32
29

UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

FACULTAD DE INGENIERIA

METODOLOGIA PARA EL DISEÑO DE UTILERIAS RESIDENTES EN MEMORIA

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A N :

DULCE MARIA MAYA REZA

OSCAR ALBERTO OLIVO DE ANDA

MEXICO D. F.

TESIS CON
FALLA DE ORIGEN

1991

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

TESIS CON FALLA DE ORIGEN

INDICE

CONTENIDO

PARTE II

PARTE III

INTRODUCCION

INTRODUCCION

Durante la etapa de investigación para la realización de este libro; numerosos temas, no implicados necesariamente en lo que es la esencia de esta tesis (Metodología para el Diseño y Programación de Utilerías Residentes en Memoria), fueron cobrando gran interes para nosotros, pues incluirlos vendría a enriquecer nuestro trabajo al permitir ubicar al lector en un contexto general y presentar información técnica complementaria valiosa.

Aunque la información presentada en este libro respeta un orden de continuidad, orientado a servir didácticamente al lector; en cada capítulo se incluye una introducción permitiendo la lectura de cada uno de ellos, de manera independiente.

Los temas de esta tesis han sido abordados de acuerdo a las pautas del método científico: planteamiento del problema, análisis, solución, resultados y conclusiones. Siguiendo esta tónica hemos estructurado la información de la manera indicada en los párrafos siguientes.

El presente trabajo se encuentra dividido en 3 partes generales. Dos de ellas estan dedicadas al tratamiento de los dos tipos de Programas Residentes en Memoria de mayor importancia en nuestros días: las Utilerías de Accesorio de Escritorio "*POP-UP*" y el programa "*SPOOLER*" de Impresión. La tercera parte del trabajo trata por separado un tema, de hecho involucrado en las dos aplicaciones antes mencionadas: "El Ambiente de Ventanas". La razón por la que decidimos tratar independientemente este tema, sin mencionar que en si mismo es ya extenso, radica en su importancia. El concepto de Programas TSR (*Terminate and Stay Resident*) no puede concebirse sin el concepto de ventanas "*pop-up*" (tambien conocidas como "emergentes" haciendo alusión a su aparicion momentánea en pantalla).

La primera parte abarca los cinco primeros capítulos de este libro. En el capítulo I se estudia, desde la perspectiva del sistema operativo MS-DOS, el desarrollo de los programas residentes en memoria desde sus inicios hasta nuestros días; también se profundiza en algunos aspectos del MS-DOS que deben ser previamente conocidos por el programador. El capítulo II aborda aspectos generales de los programas TSR, ¿Que son? ¿Que tipos de programas residentes existen? ¿Que programas pueden y qué programas deben convertirse en residentes?, etc. En el capítulo III se ponen de manifiesto los problemas implicados en la construcción de programas TSR, al tiempo que se realiza un análisis y se proponen alternativas de solución. El conjunto de las alternativas seleccionadas es incluido en el esquema general presentado en el capítulo IV. Dicho esquema constituye en sí, la metodología propuesta como guía para el diseño de utilerías residentes en memoria. Por último, en el capítulo V se pone en evidencia la importancia de presentar al usuario un ambiente amigable de trabajo, basado en la presentación por medio de ventanas; también incluye algunos aspectos técnicos básicos que, a nuestro juicio, el programador debe tener presente antes de iniciar la fase de programación del ambiente de ventanas para su utilería"*pop-up*". Con estos temas damos por concluida la primera parte del trabajo y sentamos las bases para iniciar la siguiente, cuyo tema central es "El Ambiente de Ventanas".

La segunda parte está comprendida por los capítulos VI, VII, VIII, IX y X. En esencia, todos ellos abarcan el tema de la presentación por medio de ventanas. En cada capítulo se diseña un conjunto de funciones orientado al manejo de un tipo particular de ventana.

En el capítulo VI se presentan 2 librerías de ventana básicas, la librería de funciones de bajo nivel y la librería de funciones generales de ventana. La primera, está basada en lenguaje ensamblador y contempla todas las operaciones de

propósito general de bajo nivel (específicas a la IBM PC y a su hardware) para manipular el teclado y el despliegue en pantalla. La segunda librería, contiene las funciones básicas para operaciones con ventana (creación, especificación de localización, tamaño, colores, contornos, títulos, escritura de texto en su interior, movimiento, restablecimiento del contexto de la pantalla después de su desaparición, etc.).

El capítulo VII aporta una librería especial que soporta del concepto de ventanas de ayuda de contexto sensitivo; permitiendo la posibilidad de ofrecer al usuario ayuda en "línea" independientemente de la aplicación en la que se encuentre.

El capítulo VIII viene a incrementar el potencial de nuestra biblioteca de funciones de ventana al agregar una librería de funciones para "ventanas de edición". Con el fin de ejemplificar el uso de dichas funciones, incluimos un programa de ejemplo (editor de texto), que en sí mismo puede considerarse ya una aplicación útil.

Para completar nuestra biblioteca de funciones de ventana, en el capítulo IX se incluye la librería de funciones para manejar "ventanas de menú". Estas funciones están basadas en la técnica de "menú de barra deslizante", que constituye el método más común de presentación de opciones. Para finalizar esta parte, todos los programas "ejemplo" (que demuestran el uso de las diferentes librerías de ventana) son integrados en un programa ejecutable controlado por un menú.

En el capítulo X, se lleva a la práctica el proceso (enunciado en el capítulo IV) para la elaboración de programas residentes en memoria. En esta sección presentamos un programa manejador de TSR's de propósito general, basado en el programa ejecutable de menú del capítulo anterior.

La tercera y ultima parte de este libro está dedicada al estudio particular de una de las aplicaciones residentes, quizá de mayor utilidad: "El *spooler* de impresion". Esta parte incluye sólo un capitulo, el capitulo XI, en el cual presentamos una introducción a los conceptos basicos manejados y un listado documentado de un programa de "*spool*" de impresión muy completo. Dicho programa puede manejar hasta 3 impresoras y trabajar en red.

LO QUE SE DEBE SABER DEL DOS

CAPITULO 1

HISTORIA DEL MS-DOS Y DE LOS PROGRAMAS TSR

El sistema operativo MS-DOS ha estado presente en el mundo de la computación desde hace ya una decada. Hoy día, existen diversas opiniones respecto a la eficiencia de este sistema operativo; pero pese a ello para nadie queda oculto que el MS-DOS marcó toda una era, y que en la actualidad sigue teniendo vigencia.

Gran parte de las peculiaridades del MS-DOS (como lo son aquellas que se erigen como limitantes para desarrollar un ambiente "*multitasking*" en la PC), se deben a sus orígenes. Para nosotros será interesante conocer la evolución del MS-DOS, porque al ir conociendo las demandas cambiantes que han dictado su crecimiento, nos ubicaremos en un contexto general que nos ayudará a entender cómo y porqué surgen los programas TSR.

Remontemonos a la era en la que los sistemas de cómputo personales iniciaban, fines de los 70. Era, en la que sólo se contaba con *diskettes* de una sóla cara que almacenaban 180Kb, y discos duros sólo disponibles para macrocomputadoras ("*mainframes*"). En esta epoca aparece el microprocesador 8086 de Intel, el primero de 16 *bits*, que con sus 20 líneas de direcciones podía direccionar un *megabyte* de memoria. Antes de su introducción, sólo existían microprocesadores de 8 *bits* que direccionaban 64 *kilobytes* de memoria. La mayor cantidad de memoria que podía direccionarse y la mayor capacidad de procesamiento del 8086, prometía un enorme crecimiento de las aplicaciones para computadoras personales.

En 1975, epoca de las microcomputadoras de 8 *bits*, Bill Gates y Paul Allen fundaron una compañía desarrolladora de *software* llamada "*Microsoft*". Su primer producto fue un intérprete de Basic.

Al poco tiempo convencieron a la mayoría de los fabricantes de computadoras personales de incorporar el Basic de *Microsoft* en sus máquinas. Se crearon varias versiones de Basic, algunas de las cuales funcionaban como sistema operativo -es decir, no requerían de un sistema operativo por separado que las soportara- e incluían funciones tales como entrada y salida de disco.

A principios de 1979, Tim Paterson de la compañía "*Seattle Computer Products*", diseñó una tarjeta de circuito impreso para el microprocesador 8086. Pero como no se contaba entonces con un sistema operativo para esta tarjeta pionera, hubo que escribir uno nuevo.

Así, a mediados de 1980 Tim Paterson creó el QDOS (*Quick and Dirty Operating System*) para sus sistemas 8086 S-100. Las convenciones de la interfaz del sistema operativo se escogieron de tal manera que emularan al CP/M (el *software* CP/M estándar usado entonces para los microprocesadores 8080 y Z80 de 8 *bits*), a fin de que el *software* de CP/M existente pudiera modificarse con relativa facilidad. Para fines de 1980, el QDOS había cambiado su nombre al de 86-DOS, como resultado de haber conseguido una versión mejorada.

Fue hasta este entonces que IBM empezo a reaccionar terminando con sus anos de indiferencia asumida para finalmente tomar la decisión de lanzar al mercado una computadora personal. Catorce ingenieros encabezados por Don Estridge asumieron el reto de diseñar y construir en apenas un ano la IBM PC. Uno de los primeros problemas que se plantearon fue la elección del sistema operativo. En esos momentos, IBM poseía sistemas operativos multitarea y multiusuario para sus grandes computadoras, pero ninguno capaz de implementarse en el entorno monotarea y monousuario propio de la IBM PC.

Ante este contratiempo, y en vista del escaso tiempo disponible, Don Estridge decidió encargar la tarea de implementación del nuevo sistema operativo a otra compañía. Esta compañía fue "*Microsoft* Corporation".

