sector
        MOV     DX,SECTOR_ACTUAL_NO
        LEA     BX,SECTOR               ;Donde almacenar este sector
        INT     25h                     ;Leer el sector
        POPF                            ;Sacar registro de estado
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LEE_SECTOR      ENDP

        PUBLIC  SECTOR_PREVIO
        EXTRN   INI_DESP_SEC:NEAR, ESCRIBE_ENCABEZADO:NEAR
        EXTRN   ESCRIBE_LINEA_INDICADOR:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR_ACTUAL_NO:WORD, INDICADOR_EDITOR:BYTE
DATA_SEG        ENDS
;-------------------------------------------------------------------------;
;Este procedimiento lee el sector previo si es posible          ;
;                                                               ;
; Utiliza: ESCRIBE_ENCABEZADO, LEE_SECTOR, INI_DESP_SEC         ;
;          ESCRIBE_LINEA_INDICADOR                              ;
; Lee    : SECTOR_ACTUAL_NO,AX                                  ;
;Escribe : SECTOR_ACTUAL_NO                                     ;
;-------------------------------------------------------------------------;
SECTOR_PREVIO   PROC NEAR
        PUSH    AX
        PUSH    DX
        MOV     AX,SECTOR_ACTUAL_NO
        OR      AX,AX
        JZ      NO_DECREMENTE_SECTOR
```

```
        POP     AX
        MOV     SECTOR_ACTUAL_NO,AX
        CALL    ESCRIBE_ENCABEZADO
        CALL    LEE_SECTOR
        CALL    INI_DESP_SEC
        LEA     DX,INDICADOR_EDITOR
        CALL    ESCRIBE_LINEA_INDICADOR
NO_DECREMENTE_SECTOR:
        POP     DX
        POP     AX
        RET
SECTOR_PREVIO   ENDP

        PUBLIC  SECTOR_PROXIMO
        EXTRN   INI_DESP_SEC:NEAR, ESCRIBE_ENCABEZADO:NEAR
        EXTRN   ESCRIBE_LINEA_INDICADOR:NEAR

DATA_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR_ACTUAL_NO:WORD, INDICADOR_EDITOR:BYTE
DATA_SEG        ENDS
;-------------------------------------------------------------------------;
;Utiliza:  ESCRIBE_ENCABEZADO, LEE_SECTOR, INI_DESP_SEC                   ;
;          ESCRIBE_LINEA_INDICADOR                                        ;
;Lee    :  SECTOR_ACTUAL_NO, INDICADOR_EDITOR                            ;
;Escribe:  SECTOR_ACTUAL_NO                                              ;
;-------------------------------------------------------------------------;
SECTOR_PROXIMO PROC NEAR
        PUSH    AX
        PUSH    DX
        MOV     AX,SECTOR_ACTUAL_NO
        INC     AX
        MOV     SECTOR_ACTUAL_NO,AX
        CALL    ESCRIBE_ENCABEZADO
        CALL    LEE_SECTOR
        CALL    INI_DESP_SEC
        LEA     DX,INDICADOR_EDITOR
        CALL    ESCRIBE_LINEA_INDICADOR
        POP     DX
        POP     AX
        RET
SECTOR_PROXIMO  ENDP

        PUBLIC ESCRIBE_SECTOR
;-------------------------------------------------------------------------;
;Este procedimiento escribe un sector de regreso al disco                 ;
;                                                                         ;
; Lee:  DISK_DRIVE_NO, SECTOR_NO, SECTOR                                  ;
;-------------------------------------------------------------------------;
ESCRIBE_SECTOR  PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,DISK_DRIVE_NO        ;Numero del disk driver
        MOV     CX,1                    ;Escribe un sector
```

```
        MOV     DX,SECTOR_ACTUAL_NO
        LEA     BX,SECTOR
        INT     26H
        POPF
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
ESCRIBE_SECTOR  ENDP

        PUBLIC  SELECTOR_N
        EXTRN   LEE_DECIMAL:NEAR,ESCRIBE_LINEA_INDICADOR:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   ENTRAR_NO_SECTOR:BYTE
        EXTRN   SECTOR_ACTUAL_NO:WORD
DATA_SEG        ENDS
;-------------------------------------------------------------------------;
; Este procedimiento utiliza la tecla F3 para solicitar el                ;
; numero de sector que se desea ver                                       ;
;                                                                         ;
;  AX : direccion del mensaje                                             ;
;                                                                         ;
; Utiliza: LEE_DECIMAL, ESCRIBE_LINEA_INDICADOR                           ;
;          SECTOR_PROXIMO                                                 ;
; Lee    : ENTRAR_NO_SECTOR,SECTOR_ACTUAL                                 ;
;-------------------------------------------------------------------------;
SELECTOR_N      PROC    NEAR
        PUSH    AX
        PUSH    DX
MAL_DIGITO:
        LEA     DX,ENTRAR_NO_SECTOR
        CALL    ESCRIBE_LINEA_INDICADOR
        CALL    LEE_DECIMAL
        JC      MAL_DIGITO
        DEC     AX
        MOV     SECTOR_ACTUAL_NO,AX
        CALL    SECTOR_PROXIMO
        POP     DX
        POP     AX
        RET
SELECTOR_N      ENDP

CODE_SEG        ENDS

        END
```

```
C:\TASM\WINNT>type ED.ASM

DGROUP GROUP CODE_SEG, DATA_SEG
       ASSUME CS:DGROUP

CODE_SEG  SEGMENT PUBLIC

PUBLIC ESCRIBE_DECIMAL
;-------------------------------------------------------------------------;
; Este procedimiento  escribe un numero sin signo de 16 bit              ;
; en notacion decimal                                                     ;
;                                                                         ;
;    DX  Numero sin signo de 16 bits                                      ;
;    Utiliza: ESCRIBE_DIGITO_HEX                                          ;
;-------------------------------------------------------------------------;
ESCRIBE_DECIMAL   PROC  NEAR
     PUSH  AX
     PUSH  CX
     PUSH  DX
     PUSH  SI
     MOV   AX,DX
     MOV   SI,10
     XOR   CX,CX
NO_CERO:
     XOR   DX,DX
     DIV   SI
     PUSH  DX
     INC   CX
     OR    AX,AX
     JNE   NO_CERO
LAZO_ESCRIBE_DIGITOS:
     POP   DX
     CALL  ESCRIBE_DIGITO_HEX
     LOOP  LAZO_ESCRIBE_DIGITOS
FIN_DECIMAL:
     POP   SI
     POP   DX
     POP   CX
     POP   AX
     RET
ESCRIBE_DECIMAL   ENDP

  PUBLIC   ESCRIBE_HEX
ESCRIBE_HEX  PROC  NEAR    ; punto de entrada
  PUSH  CX
  PUSH  DX
  MOV   DH,DL
  MOV   CX,4
  SHR   DL,CL
  CALL  ESCRIBE_DIGITO_HEX
  MOV   DL,DH
  AND   DL,0Fh
  CALL  ESCRIBE_DIGITO_HEX
  POP   DX
  POP   CX
  RET
ESCRIBE_HEX  ENDP
```

```
  PUBLIC  ESCRIBE_DIGITO_HEX
ESCRIBE_DIGITO_HEX  PROC  NEAR
  PUSH  DX
  CMP   DL,10
  JAE   HEX_LETRA
  ADD   DL,"0"
  JMP   SHORT SALIDA_DIG
HEX_LETRA:
  ADD   DL,"A"-10
SALIDA_DIG:
  CALL  ESCRIBE_CAR
  POP   DX
  RET
ESCRIBE_DIGITO_HEX  ENDP

        PUBLIC  ESCRIBE_CAR
        EXTRN   POSICION_CURSOR:NEAR
;-------------------------------------------------------------------------;
; Este es un procedimiento que imprime un caracter en pantalla  ;
; utilizando rutinas del BIOS, asi caracteres como espacio hacia;
; atras son tratados como cualquier otro caracter y son despla- ;
; zados.                                                        ;
; Este procedimiento se encarga de actualizar la posicion del   ;
; cursor.                                                       ;
;                                                               ;
; DL  Byte a imprimir en la pantalla                            ;
;                                                               ;
; Utiliza: POSICION_CURSOR                                      ;
;-------------------------------------------------------------------------;
ESCRIBE_CAR  PROC  NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,9                ;Llama salida de caracter/atributo
        MOV     BH,0                ;Establece despliegue pagina 0
        MOV     CX,1                ;Escribe solo un caracter
        MOV     AL,DL               ;Caracter a escribir
        MOV     BL,7                ;Atributo normal
        INT     10h                 ;Escribe caracter y atributo
        CALL    POSICION_CURSOR     ;Mover cursor siguiente posicion
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
ESCRIBE_CAR     ENDP

        PUBLIC  ESCRIBE_CAR_N_VECES
;-------------------------------------------------------------------------;
;Este procedimiento escribe N copias de un caracter             ;
;                                                               ;
; DL  Codigo del caracter                                       ;
; CX  Numero de veces a escribir el caracter                    ;
```

```
;                                                                      ;
; Utiliza: ESCRIBE_CAR                                                 ;
;----------------------------------------------------------------------;
ESCRIBE_CAR_N_VECES     PROC    NEAR
        PUSH    CX
N_VECES:
        CALL    ESCRIBE_CAR
        LOOP    N_VECES
        POP     CX
        RET
ESCRIBE_CAR_N_VECES     ENDP

        PUBLIC  ESCRIBE_PATRON
;----------------------------------------------------------------------;
; Este procedimiento escribe una linea en la pantalla, basandose      ;
; en la forma de los datos                                             ;
;                                                                      ;
;   DX    Numero de veces que se escribe un caracter                   ;
; DS:DX Direccion anterior del segmento                                ;
;                                                                      ;
; Utiliza: ESCRIBE_CAR_N_VECES                                         ;
;----------------------------------------------------------------------;
ESCRIBE_PATRON  PROC NEAR
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSHF                           ;Salva la bandera de direccion
        CLD                             ;Pone bandera de direccion p/incremento
        MOV     SI,DX                   ;Mueve incremento dentro de SI para LODS
LAZO_PATRON:
        LODSB                           ;Obtiene caracter en DL
        OR      AL,AL                   ;¿Es el fin de datos?
        JZ      FIN_PATRON              ;Si, retorne
        MOV     DL,AL                   ;No, escribir caracter N veces
        LODSB                           ;Obtiene el contador de repeticion en AL
        MOV     CL,AL                   ;Y coloca en CX para ESCRIBE_CAR_N_VECES
        XOR     CH,CH                   ;Pone a cero byte superior de CX
        CALL    ESCRIBE_CAR_N_VECES
        JMP     LAZO_PATRON
FIN_PATRON:
        POPF
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
ESCRIBE_PATRON  ENDP

        PUBLIC  ESCRIBE_ENCABEZADO
DATA_SEG        SEGMENT PUBLIC
        EXTRN   LINEA_ENCABEZADO_NO:BYTE
        EXTRN   ENCABEZADO_PARTE_1:BYTE
        EXTRN   ENCABEZADO_PARTE_2:BYTE
        EXTRN   DISK_DRIVE_NO:BYTE
```

```
        EXTRN   SECTOR_ACTUAL_NO:WORD
DATA_SEG        ENDS
        EXTRN   GOTO_XY:NEAR, LIMPIA_HASTA_FIN_DE_LINEA:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento escribe el encabezado con el numero del disk;
;driver y el sector                                                       ;
;                                                                         ;
; Utiliza: GOTO_XY,ESCRIBE_CADENA, ESCRIBE_CAR,                           ;
;          ESCRIBE_DECIMAL,LINEA_ENCABEZADO_NO,                           ;
;          ENCABEZADO_PARTE_1, ENCABEZADO_PARTE_2                         ;
;          DISK_DRIVE_NO, SECTOR_ACTUAL_NO                                ;
;-------------------------------------------------------------------------;
ESCRIBE_ENCABEZADO PROC NEAR
        PUSH    DX
        XOR     DL,DL               ;Mover cursor a linea de encabezado
        MOV     DH,LINEA_ENCABEZADO_NO
        CALL    GOTO_XY
        LEA     DX,ENCABEZADO_PARTE_1
        CALL    ESCRIBE_CADENA
        MOV     DL,DISK_DRIVE_NO
        ADD     DL,'A'
        CALL    ESCRIBE_CAR
        LEA     DX,ENCABEZADO_PARTE_2
        CALL    ESCRIBE_CADENA
        MOV     DX,SECTOR_ACTUAL_NO
        CALL    ESCRIBE_DECIMAL
        CALL    LIMPIA_HASTA_FIN_DE_LINEA
        POP     DX
        RET
ESCRIBE_ENCABEZADO ENDP

        PUBLIC ESCRIBE_CADENA
;-------------------------------------------------------------------------;
; Este procedimiento escribe una cadena de caracteres en la               ;
; pantalla. La cadena debe terminar con DB 0                              ;
;                                                                         ;
; DS:DX  Direccion de la cadena                                           ;
;                                                                         ;
; Utiliza        ESCRIBE_CAR                                              ;
;-------------------------------------------------------------------------;
ESCRIBE_CADENA  PROC NEAR
        PUSH    AX
        PUSH    DX
        PUSH    SI
        PUSHF
        CLD
        MOV     SI,DX
LAZO_CADENA:
        LODSB
        OR      AL,AL
        JZ      FIN_CADENA
        MOV     DL,AL
        CALL    ESCRIBE_CAR
        JMP     LAZO_CADENA
FIN_CADENA:
```

```
        POPF
        POP     SI
        POP     DX
        POP     AX
        RET
ESCRIBE_CADENA ENDP

        PUBLIC  ESCRIBE_LINEA_INDICADOR
        EXTRN   LIMPIA_HASTA_FIN_DE_LINEA:NEAR
        EXTRN   GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   LINEA_INDICADOR_NO:BYTE
DATA_SEG        ENDS
;-----------------------------------------------------------------------;
;Este procedimiento escribe la linea de indicaciones en la pantal;
;la y limpia hasta el final de la linea                           ;
;                                                                 ;
; PS:DX  Direccion de la linea de indicaciones                    ;
; Lee:   LINEA_INDICADOR_NO                                       ;
;-----------------------------------------------------------------------;
ESCRIBE_LINEA_INDICADOR PROC NEAR
        PUSH    DX
        XOR     BL,BL
        MOV     DH,LINEA_INDICADOR_NO
        CALL    GOTO_XY
        POP     DX
        CALL    ESCRIBE_CADENA
        CALL    LIMPIA_HASTA_FIN_DE_LINEA
        RET
ESCRIBE_LINEA_INDICADOR ENDP

        PUBLIC  ESCRIBE_ATRIBUTO_N_VECES
        EXTRN   POSICION_CURSOR:NEAR
;-----------------------------------------------------------------------;
;Este procedimiento establece los atributos de N caracteres       ;
;comenzando con el de la posicion actual del cursor               ;
;                                                                 ;
;   CX  numero de caracteres a establecer el atributo            ;
;   DL  nuevo atributo para el caracter                          ;
;                                                                 ;
; Utiliza: POSICION_CURSOR                                        ;
;-----------------------------------------------------------------------;
ESCRIBE_ATRIBUTO_N_VECES PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BL,DL               ;Coloca nuevo atributo
        XOR     BH,BH               ;Poner despliegue de pagina 0
        MOV     DX,CX               ;CX utilizado por rutina del BIOS
        MOV     CX,1
LAZO_ATRIBUTO:
        MOV     AH,8
        INT     10h
        MOV     AH,9
```

```
            INT     10h
            CALL    POSICION_CURSOR
            DEC     DX

            JNZ     LAZO_ATRIBUTO
            POP     DX
            POP     CX
            POP     BX
            POP     AX
            RET
ESCRIBE_ATRIBUTO_N_VECES ENDP

CODE_SEG  ENDS

    END
```

```
;LISTO  FANTASMA.ASM


CGROUP GROUP CODE_SEG, DATA_SEG
       ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
       PUBLIC   MOVER_A_POSICION_HEXA
       EXTRN    GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
       EXTRN    LINEAS_DESPUES_SECTOR:BYTE
DATA_SEG        ENDS
;-------------------------------------------------------------------;
;Este procedimiento mueve el cursor real a la posicion del cur-     ;
;sor fantasma en la ventana hexa                                    ;
;                                                                   ;
; Utiliza: GOTO_XY                                                  ;
; Lee    : LINEAS_DESPUES_SECTOR, CURSOR_FANTASMA_X                 ;
;          CURSOR_FANTASMA_Y                                        ;
;-------------------------------------------------------------------;
MOVER_A_POSICION_HEXA PROC NEAR
       PUSH    AX
       PUSH    CX
       PUSH    DX
       MOV     DH,LINEAS_DESPUES_SECTOR ;Hallar fila de fantasma
       ADD     DH,2                     ;Mas fila hexa y barra
       ADD     DH,CURSOR_FANTASMA_Y     ;DH fila cursor fantasma
       MOV     DL,8                     ;Lado izquierdo
       MOV     CL,3                     ;Cada columna usa 3 caractere
       MOV     AL,CURSOR_FANTASMA_X     ;se debe multiplicar por 3
       MUL     CL
       ADD     DL,AL
       CALL    GOTO_XY
       POP     DX
       POP     CX
       POP     AX
       RET
MOVER_A_POSICION_HEXA  ENDP

       PUBLIC  MOVER_A_POSICION_ASCII
       EXTRN   GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
       EXTRN    LINEAS_DESPUES_SECTOR:BYTE
DATA_SEG        ENDS
;-------------------------------------------------------------------;
; Este procedimiento mueve el cursor real al inicio del cursor      ;
; fantasma en la ventana ASCII                                      ;
;                                                                   ;
; Utiliza: GOTO_XY                                                  ;
; lee    : CURSOR_FANTAMA_Y                                         ;
;-------------------------------------------------------------------;
MOVER_A_POSICION_ASCII PROC NEAR
       PUSH    AX
       PUSH    DX
       MOV     DH,LINEAS_DESPUES_SECTOR;Hallar fila cursor fantasma
       ADD     DH,2                    ;mas fila hex y barra horiz.
       ADD     DH,CURSOR_FANTASMA_Y    ;DH fila cursor fantasma
```

```
        MOV     DL,59                           ;Lado izquierdo
        ADD     DL,CURSOR_FANTASMA_X            ;Obtener posicion X
        CALL    GOTO_XY
        POP     DX
        POP     AX
        RET
MOVER_A_POSICION_ASCII  ENDP

        PUBLIC SALVAR_CURSOR_REAL
;-------------------------------------------------------------------------;
; Este procedimiento salva la posicion del cursor real en dos   ;
; variables CURSOR_REAL_X y CURSOR_REAL_Y                       ;
;                                                               ;
; Escribe: CURSOR_REAL_X, CURSOR_REAL_Y                         ;
;-------------------------------------------------------------------------;
SALVAR_CURSOR_REAL PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                            ;Leer posicion del cursor
        MOV     BH,BH                           ;en pagina 0
        INT     10h                             ;y retornarla en DL, DH
        MOV     CURSOR_REAL_Y,DL
        MOV     CURSOR_REAL_X,DH
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
SALVAR_CURSOR_REAL ENDP

        PUBLIC  RECUPERA_CURSOR_REAL
        EXTRN   GOTO_XY:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento recupera el cursor real en su posicion origi;
;nal guardada en CURSOR_REAL_X, y en CURSOR_REAL_Y              ;
;                                                               ;
;Utiliza: GOTO_XY                                              ;
;Lee    : CURSOR_REAL_X, CURSOR_REAL_Y                         ;
;-------------------------------------------------------------------------;
RECUPERA_CURSOR_REAL PROC NEAR
        PUSH    DX
        MOV     DL,CURSOR_REAL_Y
        MOV     DH,CURSOR_REAL_X
        CALL    GOTO_XY
        POP     DX
        RET
RECUPERA_CURSOR_REAL ENDP

        PUBLIC  ESCRIBE_FANTASMA
        EXTRN   ESCRIBE_ATRIBUTO_N_VECES:NEAR
;-------------------------------------------------------------------------;
;Este procedimiento utiliza CURSOR_X y CURSOR_Y por medio de    ;
;MOVER_A..... como coordenadas para el cursor fantasma,  para   ;
```

```
; escribir el cursor fantasma                                          ;
;                                                                      ;
; Utiliza: ESCRIBE_ATRIBUTO_N_VECES, SALVAR_CURSOR_REAL               ;
;          RECUPERA_CURSOR_REAL, MOVER_A_POSICION_HEXA                ;
;          MOVER_A_POSICION_ASCII                                     ;
;----------------------------------------------------------------------;
ESCRIBE_FANTASMA  PROC NEAR
          PUSH    CX
          PUSH    DX
          CALL    SALVAR_CURSOR_REAL
          CALL    MOVER_A_POSICION_HEXA
          MOV     CX,4
          MOV     DL,70h
          CALL    ESCRIBE_ATRIBUTO_N_VECES
          CALL    MOVER_A_POSICION_ASCII
          MOV     CX,1                          ;Cursor 1 caract. de ancho
          CALL    ESCRIBE_ATRIBUTO_N_VECES
          CALL    RECUPERA_CURSOR_REAL
          POP     DX
          POP     CX
          RET
ESCRIBE_FANTASMA  ENDP

          PUBLIC  BORRA_FANTASMA
          EXTRN   ESCRIBE_ATRIBUTO_N_VECES:NEAR
;----------------------------------------------------------------------;
;Este procedimiento borra el cursor fantasma                           ;
;                                                                      ;
; Utiliza:  ESCRIBE_ATRIBUTO_N_VECES, SALVAR_CURSOR_REAL              ;
;           RECUPERAR_CURSOR_REAL, MOVER_A_POSICION_HEXA              ;
;           MOVER_A_POSICION_ASCII                                    ;
;----------------------------------------------------------------------;
BORRA_FANTASMA  PROC NEAR
          PUSH    CX
          PUSH    DX
          CALL    SALVAR_CURSOR_REAL
          CALL    MOVER_A_POSICION_HEXA
          MOV     CX,4
          MOV     DL,7
          CALL    ESCRIBE_ATRIBUTO_N_VECES
          CALL    MOVER_A_POSICION_ASCII
          MOV     CX,1
          CALL    ESCRIBE_ATRIBUTO_N_VECES
          CALL    RECUPERA_CURSOR_REAL
          POP     DX
          POP     CX
          RET
BORRA_FANTASMA  ENDP

;----------------------------------------------------------------------;
;Estos cuatro procedimientos mueven el cursor fantasma                 ;
;                                                                      ;
; Utiliza: BORRA_FANTASMA, ESCRIBE_FANTASMA                           ;
;          DESPLAZA_ARR, DESPLAZA_ABA                                 ;
; Lee    : CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y                       ;
```

```
; Escribe: CURSOR_FANTASMA_X, CURUSOR_FANTASMA_Y                    :
;                                                                   ;
        PUBLIC  FANTASMA_ARR
FANTASMA_ARR    PROC    NEAR
        CALL    BORRA_FANTASMA
        DEC     CURSOR_FANTASMA_Y
        JNS     NO_TOPE
        CALL    DESPLAZA_ABA

NO_TOPE:
        CALL    ESCRIBE_FANTASMA
        RET
FANTASMA_ARR    ENDP

        PUBLIC  FANTASMA_ABA
FANTASMA_ABA    PROC    NEAR
        CALL    BORRA_FANTASMA
        INC     CURSOR_FANTASMA_Y
        CMP     CURSOR_FANTASMA_Y,16  ;¿Cursor no esta extremo inferior
        JB      NO_FONDO
        CALL    DESPLAZA_ARR
NO_FONDO:
        CALL    ESCRIBE_FANTASMA
        RET
FANTASMA_ABA    ENDP

        PUBLIC  FANTASMA_IZQ
FANTASMA_IZQ    PROC    NEAR
        CALL    BORRA_FANTASMA
        DEC     CURSOR_FANTASMA_X
        JNS     NO_IZQ
        MOV     CURSOR_FANTASMA_X,0  ;Esta en extremo, no se mueve
NO_IZQ:
        CALL    ESCRIBE_FANTASMA
        RET
FANTASMA_IZQ    ENDP

        PUBLIC  FANTASMA_DER
FANTASMA_DER    PROC    NEAR
        CALL    BORRA_FANTASMA
        INC     CURSOR_FANTASMA_X
        CMP     CURSOR_FANTASMA_X,16
        JB      NO_DER
        MOV     CURSOR_FANTASMA_X,15
NO_DER:
        CALL    ESCRIBE_FANTASMA
        RET
FANTASMA_DER    ENDP

        EXTRN   DESP_MEDIO_SECTOR:NEAR, GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   LINEAS_DESPUES_SECTOR:BYTE
DATA_SEG        ENDS
```

```
;
; Estos dos procedimientos permiten moverse entre los dos medios
; sectores desplegados.
;
; Utiliza: ESCRIBE_FANTASMA, DESP_MEDIO_SECTOR
;          BORRA_FANTASMA, GOTO_XY, SALVAR_CURSOR_REAL
;          RECUPERAR_CURSOR_REAL
; Lee    : LINEAS_DESPUES_SECTOR
;Escribe : SECTOR_OFFSET, CURSOR_FANTASMA_Y
;
DESPLAZA_ARR    PROC NEAR
          PUSH    DX
          CALL    BORRA_FANTASMA
          CALL    SALVAR_CURSOR_REAL
          XOR     DL,DL                   ;Pone cursor para despliega
          MOV     DH,LINEAS_DESPUES_SECTOR
          ADD     DH,2
          CALL    GOTO_XY
          MOV     DX,256                  ;Despliega la segunda mit
          MOV     SECTOR_OFFSET,DX
          CALL    DESP_MEDIO_SECTOR
          CALL    RECUPERA_CURSOR_REAL
          MOV     CURSOR_FANTASMA_Y,0     ;Al tope del segundo medi
          CALL    ESCRIBE_FANTASMA
          POP     DX
          RET
DESPLAZA_ARR    ENDP

DESPLAZA_ABA    PROC NEAR
          PUSH    DX
          CALL    BORRA_FANTASMA
          CALL    SALVAR_CURSOR_REAL
          XOR     DL,DL
          MOV     DH,LINEAS_DESPUES_SECTOR
          ADD     DH,2
          CALL    GOTO_XY
          XOR     DX,DX
          MOV     SECTOR_OFFSET,DX
          CALL    DESP_MEDIO_SECTOR
          CALL    RECUPERA_CURSOR_REAL
          MOV     CURSOR_FANTASMA_Y,15
          CALL    ESCRIBE_FANTASMA
          POP     DX
          RET
DESPLAZA_ABA    ENDP
CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
          CURSOR_REAL_X    DB       0
          CURSOR_REAL_Y    DB       0
          PUBLIC CURSOR_FANTASMA_X, CURSOR_FANTASMA_Y
CURSOR_FANTASMA_X          DB       0
CURSOR_FANTASMA_Y          DB       0
DATA_SEG        ENDS
          END
```

```
; [illegible].ASM

CR      EQU     13                      ;Retorno del carro
LF      EQU     10                      ;Avance de linea
CSEGMENT GROUP  CODE_SEG
        ASSUME  CS:CSEGMENT
CODE_SEG        SEGMENT PUBLIC
        PUBLIC  ENVIA_CRLF
;-------------------------------------------------------------------------
;Esta rutina solo envia un retorno de carro y un avance de linea
;-------------------------------------------------------------------------
ENVIA_CRLF      PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     AH,2
        MOV     DL,CR
        INT     21h
        MOV     DL,LF
        INT     21h
        POP     DX
        POP     AX
        RET
ENVIA_CRLF      ENDP
        PUBLIC  LIMPIA_PANTALLA
;-------------------------------------------------------------------------
;Este procedimiento limpia la pantalla completamente
;-------------------------------------------------------------------------
LIMPIA_PANTALLA PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        SUB     AL,AL                   ;Limpiar ventana entera
        MOV     CX,0                    ;Esquina superior izq. esta en 0
        MOV     DH,24                   ;Linea inferior esta en linea 24
        MOV     DL,79                   ;Lado derecho esta en columna 79
        MOV     BH,7                    ;Utiliza atributo normal para li
        MOV     AH,6                    ;Llama funcion DESPLAZAR HACIA AB
        INT     10h                     ;Limpia pantalla
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LIMPIA_PANTALLA ENDP
        PUBLIC  GOTO_XY
;-------------------------------------------------------------------------
;Este procedimiento mueve el cursor DH fila, DL columna
;-------------------------------------------------------------------------
GOTO_XY  PROC NEAR
        PUSH    AX
        PUSH    BX
        MOV     BH,0                    ;Despliega pagina 0
        MOV     AH,2                    ;Llamada  para poner posicion de
        INT     10h
        POP     DX
        POP     AX
```