Sin embargo, *Microsoft* por una parte, entre sus propios recursos disponía únicamente del sistema operativo XENIX, una versión autorizada de UNIX de AT&T que no se podía implementar en el microprocesador 8088. Por otra parte, todos los programas existentes para la familia de micros de 8 *bits*, de la cual descendía el 8088, estaban escritos para el sistema operativo CP/M-80 de *Digital Research*. Y, para rematar, IBM había sido muy terminante en sus condiciones al fijar un límite que no excediera de un año.

Ante esta situación, *Microsoft* optó por la mejor vía que pudiera existir para solucionar el problema en ese momento: crear un sistema operativo con un alto grado de compatibilidad con CP/M-80 para permitir a los programadores transportar sin problemas sus programas al nuevo entorno. Esta decisión ha sido criticada, pues muchos arguyen que so pretexto la compatibilidad se relegaron otros aspectos que, aunque en ese momento no influirían, a la larga sí. Estos aspectos se refieren a las características heredadas del sistema operativo antecesor, que impedirían la implementación de un ambiente "*multitasking*" en la computadora.

De esta manera, en agosto de 1980, *Microsoft* se lanzó a la búsqueda de un sistema operativo similar al CP/M-80 que pudiera utilizar como embrión de su futuro proyecto. Además, como *Microsoft* también tenía que desarrollar **Basic, FORTRAN, Pascal y COBOL** para la IBM-PC, el nuevo sistema operativo debería, a su vez, ser capaz de soportar los nuevos lenguajes.

A finales de 1980, *Microsoft* hace negociaciones con *Seattle Computer Products* y le compra los derechos del *software* 86-DOS, anteriormente llamado QDOS, y que ahora sería llamado MS-DOS (*MicroSoft-Disk Operating System*).

Así, en agosto de 1981 se presentaba formalmente la nueva computadora personal IBM-PC acompañada del sistema operativo MS-DOS versión 1.0. No sólo era el comienzo de las computadoras personales, sino también el comienzo del imperio de *Microsoft*.

El MS-DOS se había puesto en camino y, como sucede con cualquier producto de *software*, las nuevas versiones aparecerían de manera continua y rápida conforme se iban detectando y corrigiendo errores, el desarrollo de *software* y *hardware* creaban nuevos requerimientos, y los usuarios demandaban más cambios. A continuación se enumeran, de manera general, las características de cada una de las versiones del DOS, desde su aparición hasta la fecha.

Versión 1.0 -Agosto de 1981-

- Cuando esta versión salió, la IBM-PC original sólo contaba con 64K de memoria RAM.
- Podía soportar sólo discos flexibles de una sola cara.
- Esta versión no requería más de 16K de memoria.
- Contaba sólo con 2 tipos de atributo de archivo: oculto (*hiden*) y sistema (*SYS*).
- No había agrupación de archivos en forma de directorios o de partición del disco, ya que los *diskettes* soportados eran muy restringidos.
- El número de archivos por *diskette* se limitaba a 64.

Versión 1.1 -Mayo de 1982-

- Proporciona soporte para discos de 2 caras, con un máximo de 112 archivos y 320K por disco.

Versión 2.0 -Marzo de 1983-

- Esta versión se introdujo con las computadoras XT.
- Requiere al menos 24K de memoria.
- Proporciona soporte para un disco duro de 10 *megabytes*.
- Incluye una estructura jerárquica de directorio para organizar los archivos en el disco.
- Soporta *diskettes* con nueve sectores por pista.
- El tamaño máximo permisible de un archivo era de 360K.
- Incorpora más atributos para los archivos: sólo lectura, oculto, sistema, etiqueta de volumen del directorio raíz o primitivo, listado del directorio con subdirectorios y *bit* de archivo.

- Incluye manejadores de archivo.
- Cuenta con los primeros medios para realizar *"multitasking"* en forma de un *spooler* de impresión.
- Los manejadores de periféricos pueden instalarse usando el archivo CONFIG.SYS, de tal modo que es posible agregar a la computadora una enorme variedad de dispositivos nuevos sin necesidad de cambiar el sistema operativo fundamental.
- Brinda la facilidad de llevar a cabo el control dinamico de memoria vía *software*.

Versión 2.1 -Octubre de 1983-

- En esta versión solo se realizaron cambios de sincronización con el fin de permitir un mejor manejo de la IBMPCjr y la PC portátil.

Versión 2.11 -Inicios de 1984-

- Esta versión fue sacada al mercado para mejorar el llamado "soporte internacional" (tablas para manejar símbolos, formato de fecha y hora, símbolos con punto decimal, etc.)

Versión 3.0 -Agosto de 1984-

- Esta version fue escrita especificamente para las computadoras AT, aun cuando corre en las XT y PC anteriores, proporcionando también ventajas para ellas.
- Soporta discos duros de 32 *megabytes* y discos flexibles de 5.25" de alta capacidad (1.2 *megabytes*).
- Pueden usarse nombres con trayectorias completas para todos los comandos de ejecución de programas DOS.
- El comando de impresion es más elaborado y flexible.
- El comando de gráficos soporta más impresoras.
- Soporta las convenciones internacionales para teclados, fecha, hora y moneda.
- El manejo uniforme de errores permite a los programadores reportar y diagnosticar errores con mayor facilidad.
- Esta versión incluye algunas de las funciones de soporte necesarias para el trabajo en red.

Versión 3.1 -Marzo de 1985-

- Esta versión añade la posibilidad de compartir discos en red.
- Principalmente en esta versión, se incorporaron soportes adicionales para redes.

Versión 3.2 -Mediados de 1986-

- Esta versión proporciona soporte para las unidades de disco flexible de 3½ pulgadas.
- Integra un formato especial en los manejadores de dispositivos periféricos.

Versión 3.3 -Febrero de 1988-

- Se proporciona un método nuevo y más flexible de soporte de idiomas nacionales llamado "conmutación código-página".
- Tiene capacidad para más de 20 manejadores de archivo.
- El comando **FASTOPEN** proporciona una manera de acelerar el acceso a los archivos, manteniendo la pista de aquellos que se hayan usado más recientemente.
- El comando **CALL** permite correr un archivo ***batch*** desde otro archivo *batch*.
- El soporte de dispositivos es expandido para soportar la nueva línea de IBM PS/2.

Versión 4.0 -Octubre de 1988-

- Esta versión incrementa el límite de archivos y la partición de memoria a *2 gigabytes*.
- Incluye un *shell* controlado por *mouse* a fin de permitir un acceso más amigable a los recursos del DOS.
- Proporciona una manera de salvar el límite de memoria de 640K del DOS al incluir soporte para el LIM/EMS 4.0, la convención para ampliación de memoria.
- Los comandos **VDISK** y **FASTOPEN** pueden usar la ampliación de memoria.
- El nuevo comando **MEM** presenta en pantalla una tabla del uso de memoria, indicando qué memoria se asigna para qué programas y otros fines.
- Es compatible con ***diskettes*** OS/2.

Versión 5.0

- El pasado marzo de este año, se anunció que la versión 5.0 del DOS estaba lista para salir al mercado. Se preve que para mediados de 1991 ya se encuentre difundida en los E.U. Esta nueva version del DOS está dirigida, principalmente, a ahorrar memoria RAM, así como ofrecer al usuario un ambiente del todo accesible, entre otros aspectos.

Una vez conocido el desarrollo de las diferentes versiones del DOS, situemonos en la época del surgimiento de los programas TSR.

Como sabemos, cada versión es resultado de nuevas demandas por parte de los diversos usuarios. La PC, su nombre lo dice (Computadora Personal), estuvo orientada desde su origen a atender un sólo usuario y una sóla tarea. Esto funcionó muy bien por algún tiempo. Sin embargo, pronto se vio clara la necesidad de poder manejar varias tareas. Los programas TSR (*Terminate and Stay Resident*), -cuyo significado literal es: programas que al terminar se quedan residentes en memoria- también conocidos como programas residentes; surgen con el fin de satisfacer los nuevos requerimientos que los usuarios demandaban. Los programas TSR permiten crear un estilo de ambiente "*multitasking*" en la PC. En el siguiente apartado profundizaremos en el conocimiento de este tipo de programas.

Las caracteristicas de la PC que dieron pie a la introducción de un ambiente de apariencia "*multitasking*" fueron las siguientes:

- La capacidad del microprocesador 8088 de poder direccionar un *megabyte* de memoria.
- La inclusión de una estructura de vectores de interrupción.
- La singularidad del teclado de la PC y su despliegue de video como parte integral de la computadora, y no como terminal de video conectada a la computadora a través del puerto serial.

Estas caracteristicas en 1982, sugerían un nuevo mundo de perspectivas de desarrollo de la PC y de utilerías para ella. No obstante, con el transcurso del tiempo se llegarían a comprobar 2 cosas: 1) La insuficiencia del microprocesador 8088 para poder simular por completo el ambiente de los equipos multiusuario-multitarea. 2) La capacidad del 8088 para manejar un estilo de "*multitasking*" que llegaría a satisfacer a un sector considerable de usuarios.

Desde su aparición, los TSR han experimentado cambios y mejoras substanciales. Podemos decir que sus orígenes arrancan desde 1982, cuando el DOS sólo contaba con una interrupción limitada para crear TSRs. Esta es la interrupción 27H de la que hablaremos también en la siguiente sección. Por ahora, sólo mencionemos que el fin primordial de *Microsoft* al incluir esta interrupción, era exclusivamente el de proporcionar al programador una herramienta para crear simplemente manejadores de interrupción residentes.

Pero, continuemos analizando la trayectoria de los programas residentes. En 1983 se incluye en la versión 2.0 del DOS otra función para manejar TSRs. Esta es la función 31H de la interrupción 21H del DOS, que a diferencia de la interrupción 27H, permitía al programador realizar programas residentes mayores de 64K. A pesar de la mejora, *Microsoft* aún reservaba para sí muchas funciones clave para la realización de utilerías residentes poderosas.

En la versión 2.0 del DOS, *Microsoft* también incluye un programa llamado **PRINT.COM**, el cual permite al usuario imprimir, y al mismo tiempo trabajar en la computadora. Era la primera vez que se ponía en evidencia la capacidad de los programas TSR. Aunque el microprocesador en realidad estaba atendiendo 2 tareas, desde la perspectiva del usuario, el cambio no se dejaba ver.

Pronto, los disenadores de *software* de diversas companias, se dieron a la tarea de "escudriñar" el **PRINT.COM** y develar los secretos que *Microsoft* conservaba para hacer aplicaciones residentes verdaderamente utiles. Al cabo de varios meses, solamente algunos empezaron a publicar sus resultados y no por completo. Desde 1984 a la fecha se han publicado numerosos ejemplos de programas TSR, pero en su mayoria constituyen pequenos programas de utileria que solo se basan en el teclado y el video para trabajar, y no hacen uso de funciones del DOS que involucren manejo de archivos. Posteriormente veremos que hacer aplicaciones que sólo impliquen manejo de video y teclado, no resulta demasiado complicado, y, aunque son acertadamente llamados "programas residentes", no constituyen el prototipo del TSR poderoso, al no resolver un problema medular, conocido con el nombre de "Problema de Reentrancia del DOS".

Afortunadamente, al cabo de algunos meses hubieron, aunque pocos, investigadores que si publicaron sus resultados; a pesar de haber tenido que enfrentar varios problemas. Primero empezaron por rastrear el **PRINT.COM** con un DEBUG descubriendo algunos aspectos interesantes: El **PRINT.COM** contenía rutinas que interceptaban las interrupciones 28H y 2FH. También se encontró que la funcion 34H era llamada para llevar a cabo una tarea importante.

Entonces, al buscar la referencia de todas estas funciones en el apéndice de Información Tecnica del manual del DOS 2.0, encontraron que dichas funciones eran para uso interno del DOS. Es decir, no-documentadas. Esto, para aquellos investigadores se venia a traducir en "no queremos decir como resolver el problema de la reentrancia". No obstante, al cabo de varios meses, los secretos de este problema se habian descubierto.

En 1984 aparece SideKick de Borland, un programa residente que en poco tiempo se difundió considerablemente. A partir de esa época han salido al mercado varios paquetes que emplean técnicas de residencia y que han venido a brindar un nuevo ambiente en la PC, asi como un ahorro considerable de tiempo.

LA VERDADERA CARA DEL DOS

El DOS es un sistema operativo diseñado expresamente para ejecutar una tarea a la vez. Responde a peticiones de servicios de entrada y salida, maneja la colocación de directorios y archivos en disco, puede comunicarse con el reloj del sistema, escribe hacia el sistema de impresión y hacia la consola de video, regresa los caracteres pulsados en el teclado, etc. El DOS es, en *esencia*, un archivo jerarquico y un servidor de dispositivos de unidad-de-registro que soporta un sólo usuario y una sóla tarea.

Durante toda la década de los 80 el MS-DOS ha dominado con autoridad suprema el mercado de los sistemas operativos para PC's. En la actualidad existen mas de 50 millones de PC's en todo el mundo trabajando con MS-DOS, cifra que representa el 99% del total de PC's.

Ante la pregunta ¿Es tan bueno el sistema operativo DOS? existen diferentes opiniones dependiendo del punto de vista que se adopte. Una primera respuesta sería no. No, si comparamos sus capacidades con las de otros sistemas operativos orientados a los microprocesadores 80386 y 80486; y una segunda respuesta sería sí, si sabemos sacarle el máximo provecho pudiendo así cubrir nuestras necesidades. Pero adentremos un poco más en estas ideas.

Por un lado, *Microsoft* ha tenido que arrastrar la vieja herencia del CP/M-80 y los límites del microprocesador 8088 (por ejemplo, la barrera de 640K de memoria RAM) en beneficio de mantener la compatibilidad ascendente entre todas sus versiones de MS-DOS. Y, precisamente esa defensa a ultranza de la compatibilidad ha impedido a *Microsoft* soportar en el MS-DOS las nuevas prestaciones incorporadas a los micros 386 y 486. Entonces nos preguntaremos ¿es tan importante la compatibilidad? nosotros creemos que sí, pues precisamente ha sido la compatibilidad la clave del gran éxito de la PC.

A lo largo de su breve historia, la PC ha mantenido una triple compatibilidad. A nivel de microprocesador: el 286 emula al 8086 y el 386 al 286. A nivel de *hardware* básico: la arquitectura de la IBM AT es compatible con la de la IBM PC. Y a nivel de sistema operativo: los programas que funcionan en una versión del MS-DOS, funcionarán también sin problema en las siguientes versiones del MS-DOS.

Durante 9 largos años, esta idea de la compatibilidad ha sido suficiente garantía para los programadores, conscientes de que sus programas se mantendrían inmunes al avance de la tecnología microinformática (nuevos microprocesadores, nuevas versiones del DOS, etc...). Y aquí surge una tesis fundamental: el éxito de un sistema operativo se mide en base al número de programas de aplicación existentes en el mercado para dicho sistema operativo.

Por mucho tiempo, el haber luchado por mantener la compatibilidad había dado resultado, pero a fines de la década de los 80, IBM advirtió que estaba perdiendo terreno frente a otros fabricantes como Compaq y, no obstante la trayectoria de compatibilidad que siempre había cuidado, decidió introducir una nueva línea de microcomputadoras con nuevas capacidades, el Sistema Personal/2 (PS/2). Antes de esto, la mayoría de los sistemas de computación personales habían tenido dificultades para cumplir con las estrictas demandas de rendimiento, comunicaciones y capacidad de almacenamiento, que cierto sector de usuarios estaba imponiendo. En sí, los modelos del Sistema Personal/2 se caracterizaron por sus canales más rápidos y amplios, una nueva arquitectura, soporte de microcódigo ampliado, y canales para acceso a memoria directa.

La tradición de IBM se suavizó un poco con el surgimiento de las PS/2 junto al esperado OS/2, cuya presentación oficial se verificó en 1988. Sin embargo es difícil que IBM vuelva a ser lo que fue durante los años 1981-1987.

La realidad es que, durante el largo período de desarrollo del MS-DOS se lograron satisfacer las necesidades de los usuarios. Meta a la que coadyuvó en gran parte el desarrollo paralelo de los programas residentes.

En el apartado anterior se mencionó que en 1984 sale al mercado *Sidekick*, un programa residente que brinda múltiples facilidades de "escritorio". Durante los años siguientes, se siguieron desarrollando numerosos programas que, aprovechandose de las ventajas de las nuevas técnicas de residencia en memoria, vinieron a ofrecer al usuario de la PC un ambiente mejorado que les permitía ahorrar copiosamente el tiempo y agilizar su trabajo.

A partir de entonces fueron muchos los usuarios que se sumaron al grupo de partidarios de los programas residentes, principalmente porque los TSRs bastaban para ver satisfechas sus necesidades. Hubo otro sector de usuarios, que al empezar a explotar el nuevo ambiente que los programas TSR ofrecían, comenzó, con el tiempo a demandar todavía mayores capacidades; estos usuarios requerían manejar un número indefinido de tareas al mismo tiempo y una cantidad muy grande de información. Es decir, sus expectativas se veían más ampliamente cubiertas por las nuevas PS/2, u otro tipo de equipo equivalente o inclusive más sofisticado.

No obstante todo esto, hoy día, el mercado de las PC con el sistema operativo MS-DOS es todavía muy grande. IBM está consciente de ello, por lo que en sus nuevos planes, tiene contemplado continuar desarrollando versiones del DOS y *software* para PC. Por su parte, las utilerías residentes para PC, tienen la ventaja de poder correr en cualquier tipo de equipo PC compatible. Y esto resulta benéficioso cuando, independientemente del equipo con que se cuente, se requiere de una aplicación específica al alcance en un TSR.

A principios de 1990, *Microsoft* e IBM anunciaron un acuerdo en el que exponían su visión conjunta del mercado de los sistemas operativos para la década de los 90. Llegaron a la conclusión de que la respuesta se centraba en 3 medidas: La primera, adoptar *Windows* como solución básica para la gama baja de computadoras personales. La segunda, seguir desarrollando versiones de MS-DOS. Y tercera, continuar difundiendo y vigilar el crecimiento del OS/2 y aplicaciones para esta gama.

En si, podemos observar que en los años 90 coexistirán dos plataformas (una de alto nivel y otra de bajo nivel), con campos de aplicación tajantemente diferenciados, pero con una fácil relacion entre si. La plataforma de alto nivel constará de un microprocesador de 32 *bits* (386/486), 4 Mb de RAM, 60 Mb en disco duro y trabajara con OS/2 o como servidor de una red local con OS/2 LAN *Server*. Y todo ello con la nueva versión OS/2 2.0, que aprovecha plenamente las capacidades de los micros 386, trabajando con 32 *bits*. La plataforma de bajo nivel estará basada en el microprocesador de 16 *bits* (8086/80286), 2 MB de RAM o menos, y podra trabajar bajo *Windows*, ya sea de forma independiente o como terminal inteligente conectada a OS/2 LAN *Server*. Esto proporcionara a los usuarios varias opciones, permitiendoles conservar sus utilerías bajo DOS, incluyendo programas residentes que constituyan aplicaciones particulares o especificas de cada usuario.