```
        RET
GOTO_XY         ENDP
        PUBLIC  POSICION_CURSOR
;-------------------------------------------------------------------------;
;Mover cursor una posicion a la derecha o a la siguiente linea            ;
;Utiliza: ENVIA_CRLF                                                      ;
;-------------------------------------------------------------------------;
POSICION_CURSOR PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3            ;Lee la posicion actual del cursor
        MOV     BH,0            ;En pagina 0
        INT     10h             ;Lee posicion cursor
        MOV     AH,2            ;Colocar nueva posicion del cursor
        INC     DL              ;Establecer columna a la siguiente
        CMP     DL,79           ;Asegurarse de que columna no sea
        JBE     BIEN            ;Ir a la siguiente linea
        CALL    ENVIA_CRLF
        JMP     ECHO
BIEN:   INT     10h
ECHO:   POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
POSICION_CURSOR ENDP
        PUBLIC  LIMPIA_HASTA_FIN_DE_LINEA
;-------------------------------------------------------------------------;
;Limpar linea desde cursor hasta final de linea                           ;
;-------------------------------------------------------------------------;
LIMPIA_HASTA_FIN_DE_LINEA PROC NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3            ;Funcion leer posicion del cursor
        MOV     BH,0            ;en pagina 0
        INT     10h             ;Obtener X Y en DL, DH
        MOV     AH,6            ;Funcion para limpiar ventana
        XOR     AL,AL           ;Limpiar ventana
        MOV     CH,DH           ;Todo en la misma linea
        MOV     CL,DL           ;Comenzar en la posicion del curs
        MOV     DL,79           ;Y detenerse en el final de la li
        MOV     BH,7            ;Utilizar atributo normal
        INT     10h
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LIMPIA_HASTA_FIN_DE_LINEA    ENDP
CODE_SEG        ENDS
        END
```

ANEXO 3

RESUMEN DEL MACRO ASSEMBLER

A3 EL ASSEMBLER

El lenguaje de ensamble ventaja de permitir escribir programas al nivel que el microprocesador comprende, pero sin forzar al programador a memorizar un conjunto de codigos numericos. Simplemente se escriben las instrucciones en mnemonico, luego se corre un programa assembler para convertirlas a sus equivalentes numericos.

El programa escrito utilizando mnemonico (abreviaciones en ingles) es llamado **programa fuente** y el programa ya traducido a codigos numericos directamente compatibles con el microprocesador es llamado **programa objeto.** Asi, el trabajo del assembler es convertir el programa fuente, orientado a los humanos, en programa objeto, que el microprocesador puede entender.

Para el presente trabajo se utilizará el Macro Assembler de IMB

A3.1 DESARROLLANDO UN PROGRAMA EN LENGUAJE DE ENSAMBLE

Aunque los programas en lenguaje de ensamble se ven bastante diferentes a los programas en BASIC, se puede seguir el mismo procedimiento para desarrollarlos. Sin embargo, en el lenguaje de ensamble la maquina esta mas involucrada. Hay seis pasos en el desarrollo de un programa en lenguaje de ensamble:

1. Definir la tarea y desarrollar el programa, esto a menudo requiere diseñar un flujograma.

2. Escribir las instrucciones del programa en la computadora utilizando un editor y luego grabar el programa.

3. Compilar el programa utilizando el program assembler. Si el assembler encuentra errores, corregirlos con el editor y volver a compilar el prgrama.

4. Convertir la salida del assembler a un programa ejecutable utilizando el encadenador (Linker)

5. Ejecutar el programa (correrlo)

6. Revisar los resultados, si difieren de la forma esperada, se deben de buscar y corregir los errores, es decir, se debe de depurar el programa.

Si el programa es corto y simple, se pueden ejecutar estos pasos rapidamente. Sin embargo, programas mas largos y complejos requieren mas tiempo en cada paso, especialmente definir el programa. La sugerencia para una mayor eficiencia en el desarrollo de programas es utilizar la tecnica de programación Top-Down (De arriba hacia abajo). Mas adelante se daran varios ejemplos.

El paso 2 se refiere a un editor. Se puede utilizar varios editor de textos que produzcan texto puramente ASCII (Caracteres regulares sin codigos de cotrol o codigos de formateo) Estos

editores incluyen WordStar, Microsoft Word, Multimate y el editor personal de IBM. Si no se dispone de ninguno de estos programas, se puede utilizar el EDLIN programa que viene en el disco del DOS.

Se debe de recordar que la computadora no puede ejecutar el programa que se escribe utilizando el editor. El programa assembler debe de compilar este programa para convertirlo en programa objeto que es el que la computadora puede comprender.

A3.2 EL ENCADENADOR (LINKER)

El DOS puede almacenar un programa en cualquier lugar conveniente de la memoria; esto libera al programador de tener que decir donde colocar el programa. Sin embargo,para usar ésta característica se debe de escribir el programa compilado a una forma que pueda ser trasladada (el término en computación es relocalizable). Esto implica usar un programa llamado LINK, el cual se encuentra en el disco del Macro Assembler.

Se hará referencia al archivo en disco que contiene un progama compilado como un **modulo objeto** Similarmente, se referirá al archivo que contiene la versión relocalizable de el programa compilado como **modulo ejecutable.** Es decir, el trabajo de LINK es crear un modulo ejecutable a partir de un modulo objeto.

También se pueden construir programas por secciones.Para hacer esto, se escribe una sección del programa y se compila. Si el assembler reporta errores se corrige y se compila de nuevo. Una vez que el primer módulo está sin errores se repite el proceso para un segundo módulo. Luego para un tercero (si lo hay), y así sucesivamente asi, eventualmente se concluirá con varios módulos objetos que en combinación, hacen todo para lo que el programa fué diseñado.

Es necesario aclarar que el encadenador se debe de utilizar para cualquier programa que se escriba, aún aquellos que solo tienen un módulo objeto.

A3.3 PROGRAMACION TOP-DOWN

Cuando se crea un programa la inclinación natural es de escribirlo como se haria con lápiz y papel: entrar la primera instrucción luego la segunda, tercera, y así sucesivamente, hasta el final.Esta "fuerza bruta", puede trabajar para programas que son cortos y simples, pero muy amenudo provoca errores y produce programas que son dificiles de comprender (y aún mas dificiles de actualizar mas tarde). Gracias a las capacidades de los microcomputadores de palabras, hay una forma mas facil, mas legible, y mas eficiente para desarrollar programas. Esta tecnica es llamada top-down.

El diseño top-down comienza con un diseño del programa, luego progresivamente se agregan los detalles. El bosquejo deverá ser una serie de lineas que digan que pasos debe de seguir el programa para ejecutar una tarea. Por ejemplo, si se desea desarrollar un programa que ejecute una devarias tareas, dependiendo de la seleccion que haga el usuario a partir de un

menú, el bosquejo podría verse como ésto:

```
; Desplegar el menú de selecciones.
; Preguntar a usuario su selección.
; Leer la selección del usuario.
; Validar la entrada.
; Si la entrada es legal realizar la tarea seleccionada.
```

Los punto y como indican que estas líneas representan comentarios en lugar de instrucciones. Realizan la misma función que REM en PASIC.

A partir de aquí se puede utilizar un editor para insertar las instrucciones entre las líneas de comentarios. Ya que cada línea define una sola tarea, se puede completar individualmente y probar cada una de ellas antes de continuar con la siguiente. Es decir, comenzando por incertar el primer grupo de instrucciones (las que despliegan el menú, en éste ejemplo), luego grabar el programa en disco y ejecutar el resto de pasos del programa (compilar, encadenar, y ejecutar). Ejecutando esto se sabe si el programa terminado parcialmente está trabajando correctamente. Si no, habrá que depurarlo y tratar de nuevo. Cuando la primera parte esté trabajando correctamente, se procede a la segunda parte, luego a la tercera, y así sucesivamente.

Esto puede parecer una forma bastante lenta de desarrollar un programa, pero tiene varias ventajas:

1. Fuerza a planificar ordenadamente la estructura del programa

2. Las líneas de comentario suministran cierta documentación al programa final.

3. Asegura que cada paso esté trabajando correctamente antes de proceder.

El precio de no seguir las indicaciones anteriores, que en muchas ocasiones no se consideran en serio, es el diseño de programas ineficientes, sin tomar en cuenta la gran cantidad de tiempo que se pierde tratando de localizar y corregir errores.

A3.4 INSTRUCCIONES FUENTE

Una vez discutido el mecanismo de desarrollar programas, estudiaran las instrucciones mismas. Un programa fuente escrito en la computadora es una secuencia de instrucciones diseñadas para ejecutar una tarea específica.

Una instrucción fuente (una línea en el programa) puede ser o bien una **instrucción en lenguaje de ensamble** o un **directiva assembler**.

Las instrucciones en lenguaje de ensamble son las que le indican al microprocesador lo que se debe hacer. Por el contrario, las directivas assembler, le dicen al assembler que hacer con las instrucciones y los datos que se entran. Las directivas se conocen también como "pseudo-operaciones".

Las instrucciones fuentes de una u otra clase pueden incluir también **operadores**, los cuales ledan información al assembler

marca de un operando en donde existe ambiguedad.
A continuación se discutirán las instrucciones en lenguaje de ensamble, directivas y operadores.

A3.4.1 Constantes en instrucciones fuente.

El ensamblador permite entrar constantes en varias formas las mas comunes son:

1. Binario - una secuencia de unos y ceros seguidos por la letra B por ejemplo, 10000101B
2. Decimal - una secuencia de digitos de cero a nueve, con o sin la letra de D: por ejemplo 129D o 129.
3. Exadecimal - una secuencia de digitos de cero a nueve y letras de la A a la F, seguida por la letra H el primer caracter debe ser uno de los digitos de 0 a 9; por ejemplo, 0E23H, y nunca E23H lo que se confundiria con un nombre de variable o una etiqueta.
4. Caracter -Una cadena de letras,numeros o simbolos encerrados "simples o dobles" IBM suministra ambas formas de esta manera se pueden poner comillas dentro de un mensaje como en "Don't enter a number here".

A3.4.2 Números Negativos

Se pueden especificar números negativos,Si el número es un valor decimal, simplemente se presede con un signo menos, si es un número binario o exadecimal se debe de entrar en complemento A dos por ejemplo, 11100000B y 0E0H son la forma en complemento A dos de el decimal -32.

A3.5 INSTRUCCIONES EN LENGUAJE DE ENSAMBLER

Cada instrucción en lenguaje de ensamble en un programa fuente puede tener hasta cuatro **campos** como sigue:
[Etiqueta:] mnemonico [operando] [;comentario] de estos, solamente el campo mnemonico es siempre operatorio la etiqueta y el campo de comentario son siempre opcionales. El campo del operando aparece unicamente solo con instrucciones que requieren un operando.(Los campos de comentario, operando y etiqueta se muestran en corchetes indicando que son obsionales; no se deben de escribir los corchetes en los programas.).
Se pueden entrar estos campos en cualquier parte de la linea. pero debe de serarse con al menos un espacio, Una instrucción en lenguaje de ensamble que utiliza los cuatro campos es:
GET[COUNT: MOV CX,01 ;Inicializa contador.

A3.5.1 Campo de Etiqueta

El campo de etiqueta asigna un nombre a una instruccion en lenguaje de ensamble esto le permite a otras instrucciones en el programa referirse a la instrucción. Asi, las etiquetas en lenguaje de ensamble sirven con el mismo propósito como los

numero de lista en programas basic.
Una etiqueta puede tener hasta 31 caracteres de longitud y debe de terminar con dos puntos (:) y puede consistir de:

Las letras de la A a la Z (o de la a a la z , el assembler no distingue entre letras mayusculas y minúsculas.)

Los digitos numericos del 0 al 9

Estos caracteres especiales: ?.@_$ la etiqueta puede comenzar con cualquier caracter excepto un digito, si se utiliza un punto (.), debe de ser el primer caracter. Los simbolos AR, A1,AY,PH, y todos los otros nombres de registros no se pueden utilizar en etiquetas.

No se pueden colocar espacios en las etiquetas, pero se puede aclarar su significado utilizando el guión bajo (_).

A3.5.2 Selección de Nombres de Etiquetas

Ya que el assembler permite entrar varias combinaciones de letras, digitos o simbolos, casi cualquier etiqueta que se pueda pensar es aceptable. Sin embargo, se recomienda seguir las siguientes indicaciones para seleccionar las etiquetas:

Hacer el nombre de etiqueta tan corto como sea posible mientras todavia sea razonable. Asi, MPH espreferible a MILLAS_POR_HORAS.

Hacer el nombre facil de escribir sin errores. Los problemas usuales de escritura se producen con varias letras identicas (tales como HHH) y caracteres parecidos (tal como la letra O y el número 0, letra l minuscula y el número 1, y la letra S y el número 5).
No utilizar etiquetas que pueden confundirce con cualquier otra. Por ejemplo, sede de evitar utilizar cosas como XXXX Y XXXY.

A3.5.3 Campo Mnémonico

El campo mnemonico contiene de dos a siete letras que son las siglas para la instrucción. Por ejemplo, MUV son las siglas para la instruccion move.
Aparte del mnemonico, muchas instrucciones requiere que se especifiquen uno o dos operandos. (por ejemplo, una instruccion ADD debe de saber cuales son los dos términos a sumar.

A3.5.4 Campo del Operando

El campo del operando le dice al microprocesador donde hallar los datos que se operaran. Este campo es obligatorio en algunas instrucciones y prohibido en otras.Si se presenta, el campo del operando contiene uno o dos operandos, separados de el mnemonico

por el menos un espacio. Si se requieren dos operandos, se debe de poner una coma (,) entre ellos.

En operaciones con dos operandos, el primero es el **operando destino** y el segundo es el **operando fuente.** El operando destino especifica el valor que el microprocesador deberá sumar a, restar de, comparar con, o almacenar en el operando destino. Por ejemplo, en la instruccion move:

```
MOV CX,DX
```

CX,DX indican mover el contenido del operando fuente en el registro DX a el operando destino en el registro CX.

El operando fuente nunca se altera por la operación mientras que el operando destino casi siempre es alterado por la operación.

A3.5.5 Campo de Comentario

Como la instrucción REM en basic este campo es opcional, permite describir instrucciones en el programa fuente, para hacer el programa más facil de comprender. Se debe de preceder el comentario con un punto y coma(;).

Se pueden colocar comentarios en una linea, para describir un bloque entero de instrucciones. Para hacer ésto, se escribe un punto y coma (;) al inicio de la linea

A3.6 DIRECTIVAS DEL ASSEMBLER

Las directivas son comandos para el assembler, y no para el microprocesador. Las directivas pueden ser utilizadas para establecer segmentos y procedimientos, definir simbolos, reservar memoria para almacenamiento temporal, y para realizar otra variedad de tareas "domesticas" importantes. A diferencia de las instrucciones en lenguaje de ensamble, la mayoria de directivas no generan codigo objeto.

Las directivas pueden tener hasta cuatro campos. Los cuales son:

```
[Nombre]  Directiva  [Operando]  [;Comentario]
```

Como los corchetes indican, solo el campo de la directiva se requiere siempre. El nombre es obligatorio con algunas directivas, prohibido con otras y opcional con el resto. Lo mismo se aplica con el operando. El campo del comentario es siempre opcional. La directiva se puede colocar el cualquier parte de la linea pero los compos deben de estar separados por al menos un espacio.

El Macro Assembler Posee alrededor de 60 directivas diferentes. En esta seccion se discutiran las mas comunes. Las directivas avanzadas se dejaran para secciones posteriores. La tabla A5. divide las directivas en tres grupos:

De datos, De listado, Y de Modo.

Tabla A3. Directivas del assembler

Tipo	Directivas		
De Datos	ASSUME	ENDP	INCLUDE
	COMMENT	ENDS	ORG
	DB	EQU	PROC
	DW	=	PUBLIC
	DD	EVEN	SEGMENT
	END	EXTRN	
De Listado	PAGE	SUBTTL	TITLE
De Modo	.286C	.8086	

Nota: No se debe de intentar memorizar el materiar de esta sección, unicamete se debe de leer y volver a el cuando se necesiten algunos detalles.

A3.6.1 Directivas de datos

Las directivas de datos se pueden dividir en cinco grupos, como se muestra en la tabla 2.2

Tabla 2.2 Directivas de datos

Directiva	Funcion
Definicion de Simbolo	
EQU	Formato: nombre EQU texto o nombre EQU expresion numerica Asigna texto o valor de una expresion numerica a nombre permanentemente.
=	Formato: nombre = expresion numerica Asigna el valor de la expresion numerica al nombre, pero puede ser reasignado.
Definicion de Dato	
DB	Formato: [nombre] DB expresion[,...] Define una variable o inicializa la capacidad de almacenamiento. DB coloca uno o mas bytes.
DW	Formato: [nombre] DW expresion[,...] Similar a DB, pero coloca palabras de dos bytes
DD	Formato: [nombre] DD expresion[,...] Coloca palabras dobles, de cuatro bytes

Tabla 2.2 continuación

	De Referencia Externa	
PUBLIC	Formato: PUBLIC simbolo[,...]	Hace que el(los) simbolo(s) definidos esten disponibles para otros modulos que posteriormente pueden ser encadenados.
EXTRN	Formato: EXTRN nombre:tipo[,...]	Especifica simbolos definidos en otros modulos
INCLUDE	Formato: INCLUDE nombre-archivo	Concatena el contenido del archivo fuente especificado con el archivo actual.
	De Especifiacion de Procedmiento/segmento	
SEGMENT	Formato: nombre de segmento SEGMENT .. .. nombre de segmento ENDS	Define los limites de un segmento, cada definicion de segmento debe de terminar con ENDS
ASSUME	Formato: ASSUME reg-seg:nombre-seg[,...] o ASSUME reg-seg:NOTHING[,...]	Le da nombre a los registors segmento (CS, DS, ES, o SS). ASSUME NOTHING càncela la instruccion ASSUME previamente especificada.
PROC	Formato: nombre PROC [NEAR] o nombre PROC [FAR] .. .. RET nombre ENDP	Asigna un nombre a una secuencia de instrucciones. Cualquier definicion PROC debe de terminar con ENDP

TABLA 2.2 DIRECTIVAS DE DATOS

ASSEMBLY CONTROL	
END	Format:END [entrada-nombre del punto]
	Marca el final de un programa fuente.
EVEN	Formato: par
	Fuerza un conteo de localización a un limite par.
ORG	Formato: ORG expresión.
	Cuenta el arreglo de localizaciones al valor de la expresión.

A3.6.2 DEFINICIUN DE DIRECTIVAS

La definición de directivas asigna un nombre simbólico a una expresión. Esta puede ser una constante de 16 bits, una referencia a una dirección, cualquier nombre simbólico, un identificador de segmento (prefijo) y un operando o un nombre de instrucción. Después de asignar el nombre, se puede utilizar en cualquier parte la expresión normalmente usada.
Las directivas EQU(igualar) y =(signo igual) son similares pero:

1. Se pueden redefinir símbolos definidos con =, mientras que con EQU son permanentes.
2. EQU puede ser utilizada para cualquier texto o expresión numérica, mientras que = sólo puede ser utilizado para expresiones numéricas.

EQ es conveniente para asignar nombres simples a números, combinación de direcciones complejas, y otras cosas que usted no necesite entrar en todo el programa.
Algunos ejemplos son:

```
K       EQU          1024 ; Nombre de una constante
TABLE   EQU  DS:[BP][SI] ; Nombrar una combinación de direcciones
SPEED   EQU  RATE        ; Asignar un nombre alternativo
COUNT   EQU  CX          ; Asignar un nombre de registro
```

Entrando en una expresión, el operando puede también incluir algunas cosas simples, en las cuales el Assembler efectúa el cálculo. Por ejemplo:

```
DBL-SPEED    EQU  2*SPEED
MINS-PER-DAY EQU  60*24
```

Algunos ejemplos del directivo "=" son:

```
Const=56     ; Esto es lo mismo que CONST EQU 56, pero ahora const
Const=57       puede estar redefinido, o puede referirse a la
Const=const+1  definición previa.
```

A3.6.3 DEFINICION DE DIRECTIVAS DE DATOS

Muchos programas usan localizaciones de memoria para guarder variables, llamados Item de datos que pueden ser modificados cuando se necesite.
Las directivas para asignar espacio para las variables son DB(define un byte), DW(define palabras) y DD(define doble palabra).

Estas directivas difieren en la cantidad de memoria que ellos asignan.
DB asigna 8 bit(1 byte), DW asigna 2 byte(una palabra) y DD asigna 4 byte(dos palabras).
Cuando se define una variable, se puede colocar cualquier valor inicial o simplemente reservar el espacio y luego insertar el valor próximo.

El formato general para la definición de datos es:

[nombre] DB expresión [,...]
[nombre] DW expresión [,...]
[nombre] DD expresión [,...]

donde [,...] indica que se pueden especificar muchas expresiones.

Una expresión puede tomar muchas formas, dependiendo de como se necesita definir la variable. Por ejemplo éste puede ser una constante.

Las siguientes instrucciones muestran los valores máximos y mínimos aceptables para variables decimales de un tamaño de un byte y una palabra.

```
BU-MAX  DB      255(máximo byte constante, sin signo)
BS-MAX  DB      127(máximo byte constante, con signo)
BS-MIN  DB      -128(mínimo byte constante, con signo)
WU-MAX  DW      65535(máxima palabra constante, sin signo)
WS-MAX  DW      32767(máxima palabra constante, con signo)
WS-MIN  DW      -32767(mínima palabra constante, con signo)
```

Para definir una variable sin dar un valor inicial, se colocará un signo de interrogación(?) en la expresión del campo. Por ejemplo, la siguiente instrucción reserva espacio en memoria para un byte o palabra, pero no almacena nada dentro.

```
HIGH-TEMP DB ?
AVG-WEIGHT DW ?
```

Note que simplemente reserva espacio para **HIGH-TEMP** y **AVG-WEIGHT**; sin inicializar nada. Se deberá de tener cuidado en no asumir que ellos contienen cero o cualquier otro valor específico. Además se podrá también reservar espacio para entrar tablas: en el siguiente ejemplo:

```
MONTHLY-SALES DW  12 DUP(?)
```

Se reserva 12 palabras en memoria. Esto es equivalente a la instrucción en BASIC

```
MONTHLYI SALES %(12)
```

El directivo DB(definir byte) también puede aceptar string de caracteres como una expresión. Así se podrá almacenar mensajes de errores, títulos de tablas y otros textos en memoria.

En el siguiente ejemplo los mensajes deberán de ser encerrados con con comillas simples.Se puede utilizar directivos para arreglo de tablas en memoria. Para hacerlo simplemente se lista los elementos de la tabla separado por comas. La siguiente secuencia arreglo dos tablas de 12 elementos, una para bytes y otra para palabras:

```
B-TABLE  DB     0,0,0,0,8-13 (tabla de byte)
         DB     -100,0,55,63,63,63
```

```
W-TABLE  DW     1025,567,-30222,0,90,-129 (tabla de palabra)
         DW     17,645,26534,367,78,-17
```

En éste ejemplo se han colocado los datos en dos filas de 6 elementos cada una, pero se pueden colocar en una sola línea, como máximo 132 caracteres.

Note en el ejemplo que los primeros cuatro elementos y los tres últimos de la tabla B tienen el mismo valor(0 y 63 respectivamente).
El assembler tiene un operador **DUP (duplicate)** que permite colocar aquellos datos que se repiten sólo una vez; por lo tanto en nuestro ejemplo de la tabla B se colocará:

```
B-TABLE  DB  4 DUP(0),8,-13,-100,0,53,3 DUP(63)
```

con lo cual se minimizaría la instrucción.

Las variables son también muy poderosas para guardar direcciones de memoria, que pueden ser referenciadas por instrucciones en su programa. Como se sabe, cada dirección tiene dos componentes:un número de segmento así como en la instrucción referida, ésta se deberá de especificar solo en el offset. Además puesto que el offset tiene una longitud de 16 bits; se puede utilizar el directivo DW para almacenar dicho título. Por ejemplo:

```
HERE-NEAR   DW  HERE
```

Se asignan los 16 bits de offset del título **HERE** al nombre HERE-NEAR.
La variable que guarda el offset es conocida como puntero. Además existe una instrucción para asignar 16 bit a la dirección y 16 bit al offset. éste directivo es DD(definir doble palabra). Por ejemplo:

```
LABEL- VECTOR  DD  FAR-HERE
```

En éste ejemplo se asignan 16 bit al offset y 16 bit al número de segmento del título FAR-HERE a la variable de 32 bit LABEL-VECTOR; ésta variable que guarda ambos términos de la dirección se conoce como vector.

A3.6.4 SEGMENTO /ESPECIFICACION DE DIRECTIVOS PARA PROCEDIMIENTO

Los directivos **SEGMENT Y ENDS** dividen el programa fuente en dos segmentos. Como se mencionó anteriormente, un programa puede tener cuatro clases de segmentos: datos, códigos, extra y stack. Como muestra la tabla 2.2 el SEGMENTO puede tener 3 operandos: tipo alineado, tipo combinado y clases.

El tipo alineado especifica qué parte del límite del segmento será inicializado cuando es almacenado en memoria.
Se puede comenzar en cualquier número par de dirección(palabra) o en una dirección divisible por 16 o 256 (página).

El tipo combinado especifica qué tipo de segmento será combinado con otro del mismo nombre.
El código, datos y segmento extra pueden ser unidos (PUBLIC) o separados (PRIVATE).
El segmento STACK deberá de ser del tipo STACK.
La clase define el orden en la cual los segmentos serán cargados. Los segmentos deberán tener la misma clase de nombre encerrado en comillas, de otra manera aquellos con nombres diferentes serán almacenados en el orden en el cual el programa los lea.
El tamaño
El manual del lenguaje Assembler describe estas opciones en detalle, pero en general se deberán de conservar las siguientes convenciones:

Para segmento de DATOS, utilice SEGMENT para public"DATA"
Para segmento CODIGO, utilice SEGMENT para public"CODE"
Para segmento EXTRA, utilice SEGMENT para public"EXTRA"
Para segmento STACK, utilice SEGMENT STACK'STACK"

Por ejemplo, un dato segmento dato puede tener area de la siguiente manera:

```
DSEG      SEGMENT       PARA PUBLIC'DATA'
 A          DB            ?
 B          DB            ?
SQUARES     DB            1,4,9,16,25,36,49,64
DSEG        ENDS
```

un código segmento se observaría de la siguiente manera:

```
C SEG       SEGMENT PARA PUBLIC'CODE'
              . .
              . .
              MOV   AX,BX
              MOV   CL,DH
              MOV   DI,CX
              . .
C SEG       ENDS
```

Las palabras SEGMENTS Y ENDS simplemente marcan el inicio y final de un segmento; no le indican al assembler que parte del segmento está siendo definido.
El directivo ASSUME indica cual segmento del registro (CS,DS,ES,o SS) pertenece a ese segmento.
ASSUME tiene el siguiente formato:

ASSUME seg-reg:seg-name[,...]
Donde seg-reg es cualquier nombre DS,CS,SS o ES y seg-name es el nombre que proporcia el directivo SEGMENT DS es la marca inicial del segmento de datos, CS es el código de segmento, SS es el segmento del STACK y ES el segmento extra.
ASSUME le ayuda al assembler a trasladar etiquetas dentro de direcciones, indicando que segmento del registro se pretende usar

para ubicar la etiqueta en la dirección. Por ejemplo:

```
ASSUME  DS: DSEG
```

Le indica al assembler, DSEG es mi segmento de dato, siempre que encuentre una mención de una etiqueta en DSEG al microprocesador se le indicará obtener la etiqueta del número de segmento de DS.
El ASSUME deberá de ubicarse inmediatamente después del segmento de código SEGMENT. Por ejemplo:

```
CSEG    Segment para public'code'
        ASSUME  CS:CSEG,DS:DSEG
        MOV     AX, DSEG, hacer una señal DS para DSEG
        MOV     DS, AX
          .
          .
        MOV     AX,BX
        MOV     CL,DH
        MOV     CX,DI
CSEG    ENDS
```

De nuevo, note que se ha cargado explícitamente la dirección del segmento de dato dentro de DS. El ASSUME no se preocupa de ésto.
Los directivos PROC y ENDP marcan el inicio y final de un procedimiento. Un procedimiento es un bloque de instrucciones que pueden ser ejecutados en varios lugares de un programa. Siempre que su programa llame un procedimiento el 80286 lo ejecuta, luego retorna al lugar donde la llamada fué efectuada.

Cualquier procedimiento deberá de comenzar con un PROC y terminar con un ENDP.

Un procedimiento deberá de tener por lo menos uno de los dos atributos de distancia : NEAR ó FAR, como? especificación del operando que sigue al PROC.
Un procedimiento NEAR sólo puede ser llamado de dentro del código segmento en el cual fué definido.
Por ejemplo:

```
CSEG    SEGMENT PARA PUBLIC'CODE
        ASSUME CS:CSEG
CALLEE  PROC
        .
        CALL CALLEE (llamada a un procedimiento)
        .
        .
        .
        RET
CALLEE  PROC NEAR (inicio del procedimiento)
        .
        .
        RET
CALLEE  ENDP
CSEG    ENDS
```

Aquí la llamada al procedimiento (CALLEE) está ocurriendo en el mismo segmento como la instrucción CALL.

Sin embargo, los dos procedimientos pueden también estar en diferentes módulos de programa(por ej: diferentes archivos). La comunicación entre módulos involucra los directivos EXTRN y PUBLIC: los cuales serán discutidos en la próxima sección.
Un procedimiento FAR puede ser llamado de cualquier código de segmento, por ejemplo:

```
 CSEG    SEGMENT PARA PUBLIC 'CODE'
         ASSUME  CS:CSEG
 CALLER PROC
         .
         .
         CALL CALLE (llamar el procedimiento)
         .
         .
         RET
CALLER   ENDP
CSEG     ENDS

CSEG1    SEGMENT PARA PUBLIC 'CODE'
         ASSUME  CS:CSEG1
CALLER   PROC  FAR  (procedimiento de llamada)
         .
         .
         RET
CALLER   ENDP
CSEG     ENDS
```

Cuando el 80286 llama un procedimiento, éste pone una dirección de retorno dentro del stack. Si el procedimiento tiene un atribubto NEAR, el 286 pone solo un offset- El contenido del pointer de instrucciones (IP)-en el stack. Si el procedimiento tiene un atributo FAR, el 286 coloca dos cosas en el stack: un número de segmento- El contenido del registro código del segmento- y un offset del IP.
Cuando se ensambla un programa, el assembler traslada cada instrucción RET dentro de código máquina lo cual le ordena al 80286 que muchos retornos de direcciones de palabras serán recobbrados del STACK.

Un RET en un procedimiento NEAR remueve dos palabras(IP y CS).
Algunas reglas que ayudarán a hacer sus procedimientos NEAR ó FAR son:
1.- Su procedimiento principal será un FAR (ésto es cierto para programas EXE).
2.- Si ud. siempre da al código de segmento el mismo nombre (eje. CSEG), haga todos los procedimientos NEAR excepto en el principal.
3.- Si ud. está usando un procedimiento que ha sido creado por alguien, usted deberá de saber si es un NEAR ó FAR.

A3.6.6 DIRECTIVAS DE REFERENCIA EXTERNA

Estas directivas le dan parte de la información entre módulos que eventualmente linkió para formar un programa.
Estos son caminos que auxilian a linkiar (LINK) en la construcción final de un programa.

Las directivas PUBLICOS hacen que uno o varios símbolos sean utilizados en otros módulos que eventualmente están linkiados en A-ha.
Un directivo PUBLICO puede listar varios nombres, individualmente (títulos proc) y nombres definidos por los directivos EQ o "=".

EXTRN es el reverso de PUBLIC, éste le ordena al linker "Yo necesito usar éste item. Desconozco dónde está ó en qué módulo está, pero necesito referirme a éste".

EXTRN tiene el formato general:
EXTRN nombre: type [,...]

Donde el nombre es el símbolo definido en algún otro módulo assembly (y declarada pública) y tipo puede ser alguno de los siguientes:
Si nombre es un símbolo en el segmento de dato ó el segmento extra, entonces el tipo puede ser un BYTE, PALABRA ó DOBLE PALABRA.
Si nombre es un nombre de procedimiento, entonces tipo puede ser NEAR ó FAR.
Si nombre es una constante definida por los directivos EQ ó "=", entonces tipo será ABS.

PUBLIC y EXTRN: son a menudo usados para dividir procedimientos. Por ejemplo, para correr un kprocedimiento llamado SORT de un programa principal, el módulo en el cual SORT ha sido definido deberá de incluir PUBLIC SORT y el módulo principal (el primero que se refirió con SORT) deberá contener EXTRNSORT: NEAR ó EXTRN SORT: FAR.

La directiva INCLUDE encadena un archivo completo de instrucciones fuente dentro del archivo en fuso y lo ensambla a la vez.
Por ejemplo: INCLUDE C:\MIMERFIL.ASM lee el contenido del archivo MIMERFIL.ARS en drive C dentro del archivo fuente, reemplazando la instrucción INCLUDE. Se puede también utilizar INCLUDE para leer macros dentro de un programa.

A3.6.7 DIRECTIVAS DE CONTROL

El assembler reconoce muchos directivos de control, pero solo END,EVEN,ORG son frecuentemente utilizados.
La directiva END marca el final de un programa, así como también le indica al assembler donde parar el ensamble. Por lo tanto se deberá de incluir END en cualquier programa fuente.
El formato general es:

END [EXPR] PUNTO LABEL

donde entry-point label identifica el lugar donde el DOS inicia la ejecución del programa. Por ejemplo:

```
END   MY-PROG
```

Marca el final del programa MY-PROG.

NOTA IMPORTANTE:

Si su programa consiste de muchos módulos deberá de etiquetar el módulo principal con END; y omitir el END en el segundo módulo. Por ejemplo:
Si su programa principal llama una sub-rutina que está separada por módulos, cada módulo de la sub-rutina no deberá de tener la etiqueta END.

La directiva EVEN puede correr programas más rápido, ésto es porque teniendo un bus de 16 bit, el 80286 puede transferir datos de 16 bit en una sola operación. Sin embargo, se toma bastante tiempo en transferir datos inicializando la transferencia de datos en una dirección impar que si se tomará datos que inicialicen en un número de dirección par.

Por lo tanto, en aplicaciones con tiempo crítico es ventajoso almacenar datos en número de direcciones par.
Para guardar datos convenientemente alineados, ordenelos primero con doble palabra (DDs), segundo con palabras(DWs) y después con bytes(DBs).

Si el segmento de datos contiene sólo el byte del pseudo-operador se deberá de colocar un EVEN delante de cada uno escepto en el primero. Ejemplo:

```
DSEG    SEGMENT PARA PUBLIC'DATA'
        HOWRS DB?
        EVEN
        MESSAGE DB ' PRESS ANY KEY TO CONTINUE'
DSEG    ENDS
```

EVENS solamente hará su trabajo si es necesario. Esto es, si el contador de localizaciones está listo apuntando a un offset par, even no hará nada. Por otro lado si el contador de localizaciones está apuntando a un offset impar, el assembler reemplazará EVEN con UNA byte, haciendo la próxima localización par. Por ejemplo: si el contador de localizaciones es fijado al offset 129H. EVEN hará que el assembler almacene la próxima unidad en 12AH.

El pseudo-operador ORG(origen) alterá el contenido del contador de localizaciones, para hacer que el assembler almacene datos o instrucciones en lugares diferentes donde normalmente los almacene. ORG es a menudo usado con tipos de programas con extensión (UNA-comandos). Por ejemplo:

```
ORG 100H
```

Le indica al assembler que inicialize el almacenamiento del programa 256 bytes más adelante de la localización actual.

A3.6.8 DIRECTIVOS DE LISTADO

La tabla 2.3 resume directivas de listado

TABLA 2.3 DIRECTIVAS DE LISTADO

Directiva	Función
PAGE	Formato: PAGE [lineas][,columnas] Fija la longitud y ancho de la página.
TITLE	Formato: TITLE text Especifica el título que será listado en la segunda línea de cada página.
SUBTTL	Formato: SUBTTL text Especifica el subtítulo que será listado en la tercera línea de cada página.

PAGE tiene el siguiente formato:

PAGE [lineas][,columnas]

Donde lineas y columnas fija la longitud y ancho de las páginas durante en ensamble. La línea puede tener un rango de 10 a 255,columnas un rango de 60 a 132 caracteres.

Los valores por default, son 57líneas y 80 caracteres para un tamaño de papel de 8½ por 11 pulgadas. Ud. puede por ejemplo, imprimir un tamaño legal de papel con una instrucción como PAGE ,, 66 , para cambiar el tamaño standar de la salida impresa usando PAGE 132.

El assembler imprime un caracter numérico y un número de página en la parte superior de cada página, con un guión en ellos.

El directivo TITLE produce un título justificado a la izquierda en la segunda línea de cada página.
El directivo SUBTTL produce un subtítulo centrado en la tercera línea de la próxima página. Este usualmente describe el contenido de la página. Por ejemplo: el inicio de un listado podría tener el siguiente encabezado:

```
TITLE    LUNAR ALT - GALAXY CENSUS PROGRAM
SUBTTL     VERSION DATA SEGMENT
```

al final del primer segmento de dato ud. podría cambiar el

subtítulo como sigue:

```
SUBTTL PLUTONION DATA SEGMENT
PAGE
```

Los títulos y subtítulos pueden tener 60 caracteres de longitud.

A3.6.9 DIRECTIVAS DE MODO

El macroassembler ha sido diseñado para trabajar con los microprocesadores de intel 8086 y 8088 así como también con el 80186. El 80286 incluye todas las instrucciones , más algunas adicionales propias . Los directivos de modo le dicen al assembler que juego de instrucciones del microprocesador reconocerá.
La primera de ellas:
Este directivo indica aceptar instrucciones del 80286, así como también instrucciones del 8086/88. Esta instrucción es necesaria si el programa incluye instrucciones que son únicas del 80286.
El segundo modo es:
Esta directiva le dice al assembler aceptar instrucciones en el modo protegido así como también instrucciones en el modo real; es decir que acepta instrucciones del 80286 y del 8086.
El último directivo de modo 8086 es el valor por default. Este directivo le dice al assembler solo aceptar instrucciones del 8086/88 enviando errores si encuentra instrucciones del 80286.
Se puede utilizar directivos 8086, para reconocer instrucciones ilegales en un programa que ha sido diseñado para correr en una PC IBM el cual contiene instrucciones del 8088. si se utilizán directivos del .286C, 286P, ó 286 colocandolas al inicio de un programa aparecerán listadas a la derecha.

A3.6.10 OPERADORES

Un operador es un modificador usado en el campo del operando en el lenguaje assembler o instrucción directiva. Este esta formado por cinco clases de operandos: Aritméticos, lógicos, relacionales.retorno de valores y atributos. la tabal 2-4 resume estos operandos:

TABAL 2-2 OPERANDOS

OPERADOR	FUNCION
ARITMETICO	
+	formato: valor1 + valor2 suma el valor1 y el valor2
-	formato: valor1 - valor2 resta el valor2 del valor1
*	formato: valor1 * valor2 multiplica valor1 por valor2
/	formato: valor1 / valor2

	divide el valor1 por el valor2 y retorna el cociente
MOD	formato: valor1 MOD valor2 divide el valor1 por el valor2 y retorna el recíduo
SHL	formato: valor1 SHL expresion, desplazamiento a la izquierda del valor del bit de la expresion
SHR	formato: valor SHR expresion desplaza valor a la derecha, el valor del bit de la expresion
AND	formato: valor1 AND valor2 evaluación lógica AND de valor1 y valor2
OR	formato: valor1 OR valor2 tomar el valor lógico de comparar el valor1 con el valor2
XOR	formato: valor1 XOR valor2 evaluación or exclusivo del valor1 y del valor2
NOT	formato: NOT valor cambia el estado de cada bit en valor, es decir tomar el complemento

RELACIONAL

EQ	formato: operando1 EQ operando2 verdadero si los dos operandos son iguales
NE	formato: operando1 NE operando2 verdadero si los dos operandos no son iguales
LT	formato: operando1 LT opreando2 verdadero si operando1 es menor que el operando2,
GT	formato: operando1 GT operando2 verdadero si operando1 es mayor que operando2
LE	formato: operando1 LE operando2 cierto si operando1 es menor o igual que operando2
GE	formato: operando1 GE operando2 verdadero si operando1 es mayor o igual que operando2

RETORNO DE VALOR

$	formato: $ retorna el valor actual del contador de localización

SEG	formato: SEG variable retorna el valor del segmento de la variable o segmento
OFFSET	formato: OFFSET variable o OFFSET etiqueta. retorna el valor offset de la variable o etiqueta.
LENGTH	formato: LENGTH variable retorna la longitud en unidades (bytes o palabras) para cualquier variable definida usando DUP
TYPE	formato: TYPE variable o TYPE etiqueta para variables, TYPE retorna 1 (byte), 2 (palabra), o 4(doble palabra). Para etiqueta, retorna 1(NEAR), o 2 (FAR)
SIZE	formato: SIZE variable retorna el producto de LENGTH y TYPE.
ATRIBUTOS	
PTR	formato: tipo PTR expresion recorre el tipo (byte o palabra) o distancia (NEAR o FAR) de un operando de dirección de memoria
DS: ES: SS: CS:	formato: seg - reg: adrr - expr ó seg - reg:etiqueta ó seg - reg: variable recorre los segmentos de atributos de una etiqueta, variable dirección
SHORT	formato: JMP SHORT etiqueta llama al assembler a que el salto a la etiqueta destino no sea mayor que 127 bytes para la siguiente instrucción
THIS	formato: THIS atributo ó THIS tipo Crea un operando en la dirección de memoria para cualquier atributo de distancia(NEAR o FAR) o cualquier atributo tipo (byte o palabra) en un offset igual al valor actual del contador de localizaciones y un atributo de segmento del segmento concíderado.
HIGH	formato: HIGH valor ó HIGH expresion retorna el byte más significativo de un valor numerico de 16 bits ó de una dirección.
LOW	formato: LOW valor ó LOW expresion retorna el byte menos significativo de un valor número o dirección.

A3.6.11 OPERADORES ARITMETICOS

Los operadores aritméticos combinan operadores números para obtener resultados numéricos. Los operadores aritméticos más comunes son: la suma (+), la resta (-), el producto (*) y la division (/).
La division retorna el cociente producido de utilizar un operador de division. por ejemplo:

```
PI_QUOT     EQU     31415/10000
```

retorna el valor de 3.
El operador MOD retorna el residuo de un operador de division. por ejemplo:

```
PI_REM      EQU     31416 MOD 10000
```

define una constante llamada PI_REM que tendrá el valor 1416.
Finalmente SHL y SHR desplazán un operando número a la izquierda o a la derecha.

```
START   EQU   THIS  FAR
        MOV   CX,100
```

Añade a la instrucción MOV un atributo FAR, el cual hara que salte directamente a esta etiqueta desde otro segmento.
Los operadores HIGH y LOW retornan el byte de mayor o menor orden de una expresión de 16 bit. Por ejemplo, si se define una constante como:

```
CONST   EQU   0ABCDH
```

La instrucción

```
MOV  AH,HIGH CONST
```

Cargará el valor 0ABH dentro del registro AX.

A5.5 ARCHIVOS COM

El DOS trabaja con dos tipos diferentes de segmentos de archivos de programas de lenguaje assembly. El primer segmento es el archivo EXE. (ejecutable); el otro el archivo COM.(comandos). En general, los programadores utilizán la técnica EXE para diseñar programas de mayor capacidad de 64k de capacidad, y utilizán archivos COM para programas de menor capacidad, ya que los archivos COM estan limitados a 64k.
En el disco se podrán observar los dos tipos de archivos: en los archivos de DOS por ejemplo se encontrarán los archivos de comandos PRINT, y FORMAT; en cambio SORT, FIND, y LINK son archivos EXE.

A5.6 REGLAS PARA CREAR UN ARCHIVO COM

La creación de archivos com requieren diferentes reglas que la

empleadas para los archivos EXE. El manual del macro Assembler ofrece mayor detalles de estas reglas, por lo tanto solo se mencionarán algunas importantes:

1. Omita todos lo segmentos de datos,stack y extra respectivamente
2. Defina solo un código de segmento, y ponga todos los datos en el (asi como también las instrucciones)
3. En la directiva ASSUME ,señale todos los cuatro registros de segmento al segmento de código.
4. Inicialice el programa con el operador:
   ORG 100H
   El cual llama al computador a cargar el programa 256 byte despues del comienzo del código de segmento. Esto es necesario porque en un programa COM, los primeros 256 bytes son ocupados por alguna llamada de un segmento de programa prefijado (PSP) el cual es dicutido con mayor detalle en el Manual de referencia del programador de Microsoft.
5. Ponga todos los datos adelante de las instrucciones del programa .Además si el dato esta incluido, inicialice el programa con una instrucción de salto JUMP que salte a la primera instrucción. Por ejemplo, para preparar un programa que mueve datos dentro de una tabla como un archivo COM, se inicializará con:

```
ENTRY:    JMP   OUR_PROG        : salto al area de datos
SOURCE    DB    10,20,30,40
DEST      DB    4 DUP(?)
OUR_PROG  PROC  NEAR            : siguen las instrucciones del
                                  programa
```

6. defina todos los procedimientos como NEAR.
7. Finalice todas las etiquetas con END: por ejemplo:
   END ENTRY

AS.7 CREACION DE UN ARCHIVO COM

Para crear un archivo COM, comience primero creando el archivo EXE (esto es, entre el programa, ensamblelo, y luego encadenelo con LINK).
Al utilizar el LINK la pantalla mostrará el siguiente mensaje:

```
Warning: No STACK segment.
```

No se preocupe, esto es una precaución no un mensaje de error.Esto es debido a que usted esta utilizando un archivo COM el cual no permite un segmento de stack.
Con un archivo EXE no se observara este mensaje. Así que para crear el archivo COM se nesecitará un paso más: convertir el archivo EXE a COM atravéz del programa del DOS EXE2BIN.
Asumiendo que su Assembler se encuentra en el drive A y que su disco de datos en el drive B, deberá de correr el programa EXE2BIN de la siguiente manera:

```
a: exe2bin proqname proqname.com
```

A5.8 OBSERVANDO LOS DATOS DE UN PROGRAMA COM

Como se mencionó en el capitulo dos, EL DEBUG, para observar los datos de un archivo EXE se utilizó el romando d, el cual desplegaba en pantalla el inicio del segmento de datos atravez de la instrucción d ds:0. En oposición a los archivos COM. los datos son introducidos en el segmento de código. Por lo tanto para desplegarlos, no se puede utilizar la instrucción cs:0, debido a que los datos no se inicializán en este segmento.
En su lugar se inicializá en el último prefijo del segmento del programa (PSP) y la instruccion de salto JMP al inicio del programa.
El PSP tiene una longitud de 256 bytes (o 100H, por esto es la razón de la instrucción ORG 100H), y la instrucción de salto JMP posee tres bytes de longitud. por esta razón, los datos inicializán 103H bytes pasados del inicio del segmento de código. por lo tanto la instrucción correcta es d cs:103.

A5.9 DIRECTIVAS AVANZADAS

Esta sección describirá las directivas más utilizadas por los programadores. la tabla 2-9 lista esta directivas en tres grupos: Directivas de datos, condicional y de listado .

Tabla 2-9 Directivas Avanzadas

Tipo	Directiva		
Dato	GROUP	LABEL	
Condicional	ELSE	IFNDEF	IF1
	ENDIF	IFDIF	IF2
	IF	IFE	
	IFDEF	IFIDN	
De listado	.LREF	%OUT	.XLIST
	.LFCOND	.SFCOND	
	.LIST	.XREF	

A5.9.1 DIRECTIVAS DE DATOS

La tabla 2-10 describe las directivas avansadas de datos.

Tabla 2-10 Directivas Avanzada de datos

Directiva	función
GROUP	formato: name GROUP seg-name[,...] reune los segmentos especificos sobre un nombre los cuales reciden entre un segmento fisico de 64k-byte.
LABEL	formato: name LABEL type define los atributos de nombre.

La directiva GROUP asigna un nombre a una coleccion de segmento. Esta hace que todos los elementos en este segmento sean direccionables con el arreglo del registro segmento . Usted puede, por ejemplo, usar GROUP para crear un archivo COM que tiene separado el segmento de codigo y el segmento de datos, en lugar de un reamento, con un salto sobre los datos.
Como otra aplicacion, suponga que usted necesita usar dos procedimiento que estan en archivos de disco diferente . El problema es que ambos procedimientos tienen un NEAR, pero ellos estan en un codigo de segmento que tienen diferente nombre en su segmento de codigo. Especificamente, PROC1 esta en CODESEG y los datos en DATASEG, mientras PROC2, esta en CODE_SEG y sus datos en DATA SEG, sin embargo el procedimiento NEAR solo puede ser llamado entre los mismos segmentos; por lo tanto la solución a este problema es utilizar un directivo de GROUP .
El directivo LABEL define el segmento,offset, y el tipo de atributo del nombre. Podrá utilizar LABEL para dar a una instrucción un atributo FAR, para poderla transferir de un segmento a otro. Por ejemplo:

```
NEAR    LABEL   FAR
        MOV     DX,0
```

A5.9.2 DIRECTIVAS DE CONDICION

Las directivas de condición le dicen al assembler a que salte a un grupo de instrucciones dependiendo de una condición cierta o falsa. Estas directivas estan formadas por dos clausulas dentro de un prograrma IF-ENDIF. En cada caso si la condición a evaluar es cierta, el codigo dentro del segmento if es ensamblado, de lo contrario, si la condición es falsa el assembler salta el segmento hacia la siguiente instrucción ENDIF.
Se puede agrupar las ocho directivas IF dentro de cuatro pares como sigue:
-IFE es cierta si expresion is 0; IF es cierta si la expresión no es cero
-IF1 es cierta cuando el assembler ejecuta el paso uno; IF2 es cierta cuando se ejecuta el paso 2
IFDEF es cierta si el simbolo es definido ó ha sido declarado externo por una directiva EXTRN; por otro lado IFNDEF es falsa.
-IFIDN es cierta si los string en los argumentos uno y dos son identicos; por otro lado es falso si ellos son diferentes.

Tabla 2-11 Directivas de condición

Directiva	Función
IFE	formato: IFE expresión cierto si la expresión es 0
IF	formato: IF expresión cierto si la expresión es 0
IF1	formato: IF1

cierto si es ensamblado en la ejecución del paso1

IF2 formato: IF2
cierto si es ensamblado en la ejecución del paso2

IFDEF formato: IFDEF simbolo
cierto si simbolo ha sido definido ó declarado externo por un directivo EXTRN.

IFNDEF formato: IFNDEF simbolo
cierto si simbolo es indefinida o no declarada externa por una directiva EXTRN.

IFIDN formato: IFIDN <string1>, <string2>
cierto si string1 y string2 son iguales. ambos corchetes son necesarios.

IFIDF formato: IFIDF <string1>,<string2>
cierto si string1 y string2 son diferentes ambos corchetes son necesarios.

Por ejemplo, para incluir una rutina de diagnóstico en una corrida de prueba, se deberá de encerrar con un IFE y ENDIF, la constante FOR_TEST_ONLY. En el ensamblado, el asembler chequeará el valor del FOR_TEST_ONLY, si es 0, el diagnóstico sera incluido (ensamblado dentro) en el rpograma.Por el contrario se FOR_TEST_ONLY tiene cualquier otro valor, el diagnóstico será omitido. El proaram podría tener el siguiente aspecto:

```
            IFE     FOR_TEST_ONLY
DIAG1:      . .                 ( test de instrucciones de diagnóstico )
            . .
            ENDIF
```

Las instrucciones entre DIAG1 y ENDIF serán ensambladas solo si una instrucción como

```
FOR_TEST_ONLY   = 0
```

Por otro lado, si una instrucción como

```
FOR_TEST_ONLY   = 1
```

Hará que el assembler salte las instrucciones entre DIAG1 y ENDIF.

A5.9.3 LA OPCION ELSE

Se podrá incluir también el termino ELSE para generar una alternativa para el caso falso. El formato general es:

```
IFxx    [ argumento ]
[ ELSE ]
ENDIF
```

La opción ELSE le permite por ejemplo, producir dos versiones de un proarama: una que despliegue el prompt y mensaje en ingles y la otra que despligue esto en español. Para hacerlo, declare una

constante llamada LANGUAGE para seleccionar la instrucción que corresponde con cada lenguaje. Si LANGUAGE es cero, el assembler producirá la versión en ingles, si LANGUAGE es uno, producirá la versión en español. La porción de programa que podria hacer esto es:

```
IFE     LANGUAJE
ELSE
```

A5.9.4 CONDICIONES ANIDADAS

Se podrá dar al assembler más de dos clausulas de cindición anidadas. Por ejemplo suponga, que la versión en español realmente no se comprende, y decide producir otras versiones que desplieguen mensajes y el prompt en alemán y frances. Para hacer esto, modificaria el programa anterior para que la constatte LANGUAGE tomará los valores 0,1,2, o 3(para ingles, español, frances o alemán) respectivamente; por lo cual la sección de mensajes podria tener la siguiente forma:

```
IFE     LANGUAJE
  .
  .             ( instrucciones producidas en ingles)
  .
ELSE
   IFE     LANGUAJE_1
     .
     .          ( instrucciones producidas en español )
     .
   ELSE
      IFE    LANGUAJE_2
        .
        .       ( instrucciones producidas en frances )
        .
      ELSE
        .
        .       ( instrucciones producidas en alemán )
      ENDIF
   ENDIF
ENDIF
```

Hubo que ser necesario separar tres ENDIF para balancear los correspondientes IFE. Además se ha sangrado los pares de IFE_ENDIF para permitir una mejor lectura del programa.

A5.9.5 DIRECTIVAS DE LISTADO

Las directivas de listado le dicen al ensamblador que imprimir y conque formato. La tabla 2-12 resume estos directivos en cuatro grupos:

tabal 2-12 Directivas de listado

Directiva	Funcion

Control de listado

.XCREF	formato:	.XCREF Desactiva el listado entre el actual CREF y el siguiente.
.CREF	formato:	.CREF restaura el listado desactivado por XCREF
.XLIST	formato:	.XLIST Desactiva el ensamblado listado entre el directivo .LIST
.LIST	formato:	.LIST restaura el listado del ensamble

Despliegue de mensajes de estado

% OUT	formato:	% OUT texto despliega el mensaje en el texto

Bloques de control de condición falsa

.LFCOND	formato:	.LFCOND Lista todos los bloques condicionales. Este es el valor por default.
.SFCOND	formato:	.SFCOND Omite los bloques con condición falsa en el listado.

A5.9.6 Directivas de control de listado

Las directivas .XCREF y .CREF excluirán las porciones de un programa de un archivo de referencia cruzado (.CRF), mientras que XLIST y .LIST lo excluirán de unarchivo .LST. Se podrá utilizar .LIST y .XLIST para imprimir procedimientos seleccionados entre programas, colocando .LIST al inicio de este, y .XLIST al final.

A5.9.7 Directivas de despliegue de mensajes de estado

La directiva %OUT desplegará mensajes mientras un programa esta siendo ensamblado, esto es muy poderoso para reportar el progreso de un ensamblaje demasiado largo. Por ejemplo, la siguiente instruccion le dice a usted que el asembler ha encontrado la segunda parte del procedimiento