Antes de terminar, queremos hacer notar que no obstante el crecimiento que ha experimentado el MS-DOS, como lo atestigua el gran numero de versiones emitidas, existe un nutrido grupo de usuarios y programadores que no han querido perder su inversión en el MS-DOS al cambiar radicalmente de sistema operativo. Estos usuarios han visto satisfechas sus necesidades al proveer al DOS de nuevos "aditamentos" que lo hacen más poderoso y del todo capaz de cubrir sus expectativas; por lo que no ven motivo para cambiarlo. Estos "aditamentos" constituyen un nuevo nivel agregado al DOS, conocido como *shell* adicional o máquina virtual.

Así, debido a que el DOS esta basado en modo texto, y debido a las exigencias actuales de sistemas basados en gráficos; se puede agregar arriba del DOS una máquina virtual basada en gráficos como *Windows*. Otro ejemplo de máquina virtual lo constituyen las utilerías residentes al contribuir a establecer mejoras sobre el MS-DOS.

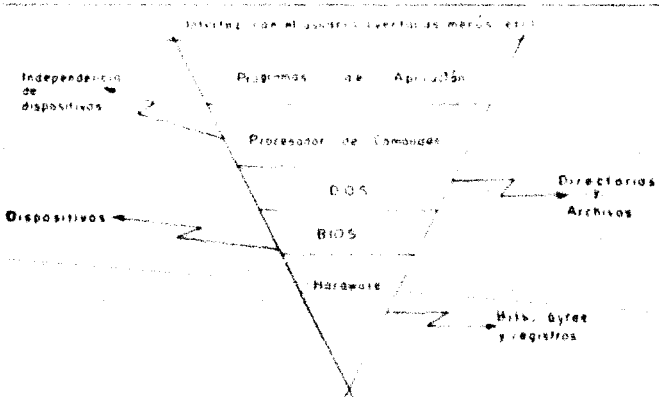
Dichas utilerías residentes en memoria son el marco de esta tesis. Por lo que antes de entrar de lleno al tema, consideramos conveniente primero haber ubicado al lector en un contexto general desde la perspectiva de los orígenes y desarrollo del MS-DOS; sistema operativo para el cual los TSRs fueron diseñados.

ESTRUCTURA DEL MS-DOS

■ CAPAS DEL SISTEMA.

Una manera de suma utilidad para contribuir a la formación de un concepto real del sistema PC bajo el MS-DOS, radica en concebirlo como una estructura jerárquica cuyas funciones son distribuidas en varios subsistemas. Cada subsistema o capa de la estructura proporciona un conjunto de servicios determinados sobre los cuales se construye la siguiente capa, o nivel más elevado. La siguiente figura muestra, de manera gráfica, la disposición jerárquica de este sistema de capas. Cada capa también recibe el nombre de "máquina virtual", debido a que cada nivel viene a aumentar el poder del sistema en su conjunto.

FIG. 1.1 CAPAS DEL SISTEMA



La base del sistema está constituida por los circuitos eléctricos que integran la computadora. A continuación se encuentra la capa del BIOS (Sistema Básico de E/S), con el cual se tiene un nuevo concepto de "sistema de computadora" al incluir funciones y características particulares (que serán vistas a continuación).

El BIOS debe estar presente al encender la máquina y antes de que se realice cualquier lectura a disco; por lo que se almacena en memoria permanente o de sólo lectura (ROM). El BIOS contiene programas para establecer la comunicación y el control del equipo periférico.

Existen varias versiones de BIOS, tales como Phoenix BIOS, el AMI BIOS y el IBM PC BIOS estándar. Las computadoras XT requieren un BIOS diferente de las AT y las PS/2 usan algo llamado ABIOS (BIOS avanzado), a fin de aprovechar al máximo la memoria disponible. El BIOS constituye la primer capa de aislamiento entre los programas y el hardware físico.

Junto con el BIOS en ROM se incluye una autoprueba de encendido, mediante la cual se verifica la integridad de la memoria, se inicializan las unidades de disco y las tarjetas controladoras, se prueba la pantalla, el reloj, los puertos de entrada y salida, y se verifica el teclado. Durante cierta parte de este proceso puede verse el contador de memoria en la pantalla.

La capa siguiente está formada por el sistema operativo (DOS) El cual integra las características requeridas en orden de operar en un medio ambiente estandarizado. El DOS define otro concepto de "sistema de computadora" de mayor nivel. El sistema operativo es cargado una vez terminada la autoprueba de encendido, etapa que recibe el nombre de inicialización. Durante esta etapa la computadora intenta encontrar un archivo de inicialización que le indique que sistema operativo va a cargar, dónde encontrarlo y como cargarlo. Busca primero en la unidad de *diskettes* A>. Si esta unidad esta vacía, intenta leer el archivo de inicialización en la unidad de disco duro C>. Cuando encuentra el archivo requerido, lo carga, ejecuta sus instrucciones y procede finalmente a cargar el sistema operativo. Este proceso consta de varios pasos, pero los esenciales son: cargar en memoria la porción residente del sistema operativo y correr el intérprete de comandos.

Precisamente es el intérprete de comandos lo que forma el siguiente nivel de la estructura. Éste, constituye la parte del sistema operativo con la que el usuario esta más familiarizado. Un *prompt*, que indica la unidad de discos en función, aparece en la parte superior izquierda de la pantalla. Línea sobre la cual se pueden ejecutar programas o comandos del DOS, por lo que recibe el nombre de Línea de Comandos.

Cuando se introduce un comando, el intérprete de comandos (también conocido como procesador de comandos) intenta determinar qué quiere hacer el usuario. Busca en los directorios de la ruta en curso un archivo ejecutable cuyo nombre coincida con el que se anotó. Si lo encuentra, procede a su ejecución; en este punto el usuario se encontrará en el nivel siguiente, corriendo la máquina virtual provista por el programa elegido.

En efecto, el nivel más alto lo definen los programas de aplicación de usuario. Incluyendo lenguajes, utilerías y aplicaciones residentes.

Además de proporcionar esta facilidad fundamental para cargar y ejecutar programas, el DOS incluye varias facilidades básicas que permiten listar, crear o borrar directorios y archivos, moverse de un directorio a otro, transferir datos de la computadora a los puertos o viceversa. También proporciona los medios para llevar a cabo lo anteriormente mencionado desde el interior de cualquier programa y ofrece la facilidad de correr, de manera automática, listas de comandos DOS, por medio de lo que conocemos como archivos *batch*.

Hemos visto, la estructura del sistema de la PC bajo el DOS, destacando su organización jerárquica y el concepto de máquina virtual; con estos antecedentes corresponde ahora hablar de las partes o módulos que integran propiamente el sistema operativo MS-DOS. Como veremos, algunas de estas partes coinciden con ciertos niveles del sistema de capas, en este apartado mencionados. Hacemos esta aclaración con el fin de evitar ambigüedades y poder distinguir entre ambas clasificaciones.

■ LOS MODULOS DEL DOS.

El MS-DOS consiste de cuatro módulos básicos que definiremos a continuación.

1.- El Registro de Inicialización o Arranque (*Boot record*).
Este registro se encuentra en el *track* 0, sector 1, lado 1, de cualquier disco formateado con el comando **FORMAT** del DOS. En discos duros el registro de arranque se localiza en el primer sector de la partición del DOS. Este registro requiere de un área equivalente a un sector de disco, lleva a cabo la identificación o reconocimiento del disco y contiene el programa de arranque inicial para el disco.

2.- El BIOS (*Basic Input Output System*) "Sistema Básico de Entada/Salida".
El BIOS constituye el primer nivel de *software* en la máquina virtual. EL BIOS consiste de 2 partes: una localizada en ROM, en la que residen las funciones mas básicas de la máquina; y otra que es cargada desde disco, la cual amplía las funciones de la primera; de manera que en conjunción pueden manejar toda la entrada/salida del sistema. El propósito del BIOS es aislar los niveles altos de *software* de posibles cambios de *hardware* en la computadora. EL BIOS proporciona un conjunto definido de servicios que sirven de base a los niveles altos de *software*. Cada fabricante de computadoras proporciona el BIOS para sus máquinas.

3.- El cuerpo del DOS propiamente. (*DOS Kernel*)
El sistema operativo es un programa que proporciona servicios de alto nivel necesarios para la implementación de programas. *Microsoft* proporciona el *DOS kernel* como un programa propietario basado en los servicios estándar del BIOS. El *DOS kernel* incluye servicios independientes de *hardware* que pueden ser usados por programas de aplicación sobre una variedad de sistemas.

Estos servicios pueden ser accesados de 2 formas: a través de interrupciones de *software*, o directamente mediante llamadas a funciones del DOS, colocando el número de la función en el registro AH y ejecutando la interrupción 21H. Los servicios del DOS se pueden agrupar en las siguientes categorías: E/S de caracteres, operaciones de directorio, control de disco, asignación dinámica de memoria, manejo de errores, operaciones de archivo, funciones de miscelánea del sistema, funciones de red, y ejecución-terminación de programas en general. El DOS construye todos estos servicios sobre la plataforma que le proporciona el BIOS.

4.- El Procesador de Comandos. (**COMMAND.COM**)

También conocido como interprete de comandos o *shell*, constituye la interfaz entre el usuario y los servicios del DOS. Hecho por el cual, mucha gente confunde este módulo con el propio sistema operativo. El Procesador de Comandos genera el indicador de solicitud de comandos (*prompt* del sistema, C>), acepta comandos, y ejecuta programas requeridos por los usuarios del sistema. En sí, el **COMMAND.COM** es un programa que controla el acceso a los recursos del sistema y genera el ambiente de trabajo tan ampliamente conocido ya por los usuarios. Consta de 3 partes: una porción residente, una sección de inicialización, y un módulo transitorio.

a) La porción residente contiene las rutinas de los procesos: *Ctrl-C* y *Ctrl-Break*, errores críticos, y el mecanismo de terminación de otros programas transitorios. Esta parte es la encargada de enviar mensajes de error y de responder al mensaje:

ABORT, RETRY, IGNORE?.

También contiene el código requerido para recargar la porción transitoria del **COMMAND.COM** cuando sea necesario.

b) La sección de inicialización es cargada sobre la porción residente al momento de encender la máquina y arrancar el sistema. En esta parte se procesa el archivo *batch* **AUTOEXEC** (que incluye la lista de comandos de usuario a ejecutar al tiempo de inicialización del sistema); en caso de no existir este archivo, este paso es omitido.

c) La parte transitoria es cargada en el tope del Area de Memoria de Programas Transitorios (TPA), área que inicia inmediatamente después de la porción residente del **COMMAND.COM**. En la sección "Organización de la Memoria del DOS" encontrada más adelante, podremos visualizar la distribución de memoria en su conjunto.

Ahora, los comandos aceptados por el **COMMAND.COM** caen en tres categorías: Comandos Internos, Comandos Externos y Archivos *Batch*.

a) Los comandos internos, a veces llamados intrínsecos, incluyen las operaciones: **COPY**, **REN**(AME), **DIR**(ECTORY), **DEL**(ETE), etc. Las rutinas que ejecutan estos comandos vienen incluidas en la parte transitoria del **COMMAND.COM**.

b) Los comandos externos, o extrínsecos (también conocidos como programas transitorios) vienen a ser los nombres de los archivos con extensión EXE, COM o BAT almacenados en disco. Antes de su ejecución, estos programas son cargados en el TPA. Ejemplos familiares de comandos externos son **CHKDSK**, **BACKUP** y **RESTORE**. Tan pronto como un comando externo haya completado su trabajo, es descartado de memoria, por ello debe ser recargado de disco cada vez que sea invocado.

c) *Los archivos batch* son archivos de texto que contienen listas de comandos de tipo intrinseco, extrínseco o *batch*. Estos archivos son procesados por un intérprete especial construido dentro de la parte transitoria del **COMMAND.COM**. El intérprete lee y ejecuta línea por línea del archivo *batch*.

Ahora, cuando el usuario teclea algo en la línea de comandos, el **COMMAND.COM** primero determina si se trata de un comando intrinseco que pueda llevar a cabo directamente. Si éste no es el caso, busca un comando externo (programa ejecutable) o archivo *batch*, con ese nombre. La búsqueda es efectuada primero en la unidad de disco y directorio corrientes, y luego en cada uno de los directorios especificados en la trayectoria del ambiente. En cada directorio inspeccionado el **COMMAND.COM** primeramente intenta encontrar un archivo con extensión .COM, en segundo lugar uno con extension .EXE y finalmente uno de extensión .BAT. Si no se encuentra archivo alguno de estos tipos, el **COMMAND.COM** desplegará el conocido mensaje de error:

"BAD COMMAND OR FILE NAME"

pero si si logra encontrar cualesquiera de ellos, procederá a su ejecucion. Este último caso, es el que nos concierne. El preámbulo anterior ha sido presentado con el fin de llegar a este punto. Será de interés conocer el proceso de ejecucion de un archivo y relacionarnos con estos conceptos para entender que sucede al ejecutarse nuestro TSR.

Para nuestro estudio, bastará analizar de manera general el procedimiento seguido por el DOS para ejecutar un programa, ya que, a pesar de que no es del todo indispensable incluir esta información (segun el contexto de esta tesis), consideramos que puede contribuir a reforzar y complementar temas que se tratarán con posterioridad, y que involucran un previo conocimiento de conceptos manejados en este punto, tales como la carga de un programa, el PSP, etc.

EJECUTANDO UN PROGRAMA.

¿Qué hace el **COMMAND.COM** al identificar en la Línea de Comandos el nombre de un archivo con extension .COM o .EXE ? Se dispone a ejecutar el programa llamando a la función **EXEC** del DOS. Esta función realiza lo siguiente:

- Verifica si existe suficiente memoria disponible para cargar el programa. Si la hay, asigna la memoria requerida. Y si no la hay, emite un mensaje de error y se abstiene de ejecutar el programa.

- Si la memoria fue asignada, la funcion **EXEC** construye una estructura de datos especial llamada Prefijo de Segmento de Programa (conocida como PSP) en la parte del fondo de la memoria asignada al programa. El PSP contiene varias ligas y apuntadores requeridos por el programa de aplicación. Más adelante trataremos el tema del PSP por separado.

- Posteriormente carga el programa dentro del espacio de memoria inmediato superior al PSP.

- A continuacion transfiere el control al "punto de entrada" del programa. Los archivos .COM siempre mantienen el "punto de entrada" en la dirección 100H (justo despues del PSP).

- Para finalizar, cuando el programa termina regresa el control a la porción residente del **COMMAND.COM**, la cual realiza un recuento del área ocupada por la porción transitoria. Si el resultado de esta prueba coincide con el valor esperado, el control es vuelto a transferir a la porción transitoria, y el *prompt* del sistema reaparece en pantalla. Pero si el resultado no coincide, la porción transitoria es vuelta a cargar desde disco y recupera el control.

- Al término del programa transitorio, se ejecuta de manera automática una función de terminación especial del DOS que libera la memoria ocupada por el programa y regresa el control al programa que originó su ejecución (en este caso el COMMAND.COM).

LA MEMORIA

El tema central de esta tesis, programas residentes en memoria, da por si mismo la respuesta al porqué de la inclusión de este apartado. Uno de los propósitos de esta tesis es el de abarcar, en lo posible, todos los puntos en ella involucrados. Del que toca hablar en esta parte es el que trata de la memoria RAM de la PC. Cuando se ejecuta por primera vez un TSR, este residirá en la memoria RAM hasta su terminación definitiva. Antes de entrar de lleno al tema de como se carga en memoria un TSR y demás aspectos concernientes, conozcamos que es, cómo se organiza y como funciona la memoria de la PC.

Empecemos por recordar los tipos de memoria disponibles en la PC, para luego dedicarnos exclusivamente al de mayor interés para nuestro estudio: el correspondiente a la memoria RAM.

Las computadoras personales y compatibles constan de 2 clases generales de memoria: 1) La Memoria de Sólo Lectura (ROM) "*Read Only Memory*": memoria permanentemente instalada en la computadora, que usualmente conserva una porción especifica del BIOS. En esta clasificación se incluye la Memoria Programable de Sólo Lectura (PROM) y demás variantes. 2) La Memoria de Acceso Aleatorio (RAM) "*Random Access Memory*". Esta, Conserva código de programas y datos no permanentes y es utilizada para cargar y ejecutar los programas de usuario.

Existen varios tipos de memoria RAM, entre los que se incluyen:

- Memoria Convencional (o Base).
- Memoria Superior.
- Memoria Alta.
- Memoria Extendida.
- Memoria Expandida.

La memoria convencional constituye el área en la que se ejecutan la mayor parte de los programas, y en la que se carga, generalmente, el sistema operativo.

La máxima cantidad de memoria convencional de la que se puede disponer es normalmente de 640 Kb, aunque esta barrera puede superarse utilizando la memoria extendida. Ésta representa el espacio de memoria situado por encima de la marca de 1 Mb.

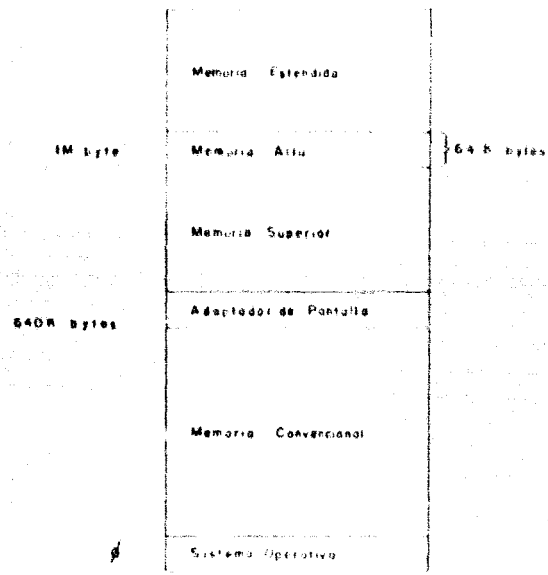
Otra forma de acceder a una mayor cantidad de memoria es mediante la utilización de memoria expandida o memoria EMS (*Expanded Memory Specification*). Dicha memoria permite ejecutar programas que manejan grandes cantidades de datos, y que no podrían ejecutarse de una manera eficiente en la memoria convencional. Dichos programas tienen que diseñarse específicamente para acceder a este tipo de memoria.

A la memoria situada en las direcciones comprendidas entre 640 Kb y 1 Mb, se le llama memoria superior, y tiene un tamaño de 384 Kb. Algunas partes del *hardware* de la computadora utilizan zonas de esta memoria.

La memoria alta está constituida por los primeros 64 Kb de la memoria extendida. Algunos comandos del sistema para equipos 286 en adelante pueden hacer uso de la memoria superior y alta.

La figura 1.2 muestra la disposición de las zonas de memoria antes mencionadas. Pero antes de terminar esta sección, queremos anotar que el estudio que iniciaremos referente a la memoria RAM, se centrará en el área de memoria convencional por constituir la zona de trabajo de los TSRs.

FIG.1.2 MAPA DE MEMORIA PARA MAQUINAS BASADAS EN EL INTEL 80286, 386 ó 486



■ ORGANIZACION DE LA MEMORIA.