```
IF2
   %OUT  inicio del segundo paso del  assembly
ENDIF
```

ANEXO 4

DATOS GENERALES SOBRE DISCOS FLEXIBLES

DATOS GENERALES SOBRE DISCOS FLEXIBLES

Cuando el proceso de formateo el DOS crea la siguiente estructura para los discos:

1. El sector 0 de cada cara (sector 0) que almacena información acerca de la estructura lógica de un disco y un programa de arranque

2. La FAT (File Allocation Table) (sectores 1 y 2) que determina la localización de los archivos en el disco.

3. Copia de la FAT (sectores 3 y 4)

4. El directorio raíz (sectores 5,6,7,.....11) que es un índice para los archivos

5. Sectores de datos (sectores 12 en adelante)

Nota: Los discos de una cara, tienen solamente cuatro sectores para el directorio.

Definiciones:

Tracks: (Pistas) Círculos concéntricos en el disco en donde se almacena la información

Sectores: División de las pistas, por lo general, cada sector almacena 512 bytes

Clusters: Unidad física mínima de asignación de espacio para los archivos. Cada cluster puede almacenar de 1 a 7 sectores

SECTOR DE BOOT

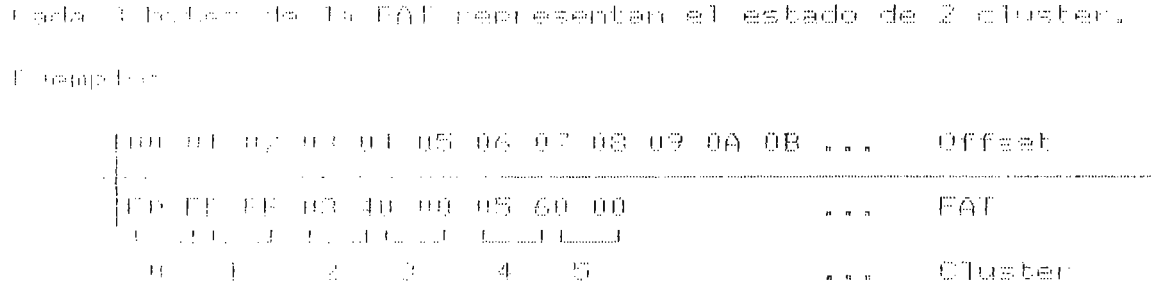
00	-	JMP (dirección del programa cargador)
03	- 0A	Nombre del sistema
0B	- 0C	Número de bytes por sector *
0D		Número de sectores por cluster
0E	- 0F	Número de sectores reservados
10		Número de copias de la FAT
11	- 12	Máximo número de entradas al directorio raíz *
13	- 14	Número de sectores totales *
15		Descripción de medio
16	- 17	Número de sectores ocupados por la FAT
18	- 19	Número de sectores ocupados por pista
1A	- 1B	Número de caras
1C	- 1F	Número de sectores ocultos

* Primero menos significante almacenado primero

FAT DE 12 BITS

La compuesta por lo que se llama entradas de FAT, cada entrada de FAT esta definida por 1 byte y medio (12 bits). Estos doce bits indican el estado de un cluster, es decir, la FAT esta compuesta por tantas entradas como sectores tenga el disco. Cada 3 bytes de la FAT representan el estado de 2 cluster.

Ejemplo:



El cluster 0 y 1 estan reservados para el sistema. A partir del cluster 2 se utilizan para los datos.

De la figura se puede ver que los cluster 2 y 3 estan compuestos por 03 40 00 a partir de un offset de 3).

Los cluster 4 y 5 estan compuestos por 05 60 00 a partir de un offset de 6).

En el directorio de un disco, para cada archivo, se utilizan 32 bytes, los cuales seran utilizados para dar informacion correspondiente a dichos archivos (logitud del mismo, inicio en la FAT, hora, fecha de ultima edicion, etc). El byte 32 de la parte del directorio que corresponde a un archivo indica el cluster de arranque a la FAT.

Asi, si en el directorio, en el byte 32, de la parte que corresponde a un archivo, aparece por ejemplo el numero 02, eso quiere decir que su mapa de localizacion en la FAT (cluster ocupados por un archivo) comienza en el cluster 02 de la FAT (un offset de 03) ademas este dato indica que el archivo esta almacenado comenzando en el cluster 02, los siguientes cluster ocupados por el archivo puede que no esten contiguos, para saber cual es el siguiente cluster utilizado por un archivo, es necesario seguirle la pista en la FAT.

En general, para acceder los datos de un archivo, no se hace por medio de cluster, si no, que a través de sectores. Para saber en que sector comienza un cluster determinado se utiliza la siguiente formula:

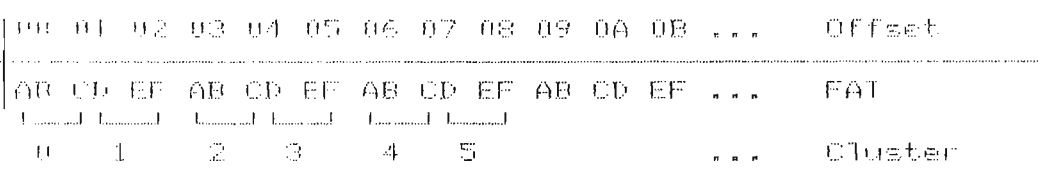
Es decir, que el sector en donde comienza, por ejemplo, el sector 4 es:

$S = (4 - 2)*12 + 12$

$S = 16$

En general, para calcular el offset, se multiplica el contenido del campo del directorio llamado ingreso a la FAT por 1.5 y se toma el entero proximo menor. A partir de este offset se tomara una palabra, la cual se interpreta de manera distinta si el cluster es par o impar.

Sabemos que cada 3 bytes definen dos cluster, estos 3 bytes los representaremos por AB CD EF, es decir, que la FAT se puede conceptualizar desde el inicio de la siguiente manera



Los bytes comprendidos entre A y F representan dos entradas de FAT (dos cluster).

Para averiguar a que cluster apunta una entrada de FAT procedemos asi:

Los bytes AB CD EF estan apuntado a DAB y EFC.

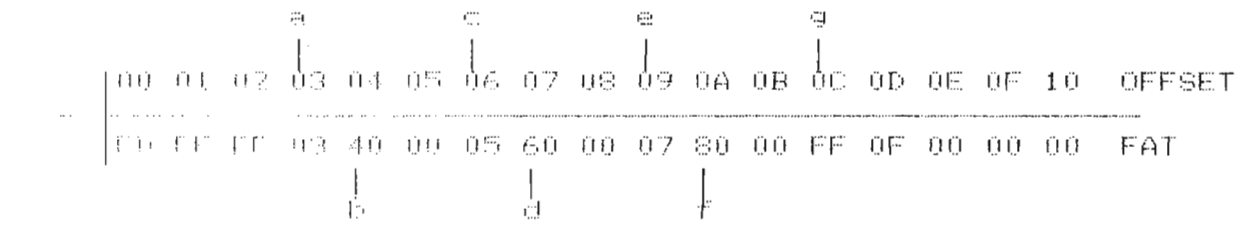
Si el numero de cluster que se multiplico por 1.5 es impar el offset calculado apunta a la letra C de los bytes AB CD DE y el cluster a que apunta la entrada vendra dado por EFC

Si el numero de cluster que se multiplica por 1.5 es par el offset calculado apunta a la letra A de AB CD EF y la entrada de FAT apunta a DAB

No siempre las entradas de la FAT apunta al siguiente cluster del archivo, si no, que a veces las entradas indican el estado del cluster como se muestra en la siguiente tabla:

Codigo	Significado
0	Libre para asignarse
FF0-FF6	Reservado
FF7	Cluster dañado
FFF	Fin de archivo

Ejemplo:
Si en el sector 5 (directorio) el valore del byte correspondiente al ingreso de la FAT tiene 02 hallar todos los cluster que pertenecen al archivo utilizando la FAT que se muestra en la siguiente figura:



1. Calcular offset:

02*1,5 = 3 (a)

Numero de cluster original (2, par) leer en AB CD EF a partir del punto A

AB CD EF
03 40 05

DAB = 003

El siguiente cluster del archivo es 003

2. Calcular el nuevo offset

03*1,5 = 4,5 = 4 (b)

Numero de cluster original (3, impar) leer CD EF

CD EF
40 00

EFC = 004

El siguiente cluster del archivo es 004

3. Calcular nuevo offset:

04*1,5 = 6 (c)

Como 4 es par leer AB CD EF

AB CD EF
05 60 00

DAB = 005

El siguiente cluster del archivo es 005

4. 05*1,5 = 7,5 ≡ 7 (d)

(5, impar) CD EF
80 00

EFC = 006

5. 06*1,5 = 9 (e)

(6, par) AB CD EF
07 80

DAB = 007

6. 07*1,5 = 10,5 ≡ 10 (f)

CD EF
80 00

EFC = 008

7. 08*1,5 = 12 (g)

AB CD
FF 0F

DAB = FFF (fin de archivo)

Clusters del archivo son: 2,3,4,5,6,7,8

ANEXO 5

PIN-OUT DEL UART Y DEL RS-232C

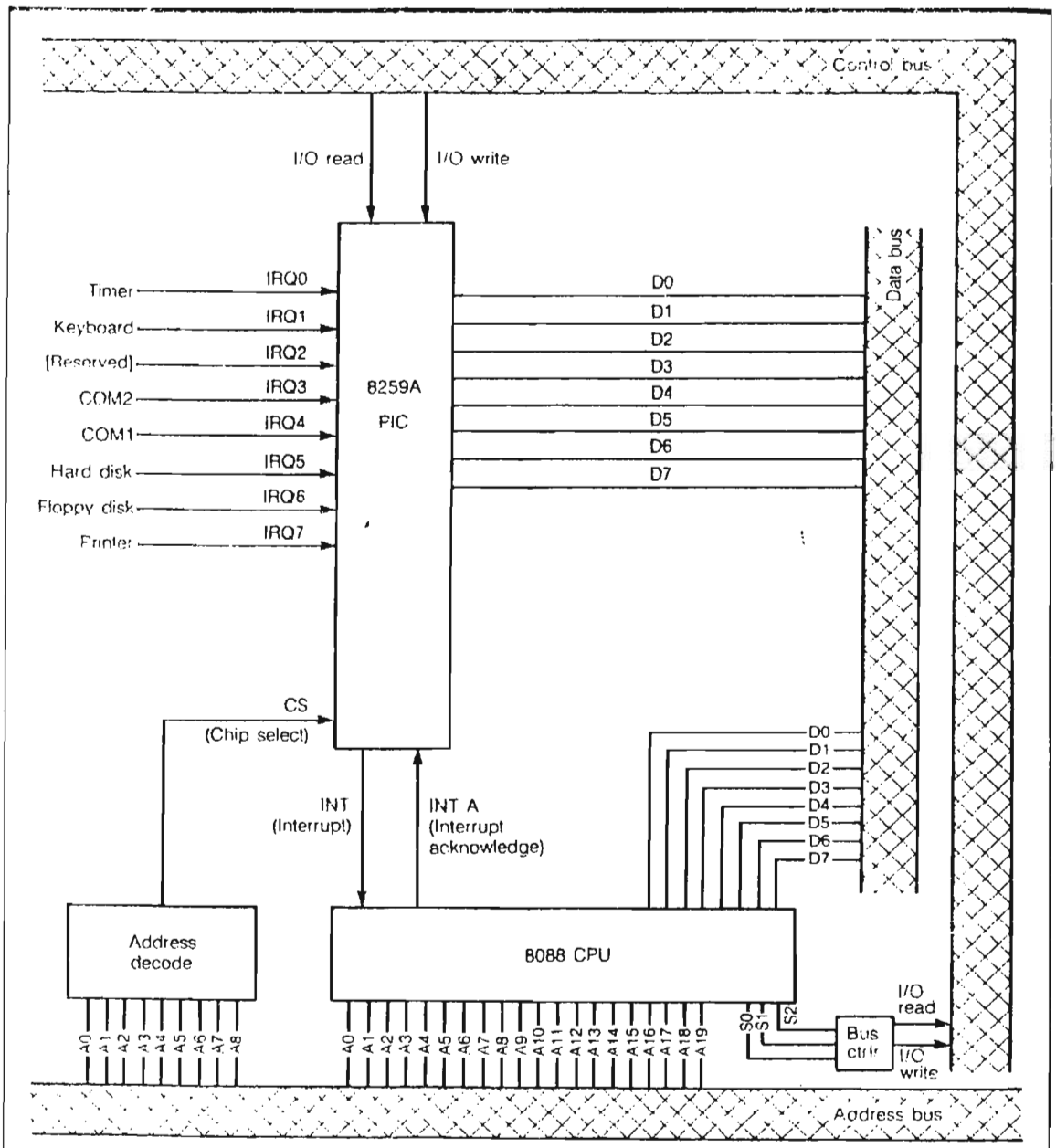


Figure 14.1: IBM PC interrupt structure

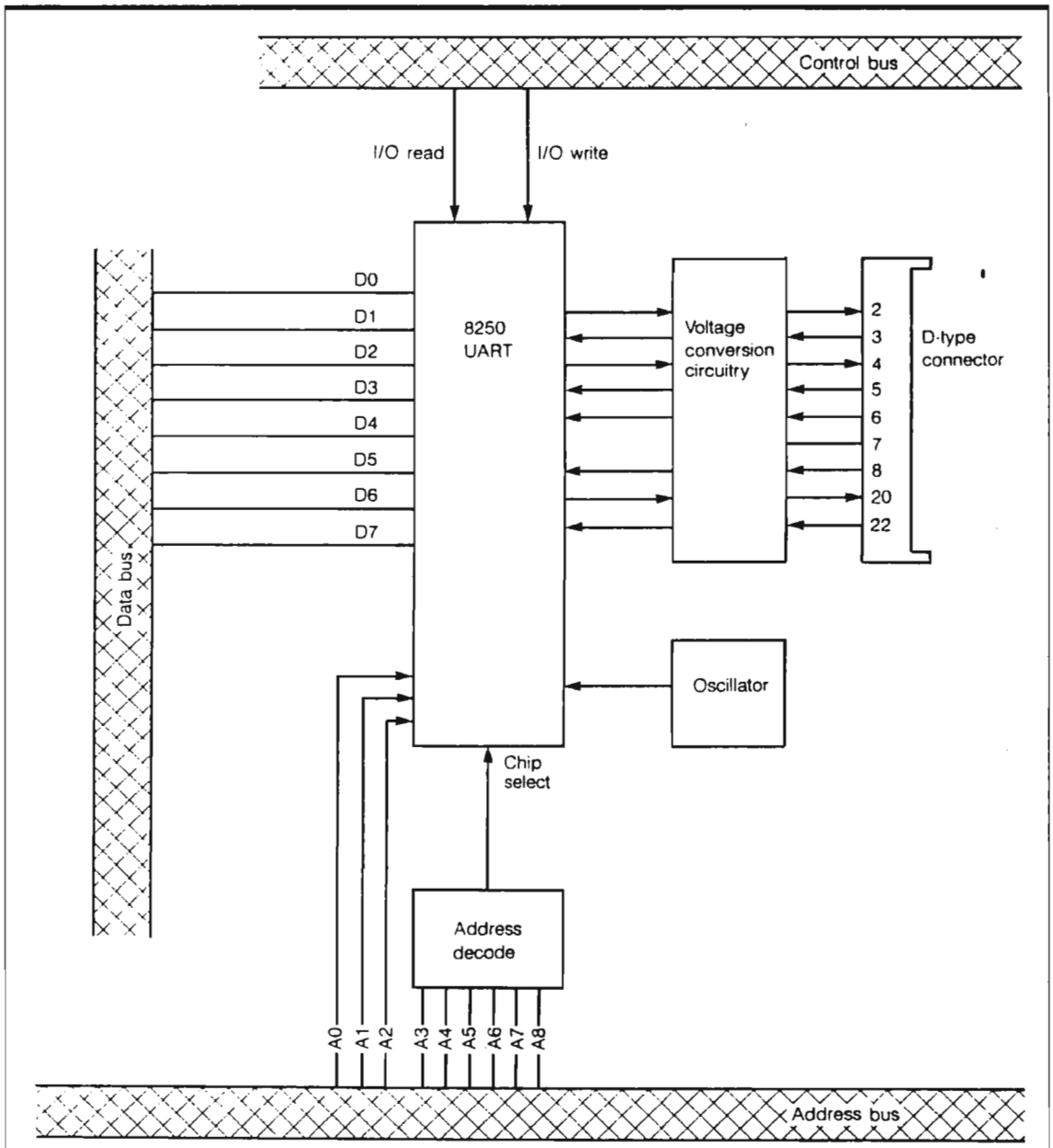
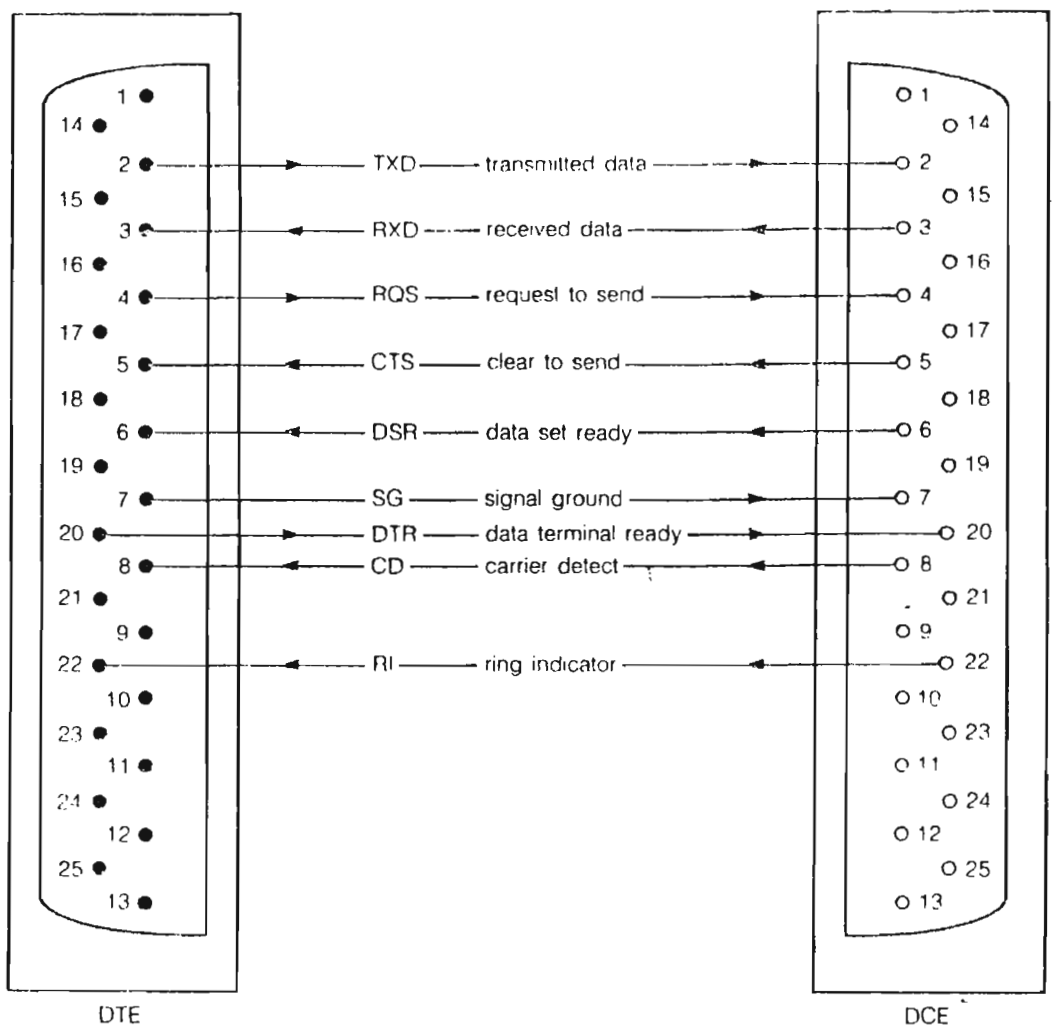


Figure 14.2: Serial interface logic diagram



Standard DTE (e.g., IBM PC) to DCE (e.g., modem) connections

Anexo 5.3

The full set of RS-232 circuits

Pin	Circuit	Abbreviation	Full name	Direction
Data				
2	BA	TXD	Transmitted data	DTE to DCE
3	BB	RXD	Received data	DCE to DTE
Primary handshaking lines:				
6	CC	DSR	Data set ready	DCE to DTE
20	CD	DTR	Data terminal ready	DTE to DCE
Secondary handshaking lines:				
4	CA	RQS	Request to send	DTE to DCE
5	CB	CTS	Clear to send	DCE to DTE
Modem lines:				
8	CF	CD	Carrier detect	DCE to DTE
22	CE	RI	Ring indicator	DCE to DTE
Ground or common:				
7	AB	SG	Signal ground	
Less commonly used circuits:				
1	AA		Protective ground	
12	SCF		Secondary received line signal det.	DCE to DTE
13	SCB		Secondary clear to send	DTE to DCE
14	SBA		Secondary Transmitted Data	DTE to DCE
15	DB		Transmitter signal element timing	DCE to DTE
16	SBB		Secondary received data	DCE to DTE
17	DD		Receiver signal element timing	DCE to DTE
19	SCA		Secondary request to send	DTE to DCE
21	CG		Signal quality detector	DCE to DTE
23	CH		Data signal rate selector	DTE to DCE
23	CI		Data signal rate selector	DCE to DTE
24	DA		Transmitter signal element timing	DTE to DCE

Anexo 5.

ANEXO 6

PIN-OUT DE LOS CHIPS DE SOPORTE DEL SISTEMA 286

intel

ADVANCE INFORMATION

iAPX 286/10
HIGH PERFORMANCE MICROPROCESSOR WITH MEMORY MANAGEMENT AND PROTECTION

(80286-8, 80286-6, 80286-4)

- **High Performance Processor (Up to six times iAPX 86)**
- **Large Address Space:**
  - **—16 Megabytes Physical**
  - **—1 Gigabyte Virtual per Task**
- **Integrated Memory Management, Four-Level Memory Protection and Support for Virtual Memory and Operating Systems**
- **Two iAPX 86 Upward Compatible Operating Modes:**
  - **—iAPX 86 Real Address Mode**
  - **—Protected Virtual Address Mode**
- **Range of clock rates**
  - **—8 MHz for 80286-8**
  - **—6 MHz for 80286-6**
  - **—4 MHz for 80286-4**
- **Optional Processor Extension:**
  - **—iAPX 286/20 High Performance 80-bit Numeric Data Processor**
- **Complete System Development Support:**
  - **—Development Software: Assembler, PL/M, Pascal, FORTRAN, and System Utilities**
  - **—In-Circuit-Emulator (ICE™-286)**
- **High Bandwidth Bus Interface (8 Megabyte/Sec)**
- **Available In EXPRESS:**
  - **—Standard Temperature Range**

The iAPX 286/10 (80286 part number) is an advanced, high-performance microprocessor with specially optimized capabilities for multiple user and multi-tasking systems. The 80286 has built-in memory protection that supports operating system and task isolation as well as program and data privacy within tasks. An 8 MHz iAPX 286/10 provides up to six times greater throughout than the standard 5 MHz iAPX 86/10. The 80286 includes memory management capabilities that map up to 2^{30} (one gigabyte) of virtual address space per task into 2^{24} bytes (16 megabytes) of physical memory.

The iAPX 286 is upward compatible with iAPX 86 and 88 software. Using iAPX 86 real address mode, the 80286 is object code compatible with existing iAPX 86, 88 software. In protected virtual address mode, the 80286 is source code compatible with iAPX 86, 88 software and may require upgrading to use virtual addresses supported by the 80286's integrated memory management and protection mechanism. Both modes operate at full 80286 performance and execute a superset of the iAPX 86 and 88's instructions.

The 80286 provides special operations to support the efficient implementation and execution of operating systems. For example, one instruction can end execution of one task, save its state, switch to a new task, load its state, and start execution of the new task. The 80286 also supports virtual memory systems by providing a segment-not-present exception and restartable instructions.

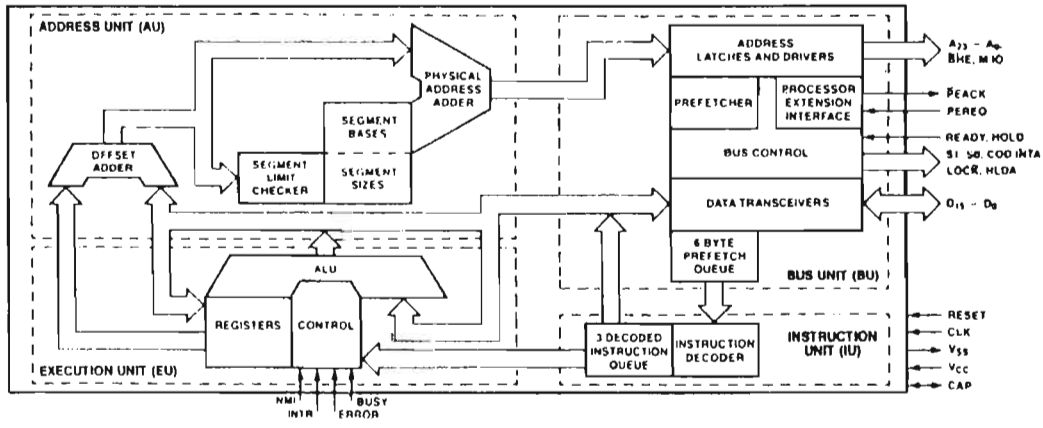
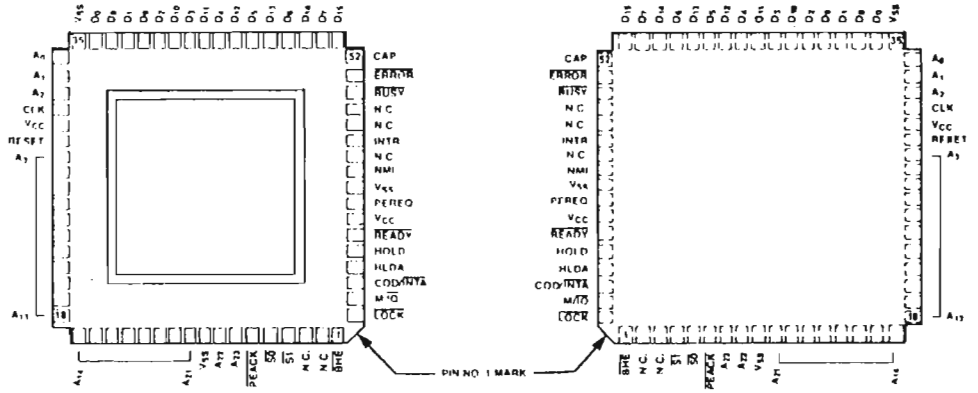


Figure 1. 80286 Internal Block Diagram

Component Pad View—As viewed from underside of component when mounted on the board.

P.C. Board View—As viewed from the component side of the P.C. board.



NOTE: N.C. pads must not be connected.

Figure 2. 80286 Pin Configuration

Table 1. Pin Description

The following pin function descriptions are for the 80286 microprocessor:

Symbol	Type	Name and Function
CLK	I	**System Clock** provides the fundamental timing for iAPX 286 systems. It is divided by two inside the 80286 to generate the processor clock. The internal divide-by-two circuitry can be synchronized to an external clock generator by a LOW to HIGH transition on the RESET input.
D_{15}-D_0	I/O	**Data Bus** inputs data during memory, I/O, and interrupt acknowledge read cycles; outputs data during memory and I/O write cycles. The data bus is active HIGH and floats to 3-state OFF during bus hold acknowledge.
A_{23}-A_0	O	**Address Bus** outputs physical memory and I/O port addresses. A0 is LOW when data is to be transferred on pins D_{7-0}. A_{23}-A_{16} are LOW during I/O transfers. The address bus is active HIGH and floats to 3-state OFF during bus hold acknowledge.
$\overline{BHE}$	O	**Bus High Enable** indicates transfer of data on the upper byte of the data bus, D_{15-8}. Eight-bit oriented devices assigned to the upper byte of the data bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is active LOW and floats to 3-state OFF during bus hold acknowledge.

$\overline{BHE}$ and A0 Encodings

$\overline{BHE}$ Value	A0 Value	Function
0	0	Word transfer
0	1	Byte transfer on upper half of data bus (D_{15-8})
1	0	Byte transfer on lower half of data bus (D_{7-0})
1	1	Reserved

Table 1. Pin Description (Cont.)

Symbol	Type	Name and Function
$\overline{S1}$, $\overline{S0}$	O	**Bus Cycle Status** indicates initiation of a bus cycle and, along with M/$\overline{IO}$ and COD/$\overline{INTA}$, defines the type of bus cycle. The bus is in a T_S state whenever one or both are LOW. $\overline{ST}$ and $\overline{S0}$ are active LOW and float to 3-state OFF during bus hold acknowledge.

80286 Bus Cycle Status Definition

COD $\overline{INTA}$	M/$\overline{IO}$	$\overline{S1}$	$\overline{S0}$	Bus cycle initiated
0 (LOW)	0	0	0	Interrupt acknowledge
0	0	0	1	Reserved
0	0	1	0	Reserved
0	0	1	1	None, not a status cycle
0	1	0	0	IF A1 = 1 then halt, else shutdown
0	1	0	1	Memory data read
0	1	1	0	Memory data write
0	1	1	1	None, not a status cycle
1 (HIGH)	0	0	0	Reserved
1	0	0	1	I/O read
1	0	1	0	I/O write
1	0	1	1	None, not a status cycle
1	1	0	0	Reserved
1	1	0	1	Memory instruction read
1	1	1	0	Reserved
1	1	1	1	None, not a status cycle

M/$\overline{IO}$	O	**Memory/IO Select** distinguishes memory access from I/O access. If HIGH during T_S, a memory cycle or a halt/shutdown cycle is in progress. If LOW, an I/O cycle or an interrupt acknowledge cycle is in progress. M/$\overline{IO}$ floats to 3-state OFF during bus hold acknowledge.
COD/$\overline{INTA}$	O	**Code/Interrupt Acknowledge** distinguishes instruction fetch cycles from memory data read cycles. Also distinguishes Interrupt acknowledge cycles from I/O cycles. COD/$\overline{INTA}$ floats to 3-state OFF during bus hold acknowledge. Its timing is the same as M/$\overline{IO}$.
$\overline{LOCK}$	O	**Bus Lock** indicates that other system bus masters are not to gain control of the system bus following the current bus cycle. The $\overline{LOCK}$ signal may be activated explicitly by the "LOCK" instruction prefix or automatically by 80286 hardware during memory XCHG instructions, interrupt acknowledge, or descriptor table access. $\overline{LOCK}$ is active LOW and floats to 3-state OFF during bus hold acknowledge.
$\overline{READY}$	I	**Bus Ready** terminates a bus cycle. Bus cycles are extended without limit until terminated by $\overline{READY}$ LOW. $\overline{READY}$ is an active LOW synchronous input requiring setup and hold times relative to the system clock be met for correct operation. $\overline{READY}$ is ignored during bus hold acknowledge.
HOLD HLDA	I O	**Bus Hold Request and Hold Acknowledge** control ownership of the 80286 local bus. The HOLD input allows another local bus master to request control of the local bus. When control is granted, the 80286 will float its bus drivers to 3-state OFF and then activate HLDA, thus entering the bus hold acknowledge condition. The local bus will remain granted to the requesting master until HOLD becomes inactive which results in the 80286 deactivating HLDA and regaining control of the local bus. This terminates the bus hold acknowledge condition. HOLD may be asynchronous to the system clock. These signals are active HIGH.
INTR	I	**Interrupt Request** requests the 80286 to suspend its current program execution and service a pending external request. Interrupt requests are masked whenever the interrupt enable bit in the flag word is cleared. When the 80286 responds to an interrupt request, it performs two interrupt acknowledge bus cycles to read an 8-bit interrupt vector that identifies the source of the interrupt. To assure program interruption, INTR must remain active until the first interrupt acknowledge cycle is completed. INTR is sampled at the beginning of each processor cycle and must be active HIGH at least two processor cycles before the current instruction ends in order to interrupt before the next instruction. INTR is level sensitive, active HIGH, and may be asynchronous to the system clock.
NMI	I	**Non-maskable Interrupt Request** interrupts the 80286 with an internally supplied vector value of 2. No interrupt acknowledge cycles are performed. The interrupt enable bit in the 80286 flag word does not affect this input. The NMI input is active HIGH, may be asynchronous to the system clock, and is edge triggered after internal synchronization. For proper recognition, the input must have been previously LOW for at least four system clock cycles and remain HIGH for at least four system clock cycles.

Table 1. Pin Description (Cont.)

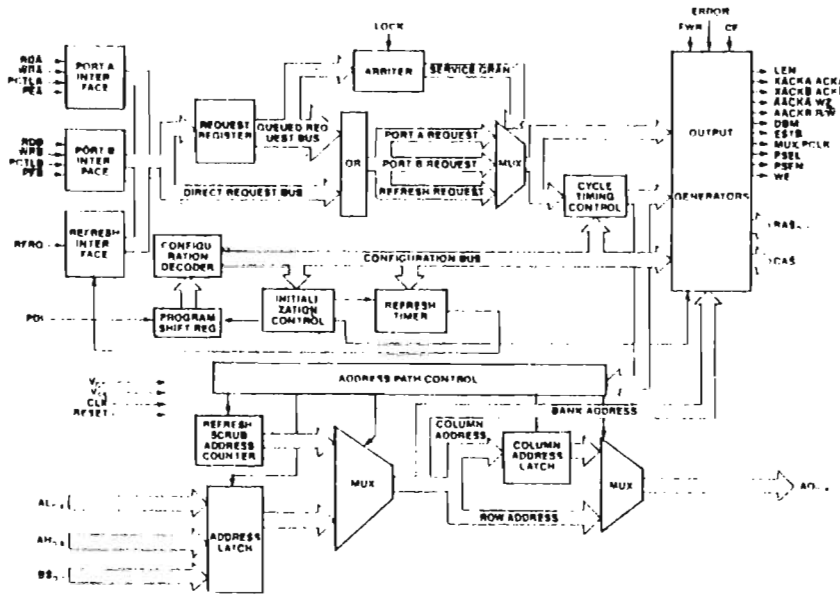
Symbol	Type	Name and Function
PEREQ PEACK	I O	**Processor Extension Operand Request and Acknowledge** extend the memory management and protection capabilities of the 80286 to processor extensions. The PEREQ input requests the 80286 to perform a data operand transfer for a processor extension. The PEACK output signals the processor extension when the requested operand is being transferred. PEREQ is active HIGH and floats to 3-state OFF during bus hold acknowledge. PEACK may be asynchronous to the system clock. PEACK is active LOW.
BUSY ERROR	I I	**Processor Extension Busy and Error** indicate the operating condition of a processor extension to the 80286. An active BUSY input stops 80286 program execution on WAIT and some ESC instructions until BUSY becomes inactive (HIGH). The 80286 may be interrupted while waiting for BUSY to become inactive. An active ERROR input causes the 80286 to perform a processor extension interrupt when executing WAIT or some ESC instructions. These inputs are active LOW and may be asynchronous to the system clock.
RESET	I	**System Reset** clears the internal logic of the 80286 and is active HIGH. The 80286 may be re-initialized at any time with a LOW to HIGH transition on RESET which remains active for more than 16 system clock cycles. During RESET active, the output pins of the 80286 enter the state shown below: **80286 Pin State During Reset** **Pin Value** / **Pin Names** 1 (HIGH) / $\overline{S0}$, $\overline{S1}$, $\overline{PEACK}$, A23-A0, $\overline{BHE}$, $\overline{LOCK}$ 0 (LOW) / M/$\overline{IO}$, COD/$\overline{INTA}$, HLDA 3-state OFF / D_{15}-D_0 Operation of the 80286 begins after a HIGH to LOW transition on RESET. The HIGH to LOW transition of RESET must be synchronous to the system clock. Approximately 50 system clock cycles are required by the 80286 for internal initializations before the first bus cycle to fetch code from the power-on execution address is performed. A LOW to HIGH transition of RESET synchronous to the system clock will end a processor cycle at the second HIGH to LOW transition of the system clock. The LOW to HIGH transition of RESET may be asynchronous to the system clock; however, in this case it cannot be predetermined which phase of the processor clock will occur during the next system clock period. Synchronous LOW to HIGH transitions of RESET are required only for systems where the processor clock must be phase synchronous to another clock.
V_{SS}	I	**System Ground:** 0 Volts.
V_{CC}	I	**System Power:** +5 Volt Power Supply.
CAP	I	**Substrate Filter Capacitor:** a 0.047μf ± 20% 12V capacitor must be connected between this pin and ground. This capacitor filters the output of the internal substrate bias generator. A maximum DC leakage current of 1 μa is allowed through the capacitor. For correct operation of the 80286, the substrate bias generator must charge this capacitor to its operating voltage. The capacitor chargeup time is 5 milliseconds (max.) after V_{CC} and CLK reach their specified AC and DC parameters. RESET may be applied to prevent spurious activity by the CPU during this time. After this time, the 80286 processor clock can be phase synchronized to another clock by pulsing RESET LOW synchronous to the system clock.

intel

8207
DUAL-PORT DYNAMIC RAM CONTROLLER

- **Provides All Signals Necessary to Control 16K, 64K and 256K Dynamic RAMs**
- **Directly Addresses and Drives up to 2 Megabytes without External Drivers**
- **Supports Single and Dual-Port Configurations**
- **Automatic RAM Initialization in All Modes**
- **Four Programmable Refresh Modes**
- **Transparent Memory Scrubbing in ECC Mode**
- **Fast Cycle Support for 8 MHz 80286 with 8207-16**
- **Slow Cycle Support for 8 MHz, 10 MHz 8086/88, 80186/188 with 8207-8, 8207-10**
- **Provides Signals to Directly Control the 8206 Error Detection and Correction Unit**
- **Supports Synchronous or Asynchronous Operation on Either Port**
- **68 Lead JEDEC Type A Leadless Chip Carrier (LCC) and Pin Grid Array (PGA), Both in Ceramic.**

The Intel 8207 Dual-Port Dynamic RAM Controller is a high-performance, systems-oriented, Dynamic RAM controller that is designed to easily interface 16K, 64K and 256K Dynamic RAMs to Intel and other microprocessor systems. A dual-port interface allows two different busses to independently access memory. When configured with an 8206 Error Detection and Correction Unit the 8207 supplies the necessary logic for designing large error-corrected memory arrays. This combination provides automatic memory initialization and transparent memory error scrubbing.



210463-1

Figure 1. 8207 Block Diagram

Table 1. Pin Description

Symbol	Pin	Type	Name and Function
LEN	1	O	**ADDRESS LATCH ENABLE:** In two-port configurations, when Port A is running with iAPX 286 Status Interface mode, this output replaces the ALE signal from the system bus controller of port A and generates an address latch enable signal which provides optimum setup and hold timing for the 8207. This signal is used in Fast Cycle operation only.
$\overline{XACKA}$/ $\overline{ACKA}$	2	O	**TRANSFER ACKNOWLEDGE PORT A/ACKNOWLEDGE PORT A:** In non-ECC mode, this pin is $\overline{XACKA}$ and indicates that data on the bus is valid during a read cycle or that data may be removed from the bus during a write cycle for Port A. $\overline{XACKA}$ is a Multibus-compatible signal. In ECC mode, this pin is $\overline{ACKA}$ which can be configured, depending on the programming of the X program bit, as an $\overline{XACK}$ or $\overline{AACK}$ strobe. The SA programming bit determines whether the $\overline{AACK}$ will be an early $\overline{EAACKA}$ or a late $\overline{LAACKA}$ interface signal.
$\overline{XACKB}$/ $\overline{ACKB}$	3	O	**TRANSFER ACKNOWLEDGE PORT B/ACKNOWLEDGE PORT B:** In non-ECC mode, this pin is XACKB and indicates that data on the bus is valid during a read cycle or that data may be removed from the bus during a write cycle for Port B. $\overline{XACKB}$ is a Multibus-compatible signal. In ECC mode, this pin is $\overline{ACKB}$ which can be configured, depending on the programming of the X program bit, as an $\overline{XACK}$ or $\overline{AACK}$ strobe. The SB programming bit determines whether the $\overline{AACK}$ will be an early $\overline{EAACKB}$ or a late $\overline{LAACKB}$ interface signal.
$\overline{AACKA}$/ $\overline{WZ}$	4	O	**ADVANCED ACKNOWLEDGE PORT A/WRITE ZERO:** In non-ECC mode, this pin is $\overline{AACKA}$ and indicates that the processor may continue processing and that data will be available when required. This signal is optimized for the system by programming the SA program bit for synchronous or asynchronous operation. In ECC mode, after a RESET, this signal will cause the 8206 to force the data to all zeros and generate the appropriate check bits.
$\overline{AACKB}$/ R/$\overline{W}$	5	O	**ADVANCED ACKNOWLEDGE PORT B/READ/WRITE:** In non-ECC mode, this pin is $\overline{AACKB}$ and indicates that the processor may continue processing and that data will be available when required. This signal is optimized for the system by programming the SB program bit for synchronous or asynchronous operation. In ECC mode, this signal causes the 8206 EDCU to latch the syndrome and error flags and generate check bits.
$\overline{DBM}$	6	O	**DISABLE BYTE MARKS:** This is an ECC control output signal indicating that a read or refresh cycle is occurring. This output forces the byte address decoding logic to enable all 8206 data output buffers. In ECC mode, this output is also asserted during memory initialization and the 8-cycle dynamic RAM wake-up exercise. In non-ECC systems this signal indicates that either a read, refresh or 8-cycle warm-up is in progress.
$\overline{ESTB}$	7	O	**ERROR STROBE:** In ECC mode, this strobe is activated when an error is detected and allows a negative-edge triggered flip-flop to latch the status of the 8206 EDCU CE for systems with error logging capabilities. $\overline{ESTB}$ will not be issued during refresh cycles.
LOCK	8	I	**LOCK:** This input instructs the 8207 to lock out the port not being serviced at the time LOCK was issued.
V_{CC}	9 43	I	**DRIVER POWER:** + 5 volts. Supplies V_{CC} for the output drivers. **LOGIC POWER:** + 5 volts. Supplies V_{CC} for the internal logic circuits.
CE	10	I	**CORRECTABLE ERROR:** This is an ECC input from the 8206 EDCU which instructs the 8207 whether a detected error is correctable or not. A high input indicates a correctable error. A low input inhibits the 8207 from activating WE to write the data back into RAM. This should be connected to the CE output of the 8206.

Table 1. Pin Description for the 286 Mode (Also Contains Pins Identical in Other Modes)

Symbol	Pin Type Input (I) Output (O)	Pin Number	Identical In All Modes	Functions
$\overline{BHE}$	I/O	1	YES	**BUS HIGH ENABLE** indicates transfer of data on the upper byte of the data bus, D15-8. Eight bit devices assigned to the upper byte of the data bus would normally use $\overline{BHE}$ to condition chip select function. $\overline{BHE}$ is active LOW and floats to Tri-State OFF when the 82258 does not own the bus.
$\overline{RD}$	I	2	NO	**READ** command in conjunction with chip select ($\overline{CS}$) enables reading out of the 82258 register, addressed by the address lines A7-A0. $\overline{RD}$ is an active LOW signal and is asynchronous to the 82258 clock.
$\overline{WR}$	I	3	NO	**WRITE** command along with $\overline{CS}$ is used for writing into the 82258 registers. $\overline{WR}$ is an active LOW signal and is asynchronous to the 82258 clock.
DREQ3-DREQ0	I	4—7	YES	**DMA REQUEST** input signals are used for externally synchronized DMA transfers. If channel 3 is used as a Multiplexor channel, DREQ3 is defined as I/O Request (IOREQ) signal. These signals are active HIGH signals and are asynchronous to the 82258 clock.
$\overline{CS}$	I	8	NO	**CHIP SELECT** is used to enable a processor to access the 82258 registers. This access is additionally controlled either by bus status signals or by the Read or Write command signals. $\overline{CS}$ is an active LOW signal, asynchronous to the 82258 clock.
$\overline{READY}$	I	10	NO	**BUS READY** terminates a bus cycle. Bus cycles are extended without limit until terminated by an active $\overline{READY}$. $\overline{READY}$ is an active LOW, synchronous input, requiring set up and hold times relative to system clock to be met for correct operation.
$\overline{S1}$, $\overline{S0}$	I/O	11,13	YES	**BUS CYCLE STATUS** signals control the support circuitry. The beginning of a bus cycle is indicated by $\overline{S1}$, or $\overline{S0}$, or both going active. The termination of a bus cycle is indicated by all the status signals going inactive in the 186 mode or the bus ready ($\overline{READY}$) going active in the 286 mode. Both $\overline{S0}$ & $\overline{S1}$ are active LOW signals. $\overline{S0}$, $\overline{S1}$ along with $\overline{S2}$ (in the 186 mode) or M/$\overline{IO}$ (in the 286 mode) define the type of bus cycle. $\overline{S2}$ and M/$\overline{IO}$ have the same meaning but, in the 186 mode $\overline{S2}$ signal can be active only when at least one of $\overline{S1}$ and $\overline{S0}$ is active, whereas in the 286 mode the M/$\overline{IO}$ signal is valid with the address on address lines.

$\overline{BHE}$ and A0 Encoding

$\overline{BHE}$ Value	A0 Value	Function
0	0	Word Transfer (D15-0)
0	1	Byte Transfer on upper half of data bus (D15-8)
1	0	Byte Transfer on lower half of data bus (D7-0)
1	1	Odd addressed byte on 8 bit bus (D7-0)

Table 1. Pin Description for the 286 Mode (Also Contains Pins Identical in Other Modes) (Continued)

Symbol	Pin Type Input (I) Output (O)	Pin Number	Identical In All Modes	Functions
				The 82258 Bus Cycle Status Definitions (82258 Local Bus Master, All Signals (O))

M/$\overline{IO}$ or $\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Bus Cycle Initiated
0	0	0	Read I/O Vector (For Multiplexor channel)
0	0	1	Read from I/O space
0	1	0	Write into I/O space
0	1	1	None. (Does not occur in the 186 mode)
1	0	0	None. (Does not occur)
1	0	1	Read from memory space
1	1	0	Write into memory space
1	1	1	None; not a bus cycle

When the 82258 is not a bus master of the local bus, the status signals are used as inputs for detection of synchronous accesses to the 82258

Interpretation of the Status and $\overline{CS}$ Signals by the 82258 (82258 Slave, All Signals (I))

$\overline{CS}$	$\overline{S1}$	$\overline{S0}$	Interpretation
1	X	X	82258 not selected (No action)
0	0	0	No 82258 access (No action)
0	0	1	Read from an 82258 register
0	1	0	Write into an 82258 register
0	1	1	Not a bus cycle*

*: The 82258 is selected but no synchronous access is activated. The 82258 monitors $\overline{RD}$ and $\overline{WR}$ signals for detection of an asynchronous access.

CLK	I	12	NO	**SYSTEM CLOCK** provides the fundamental system timing. It is divided by two to generate the 82258 internal clock. CLK is an active HIGH signal which can be connected directly to the 82284 CLK output. The internal divide-by-two circuitry is synchronized to the external clock generator by a LOW to HIGH transition on the RESET input, or by first HIGH to LOW transition on the Status Input $\overline{S0}$ or $\overline{S1}$ after RESET.
M/$\overline{IO}$	O	14	NO	**MEMORY/$\overline{IO}$ SELECT** distinguishes between memory and I/O space addresses.
RESET	I	15	YES	**SYSTEM RESET** forces the 82258 to the initial state. RESET is an active HIGH signal and must be synchronous to the system clock. Reset must be activated for at least 16 CLK cycles.

Table 1. Pin Description for the 286 Mode (Also Contains Pins Identical in Other Modes) (Continued)

Symbol	Pin Type Input (I) Output (O)	Pin Number	Identical In All Modes	Functions
HOLD HLDA	O I	16 17	NO	**BUS HOLD REQUEST AND HOLD ACKNOWLEDGE** control ownership of the local 82258 bus. When active, HOLD indicates a request for the control of the local bus. HOLD goes inactive when the 82258 relinquishes the bus. HLDA, when active, indicates that the 82258 can acquire the control of the bus. When HLDA goes inactive, the 82258 must relinquish the bus at the end of its current cycle. HLDA may be asynchronous to the system clock. Both HOLD and HLDA are active HIGH signals.
D15-D0	I/O	18-25, 27-34	NO	**DATA BUS** is the bidirectional 16 bit bus. For use with an 8 bit bus, only the lower 8 data lines D0–D7 are relevant. The data bus is active HIGH.
A0-A7	I/O	35-42	NO	**ADDRESS LINES** A0–A7 are the lower 8 address lines for DMA transfers. They are also used to input the register address when the processor accesses an 82258 register. All lines are active HIGH.
A8-A23	O	44-59	NO	**ADDRESS LINES** A8–A23 form the remainder of the 82258 address bus. Address bus is active HIGH. ***Pin A21 must have a pull-up resistor (n 10k Ω) connected to it to ensure that it is high during reset.***
$\overline{DACK0}$-$\overline{DACK3}$	O	61-64	YES	**DMA ACKNOWLEDGE** signal acknowledges the requests of the corresponding DREQ signal. $\overline{DACK}$i goes active when the requested transfers are performed on the channel i in response to a DREQi. If channel 3 is in the multiplexor mode, $\overline{DACK3}$ is defined as I/O acknowledge (IOACK). These signals are active LOW.
$\overline{EOD0}$-$\overline{EOD3}$	I/O	65-68	YES	**END OF DMA** signals are open drain drivers with internal high impedance pull up resistors (an external pull-up resistor is required) and can be used as quasi-bi-directional lines. These signals are active LOW. As OUTPUTs the signals are activated (if enabled) for two T-STATE cycles at the end of the DMA transfer of the corresponding channel or they are activated under program control (End of DMA output or interrupt output) $\overline{EOD}$s acts as "End of DMA" level triggered INPUTs if the signals are held high internally but forced low by the external circuitry for at least 250 ns. The current transfer is aborted and the 82258 continues with the next command $\overline{EOD2}$ can also be used as a common active high interrupt signal (INTOUT) for all four channels. In this mode, this signal is a push-pull output and not an open drain output. Other EODi pins may still be used in their regular I/O mode.
V_{SS}	I	9, 43	YES	**SYSTEM GROUND:** 0 Volt.
V_{CC}	I	26, 60	YES	**SYSTEM POWER:** + 5V Power Supply Pin.

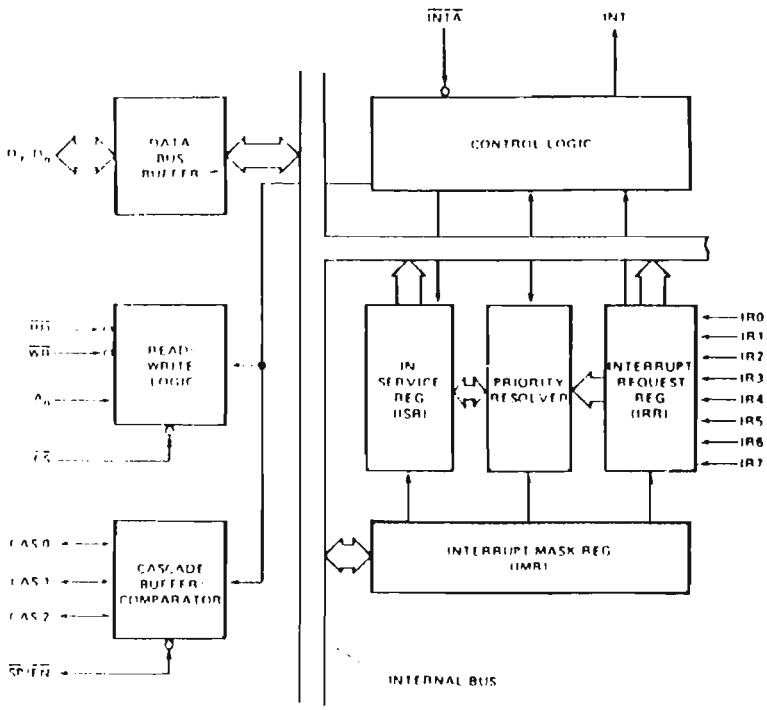
8259A
PROGRAMMABLE INTERRUPT CONTROLLER
8259A/8259A-2/8259A-8

- 8086, 8088 Compatible
- MCS-80®, MCS-85® Compatible
- Eight-Level Priority Controller
- Expandable to 64 Levels
- Programmable Interrupt Modes
- Individual Request Mask Capability
- Single +5V Supply (No Clocks)
- 28-Pin Dual-In-Line Package
- Available in EXPRESS
  - Standard Temperature Range
  - Extended Temperature Range

The Intel 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry. It is packaged in a 28-pin DIP, uses NMOS technology and requires a single +5V supply. Circuitry is static, requiring no clock input.

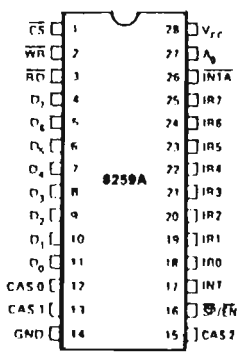
The 8259A is designed to minimize the software and real time overhead in handling multi-level priority interrupts. It has several modes, permitting optimization for a variety of system requirements.

The 8259A is fully upward compatible with the Intel 8259. Software originally written for the 8259 will operate the 8259A in all 8259 equivalent modes (MCS-80/85, Non-Buffered, Edge Triggered).



231468-1

Figure 1. Block Diagram



231468-2

Figure 2. Pin Configuration

Table 1. Pin Description

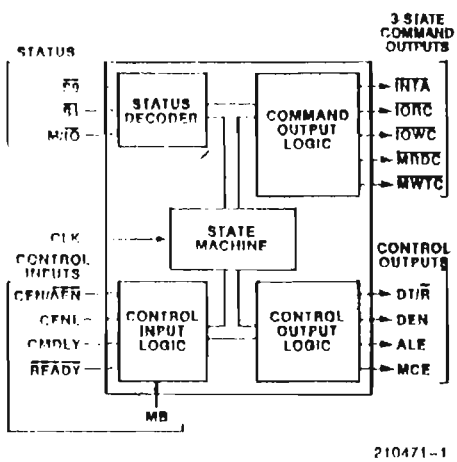
Symbol	Pin No.	Type	Name and Function
V_{CC}	28	I	**SUPPLY:** +5V Supply
GND	14	I	**GROUND**
$\overline{CS}$	1	I	**CHIP SELECT:** A low on this pin enables $\overline{RD}$ and $\overline{WR}$ communication between the CPU and the 8259A. INTA functions are independent of $\overline{CS}$
$\overline{WR}$	2	I	**WRITE:** A low on this pin when $\overline{CS}$ is low enables the 8259A to accept command words from the CPU.
$\overline{RD}$	3	I	**READ:** A low on this pin when $\overline{CS}$ is low enables the 8259A to release status onto the data bus for the CPU.
D_7-D_0	4-11	I/O	**BIDIRECTIONAL DATA BUS:** Control, status and interrupt-vector information is transferred via this bus.
CAS_0-CAS_2	12, 13, 15	I/O	**CASCADE LINES:** The CAS lines form a private 8259A bus to control a multiple 8259A structure. These pins are outputs for a master 8259A and inputs for a slave 8259A.
$\overline{SP}/\overline{EN}$	16	I/O	**SLAVE PROGRAM/ENABLE BUFFER:** This is a dual function pin. When in the Buffered Mode it can be used as an output to control buffer transceivers (EN). When not in the buffered mode it is used as an input to designate a master (SP = 1) or slave (SP = 0).
INT	17	O	**INTERRUPT:** This pin goes high whenever a valid interrupt request is asserted. It is used to interrupt the CPU, thus it is connected to the CPU's interrupt pin.
IR_0-IR_7	18-25	I	**INTERRUPT REQUESTS:** Asynchronous inputs. An interrupt request is executed by raising an IR input (low to high), and holding it high until it is acknowledged (Edge Triggered Mode), or just by a high level on an IR input (Level Triggered Mode).
$\overline{INTA}$	26	I	**INTERRUPT ACKNOWLEDGE:** This pin is used to enable 8259A interrupt-vector data onto the data bus by a sequence of interrupt acknowledge pulses issued by the CPU.
A_0	27	I	**A0 ADDRESS LINE:** This pin acts in conjunction with the $\overline{CS}$, $\overline{WR}$, and $\overline{RD}$ pins. It is used by the 8259A to decipher various Command Words the CPU writes and status the CPU wishes to read. It is typically connected to the CPU A0 address line (A1 for 8086, 8088).

82288
BUS CONTROLLER FOR 80286 PROCESSORS
82288-12, 82288-10, 82288-8, 82288-6

- Provides Commands and Control for Local and System Bus
- Offers Wide Flexibility in System Configurations
- Flexible Command Timing
- Optional MULTIBUS® Compatible Timing
- Control Drivers with 16 mA I_{OL} and 3-State Command Drivers with 32 mA I_{OL}
- Single +5V Supply
- Available in 20 pin Cerdip Package
  (See Packaging Spec, Order #231369)

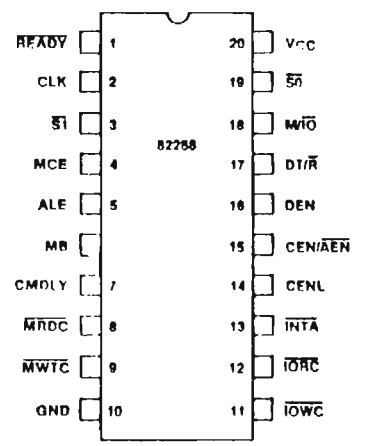
The Intel 82288 Bus Controller is a 20-pin HMOS component for use in 80286 microsystems. The bus controller provides command and control outputs with flexible timing options. Separate command outputs are used for memory and I/O devices. The data bus is controlled with separate data enable and direction control signals.

Two modes of operation are possible via a strapping option: MULTIBUS compatible bus cycles, and high speed bus cycles.



210471-1

Figure 1. 82288 Block Diagram



210471-2

Figure 2. 82288 Pin Configuration

*MULTIBUS is a registered trademark of Intel Corporation.

Table 1. Pin Description

The following pin function descriptions are for the 82288 bus controller.

Symbol	Type	Name and Function
CLK	I	**SYSTEM CLOCK** provides the basic timing control for the 82288 in an 80286 microsystem. Its frequency is twice the internal processor clock frequency. The falling edge of this input signal establishes when inputs are sampled and command and control outputs change.
$\overline{S0}$, $\overline{S1}$	I	**BUS CYCLE STATUS** starts a bus cycle and, along with M/$\overline{IO}$, defines the type of bus cycle. These inputs are active LOW. A bus cycle is started when either $\overline{S1}$ or $\overline{S0}$ is sampled LOW at the falling edge of CLK. Setup and hold times must be met for proper operation.

80286 Bus Cycle Status Definition

M/$\overline{IO}$	$\overline{S1}$	$\overline{S0}$	Type of Bus Cycle
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	None; Idle
1	0	0	Halt or Shutdown
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	None; Idle

M/$\overline{IO}$	I	**MEMORY OR I/O SELECT** determines whether the current bus cycle is in the memory space or I/O space. When LOW, the current bus cycle is in the I/O space. Setup and hold times must be met for proper operation.
MB	I	**MULTIBUS MODE SELECT** determines timing of the command and control outputs. When HIGH, the bus controller operates with MULTIBUS compatible timings. When LOW, the bus controller optimizes the command and control output timing for short bus cycles. The function of the CEN/$\overline{AEN}$ input pin is selected by this signal. This input is typically a strapping option and not dynamically changed.
CENL	I	**COMMAND ENABLE LATCHED** is a bus controller select signal which enables the bus controller to respond to the current bus cycle being initiated. CENL is an active HIGH input latched internally at the end of each T_S cycle. CENL is used to select the appropriate bus controller for each bus cycle in a system where the CPU has more than one bus it can use. This input may be connected to V_{CC} to select this 82288 for all transfers. No control inputs affect CENL. Setup and hold times must be met for proper operation.
CMDLY	I	**COMMAND DELAY** allows delaying the start of a command. CMDLY is an active HIGH input. If sampled HIGH, the command output is not activated and CMDLY is again sampled at the next CLK cycle. When sampled LOW the selected command is enabled. If $\overline{READY}$ is detected LOW before the command output is activated, the 82288 will terminate the bus cycle, even if no command was issued. Setup and hold times must be satisfied for proper operation. This input may be connected to GND if no delays are required before starting a command. This input has no effect on 82288 control outputs.
$\overline{READY}$	I	**READY** indicates the end of the current bus cycle. $\overline{READY}$ is an active LOW input. MULTIBUS mode requires at least one wait state to allow the command outputs to become active. $\overline{READY}$ must be LOW during reset, to force the 82288 into the idle state. Setup and hold times must be met for proper operation. The 82284 drives $\overline{READY}$ LOW during RESET.

Table 1. Pin Description (Continued)

Symbol	Type	Name and Function
CEN/$\overline{AEN}$	I	**COMMAND ENABLE/ADDRESS ENABLE** controls the command and DEN outputs of the bus controller. CEN/$\overline{AEN}$ inputs may be asynchronous to CLK. Setup and hold times are given to assure a guaranteed response to synchronous inputs. This input may be connected to V_{CC} or GND. When MB is HIGH this pin has the $\overline{AEN}$ function. $\overline{AEN}$ is an active LOW input which indicates that the CPU has been granted use of a shared bus and the bus controller command outputs may exit 3-state OFF and become inactive (HIGH). $\overline{AEN}$ HIGH indicates that the CPU does not have control of the shared bus and forces the command outputs into 3-state OFF and DEN inactive (LOW). $\overline{AEN}$ would normally be controlled by an 82289 bus arbiter which activates $\overline{AEN}$ when that arbiter owns the bus to which the bus controller is attached. When MB is LOW this pin has the CEN function. CEN is an unlatched active HIGH input which allows the bus controller to activate its command and DEN outputs. With MB LOW, CEN LOW forces the command and DEN outputs inactive but does not tristate them.