El DOS puede manejar hasta 1 Mb de memoria contigua RAM. En las IBM PC's y compatibles, la memoria ocupada por el MS-DOS empieza en la dirección 0000H y puede alcanzar la 09FFFFH; estas direcciones delimitan un espacio de 640 *Kbytes* de RAM que son referidos comunmente como "memoria convencional". La memoria convencional incluye los módulos del DOS (DOS *Kernel*, BIOS, COMMAND.COM -parte residente-), así como *buffers* de disco, vectores de interrupción, y un espacio reservado para alojar cualesquiera programas cargados en el sistema; este espacio recibe el nombre de Area de Programas Transitorios (TPA). Los 384 Kb restantes de RAM, encontrados encima de la memoria convencional, estan reservados para los manejadores de *hardware* en ROM, *buffers* para refresco de video, etc.

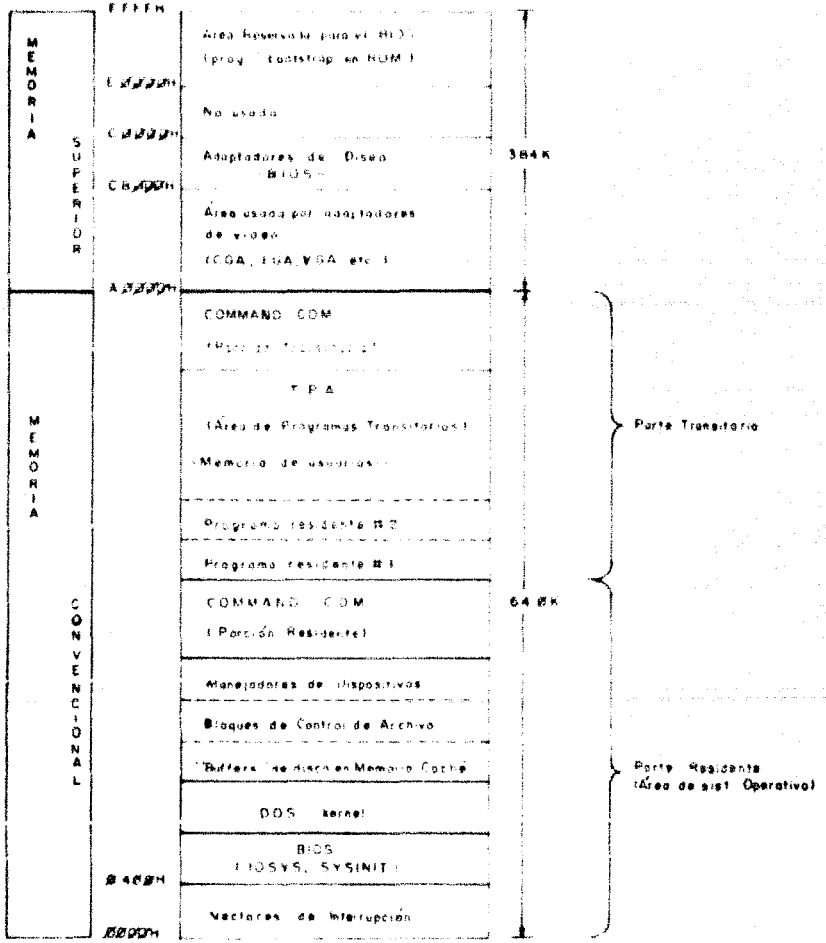
Ahora, veamos la organización de la memoria de la PC bajo otra perspectiva. Una vez cargado el sistema operativo, la memoria RAM es dividida en 2 secciones generales especializadas (que a continuación se describen) : el Area de Sistema Operativo y el Area de programas Transitorios.

El Area de Sistema Operativo empieza en al dirección 0000H, esto es, ocupa la porcion más baja de RAM. Esta área mantiene: La tabla de vectores de interrupcion (que ocupa un espacio de 1024 *bytes*), las tablas del BIOS, el propio sistema operativo con sus tablas y *buffers*, cualquier manejador instalable adicional especificado en el archivo CONFIG.SYS, y la parte resiente del intérprete COMMAND.COM. Como vemos incluye casi toda la memoria convencional excluyendo el TPA. La cantidad de memoria ocupada por el Area de Sistema Operativo varia según la versión de MS-DOS usada, el numero *buffers* de disco y el tamaño de los dispositivos manejadores instalados.

El Area de Programas Transitorios está constituida por el resto de RAM, e incluye el espacio de TPA y el área ocupada por la parte transitoria del COMMAND.COM. En el TPA se lleva a cabo la asignación dinamica de memoria y se alojan los programas de usuario.

Los programas de usuario que generalmente son cargados en primera instancia, son los TSR. El área de memoria restante en el TPA, después de haber cargado programas residentes, será el nuevo espacio disponible para satisfacer los requerimientos de memoria de otros programas. Para darnos una idea del consumo de espacio, después de haber cargado *Sidekick*, permanecen libres en el TPA aproximadamente 550 *Kbytes*. La zona de programas transitorios TPA (*Transient Program Area*) debe su nombre al hecho de que los programas de usuario son transitorios en esta área; es decir, radican sólo de manera temporal.

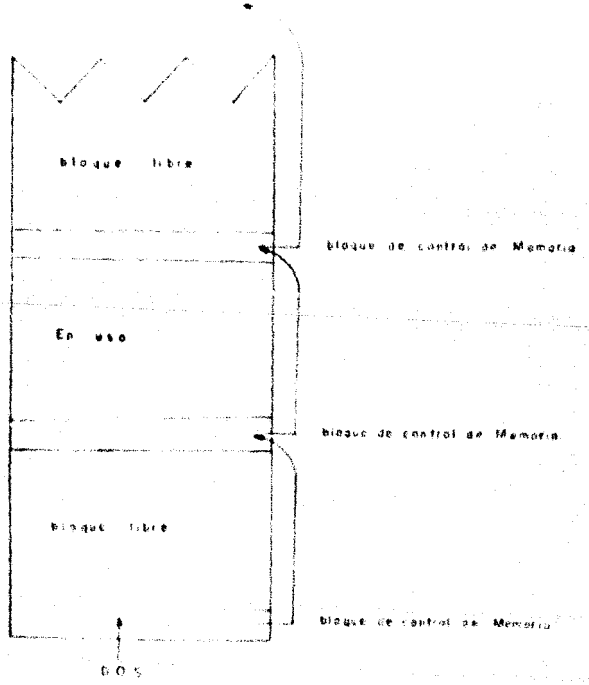
FIG.1.3 MAPA DE MEMORIA DE UNA MAQUINA CON 1Mb DE MEMORIA



■ ASIGNACION Y ADMINISTRACION DE MEMORIA.

El DOS, para poder controlar las operaciones de asignación o administración de memoria, organiza el TPA en un conjunto de bloques que permanecen encadenados entre sí. Cada bloque es acompañado de un Bloque de Control de Memoria -MCB- (*Memory Control Block*) de 16 *bytes* (es decir, el equivalente a un párrafo de memoria). Mientras que los Bloques de Memoria proporcionan el espacio requerido para mantener programas y datos, los MCB contienen información característica de su bloque asociado. Esto es, llevan un control del tamaño del bloque en párrafos, indican si es o no el último bloque de la cadena y si se encuentra libre u ocupado. Cada MCB, también, apunta al siguiente MCB de la cadena. La fig. 1.4 muestra de manera conceptual, la disposición de los bloques de memoria.

FIG.1.4 ENCADENAMIENTO DE BLOQUES DE MEMORIA



A partir de un apuntador del DOS, que direcciona al primer MCB, se inicia la cadena. Cada MCB, basándose en el tamaño de su bloque de memoria asociado, puede calcular el *offset* para llegar al siguiente MCB. La direccion del MCB corriente, más el tamaño del bloque de memoria que representa, más 1 será igual a la dirección del siguiente MCB de la cadena. Tenida la dirección del primer bloque de memoria asignado por el DOS, se puede rastrear toda la cadena para los fines que convengan. Por último cabe mencionar que si en la cadena se tienen 2 bloques libres contiguos, el DOS los combina dentro de un sólo bloque de memoria.

Cuando el DOS recibe una petición de memoria, primero inspecciona la cadena de bloques con el fin de corroborar la existencia de algun bloque lo suficientemente grande para poder cubrir la petición. Si este no existe, o si el DOS encuentra alguna anormalidad en la cadena, tal como alguna fractura o interrupción que impida su continuidad, se despliega un mensaje de error de asignación de memoria y el sistema detiene su operación. Si por el contrario se verifica que no existe problema para asignar la memoria solicitada, el DOS utiliza alguna de las siguientes estrategias de asignación:

- Estrategia del Primer Ajuste (*First Fit*)
  El DOS asigna el primer bloque de memoria encontrado sobre la cadena que pueda cubrir la petición.

- Estrategia del Mejor Ajuste (*Best Fit*)
  El DOS asigna el bloque de memoria más pequeño, pero suficientemente grande para llenar la peticion.

- Estrategia del Ultimo Ajuste (*Last Fit*)
  El DOS asigna el bloque de memoria mas alto (es decir, procurando utilizar primero las zonas del tope de la memoria disponible al usuario) y de tamaño adecuado para satisfacer la petición.

A partir de la versión 3.3 del DOS, se encuentra disponible una función que permite cambiar el tipo de estrategia de asignación de memoria. La estrategia *default* del DOS es la del primer ajuste. Y siendo la mejor, no hay razón para cambiarla a menos que así se requiera para fines especiales.

No siendo necesario tocar el punto de los tipos de estrategias disponibles bajo el DOS, hemos querido mencionarlo a manera de complemento y de poder presentar la información de forma hilada. Ahora corresponde hablar de las funciones involucradas en la asignación de memoria.

El DOS cuenta con 3 funciones básicas involucradas en la asignación/desasignación de memoria, accesadas por medio de la Interrupción 21H:

- Función 48H. Asigna un bloque de memoria.
- Función 49H. Libera un bloque de memoria asignado por medio de la función 48H.
- Función 4AH. Modifica un bloque de memoria.
  Permite cambiar el tamaño de un bloque previamente asignado. Esta función normalmente es llamada para contraer bloques.

Existen varios motivos por los que una aplicación puede requerir de los servicios de las funciones de asignación/desasignación: Por *ejemplo*, al momento de cargar un archivo .COM, intrínsecamente se lleva a cabo una función de asignación al introducirlo en la memoria disponible del sistema. Otro ejemplo se tiene cuando los programas requieren memoria adicional para la creación de *buffers* o almacenamiento de datos. Más adelante veremos como nuestras utilerías residentes requieren de la creación de áreas temporales de datos para soportar un ambiente de ventanas. Y por último, mencionemos un caso que nos concierne en la realización de nuestro programa de *spooler* de impresión (aplicación residente de gran utilidad).

Este programa, al ser cargado, mantiene en RAM 2 rutinas manejadoras de cola de *spool*: una para memoria convencional y otra para *spool* en disco. Como el *default* es el *spool* para memoria convencional, la memoria ocupada por el código correspondiente al *spool* en disco será liberada para otros usos (es aquí donde haremos uso de una función de desasignación). Si, por el contrario, lo que se quiere es emplear el *spool* en disco, bastará modificar el parámetro *default* antes de cargar nuestra utilería. Para finalizar, recordemos que un programa residente poderoso tiene que brindar al usuario la posibilidad de poder eliminarse de memoria en cualquier momento, sin necesidad de apagar el sistema. Caso en el que también haremos uso explícito de las funciones de desasignación de memoria referidas en este apartado.

En el tema de administración de memoria se encuentran implícitos los aspectos relativos a la asignación, estructura, organización de la memoria, etc. Si pretendiéramos hablar de la administración de la memoria en su totalidad, tendríamos que tocar puntos que de ninguna manera tienen relación con esta tesis. Por ello, consideramos conveniente aclarar que para nuestro caso, administrar la memoria se traducirá en saber introducir y eliminar programas residentes en el orden más apropiado, conocer las pautas a seguir para lograr la coexistencia pacífica de varios TSRs en memoria, y aprender a manejar programas residentes procurando el mayor ahorro posible en memoria, entre otros.

Cada uno de estos puntos constituyen, prácticamente, temas que deben ser tratados por separado. A lo largo de este trabajo se irán desglosando oportunamente. Por ahora, y antes de finalizar la parte teórica relativa a los antecedentes, es necesario profundizar en el conocimiento de la constitución y funcionamiento de los bloques de control de memoria del DOS (punto del que se habló de manera somera al inicio de esta sección).

Los conceptos que a continuación se presentan sientan las bases para el desarrollo del capítulo "Problemas Involucrados en la Construcción de TSRs Poderosos" (Sección: Como Remover un TSR de Memoria).

LOS BLOQUES DE CONTROL DE MEMORIA DEL DOS.

Como ya se había mencionado, para mantener la pista de la asignación de memoria, a partir de la versión 2.0 del DOS, se construye un bloque de control de memoria (MCB) de 16 *bytes* antecediendo a cada bloque de memoria de la cadena.

Ahora bien, a pesar de que la información relativa al formato del MCB (también conocido como "cabecera de arena" -*arena header*- tomado de la terminología de UNIX), no está oficialmente documentada por *Microsoft*, existen varias publicaciones y libros de consulta que abordan este tema. De estas fuentes sabemos que únicamente los primeros 5 *bytes* del MCB son usados. Veamos a continuación que representan cada uno de ellos.

El primer *byte* (*byte* 0) solo puede tener un valor de 4DH o 5AH. Este ultimo valor indica que la cabecera en cuestión es la última en la cadena, y toda la memoria encontrada arriba de ésta permanece sin usar. El hexadecimal 4D significa que el bloque de control se encuentra en alguna parte intermedia de la cadena. La memoria encontrada directamente arriba de este MCB usualmente pertenece a otro programa o al DOS mismo.

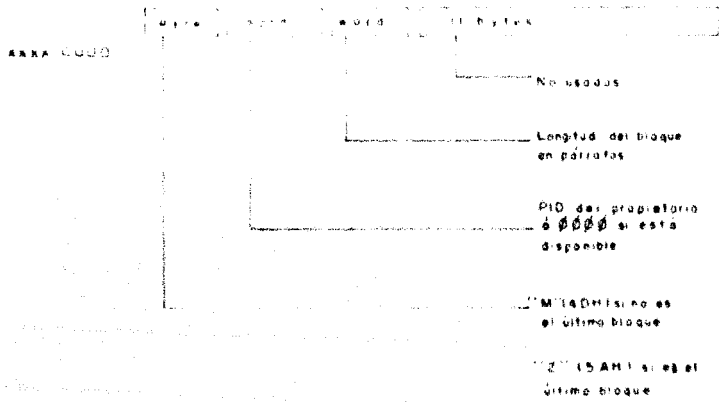
Los siguientes dos *bytes* contienen la dirección de segmento del PSP del programa al que corresponde ese MCB. Si su contenido es "0", debemos interpretar que el bloque de memoria está libre para satisfacer la demanda de memoria de algún programa. Cualquier otro valor representa una dirección de segmento.

Al inspeccionar el contenido de los MCB con un DEBUG, podremos comprobar que efectivamente cada programa posee su propio bloque de PSP, su bloque de medio ambiente, y el área de memoria que haya reservado por medio de la función 48H.

Los *bytes* 3 y 4 (es decir, los *bytes* cuarto y quinto en la cabecera) indican el tamano (en párrafos) del bloque de memoria en cuestión. Una vez encontrado un bloque de control, se puede encontrar el siguiente sumando a la dirección de segmento del MCB corriente, la longitud del bloque de memoria mas 1. Es por ello que se dice que los MCB están ligados lógicamente. Y basta explorar la cadena para que el DOS pueda determinar la estructura de la memoria corriente.

¿Cómo encontrar el primer MCB? El DOS mantiene un registro con la dirección del primer bloque de control de memoria. Sin embargo, la colocación de este registro puede variar dependiendo de la versión de DOS empleada, de que se tenga instalado en el sistema disco duro o a veces del tipo de máquina que se este utilizando. Por ello nosotros recomendamos emplear el siguiente método: explorar la memoria desde el fondo buscando un parrafo cuyo primer *byte* tenga un valor de 4DH, lo cual implica haber encontrado segmento y *offset* del primer MCB. A la dirección de segmento encontrada más 1, se le suma la longitud del bloque de memoria para obtener la dirección del siguiente MCB en la cadena. Estando en este punto, cualquier programa podrá examinar la cadena y determinar la información de asignación crítica.

FIG.1.5 ESTRUCTURA DE UN BLOQUE DE CONTROL DE MEMORIA DEL DOS



RASTREANDO LA CADENA DE MCB's

El DOS utiliza los MCB para subdividir la memoria en diferentes procesos. Cada MCB es semejante a un registro de 16 *bytes* que, en resumen, puede determinar: la cantidad de memoria que viene despues de el, que proceso posee esa parte de la memoria y si es el ultimo MCB en la cadena. (Ver fig.1.5).

El *byte* 0 de un MCB mantendra un letra "M" (4DH) si no es el último de la cadena, y una letra "Z" (5AH) en caso contrario.

Los *bytes* 1 y 2 forman una palabra (*byte* bajo y *byte* alto respectivamente) que identifica al programa "propietario" de la memoria que acompaña a ese MCB. Esta palabra (número hexadecimal de 4 digitos) constituye lo que se conoce como ID del Proceso (PID) o Identificador de Proceso; que, para versiones corrientes del DOS, no es otra cosa que la dirección de segmento del Prefijo de Segmento del Programa (PSP) del proceso que asignó memoria en ese bloque.

Los *bytes* 3 y 4 forman una segunda palabra que contiene el número de párrafos de memoria asignados al bloque de memoria asociado a ese MCB particular por el sistema operativo. (Notar que el MCB no se incluye en la cuenta).

Los 11 *bytes* restantes no son usados por las versiones corrientes del DOS. Razón por la cual es posible que puedan contener valores dejados por otros programas.

La fig. 1.6 muestra una sesión de DEBUG en la que podemos apreciar la liga lógica existente entre los MCB del DOS. Después de analizar este ejemplo, entenderemos completamente los conceptos que aquí hemos venido manejando y seremos capaces de poder rastrear correctamente la cadena de bloques en RAM.