ALE	O	**ADDRESS LATCH ENABLE** controls the address latches used to hold an address stable during a bus cycle. This control output is active HIGH. ALE will not be issued for the halt bus cycle and is not affected by any of the control inputs.
MCE	O	**MASTER CASCADE ENABLE** signals that a cascade address from a master 8259A interrupt controller may be placed onto the CPU address bus for latching by the address latches under ALE control. The CPU's address bus may then be used to broadcast the cascade address to slave interrupt controllers so only one of them will respond to the interrupt acknowledge cycle. This control output is active HIGH. MCE is only active during interrupt acknowledge cycles and is not affected by any control input. Using MCE to enable cascade address drivers requires latches which save the cascade address on the falling edge of ALE.
DEN	O	**DATA ENABLE** controls when data transceivers connected to the local data bus should be enabled. DEN is an active HIGH control output. DEN is delayed for write cycles in the MULTIBUS mode.
DT/$\overline{R}$	O	**DATA TRANSMIT/RECEIVE** establishes the direction of data flow to or from the local data bus. When HIGH, this control output indicates that a write bus cycle is being performed. A LOW indicates a read bus cycle. DEN is always inactive when DT/$\overline{R}$ changes states. This output is HIGH when no bus cycle is active. DT/$\overline{R}$ is not affected by any of the control inputs.
$\overline{IOWC}$	O	**I/O WRITE COMMAND** instructs an I/O device to read the data on the data bus. This command output is active LOW. The MB and CMDLY inputs control when this output becomes active. $\overline{READY}$ controls when it becomes inactive.
$\overline{IORC}$	O	**I/O READ COMMAND** instructs an I/O device to place data onto the data bus. This command output is active LOW. The MB and CMDLY inputs control when this output becomes active. $\overline{READY}$ controls when it becomes inactive.
$\overline{MWTC}$	O	**MEMORY WRITE COMMAND** instructs a memory device to read the data on the data bus. This command output is active LOW. The MB and CMDLY inputs control when this output becomes active. $\overline{READY}$ controls when it becomes inactive.
$\overline{MRDC}$	O	**MEMORY READ COMMAND** instructs the memory device to place data onto the data bus. This command output is active LOW. The MB and CMDLY inputs control when this output becomes active. $\overline{READY}$ controls when it becomes inactive.

Table 1. Pin Description (Continued)

Symbol	Type	Name and Function
$\overline{INTA}$	I	**INTERRUPT ACKNOWLEDGE** tells an interrupting device that its interrupt request is being acknowledged. This command output is active LOW. The MB and CMDLY inputs control when this output becomes active. $\overline{READY}$ controls when it becomes inactive.
V_{CC}		**System Power:** +5V Power Supply
GND		**System Ground:** 0V

Table 2. Command and Control Outputs for Each Type of Bus Cycle

Type of Bus Cycle	M/$\overline{IO}$	$\overline{S1}$	$\overline{S0}$	Command Activated	DT/$\overline{R}$ State	ALE, DEN Issued?	MCE Issued?
Interrupt Acknowledge	0	0	0	$\overline{INTA}$	LOW	YES	YES
I/O Read	0	0	1	$\overline{IORC}$	LOW	YES	NO
I/O Write	0	1	0	$\overline{IOWC}$	HIGH	YES	NO
None; Idle	0	1	1	None	HIGH	NO	NO
Halt/Shutdown	1	0	0	None	HIGH	NO	NO
Memory Read	1	0	1	$\overline{MRDC}$	LOW	YES	NO
Memory Write	1	1	0	$\overline{MWTC}$	HIGH	YES	NO
None; Idle	1	1	1	None	HIGH	NO	NO

Operating Modes

Two types of buses are supported by the 82288: MULTIBUS and non-MULTIBUS. When the MB input is strapped HIGH, MULTIBUS timing is used. In MULTIBUS mode, the 82288 delays command and data activation to meet IEEE-796 requirements on address to command active and write data to command active setup timing. MULTIBUS mode requires at least one wait state in the bus cycle since the command outputs are delayed. The non-MULTIBUS mode does not delay any outputs and does not require wait states. The MB input affects the timing of the command and DEN outputs.

Command and Control Outputs

The type of bus cycle performed by the local bus master is encoded in the M/$\overline{IO}$, $\overline{S1}$, and $\overline{S0}$ inputs. Different command and control outputs are activated depending on the type of bus cycle. Table 2 indicates the cycle decode done by the 82288 and the effect on command, DT/$\overline{R}$, ALE, DEN, and MCE outputs.

Bus cycles come in three forms: read, write, and halt. Read bus cycles include memory read, I/O read, and interrupt acknowledge. The timing of the associated read command outputs ($\overline{MRDC}$, $\overline{IORC}$, and $\overline{INTA}$), control outputs (ALE, DEN, DT/$\overline{R}$) and control inputs (CEN/$\overline{AEN}$, CENL, CMDLY, MB, and $\overline{READY}$) are identical for all read bus cycles. Read cycles differ only in which command output is activated. The MCE control output is only asserted during interrupt acknowledge cycles.

Write bus cycles activate different control and command outputs with different timing than read bus cycles. Memory write and I/O write are write bus cycles whose timing for command outputs ($\overline{MWTC}$ and $\overline{IOWC}$), control outputs (ALE, DEN, DT/$\overline{R}$) and control inputs (CEN/$\overline{AEN}$, CENL, CMDLY, MB, and $\overline{READY}$) are identical. They differ only in which command output is activated.

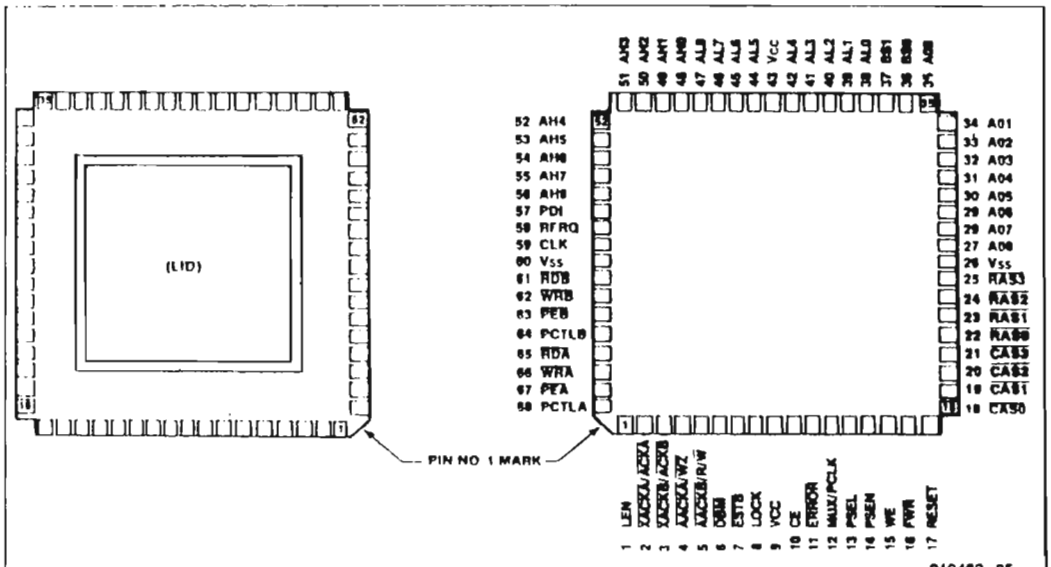
Halt bus cycles are different because no command or control output is activated. All control inputs are ignored until the next bus cycle is started via $\overline{S1}$ and $\overline{S0}$.

Table 1. Pin Description (Continued)

Symbol	Pin	Type	Name and Function
$\overline{ERROR}$	11	I	**ERROR:** This is an ECC input from the 8206 EDCU and instructs the 8207 that an error was detected. This pin should be connected to the ERROR output of the 8206.
MUX/ PCLK	12	O	**MULTIPLEXER CONTROL/PROGRAMMING CLOCK:** Immediately after a RESET this pin is used to clock serial programming data into the PDI pin. In normal two-port operation, this pin is used to select memory addresses from the appropriate port. When this signal is high, port A is selected and when it is low, port B is selected. This signal may change state before the completion of a RAM cycle, but the RAM address hold time is satisfied.
PSEL	13	O	**PORT SELECT:** This signal is used to select the appropriate port for data transfer. When this signal is high port A is selected and when it is low port B is selected.
PSEN	14	O	**PORT SELECT ENABLE:** This signal used in conjunction with PSEL provides contention-free port exchange on the data bus. When PSEN is low, port selection is allowed to change state.
WE	15	O	**WRITE ENABLE:** This signal provides the dynamic RAM array the write enable input for a write operation.
$\overline{FWR}$	16	I	**FULL WRITE:** This is an ECC input signal that instructs the 8207, in an ECC configuration, whether the present write cycle is normal RAM write (full write) or a RAM partial write (read-modify-write) cycle.
RESET	17	I	**RESET:** This signal causes all internal counters and state flip-flops to be reset and upon release of RESET, data appearing at the PDI pin is clocked in by the PCLK output. The states of the PDI, PCTLA, PCTLB and RFRQ pins are sampled by RESET going inactive and are used to program the 8207. An 8-cycle dynamic RAM warm-up is performed after clocking PDI bits into the 8207.
$\overline{CAS0}$–$\overline{CAS3}$	18–21	O	**COLUMN ADDRESS STROBE:** These outputs are used by the dynamic RAM array to latch the column address, present on the AO0–8 pins. These outputs are selected by the BS0 and BS1 as programmed by program bits RB0 and RB1. These outputs drive the dynamic RAM array directly and need no external drivers.
$\overline{RAS0}$–$\overline{RAS3}$	22–25	O	**ROW ADDRESS STROBE:** These outputs are used by the dynamic RAM array to latch the row address, present on the AO0–8 pins. These outputs are selected by the BS0 and BS1 as programmed by program bits RB0 and RB1. These outputs drive the dynamic RAM array directly and need no external drivers.
V_{SS}	26 60	I I	**DRIVER GROUND:** Provides a ground for the output drivers. **LOGIC GROUND:** Provides a ground for the remainder of the device.
AO0–AO8	35–27	O	**ADDRESS OUTPUTS:** These outputs are designed to provide the row and column addresses of the selected port to the dynamic RAM array. These outputs drive the dynamic RAM array directly and need no external drivers.
BS0–BS1	36–37	I	**BANK SELECT:** These inputs are used to select one of four banks of the dynamic RAM array as defined by the program bits RB0 and RB1.
AL0–AL8	38–42 44–47	I	**ADDRESS LOW:** These lower-order address inputs are used to generate the row address for the internal address multiplexer.
AH0–AH8	48–56	I	**ADDRESS HIGH:** These higher-order address inputs are used to generate the column address for the internal address multiplexer.

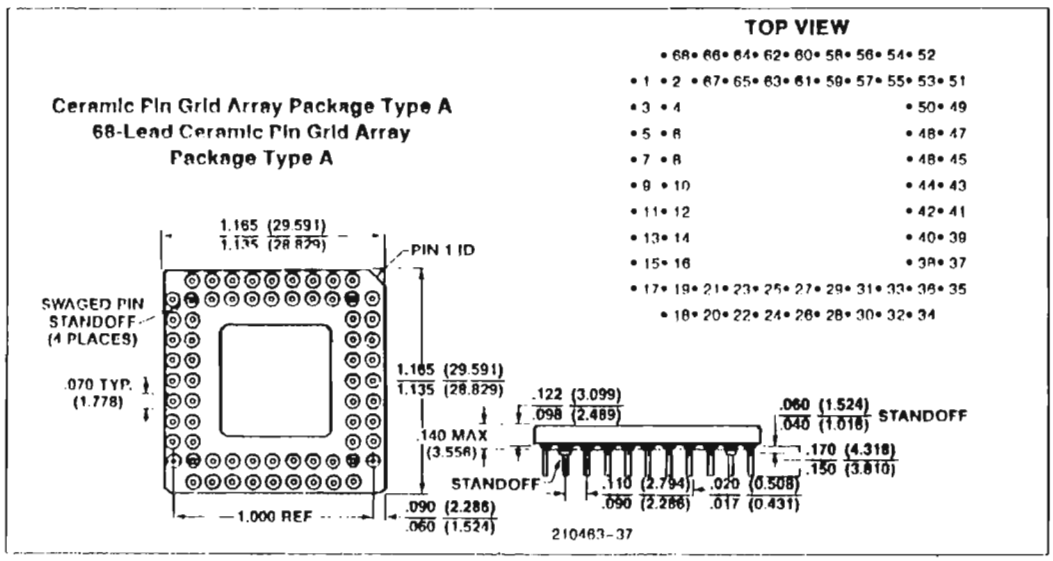
Table 1. Pin Description (Continued)

Symbol	Pin	Type	Name and Function
PDI	57	I	**PROGRAM DATA INPUT:** This input programs the various user-selectable options in the 8207. The PCLK pin shifts programming data into the PDI input from optional external shift registers. This pin may be strapped high or low to a default ECC (PDI = Logic "1") or non-ECC (PDI = Logic "0") mode configuration.
RFRQ	58	I	**REFRESH REQUEST:** This input is sampled on the falling edge of RESET. If it is high at RESET, then the 8207 is programmed for internal refresh request or external refresh request with failsafe protection. If it is low at RESET, then the 8207 is programmed for external refresh without failsafe protection or burst refresh. Once programmed the RFRQ pin accepts signals to start an external refresh with failsafe protection or external refresh without failsafe protection or a burst refresh.
CLK	59	I	**CLOCK:** This input provides the basic timing for sequencing the internal logic.
$\overline{RDB}$	61	I	**READ FOR PORT B:** This pin is the read memory request command input for port B. This input also directly accepts the $\overline{S1}$ status line from Intel processors.
$\overline{WRB}$	62	I	**WRITE FOR PORT B:** This pin is the write memory request command input for port B. This input also directly accepts the $\overline{S0}$ status line from Intel processors.
$\overline{PEB}$	63	I	**PORT ENABLE FOR PORT B:** This pin serves to enable a RAM cycle request for port B. It is generally decoded from the port address.
PCTLB	64	I	**PORT CONTROL FOR PORT B:** This pin is sampled on the falling edge of RESET. If low after RESET, the 8207 is programmed to accept memory read and write commands, Multibus commands or iAPX 286 status inputs. If high after RESET, the 8207 is programmed to accept status inputs from iAPX 86 or iAPX 186 processors. The $\overline{S2}$ status line should be connected to this input if programmed to accept iAPX 86 or iAPX 186 status inputs. When programmed to accept commands or iAPX 286 status, it should be tied low or it may be used as a Multibus-compatible inhibit signal.
$\overline{RDA}$	65	I	**READ FOR PORT A:** This pin is the read memory request command input for port A. This input also directly accepts the $\overline{S1}$ status line from Intel processors.
$\overline{WRA}$	66	I	**WRITE FOR PORT A:** This pin is the write memory request command input for port A. This input also directly accepts the $\overline{S0}$ status line from Intel processors.
$\overline{PEA}$	67	I	**PORT ENABLE FOR PORT A:** This pin serves to enable a RAM cycle request for port A. It is generally decoded from the port address.
PCTLA	68	I	**PORT CONTROL FOR PORT A:** This pin is sampled on the falling edge of RESET. If low after RESET, the 8207 is programmed to accept memory read and write commands, Multibus commands or iAPX 286 status inputs. If high after RESET, the 8207 is programmed to accept status inputs from iAPX 86 or iAPX 186 processors. The $\overline{S2}$ status line should be connected to this input if programmed to accept iAPX 86 or iAPX 186 status inputs. When programmed to accept commands or iAPX 286 status, it should be tied low or it may be connected to INHIBIT when operating with Multibus.



NOTE:
LCC is mounted lid-down into socket.

Figure 19. 8207 Pinout Diagram



8207 Pin Grid Array (PGA) Pin-Out

Packaging

The 8207 is packaged in a 68 lead JEDEC Type A Leadless Chip Carrier (LCC) and in Pin Grid Array (PGA), both in Ceramic. The package designations are R and A respectively.

eg: R 8207-8 LCC, 8 MHz DRAM Controller
eg: A 8207-16 PGA, 16 MHz DRAM Controller

NOTE:
The pin-out of the PGA is the same as the socketed pinout of the LCC.

PRELIMINARY

8254
PROGRAMMABLE INTERVAL TIMER

- Compatible with All Intel and Most Other Microprocessors
- Handles Inputs from DC to 10 MHz
  - 5 MHz 8254-5
  - 8 MHz 8254
  - 10 MHz 8254-2
- Status Read-Back Command
- Six Programmable Counter Modes
- Three Independent 16-Bit Counters
- Binary or BCD Counting
- Single +5V Supply
- Available in EXPRESS
  - Standard Temperature Range

The Intel® 8254 is a counter/timer device designed to solve the common timing control problems in microcomputer system design. It provides three independent 16-bit counters, each capable of handling clock inputs up to 10 MHz. All modes are software programmable. The 8254 is a superset of the 8253.

The 8254 uses HMOS technology and comes in a 24-pin plastic or CERDIP package.

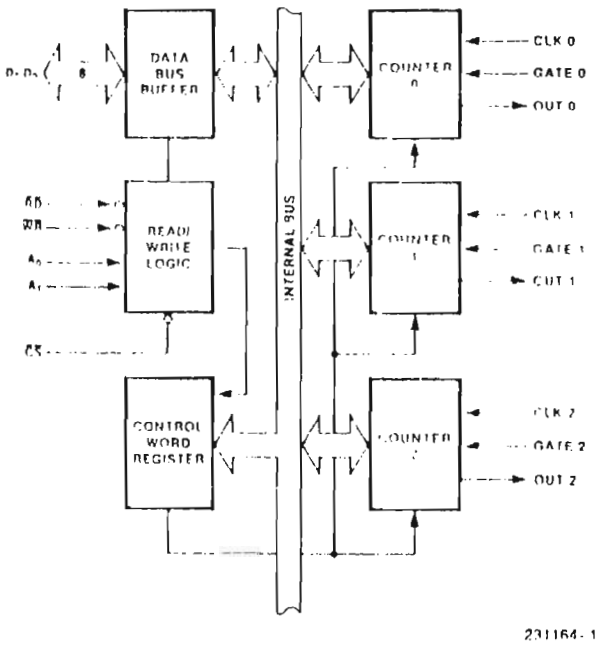


Figure 1. 8254 Block Diagram

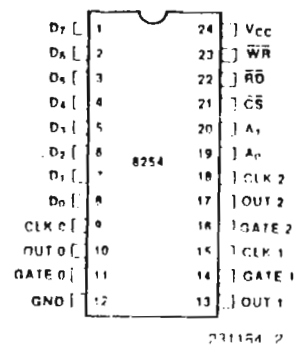


Figure 2. Pin Configuration

Table 1. Pin Description

Symbol	Pin No.	Type	Name and Function
D_7-D_0	1-8	I/O	**DATA:** Bi-directional three state data bus lines, connected to system data bus.
CLK 0	9	I	**CLOCK 0:** Clock input of Counter 0.
OUT 0	10	O	**OUTPUT 0:** Output of Counter 0.
GATE 0	11	I	**GATE 0:** Gate input of Counter 0.
GND	12		**GROUND:** Power supply connection.
V_{CC}	24		**POWER:** +5V power supply connection.
$\overline{WR}$	23	I	**WRITE CONTROL:** This input is low during CPU write operations.
$\overline{RD}$	22	I	**READ CONTROL:** This input is low during CPU read operations.
$\overline{CS}$	21	I	**CHIP SELECT:** A low on this input enables the 8254 to respond to $\overline{RD}$ and $\overline{WR}$ signals. $\overline{RD}$ and $\overline{WR}$ are ignored otherwise.
A_1, A_0	20-19	I	**ADDRESS:** Used to select one of the three Counters or the Control Word Register for read or write operations. Normally connected to the system address bus.

A_1	A_0	Selects
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word Register

CLK 2	18	I	**CLOCK 2:** Clock input of Counter 2.
OUT 2	17	O	**OUT 2:** Output of Counter 2.
GATE 2	16	I	**GATE 2:** Gate input of Counter 2.
CLK 1	15	I	**CLOCK 1:** Clock input of Counter 1.
GATE 1	14	I	**GATE 1:** Gate input of Counter 1.
OUT 1	13	O	**OUT 1:** Output of Counter 1.

FUNCTIONAL DESCRIPTION

General

The 8254 is a programmable interval timer/counter designed for use with Intel microcomputer systems. It is a general purpose, multi timing element that can be treated as an array of I/O ports in the system software.

The 8254 solves one of the most common problems in any microcomputer system, the generation of accurate time delays under software control. Instead of setting up timing loops in software, the programmer configures the 8254 to match his requirements and programs one of the counters for the desired delay. After the desired delay, the 8254 will interrupt the CPU. Software overhead is minimal and variable length delays can easily be accommodated.

Some of the other counter/timer functions common to microcomputers which can be implemented with the 8254 are:

- Real time clock
- Event-counter
- Digital one-shot
- Programmable rate generator
- Square wave generator
- Binary rate multiplier
- Complex waveform generator
- Complex motor controller

Block Diagram

DATA BUS BUFFER

This 3-state, bi-directional, 8-bit buffer is used to interface the 8254 to the system bus (see Figure 3).

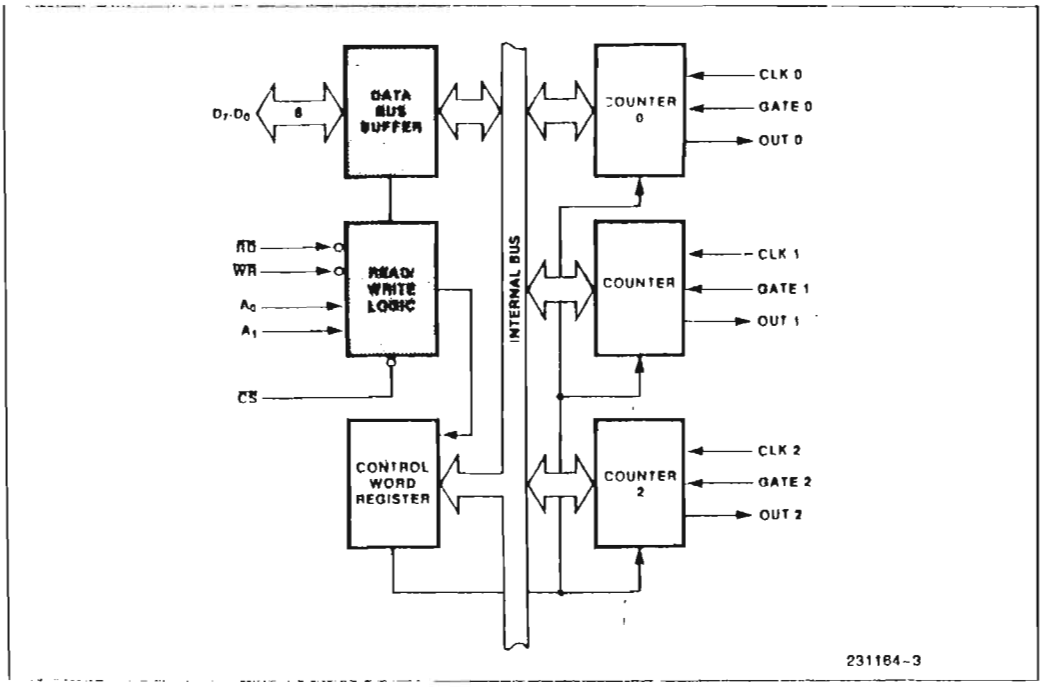


Figure 3. Block Diagram Showing Data Bus Buffer and Read/Write Logic Functions

READ/WRITE LOGIC

The Read/Write Logic accepts inputs from the system bus and generates control signals for the other functional blocks of the 8254. A_1 and A_0 select one of the three counters or the Control Word Register to be read from/written into. A "low" on the $\overline{RD}$ input tells the 8254 that the CPU is reading one of the counters. A "low" on the $\overline{WR}$ input tells the 8254 that the CPU is writing either a Control Word or an initial count. Both $\overline{RD}$ and $\overline{WR}$ are qualified by $\overline{CS}$; $\overline{RD}$ and $\overline{WR}$ are ignored unless the 8254 has been selected by holding $\overline{CS}$ low.

CONTROL WORD REGISTER

The Control Word Register (see Figure 4) is selected by the Read/Write Logic when $A_1,A_0 = 11$. If the CPU then does a write operation to the 8254, the data is stored in the Control Word Register and is interpreted as a Control Word used to define the operation of the Counters.

The Control Word Register can only be written to; status information is available with the Read-Back Command.

COUNTER 0, COUNTER 1, COUNTER 2

These three functional blocks are identical in operation, so only a single Counter will be described. The internal block diagram of a single counter is shown in Figure 5.

The Counters are fully independent. Each Counter may operate in a different Mode.

The Control Word Register is shown in the figure; it is not part of the Counter itself, but its contents determine how the Counter operates.

The status register, shown in Figure 5, when latched, contains the current contents of the Control Word Register and status of the output and null count flag. (See detailed explanation of the Read-Back command.)

The actual counter is labelled CE (for "Counting Element"). It is a 16-bit presettable synchronous down counter.

OL_M and OL_L are two 8-bit latches. OL stands for "Output Latch"; the subscripts M and L stand for "Most significant byte" and "Least significant byte"

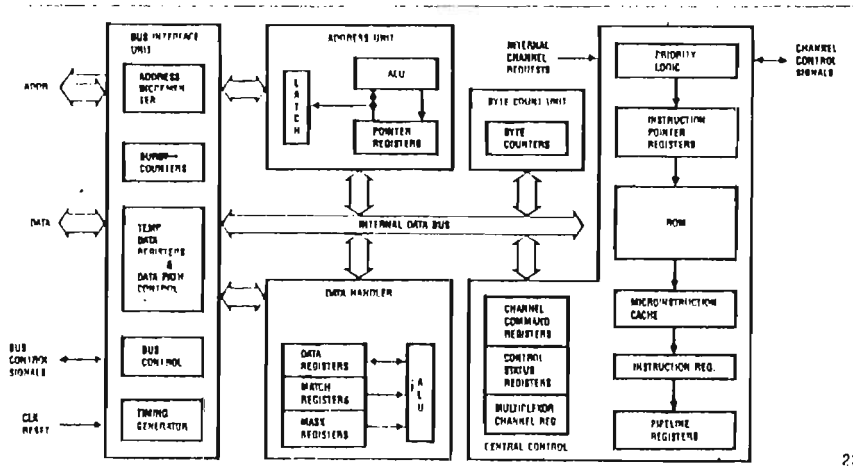
82258
ADVANCED DIRECT MEMORY ACCESS COPROCESSOR (ADMA)

- High Performance 16 Bit DMA Coprocessor for the 80386, 80286 and 80186 Families
  - 8 MByte/sec Maximum Transfer Rate in 8 MHz 80286 Systems
- Four Independently Programmable Channels
- Multiplexor Channel Capability to Support Up to 32 Subchannels
- On Chip Bus Interface for the Whole 8086 Architecture
  - 80286
  - 80186/188
  - 8086/88
- Command Chaining for CPU Independent Processing
- Automatic Data Chaining for Gathering and Scattering of Data Blocks
- 16 MByte Addressing Range
- 16 MByte Block Transfer Capability
- "On the Fly" Compare, Translate and Verify Operations
- Automatic Assembly/Disassembly of Data
- Programmable Bus Loading
- 6 and 8 MHz Speed Selections
- Available in 68-Pin LCC and PGA Packages

(See Packaging Spec, Order # 231369)

INTRODUCTION

Intel's 82258, Advanced Direct Memory Access Coprocessor is a high performance, 16 bit DMA processor optimized for the 80286, 80186 and the 8086 families of CPUs and compatible with 80386 CPU. It has on-chip bus interface for the whole 8086 family architecture. Four high speed, independently programmable DMA channels can achieve a maximum cumulative transfer rate of 8 MByte/sec in an 8 MHz 80286 system and 4 MByte/sec in 8 MHz 8086/8018[illegible]ystems. Channel 3 can be used as a Multiplexor channel, whereby, it supports 32 subchannels. This flexibility allows one to use a single DMA channel to handle a large number of slow and medium speed I/O devices. Advanced capabilities like Command and Data chaining and "On the fly" operations allow the 82258 to remove the I/O management load from the processor. The 82258 addresses the full 80286 CPU memory (16 MB for 80286), thus simplifying the system design. Automatic assembly/disassembly of data allows 16 bit processors to interface with common 8 bit peripherals and vice-versa. Remote mode of operation, where the 82258 has its own resident bus, allows modular system design. The 82258 complements the high performance, multitasking capabilities of the 80286.



231263-1

Figure 1. 82258 Internal Block Diagram

FABRICATION

The 82258 is a 68 pin device, fabricated in Intel HMOS II technology. It is packaged in JEDEC type A hermetic leadless chip carrier and pin grid array.

PIN DEFINITIONS AND FUNCTIONS

The 82258 has four operational modes

286

186 for the 80186/88 and the 8086/88 (Min mode) CPUs

8086 for the 8086/88 (Max. mode) CPUs

Bypass

Pins of the 82258 have different definitions for different modes. 286 and remote modes have the same non multiplexed bus structure and similar pin descriptions. Similarly, the 186 and the 8086 modes have multiplexed bus and similar pin description.

PINNING IN THE 286 MODE

In the 286 mode, the bus signals and the bus timings of the 82258 are the same as those of the 80286 processor. The processor can access the internal registers of the 82258 and these accesses must be supported by the bus signals. Therfore, some of the bus control signals are bidirectional and some additional bus control signals are necessary.

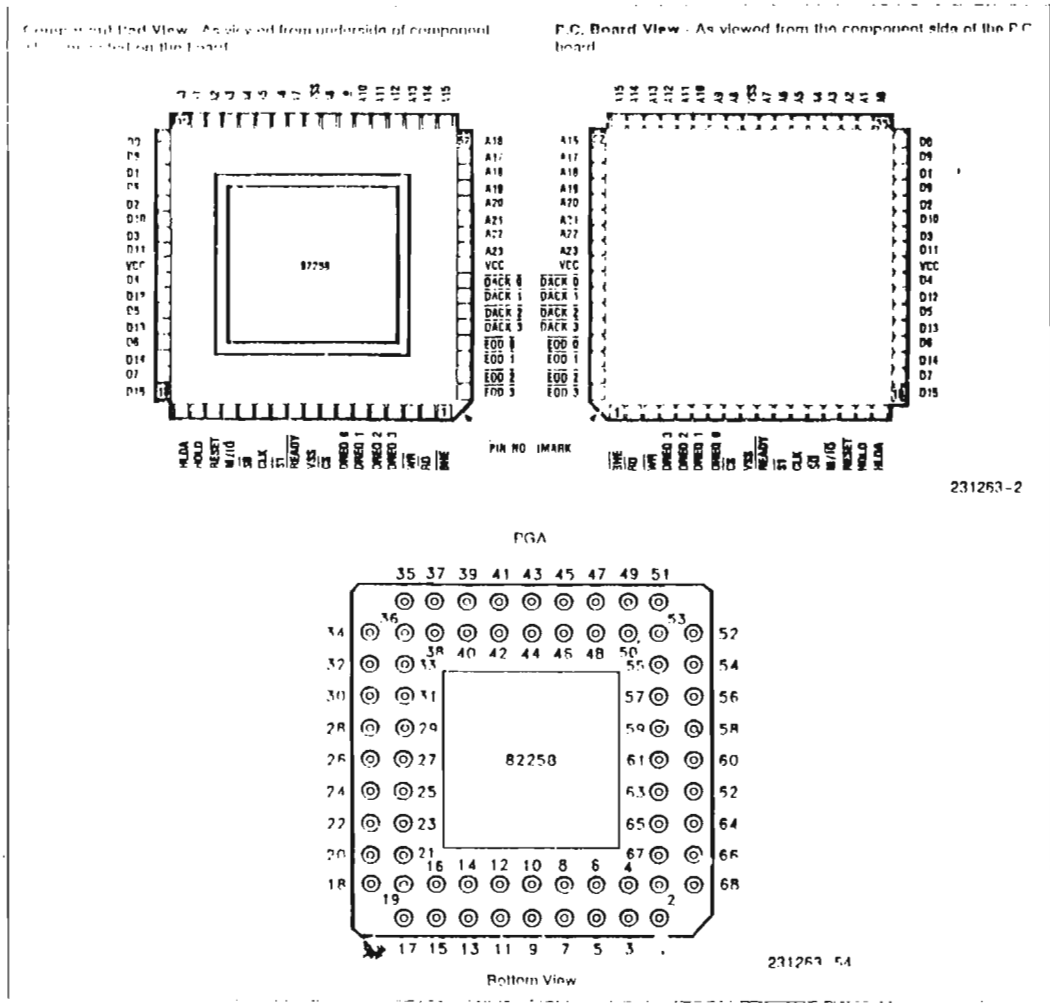


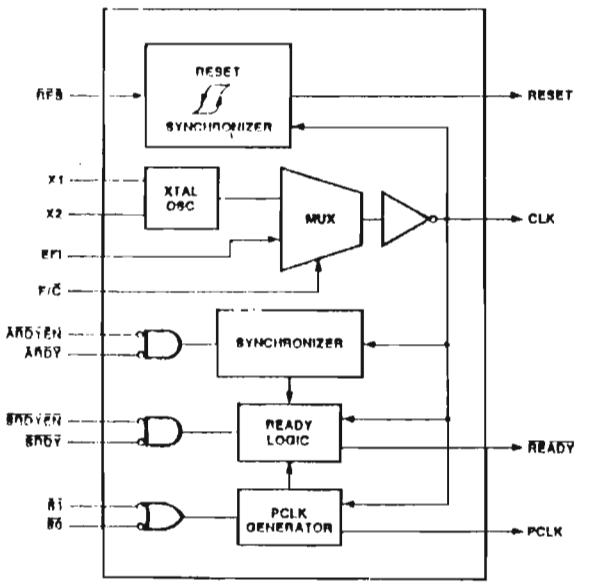
Figure 2. Pin Configuration In 286 Mode

ADVANCE INFORMATION

82C284
CLOCK GENERATOR AND READY INTERFACE FOR 80286 PROCESSORS
(82C284-12, 82C284-10, 82C284-8, 82C284-6)

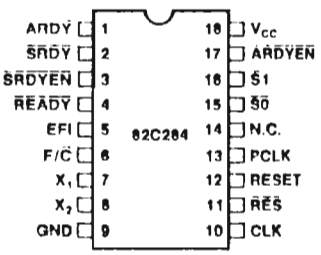
- Generates System Clock for 80286 Processors
- Uses Crystal or TTL Signal for Frequency Source
- Provides Local READY and MULTIBUS®* READY Synchronization
- Single +5V Power Supply
- CHMOS III Technology
- Generates System Reset Output from Schmitt Trigger Input
- Available in EXPRESS
  - Standard Temperature Range
  - Extended Temperature Range
- Available in 18-Lead Cerdip Package
  (See Packaging Spec, Order #231369)

The 82C284 is a clock generator/driver which provides clock signals for 80286 processors and support components. It also contains logic to supply $\overline{\text{READY}}$ to the CPU from either asynchronous or synchronous sources and synchronous RESET from an asynchronous input with hysteresis.



210453-1

Figure 1. 82C284 Block Diagram



210453-2

Figure 2. 82C284 Pin Configuration

*MULTIBUS is a patented bus of Intel.

Table 1. Pin Description

The following pin function descriptions are for the 82C284 clock generator.

Symbol	Type	Name and Function
CLK	O	**SYSTEM CLOCK** is the signal used by the processor and support devices which must be synchronous with the processor. The frequency of the CLK output has twice the desired internal processor clock frequency. CLK can drive both TTL and MOS level inputs.
F/$\overline{C}$	I	**FREQUENCY/CRYSTAL SELECT** is a strapping option to select the source for the CLK output. When F/$\overline{C}$ is strapped LOW, the internal crystal oscillator drives CLK. When F/$\overline{C}$ is strapped HIGH, the EFI input drives the CLK output.
X1, X2	I	**CRYSTAL IN** are the pins to which a parallel resonant fundamental mode crystal is attached for the internal oscillator. When F/$\overline{C}$ is LOW, the internal oscillator will drive the CLK output at the crystal frequency. The crystal frequency must be twice the desired internal processor clock frequency.
EFI	I	**EXTERNAL FREQUENCY IN** drives CLK when the F/$\overline{C}$ input is strapped HIGH. The EFI input frequency must be twice the desired internal processor clock frequency.
PCLK	O	**PERIPHERAL CLOCK** is an output which provides a 50% duty cycle clock with 1/2 the frequency of CLK. PCLK will be in phase with the internal processor clock following the first bus cycle after the processor has been reset.
$\overline{ARDYEN}$	I	**ASYNCHRONOUS READY ENABLE** is an active LOW input which qualifies the $\overline{ARDY}$ input. $\overline{ARDYEN}$ selects $\overline{ARDY}$ as the source of ready for the current bus cycle. Inputs to $\overline{ARDYEN}$ may be applied asynchronously to CLK. Setup and hold times are given to assure a guaranteed response to synchronous inputs.
$\overline{ARDY}$	I	**ASYNCHRONOUS READY** is an active LOW input used to terminate the current bus cycle. The $\overline{ARDY}$ input is qualified by $\overline{ARDYEN}$. Inputs to $\overline{ARDY}$ may be applied asynchronously to CLK. Setup and hold times are given to assure a guaranteed response to synchronous outputs.
$\overline{SRDYEN}$	I	**SYNCHRONOUS READY ENABLE** is an active LOW input which qualifies $\overline{SRDY}$. $\overline{SRDYEN}$ selects $\overline{SRDY}$ as the source for $\overline{READY}$ to the CPU for the current bus cycle. Setup and hold times must be satisfied for proper operation.
$\overline{SRDY}$	I	**SYNCHRONOUS READY** is an active LOW input used to terminate the current bus cycle. The $\overline{SRDY}$ input is qualified by the $\overline{SRDYEN}$ input. Setup and hold times must be satisfied for proper operation.
$\overline{READY}$	O	**READY** is an active LOW output which signals the current bus cycle is to be completed. The $\overline{SRDY}$, $\overline{SRDYEN}$, $\overline{ARDY}$, $\overline{ARDYEN}$, $\overline{S1}$, $\overline{S0}$ and $\overline{RES}$ inputs control $\overline{READY}$ as explained later in the $\overline{READY}$ generator section. $\overline{READY}$ is an open drain output requiring an external pull-up resistor.

Table 1. Pin Description (Continued)

The following pin function descriptions are for the 82C284 clock generator.

Symbol	Type	Name and Function
$\overline{S0}$, $\overline{S1}$	I	**STATUS** input prepare the 82284 for a subsequent bus cycle. $\overline{S0}$ and $\overline{S1}$ synchronize PCLK to the internal processor clock and control $\overline{READY}$. These inputs have internal pull-up resistors to keep them HIGH if nothing is driving them. Setup and hold times must be satisfied for proper operation.
RESET	O	**RESET** is an active HIGH output which is derived from the $\overline{RES}$ input. RESET is used to force the system into an initial state. When RESET is active, $\overline{READY}$ will be active (LOW).
$\overline{RES}$	I	**RESET IN** is an active LOW input which generates the system reset signal, RESET. Signals to $\overline{RES}$ may be applied asynchronously to CLK. A Schmitt trigger input is provided on $\overline{RES}$, so that an RC circuit can be used to provide a time delay. Setup and hold times are given to assure a guaranteed response to synchronous inputs.
V_{CC}		**SYSTEM POWER:** +5V Power Supply
GND		**SYSTEM GROUND:** 0V

FUNCTIONAL DESCRIPTION

Introduction

The 82C284 generates the clock, ready, and reset signals required for 80286 processors and support components. The 82C284 is packaged in an 18-pin DIP and contains a crystal controlled oscillator, clock generator, peripheral clock generator, Multibus ready synchronization logic and system reset generation logic.

Clock Generator

The CLK output provides the basic timing control for an 80286 system. CLK has output characteristics sufficient to drive MOS devices. CLK is generated by either an internal crystal oscillator or an external source as selected by the F/$\overline{C}$ strapping option. When F/$\overline{C}$ is LOW, the crystal oscillator drives the CLK output. When F/$\overline{C}$ is HIGH, the EFI input drives the CLK output.

The 82C284 provides a second clock output, PCLK, for peripheral devices. PCLK is CLK divided by two. PCLK has a duty cycle of 50% and MOS output drive characteristics. PCLK is normally synchronized to the internal processor clock.

After reset, the PCLK signal may be out of phase with the internal processor clock. The $\overline{S1}$ and $\overline{S0}$ signals of the first bus cycle are used to synchronize PCLK to the internal processor clock. The phase of the PCLK output changes by extending its HIGH time beyond one system clock (see waveforms). PCLK is forced HIGH whenever either $\overline{S0}$ or $\overline{S1}$ were active (LOW) for the two previous CLK cycles. PCLK continues to oscillate when both $\overline{S0}$ and $\overline{S1}$ are HIGH.

Since the phase of the internal processor clock will not change except during reset, the phase of PCLK will not change except during the first bus cycle after reset.

Oscillator

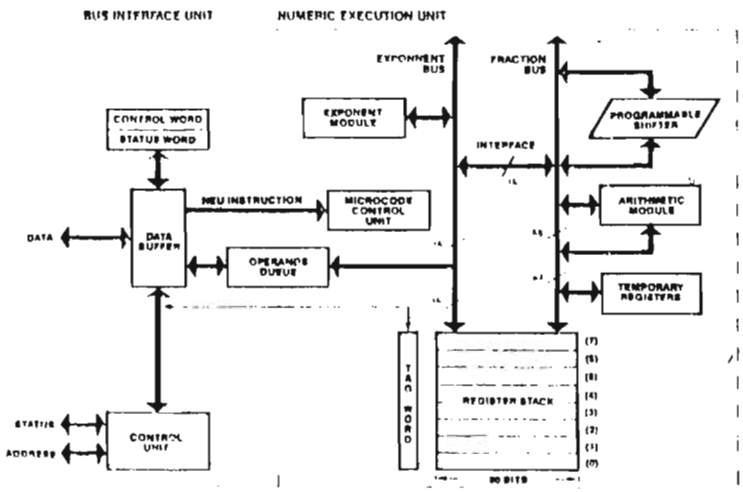
The oscillator circuit of the 82C284 is a linear Pierce oscillator which requires an external parallel resonant, fundamental mode, crystal. The output of the oscillator is internally buffered. The crystal frequency chosen should be twice the required internal processor clock frequency. The crystal should have a typical load capacitance of 32 pF.

X1 and X2 are the oscillator crystal connections. For stable operation of the oscillator, two loading capacitors are recommended, as shown in Table 2. The sum of the board capacitance and loading capacitance should equal the values shown. It is advisable to limit stray board capacitances (not including the effect of the loading capacitors or crystal capacitance) to less than 10 pF between the X1 and X2 pins. Decouple V_{CC} and GND as close to the 82C284 as possible.

80287
80-BIT HMOS
NUMERIC PROCESSOR EXTENSION
(80287-3, 80287-6, 80287-8, 80287-10)

- High Performance 80-Bit Internal Architecture
- Implements Proposed IEEE Floating Point Standard 754
- Expands 80286 Data types to Include 32-, 64-, 80-Bit Floating Point, 32-, 64-Bit Integers and 18-Digit BCD Operands
- Object Code Compatible with 8087
- Built-In Exception Handling
- Operates in Both Real and Protected Mode 80286 Systems
- 8x80-Bit, Individually Addressable, Numeric Register Stack
- Protected Mode Operation Completely Conforms to the 80286 Memory Management and Protection Mechanisms
- Directly Extends 80286 Instruction Set to Trigonometric, Logarithmic, Exponential and Arithmetic Instructions for All Data types
- Operates with 80386 CPU without Software Modification
- Available in EXPRESS—Standard Temperature Range
- Available in 40 pin-CERDIP package (see Packaging Spec, Order #231369)

The Intel 80287 is a high performance numerics processor extension that extends the 80286 architecture with floating point, extended integer and BCD data types. The 80286/80287 computing system fully conforms to the proposed IEEE Floating Point Standard. Using a numerics oriented architecture, the 80287 adds over fifty mnemonics to the 80286/80287 instruction set, making the 80286/80287 a complete solution for high performance numeric processing. The 80287 is implemented in N-channel, depletion load, silicon gate technology (HMOS) and packaged in a 40-pin cerdip package. The 80286/80287 is object code compatible with the 8086/8087 and 8088/8087.



210920-1

Figure 1. 80287 Block Diagram

210920-2

NOTE:
N/C Pins should not be connected

Figure 2.
80287 Pin Configuration

Table 1. 80287 Pin Description

Symbols	Type	Name and Function
CLK	I	**CLOCK INPUT:** this clock provides the basic timing for internal 80287 operations. Special MOS level inputs are required. The 82284 or 8284A CLK outputs are compatible to this input.
CKM	I	**CLOCK MODE SIGNAL:** indicates whether CLK input is to be divided by 3 or used directly. A HIGH input will cause CLK to be used directly. This input must be connected to V_{CC} or V_{SS} as appropriate. This input must be either HIGH or LOW 20 CLK cycles before RESET goes LOW.
RESET	I	**SYSTEM RESET:** causes the 80287 to immediately terminate its present activity and enter a dormant state. RESET is required to be HIGH for more than 4 80287 CLK cycles. For proper initialization the HIGH-LOW transition must occur no sooner than 50 μs after V_{CC} and CLK meet their D.C. and A.C. specifications.
D15–D0	I/O	**DATA:** 16-bit bidirectional data bus. Inputs to these pins may be applied asynchronous to the 80287 clock.
$\overline{BUSY}$	O	**BUSY STATUS:** asserted by the 80287 to indicate that it is currently executing a command.
$\overline{ERROR}$	O	**ERROR STATUS:** reflects the ES bit of the status word. This signal indicates that an unmasked error condition exists.
PEREQ	O	**PROCESSOR EXTENSION DATA CHANNEL OPERAND TRANSFER REQUEST:** a HIGH on this output indicates that the 80287 is ready to transfer data. PEREQ will be disabled upon assertion of $\overline{PEACK}$ or upon actual data transfer, whichever occurs first, if no more transfers are required.
$\overline{PEACK}$	I	**PROCESSOR EXTENSION DATA CHANNEL OPERAND tRANSFER ACKNOWLEDGE:** acknowledges that the request signal (PEREQ) has been recognized. Will cause the request (PEREQ) to be withdrawn in case there are no more transfers required. $\overline{PEACK}$ may be asynchronous to the 80287 clock.
$\overline{NPRD}$	I	**NUMERIC PROCESSOR READ:** Enables transfer of data from the 80287. This input may be asynchronous to the 80287 clock.
$\overline{NPWR}$	I	**NUMERIC PROCESSOR READ:** Enables transfer of data from the 80287. This input may be asynchronous to the 80287 clock.
$\overline{NPS1}$, NPS2	I	**NUMERIC PROCESSOR SELECTS:** indicate the CPU is performing an ESCAPE instruction. Concurrent assertion of these signals (i.e., $\overline{NPS1}$ is LOW and NPS2 is HIGH) enables the 80287 to perform floating point instrucctions. No data transfers involving the 80287 will occur unless the device is selected via these lines. These inputs may be asynchronous to the 80287 clock.
CMD1, CMD0	I	**COMMAND LINES:** These, along with select inputs, allow the CPU to direct the operation of the 80287. These inputs may be asynchronous to the 80287 clock.

Table 1. 80187 Pin Description (Continued)

Symbols	Type	Name and Function
V_{SS}	I	System ground, both pins must be connected to ground.
V_{CC}	I	+5V supply

FUNCTIONAL DESCRIPTION

The 80287 Numeric Processor Extension (NPX) provides arithmetic instructions for a variety of numeric data types in 80286/80287 systems. It also executes numerous built-in transcendental functions (e.g., tangent and log functions). The 80287 executes instructions in parallel with an 80286. It effectively extends the register and instruction set of an 80286 system for existing 80286 data types and adds several new data types as well. Figure 3 presents the program visible register model of the 80286/80287. Essentially, the 80287 can be treated as an additional resource or an extension to the 80286 that can be used as a single unified system, the 80286/80287.

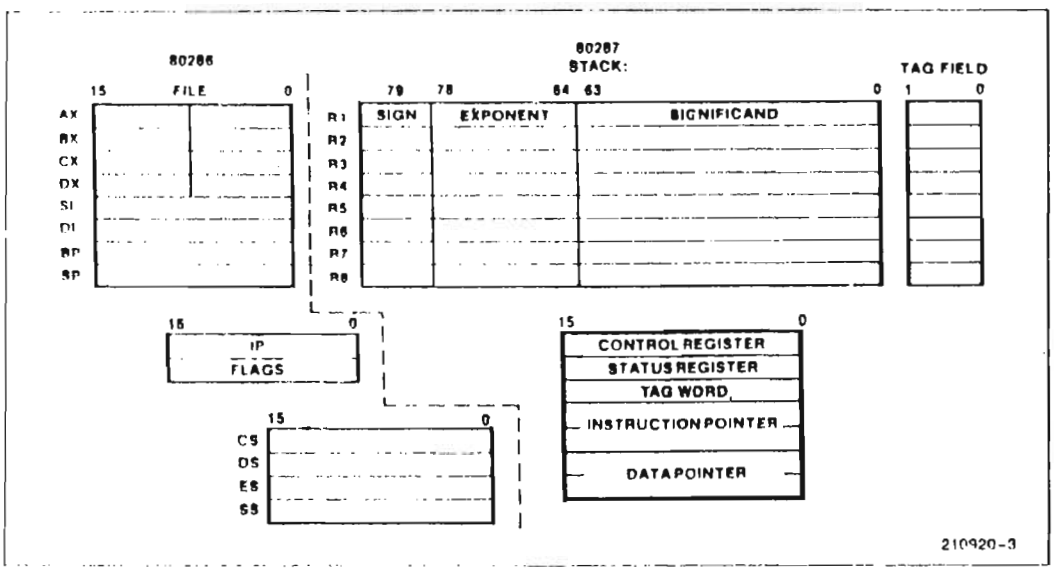


Figure 3. 80286/80287 Architecture

The 80287 has two operating modes similar to the two modes of the 80286. When reset, 80287 is in the real address mode. It can be placed in the protected virtual address mode by executing the SETPM ESC instruction. The 80287 cannot be switched back to the real address mode except by reset. In the real address mode, the 80286/80287 is completely software compatible with 8086/8087 and 8088/8087.

Once in protected mode, all references to memory for numerics data or status information, obey the 80286 memory management and protection rules giving a fully protected extension of the 80286 CPU. In the protected mode, 80286/80287 numerics software is also completely compatible with 8086/8087 and 8088/8087.

Table 6. 80287 Extensions to the 80286 Instruction Set (Continued)

Constants		Optional 8,16 Bit Displacement	Clock Count Range 32 Bit Real	32 Bit Integer	64 Bit Real	16 Bit Integer
	MF r		00	01	10	11
FLDZ LOAD +0.0 into ST(0)	ESCAPE 0 0 1 | 1 1 1 0 1 1 1 0		11-17			
FLD1 LOAD +1.0 into ST(0)	ESCAPE 0 0 1 | 1 1 1 0 1 0 0 0		15-21			
FLDPI LOAD π into ST(0)	ESCAPE 0 0 1 | 1 1 1 0 1 0 1 1		16-22			
FLDL2T LOAD $\log_2 10$ into ST(0)	ESCAPE 0 0 1 | 1 1 1 0 1 0 0 1		16-22			
FLDL2E LOAD $\log_2 e$ into ST(0)	ESCAPE 0 0 1 | 1 1 1 0 1 0 1 0		15-21			
FLDLG2 LOAD $\log_{10} 2$ into ST(0)	ESCAPE 0 0 1 | 1 1 1 0 1 1 0 0		18-24			
FLDLN2 LOAD $\log_e 2$ into ST(0)	ESCAPE 0 0 1 | 1 1 1 0 1 1 0 1		17-23			
Arithmetic						
FADD Addition						
Integer/Real Memory with ST(0)	ESCAPE MF 0 | MOD 0 0 0 R/M	DISP	90-120	108-143	95-125	102-137
ST(i) and ST(0)	ESCAPE d P 0 | 1 1 0 0 0 ST(i)		70-100 (Note 1)			
FSUB Subtraction						
Integer/Real Memory with ST(0)	ESCAPE MF 0 | MOD 1 0 R R/M	DISP	90-120	108-143	95-125	102-137
ST(i) and ST(0)	ESCAPE d P 0 | 1 1 1 0 R R/M		70-100 (Note 1)			
FMUL Multiplication						
Integer/Real Memory with ST(0)	ESCAPE MF 0 | MOD 0 0 1 R/M	DISP	110-125	130-144	112-168	124-138
ST(i) and ST(0)	ESCAPE d P 0 | 1 1 0 0 1 R/M		90-145 (Note 1)			
FDIV Division						
Integer/Real Memory with ST(0)	ESCAPE MF 0 | MOD 1 1 R R/M	DISP	215-225	230-243	220-230	224-238
ST(i) and ST(0)	ESCAPE d P 0 | 1 1 1 1 R R/M		193-203 (Note 1)			
FSQRT Square Root of ST(0)	ESCAPE 0 0 1 | 1 1 1 1 1 0 1 0		180-186			
FSCALE Scale ST(0) by ST(1)	ESCAPE 0 0 1 | 1 1 1 1 1 1 0 1		32-38			
FPREM Partial Remainder of ST(0) ÷ ST(1)	ESCAPE 0 0 1 | 1 1 1 1 1 0 0 0		15-190			
FRNDINT Round ST(0) to Integer	ESCAPE 0 0 1 | 1 1 1 1 1 1 0 0		16-50			

210920-10

NOTE:
1. If P = 1 then add 5 clocks

Table 6. 80287 Extensions to the 80286 Instruction Set (Continued)

		Optional 8,16 Bit Displacement	Clock Count Range
FXTRACT – Extract Components of ST(0)	ESCAPE 0 0 1 | 1 1 1 1 0 1 0 0		27–55
FABS – Absolute Value of ST(0)	ESCAPE 0 0 1 | 1 1 1 0 0 0 0 1		10–17
FCHS – Change Sign of ST(0)	ESCAPE 0 0 1 | 1 1 1 0 0 0 0 0		10–17
Transcendental			
FPTAN – Partial Tangent of ST(0)	ESCAPE 0 0 1 | 1 1 1 1 0 0 1 0		30–540
FPATAN – Partial Arctangent of ST(0), ST(1)	ESCAPE 0 0 1 | 1 1 1 1 0 0 1 1		250–800
F2XM1 – $2^{ST(0)} - 1$	ESCAPE 0 0 1 | 1 1 1 1 0 0 0 0		310–630
FYL2X – $ST(1) \cdot Log_2 [ST(0)]$	ESCAPE 0 0 1 | 1 1 1 1 0 0 0 1		900–1100
FYL2XP1 – $ST(1) \cdot Log_2 [ST(0) + 1]$	ESCAPE 0 0 1 | 1 1 1 1 1 0 0 1		700–1000
Processor Control			
FINIT – Initialize NPX	ESCAPE 0 1 1 | 1 1 1 0 0 0 1 1		2–8
FSETPM – Enter Protected Mode	ESCAPE 0 1 1 | 1 1 1 0 0 1 0 0		2–8
FSTSWAX – Store Control Word	ESCAPE 1 1 1 | 1 1 1 0 0 0 0 0		10–18
FLDCW – Load Control Word	ESCAPE 0 0 1 | MOD 1 0 1 R/M	DISP	7–14
FSTCW – Store Control Word	ESCAPE 0 0 1 | MOD 1 1 1 R/M	DISP	12–18
FSTSW – Store Status Word	ESCAPE 1 0 1 | MOD 1 1 1 R/M	DISP	12–18
FCLEX – Clear Exceptions	ESCAPE 0 1 1 | 1 1 1 0 0 0 1 0		2–8
FSTENV – Store Environment	ESCAPE 0 0 1 | MOD 1 1 0 R/M	DISP	40–50
FLDENV – Load Environment	ESCAPE 0 0 1 | MOD 1 0 0 R/M	DISP	35–45
FSAVE – Save State	ESCAPE 1 0 1 | MOD 1 1 0 R/M	DISP	205–215
FRSTOR – Restore State	ESCAPE 1 0 1 | MOD 1 0 0 R/M	DISP	205–215
FINCSTP – Increment Stack Pointer	ESCAPE 0 0 1 | 1 1 1 1 0 1 1 1		6–12
FDECSTP – Decrement Stack Pointer	ESCAPE 0 0 1 | 1 1 1 1 0 1 1 0		6–12

210920-21

ANEXO 7

COMUNICACIONES EN LENGUAJE C

COMUNICACIONES EN LENGUAJE C

El lenguaje C no tiene funciones estadar o procedimientos que esten diseñados para utilizarse con comunicaciones en serie. Por lo tanto la mayor parte de la programación se debe hacer con el sistema operativo o con llamadas a nivel de hardware. No se debe de esperar mucha ayuda de las librerias suministradas con los compiladores, aunque se puede comprar librerias especiales separadamente.

Los diferentes compiladores tienen diferentes métodos para el acceso de funciones de bajo nivel, y por lo tanto, cualquier discusión en relación a las comunicaciones serie, debe necesariamente referirse a un compilador especifico. En este anexo se hara referencia al Microsoft C versión 3 para IBM PC, que de aqui en adelante sera referenciado como MSC.

NOTA: Existen varios métodos de diseñar programas para comunicaciones serie, en este anexo solo se hara referencia al método que realiza comunicaciones serie por medio de UART, si se desea estudiar los métodos que utilizan las rutinas del DOS y del BIOS, consultar el libro que aparece como primera referencia en la bibliografia.

PROGRAMACION DEL UART POR MEDIO DEL C

En general para accesar los registros del INS 8250 directamente, es necesario poder utilizar las instrucciones IN y OUT del 8088, el MSC suministra dos funciones para hacer exactamente esto:

i, =inp(port); lee un byte desde el puerto (entero sin signo)
outp(port,byte); envia el valor (entero) al puerto

Referirse a la tabla 7.11, la cual da los direcciones I/O para los diferentes registros del UART en COM1 y COM2. Ahi se vera que el registro de estado de linea para COM1 está en 3FDh.
Para leerlo se puede entrar,

```
unsigned portad =0x3FD;
int lsbatus;
lsbatus =inp(portad);
```

El registro de control del modem para COM2 esta en 2F0h. Para enviar el byte modbyte a el, se puede escribir:

```
unsigned portad = 0x2F0;
    outp(portad,modbyte);
```

ANEXO 8

JUEGO DE INSTRUCCIONES DEL 80286

80286 INSTRUCTION SET SUMMARY

FUNCTION	FORMAT	CLOCK COUNT: Real Address Mode	CLOCK COUNT: Protected Virtual Address Mode	COMMENTS: Real Address Mode	COMMENTS: Protected Virtual Address Mode
DATA TRANSFER					
MOV = Move:					
Register to Register/Memory	1 0 0 0 1 0 0 w | mod reg r/m	2,3*	2,3*	2	9
Register/memory to register	1 0 0 0 1 0 1 w | mod reg r/m	2,5*	2,5*	2	9
Immediate to register/memory	1 1 0 0 0 1 1 w | mod 0 0 0 r/m | data | data if w = 1	2,3*	2,3*	2	9
Immediate to register	1 0 1 1 w reg | data | data if w = 1	2	2		
Memory to accumulator	1 0 1 0 0 0 0 w | addr-low | addr-high	5	5	2	9
Accumulator to memory	1 0 1 0 0 0 1 w | addr-low | addr-high	3	3	2	9
Register/memory to segment register	1 0 0 0 1 1 1 0 | mod 0 reg r/m	2,5*	17,19*	2	9,10,11
Segment register to register/memory	1 0 0 0 1 1 0 0 | mod 0 reg r/m	2,3*	2,3*	2	9
PUSH = Push:					
Memory	1 1 1 1 1 1 1 1 | mod 1 1 0 r/m	5*	5*	2	9
Register	0 1 0 1 0 reg	3	3	2	9
Segment register	0 0 0 reg 1 1 0	3	3	2	9
Immediate	0 1 1 0 1 0 s 0 | data | data if s = 0	3	3	2	9
PUSHA = Push All	0 1 1 0 0 0 0 0	17	17	2	9
POP = Pop:					
Memory	1 0 0 0 1 1 1 1 | mod 0 0 0 r/m	5*	5*	2	9
Register	0 1 0 1 1 reg	5	5	2	9
Segment register	0 0 0 reg 1 1 1 (reg ≠ 01)	5	20	2	9,10,11
POPA = Pop All	0 1 1 0 0 0 0 1	19	19	2	9
XCHG = Exchange:					
Register/memory with register	1 0 0 0 0 1 1 w | mod reg r/m	3,5*	3,5*	2,7	7,9
Register with accumulator	1 0 0 1 0 reg	3	3		
IN = Input from:					
Fixed port	1 1 1 0 0 1 0 w | port	5	5		14
Variable port	1 1 1 0 1 1 0 w	5	5		14
OUT = Output to:					
Fixed port	1 1 1 0 0 1 1 w | port	3	3		14
Variable port	1 1 1 0 1 1 1 w	3	3		14
XLAT = Translate byte to AL	1 1 0 1 0 1 1 1	5	5		9
LEA = Load EA to register	1 0 0 0 1 1 0 1 | mod reg r/m	3*	3*		
LDS = Load pointer to DS	1 1 0 0 0 1 0 1 | mod reg r/m (mod ≠ 11)	7*	21*	2	9,10,11
LES = Load pointer to ES	1 1 0 0 0 1 0 0 | mod reg r/m (mod ≠ 11)	7*	21*	2	9,10,11
LAHF = Load AH with flags	1 0 0 1 1 1 1 1	2	2		
SAHF = Store AH into flags	1 0 0 1 1 1 1 0	2	2		
PUSHF = Push flags	1 0 0 1 1 1 0 0	3	3	2	9
POPF = Pop flags	1 0 0 1 1 1 0 1	5	5	2,4	9,15

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

80286 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT	CLOCK COUNT: Real Address Mode	CLOCK COUNT: Protected Virtual Address Mode	COMMENTS: Real Address Mode	COMMENTS: Protected Virtual Address Mode
ARITHMETIC					
ADD = Add:					
Reg/memory with register to either	0 0 0 0 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
Immediate to register/memory	1 0 0 0 0 0 s w | mod 0 0 0 r/m | data | data if s w = 01	3,7*	3,7*	2	9
Immediate to accumulator	0 0 0 0 0 1 0 w | data | data if w = 1	3	3		
ADC = Add with carry:					
Reg/memory with register to either	0 0 0 1 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
Immediate to register/memory	1 0 0 0 0 0 s w | mod 0 1 0 r/m | data | data if s w = 01	3,7*	3,7*	2	9
Immediate to accumulator	0 0 0 1 0 1 0 w | data | data if w = 1	3	3		
INC = Increment:					
Register/memory	1 1 1 1 1 1 1 w | mod 0 0 0 r/m	2,7*	2,7*	2	9
Register	0 1 0 0 0 reg	2	2		
SUB = Subtract:					
Reg/memory and register to either	0 0 1 0 1 0 d w | mod reg r/m	2,7*	2,7*	2	9
Immediate from register/memory	1 0 0 0 0 0 s w | mod 1 0 1 r/m | data | data if s w = 01	3,7*	3,7*	2	9
Immediate from accumulator	0 0 1 0 1 0 w | data | data if w = 1	3	3		
SBB = Subtract with borrow:					
Reg/memory and register to either	0 0 0 1 1 0 d w | mod reg r/m	2,7*	2,7*	2	9
Immediate from register/memory	1 0 0 0 0 0 s w | mod 0 1 1 r/m | data | data if s w = 01	3,7*	3,7*	2	9
Immediate from accumulator	0 0 0 1 1 1 0 w | data | data if w = 1	3	3		
DEC = Decrement:					
Register/memory	1 1 1 1 1 1 1 w | mod 0 0 1 r/m	2,7*	2,7*	2	9
Register	0 1 0 0 1 reg	2	2		
CMP = Compare:					
Register/memory with register	0 0 1 1 1 0 1 w | mod reg r/m	2,6*	2,6*	2	9
Register with register/memory	0 0 1 1 1 0 0 w | mod reg r/m	2,7*	2,7*	2	9
Immediate with register/memory	1 0 0 0 0 0 s w | mod 1 1 1 r/m | data | data if s w = 01	3,6*	3,6*	2	9
Immediate with accumulator	0 0 1 1 1 1 0 w | data | data if w = 1	3	3		
NEG = Change sign	1 1 1 1 0 1 1 w | mod 0 1 1 r/m	2	7*	2	7
AAA - ASCII adjust for add	0 0 1 1 0 1 1 1	3	3		
DAA - Decimal adjust for add	0 0 1 0 0 1 1 1	3	3		
AAS - ASCII adjust for subtract	0 0 1 1 1 1 1 1	3	3		
DAS - Decimal adjust for subtract	0 0 1 0 1 1 1 1	3	3		
MUL - Multiply (unsigned)	1 1 1 1 0 1 1 w | mod 1 0 0 r/m				
Register-Byte		13	13		
Register-Word		21	21		
Memory-Byte		16*	16*	2	9
Memory-Word		24*	24*	2	9
IMUL - Integer multiply (signed)	1 1 1 1 0 1 1 w | mod 1 0 1 r/m				
Register-Byte		13	13		
Register-Word		21	21		
Memory-Byte		16*	16*	2	9
Memory-Word		24*	24*	2	9
IMUL = Integer immediate multiply (signed)	0 1 1 0 1 0 s 1 | mod reg r/m | data | data if s = 0	21,24*	21,24*	2	9
DIV - Divide (unsigned)	1 1 1 1 0 1 1 w | mod 1 1 0 r/m				
Register-Byte		14	14	6	6
Register-Word		22	22	6	6
Memory-Byte		17*	17*	2,6	6,9
Memory-Word		25*	25*	2,6	6,9

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

80286 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT	CLOCK COUNT: Real Address Mode	CLOCK COUNT: Protected Virtual Address Mode	COMMENTS: Real Address Mode	COMMENTS: Protected Virtual Address Mode
ARITHMETIC (Continued):					
IDIV – Integer divide (signed)	1 1 1 1 0 1 1 w | mod 1 1 1 r/m				
Register Byte		17	17	6	6
Register Word		25	25	6	6
Memory Byte		20*	20*	2,6	6,9
Memory Word		28*	28*	2,8	6,9
AAM – ASCII adjust for multiply	1 1 0 1 0 1 0 0 | 0 0 0 0 1 0 1 0	16	16		
AAD – ASCII adjust for divide	1 1 0 1 0 1 0 1 | 0 0 0 0 1 0 1 0	14	14		
CBW – Convert byte to word	1 0 0 1 1 0 0 0	2	2		
CWD – Convert word to double word	1 0 0 1 1 0 0 1	2	2		
LOGIC					
Shift/Rotate Instructions:					
Register/Memory by 1	1 1 0 1 0 0 0 w | mod TTT r/m	2,7*	2,7*	2	9
Register/Memory by CL	1 1 0 1 0 0 1 w | mod TTT r/m	5+n,8+n*	5+n,8+n*	2	9
Register/Memory by Count	1 1 0 0 0 0 0 w | mod TTT r/m | count	5+n,8+n*	5+n,8+n*	2	9
	TTT Instruction: 0 0 0 ROL; 0 0 1 ROR; 0 1 0 RCL; 0 1 1 RCR; 1 0 0 SHL/SAL; 1 0 1 SHR; 1 1 1 SAR				
AND = And:					
Reg/memory and register to either	0 0 1 0 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
Immediate to register/memory	1 0 0 0 0 0 0 w | mod 1 0 0 r/m | data | data if w = 1	3,7*	3,7*	2	9
Immediate to accumulator	0 0 1 0 0 1 0 w | data | data if w = 1	3	3		
TEST = And function to flags, no result:					
Register/memory and register	1 0 0 0 0 1 0 w | mod reg r/m	2,6*	2,6*	2	9
Immediate data and register/memory	1 1 1 1 0 1 1 w | mod 0 0 0 r/m | data | data if w = 1	3,6*	3,6*	2	9
Immediate data and accumulator	1 0 1 0 1 0 0 w | data | data if w = 1	3	3		
OR = Or:					
Reg/memory and register to either	0 0 0 0 1 0 d w | mod reg r/m	2,7*	2,7*	2	9
Immediate to register/memory	1 0 0 0 0 0 0 w | mod 0 0 1 r/m | data | data if w = 1	3,7*	3,7*	2	9
Immediate to accumulator	0 0 0 0 1 1 0 w | data | data if w = 1	3	3		
XOR = Exclusive or:					
Reg/memory and register to either	0 0 1 1 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
Immediate to register/memory	1 0 0 0 0 0 0 w | mod 1 1 0 r/m | data | data if w = 1	3,7*	3,7*	2	9
Immediate to accumulator	0 0 1 1 0 1 0 w | data | data if w = 1	3	3		
NOT – Invert register/memory	1 1 1 1 0 1 1 w | mod 0 1 0 r/m	2,7*	2,7*	2	9
STRING MANIPULATION:					
MOVS – Move byte/word	1 0 1 0 0 1 0 w	5	5	2	9
CMPS – Compare byte/word	1 0 1 0 0 1 1 w	8	8	2	9
SCAS – Scan byte/word	1 0 1 0 1 1 1 w	7	7	2	9
LODS – Load byte/wd to AL/AX	1 0 1 0 1 1 0 w	5	5	2	9
STOS – Stor byte/wd from AL/A	1 0 1 0 1 0 1 w	3	3	2	9
INS – Input byte/wd from DX port	0 1 1 0 1 1 0 w	5	5	2	9,14
OUTS – Output byte/wd to DX port	0 1 1 0 1 1 1 w	5	5	2	9,14

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

80286 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT	CLOCK COUNT: Real Address Mode	CLOCK COUNT: Protected Virtual Address Mode	COMMENTS: Real Address Mode	COMMENTS: Protected Virtual Address Mode
STRING MANIPULATION (Continued):					
Repeated by count in CX					
MOVS – Move string	1 1 1 1 0 0 1 0 | 1 0 1 0 0 1 0 w	5 + 4n	5 + 4n	2	9
CMPS – Compare string	1 1 1 1 0 0 1 z | 1 0 1 0 0 1 1 w	5 + 9n	5 + 9n	2,8	8,9
SCAS – Scan string	1 1 1 1 0 0 1 z | 1 0 1 0 1 1 1 w	5 + 8n	5 + 8n	2,8	8,9
LODS – Load string	1 1 1 1 0 0 1 0 | 1 0 1 0 1 1 0 w	5 + 4n	5 + 4n	2,8	8,9
STOS – Store string	1 1 1 1 0 0 1 0 | 1 0 1 0 1 0 1 w	4 + 3n	4 + 3n	2,8	8,9
INS – Input string	1 1 1 1 0 0 1 0 | 0 1 1 0 1 1 0 w	5 + 4n	5 + 4n	2	9,14
OUTS – Output string	1 1 1 1 0 0 1 0 | 0 1 1 0 1 1 1 w	5 + 4n	5 + 4n	2	9,14
CONTROL TRANSFER					
CALL = Call:					
Direct within segment	1 1 1 0 1 0 0 0 | disp low | disp-high	7 + m	7 + m	2	18
Register/memory indirect within segment	1 1 1 1 1 1 1 1 | mod 0 1 0 r/m	7 + m,11 + m*	7 + m,11 + m*	2,8	8,9,18
Direct intersegment	1 0 0 1 1 0 1 0 | segment offset | segment selector	13 + m	26 + m	2	11,12,18
Protected Mode Only (Direct Intersegment):					
Via call gate to same privilege level			41 + m		8,11,12,18
Via call gate to different privilege level, no parameters			82 + m		8,11,12,18
Via call gate to different privilege level, x parameters			86 + 4x + m		8,11,12,18
Via TSS			177 + m		8,11,12,18
Via task gate			182 + m		8,11,12,18
Indirect intersegment	1 1 1 1 1 1 1 1 | mod 0 1 1 r/m (mod ≠ 11)	16 + m	29 + m*	2	8,9,11,12,18
Protected Mode Only (Indirect Intersegment):					
Via call gate to same privilege level			44 + m*		8,9,11,12,18
Via call gate to different privilege level, no parameters			83 + m*		8,9,11,12,18
Via call gate to different privilege level, x parameters			90 + 4x + m*		8,9,11,12,18
Via TSS			180 + m*		8,9,11,12,18
Via task gate			185 + m*		8,9,11,12,18
JMP = Unconditional jump:					
Short/long	1 1 1 0 1 0 1 1 | disp low	7 + m	7 + m		18
Direct within segment	1 1 1 0 1 0 0 1 | disp low | disp-high	7 + m	7 + m		18
Register/memory indirect within segment	1 1 1 1 1 1 1 1 | mod 1 0 0 r/m	7 + m,11 + m*	7 + m,11 + m*	2	9,18
Direct intersegment	1 1 1 0 1 0 1 0 | segment offset | segment selector	11 + m	23 + m		11,12,18
Protected Mode Only (Direct Intersegment):					
Via call gate to same privilege level			38 + m		8,11,12,18
Via TSS			175 + m		8,11,12,18
Via task gate			180 + m		8,11,12,18
Indirect intersegment	1 1 1 1 1 1 1 1 | mod 1 0 1 r/m (mod ≠ 11)	15 + m*	26 + m*	2	8,9,11,12,18
Protected Mode Only (Indirect Intersegment):					
Via call gate to same privilege level			41 + m*		8,9,11,12,18
Via TSS			178 + m*		8,9,11,12,18
Via task gate			183 + m*		8,9,11,12,18
RET = Return from CALL:					
Within segment	1 1 0 0 0 0 1 1	11 + m	11 + m	2	8,9,18
Within seg adding immed to SP	1 1 0 0 0 0 1 0 | data low | data-high	11 + m	11 + m	2	8,9,18
Intersegment	1 1 0 0 1 0 1 1	15 + m	25 + m	2	8,9,11,12,18
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0 | data low | data-high	15 + m		2	8,9,11,12,18
Protected Mode Only (RET):					
To different privilege level			55 + m		9,11,12,18

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

80286 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT	CLOCK COUNT: Real Address Mode	CLOCK COUNT: Protected Virtual Address Mode	COMMENTS: Real Address Mode	COMMENTS: Protected Virtual Address Mode
CONTROL TRANSFER (Continued):					
JE/JZ – Jump on equal/zero	0 1 1 1 0 1 0 0 | disp	7 + m or 3	7 + m or 3		18
JL/JNGE – Jump on less/not greater or equal	0 1 1 1 1 1 0 0 | disp	7 + m or 3	7 + m or 3		18
JLE/JNG – Jump on less or equal/not greater	0 1 1 1 1 1 1 0 | disp	7 + m or 3	7 + m or 3		18
JB/JNAE – Jump on below/not above or equal	0 1 1 1 0 0 1 0 | disp	7 + m or 3	7 + m or 3		18
JBE/JNA – Jump on below or equal/not above	0 1 1 1 0 1 1 0 | disp	7 + m or 3	7 + m or 3		18
JP/JPE – Jump on parity/parity even	0 1 1 1 1 0 1 0 | disp	7 + m or 3	7 + m or 3		18
JO – Jump on overflow	0 1 1 1 0 0 0 0 | disp	7 + m or 3	7 + m or 3		18
JS – Jump on sign	0 1 1 1 1 0 0 0 | disp	7 + m or 3	7 + m or 3		18
JNE/JNZ – Jump on not equal/not zero	0 1 1 1 0 1 0 1 | disp	7 + m or 3	7 + m or 3		18
JNL/JGE – Jump on not less/greater or equal	0 1 1 1 1 1 0 1 | disp	7 + m or 3	7 + m or 3		18
JNLE/JG – Jump on not less or equal/greater	0 1 1 1 1 1 1 1 | disp	7 + m or 3	7 + m or 3		18
JNB/JAE – Jump on not below/above or equal	0 1 1 1 0 0 1 1 | disp	7 + m or 3	7 + m or 3		18
JNBE/JA – Jump on not below or equal/above	0 1 1 1 0 1 1 1 | disp	7 + m or 3	7 + m or 3		18
JNP/JPO – Jump on not parity/parity odd	0 1 1 1 1 0 1 1 | disp	7 + m or 3	7 + m or 3		18
JNO – Jump on not overflow	0 1 1 1 0 0 0 1 | disp	7 + m or 3	7 + m or 3		18
JNS – Jump on not sign	0 1 1 1 1 0 0 1 | disp	7 + m or 3	7 + m or 3		18
LOOP – Loop CX times	1 1 1 0 0 0 1 0 | disp	8 + m or 4	8 + m or 4		18
LOOPZ/LOOPE – Loop while zero/equal	1 1 1 0 0 0 0 1 | disp	8 + m or 4	8 + m or 4		18
LOOPNZ/LOOPNE – Loop while not zero/equal	1 1 1 0 0 0 0 0 | disp	8 + m or 4	8 + m or 4		18
JCXZ – Jump on CX zero	1 1 1 0 0 0 1 1 | disp	8 + m or 4	8 + m or 4		18
ENTER – Enter Procedure	1 1 0 0 1 0 0 0 | data-low | data-high				
L = 0		11	11	2,9	8,9
L = 1		15	15	2,9	8,9
L > 1		12 + 4(L – 1)	16 + 4(L – 1)	2,9	8,9
LEAVE – Leave Procedure	1 1 0 0 1 0 0 1	5	5	2,9	8,9
INT = Interrupt:					
Type specified	1 1 0 0 1 1 0 1 | type	23 + m		2,7,8	
Type 3	1 1 0 0 1 1 0 0	23 + m		2,7,8	
INTO – Interrupt on overflow	1 1 0 0 1 1 1 0	24 + m or 3 (3 if no interrupt)	(3 if no interrupt)	2,8,8	
Protected Mode Only:					
Via interrupt or trap gate to same privilege level			40 + m		7,8,11,12,18
Via interrupt or trap gate to fit different privilege level			78 + m		7,8,11,12,18
Via Task Gate			167 + m		7,8,11,12,18
IRET – Interrupt return	1 1 0 0 1 1 1 1	17 + m	31 + m	2,4	8,9,11,12,15,18
Protected Mode Only:					
To different privilege level			55 + m		8,9,11,12,15,18
To different task (NT = 1)			169 + m		8,9,11,12,18
BOUND – Detect value out of range	0 1 1 0 0 0 1 0 | mod reg r/m	33	13 (Use INT clock count if exception 5)	2,6	8,9,11,12,18

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

80286 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT	CLOCK COUNT: Real Address Mode	CLOCK COUNT: Protected Virtual Address Mode	COMMENTS: Real Address Mode	COMMENTS: Protected Virtual Address Mode
PROCESSOR CONTROL					
CLC – Clear carry	1 1 1 1 1 0 0 0	2	2		
CMC – Complement carry	1 1 1 1 0 1 0 1	2	2		
STC – Set carry	1 1 1 1 1 0 0 1	2	2		
CLD – Clear direction	1 1 1 1 1 1 0 0	2	2		
STD – Set direction	1 1 1 1 1 1 0 1	2	2		
CLI – Clear interrupt	1 1 1 1 1 0 1 0	3	3		14
STI – Set interrupt	1 1 1 1 1 0 1 1	2	2		14
HLT – Halt	1 1 1 1 0 1 0 0	2	2		13
WAIT – Wait	1 0 0 1 1 0 1 1	3	3		
LOCK – Bus lock prefix	1 1 1 1 0 0 0 0	0	0		14
CTS – Clear task switched flag	0 0 0 0 1 1 1 1 | 0 0 0 0 0 1 1 0	2	2	2	13
ESC – Processor Extension Escape	1 1 0 1 1 T T T | mod LLL r/m (TTT LLL are opcode to processor extension)	9-20*	9-20*	5,8	8,17
SEG – Segment Override Prefix	001 reg 110	0	0		
PROTECTION CONTROL					
LGDT – Load global descriptor table register	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 1 | mod 010 r/m	11*	11*	2,3	9,13
SGDT – Store global descriptor table register	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 1 | mod 000 r/m	11*	11*	2,3	9
LIDT – Load interrupt descriptor table register	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 1 | mod 011 r/m	12*	12*	2,3	9,13
SIDT – Store interrupt descriptor table register	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 1 | mod 001 r/m	12*	12*	2,3	9
LLDT – Load local descriptor table register from register memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 010 r/m		17,19*	1	9,11,13
SLDT – Store local descriptor table register to register/memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 000 r/m		2,3*	1	9
LTR – Load task register from register/memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 011 r/m		17,19*	1	9,11,13
STR – Store task register to register/memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 001 r/m		2,3*	1	9
LMSW – Load machine status word from register/memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 1 | mod 110 r/m	3,6*	3,6*	2,3	9,13
SMSW – Store machine status word	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 1 | mod 100 r/m	2,3*	2,3*	2,3	9
LAR – Load access rights from register/memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 1 0 | mod reg r/m		14,16*	1	9,11,16
LSL – Load segment limit from register/memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 1 1 | mod reg r/m		14,16*	1	9,11,16
ARPL – Adjust requested privilege level from register/memory	0 1 1 0 0 0 1 1 | mod reg r/m		10*,11*	2	8,9
VERR – Verify read access register/memory	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 100 r/m		14,16*	1	9,11,16
VERW – Verify write access	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 101 r/m		14,16*	1	9,11,16

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

Footnotes

The effective Address (EA) of the memory operand is computed according to the mod and r/m fields:

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP

DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low

SEGMENT OVERRIDE PREFIX

0	0	1	reg	1	1	0

reg is assigned according to the following:

reg	Segment Register
00	ES
01	CS
10	SS
11	DS

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)
000 AX	000 AL
001 CX	001 CL
010 DX	010 DL
011 BX	011 BL
100 SP	100 AH
101 BP	101 CH
110 SI	110 DH
111 DI	111 BH

The physical addresses of all operands addressed by the BP register are computed using the SS segment register. The physical addresses of the destination operands of the string primitive operations (those addressed by the DI register) are computed using the ES segment, which may not be overridden.