FIG.1.6 SESION DE DEBUG QUE RASTREA LA CADENA DE MCB's

```
                        ; OBTENER EL SEGMENTO DE LA COPIA CORRIENTE
                        ; DEL COMMAND.COM, ALMACENADO EN EL VECTOR
                        ; DE INTERRUPCION 2EH.

-D 0000:00BA L 2
0000:00B0                 970B

                        ; EL MCB ESTA LOCALIZADO EN EL PARRAFO ANT.
                        ; B97-1 = B96.
                        ; DESPLEGAR EL MCB.

-D 0B96:0000 L 10
0B96:0000 4D 97 0B D3 00 00 00 00-00 00 00 00 00 00 00 00 M....


                        ; BYTE 0 = 4DH, INDICANDO UN MCB.
                        ; BYTES 1 Y 2 FORMAN LA PALABRA SEÑALANDO
                        ; EL PROCESO PROPIETARIO, 0B97 EN ESTE CASO
                        ; BYTES 3 Y 4  FORMAN LA PALABRA QUE INDICA
                        ; LA LONGITUD DEL MCB.
                        ; (DIR MCB + 1) + (LONG BLOQUE) = SIGUIENTE
                        ; MCB EN LA CADENA.
-H B97 D3
0C6A     0AC4

                        ; ESTA INSTRUCCION MUESTRA SUMA Y DIFERENCIA
                        ; ENTRE LAS CANTIDADES SEÑALADAS.
                        ; USANDO LA SUMA DESPLEGAR EL SIGUIENTE MCB

-D 0C6A:0000 L 10
0C6A:0000 4D 7A 0C 03 00 00 00 00-00 00 00 00 00 00 00 00 Mz...

                        ; CALCULAR LA DIRECCION DEL PROXIMO BLOQUE
                        ; DE LA MISMA MANERA.
```

```
-H C6B 3
0C6E   0C68

-D 0C6E:0000 L 10
0C6E:0000 4D 97 0B 0A 00 00 00 00-00 00 00 00 00 00 00 00 M....

                        ; CADA BLOQUE ES UN  ESLABON  EN LA CADENA.
                        ; EL DOS RASTREA ESTA CADENA CADA  VEZ  QUE
                        ; RECUPERA EL CONTROL.

-H C6F A
0C79   0C65

-D 0C79:0000 L 10
0C79:0000 5A 7A 0C 86 73 00 00 00-00 00 00 00 00 00 00 00 Zz..S..

                        ; EL VALOR 5AH  INDICA QUE  EL BLOQUE  ES EL
                        , ULTIMO EN LA CADENA.  LA  DIRECCION  TOTAL
                        , DEBERA  CORRESPONDER  AL  FINAL  DE  LA
                        , MEMORIA.

-H C7A 7386
8000   98F4

                        ; 8000H = 512  K  (DE MI MAQUINA).
                        ; SALIR DEL DEBUG.


-Q
```

EL PREFIJO DE SEGMENTO DE PROGRAMA (PSP)

Hemos separado este tema para su estudio particular, debido a que el Termino "PSP" sera mencionado muy frecuentemente a lo largo de este trabajo. La razón de su importancia, radica en que siendo una estructura que el DOS construye para cada programa, y que es usada por el mismo para reconocer el proceso corriente en el sistema; al establecer un ambiente de TSRs, deberemos cuidar que el DOS vaya distinguiendo, en cada momento, qué proceso es el que se encuentra presente en el sistema (si un TSR o el programa que éste interrumpe). Ahora, para que el DOS pueda reconocer como proceso actual a un TSR, al momento de su activación, deberá llevar a cabo algo que se conoce con el nombre de "conmutación de PSPs", y que consiste en cambiar el contenido de una localidad mantenida por el DOS en la que se encuentra el PSP del proceso presente, al de la dirección del PSP del TSR. Posteriormente, una vez que el TSR haya terminado tenemos que volver a hacer la conmutación, pero ahora para devolver al DOS el PSP del programa interrumpido. Estos conceptos son ilustrados en el capítulo III. Por ahora, sólo conviene aclarar que, no basta realizar la conmutación de PSPs para que el DOS pueda aceptar al TSR como proceso corriente. Indudablemente esta fase es de vital importancia, sin embargo existen otras que deben ser llevadas a cabo para asegurar el buen funcionamiento de todo programa residente, y que vienen a complementar el proceso de conmutación de PSPs, éstas : la "conmutación de pila" y la "conmutación de DTA", que serán vistas en el capitulo antes referido.

El PSP es un área de control de 256 *bytes* encontrada al inicio de la memoria destinada por el DOS a cada programa en el área de TPA. El PSP contiene muchos campos que el DOS utiliza para manejar el ambiente de procesamiento del programa. A continuación estudiaremos, de manera general, cada uno de ellos.

ESTRUCTURA DEL PSP.

El PSP está formado por varios campos que podemos apreciar en la fig. 1.7. Pero antes de hablar en particular de cada uno de ellos, queremos aclarar que la mayoría de los aquí presentados forman parte de la información confidencial de *Microsoft* e IBM; y que ha sido posible incluir gracias a las investigaciones de escritores de *software* (también conocidos como *hackers*), que en su tarea de descubrir el funcionamiento de cualquier tipo de paquete lanzado al mercado, han querido compartir con el mundo sus resultados.

Mientras no se especifique lo contrario, las características del PSP aquí mencionadas, son verdaderas para las versiones del DOS siguientes: 2.0, 2.1, 3.0, 3.1, 3.2, 3.3 y 4.0. A continuación presentaremos la documentación de cada uno de los campos del PSP.

(PSP:0) Llamada a interrupción para terminación de proceso.

Este campo contiene la instrucción de máquina **INT 20H**; su propósito es soportar programas convertidos de CP/M a DOS. Bajo CP/M los programas son terminados por sí mismos con una llamada o salto a la localidad de memoria 0.

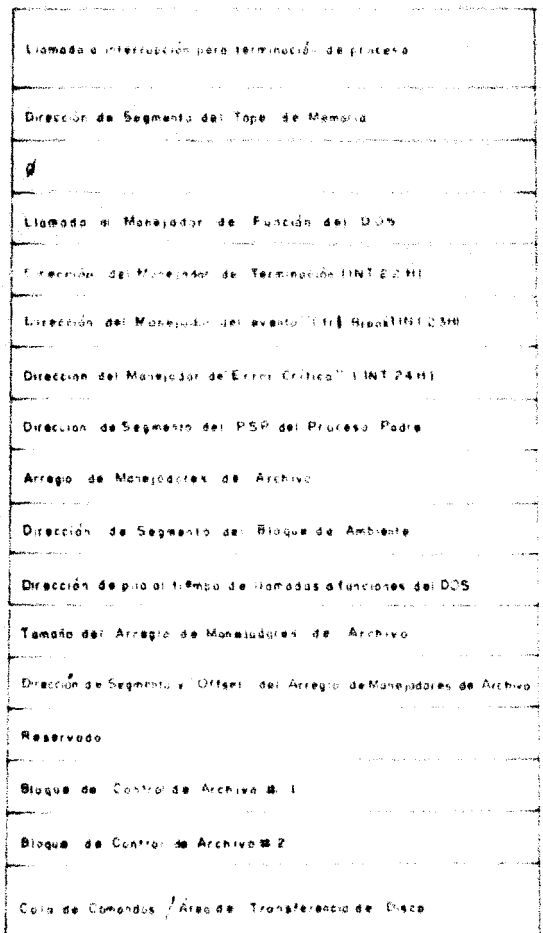
(PSP:2) Dirección de Segmento del Tope de la Memoria.

Cuando un programa es ejecutado, el DOS asigna un bloque de memoria dentro del cual el programa es cargado. Este campo contiene la dirección de segmento del tope de dicho bloque de memoria.

(PSP:4) Dirección del Manejador de Terminación.

En este campo el DOS salva el contenido del vector de interrupción 22H al momento de ejecutarse un programa. Al término de éste, el DOS restablece el vector tomando el valor salvado en este campo. El vector de interrupción 22H apunta al programa manejador de terminación del sistema.

FIG.1.7 PREFIJO DE SEGMENTO DE PROGRAMA (PSP)



(PSP:EH) Dirección del Manejador del evento Ctrl-Break.

Al ejecutarse un programa, el DOS salva en este campo el contenido del vector 23H (que apunta al manejador Ctrl-Break). Cuando el programa termina, el DOS restablece el vector 23H a su valor original recuperando el contenido de este campo.

(PSP:12H) Dirección del Manejador de Error Crítico.

Cuando un programa es ejecutado, el DOS salva el contenido previo del vector de interrupción 24H en este campo. Cuando el programa finaliza, el DOS restablece este vector utilizando el campo salvado. El vector de interrupción 24H apunta al manejador de Error Crítico del DOS

Los tres vectores de interrupción anteriormente mencionados (22H, 23H y 24H) son restablecidos cuando un TSR termina y se declara a si mismo residente. Así si el TSR necesita interceptar dichas interrupciones, deberá ligarse a ellas cada vez que sea activado. Los conceptos de "ligado de interrupciones" serán vistos en el siguiente apartado.

(PSP:16H) Dirección de Segmento del PSP del Proceso Padre.

El programa en ejecución en el sistema es llamado proceso padre. Se identifica como proceso hijo a aquel que haya sido ejecutado como resultado de una nueva llamada al DOS por parte del proceso padre. Usualmente el proceso padre es el procesador de comandos del DOS (**COMMAND.COM**), aunque efectivamente cualquier programa puede ser padre de otro.
Este campo del PSP contiene la dirección de segmento del PSP del programa padre. Como el procesador de comandos no tiene otro padre que el DOS, este campo en el PSP del **COMMAND.COM** contiene la dirección de segmento de PSP del mismo procesador de comandos. Es decir, contiene un apuntador a si mismo.

(PSP:18H) Tabla Manejadora de Archivos.

Este campo está formado por un arreglo de 20 *bytes*, cada uno de los cuales representa un manejador de archivo. Cuando un programa determinado abre un archivo, el DOS asigna a dicho programa un manejador de la tabla para ser usado cuando requiera de llamadas al DOS para leer o escribir registros en el archivo.

Los programas en lenguaje C que usan funciones de flujo de E/S, no direccionan manejadores de archivo; directamente emplean el apuntador a la estructura FILE definida en el archivo **stdio**. Sin embargo, en si la librería de funciones de C que soporta flujo de E/S si hace uso de los manejadores, ocultandolos de los programas que llamen a dichas funciones. El conjunto de los 20 manejadores, que forman este campo, a su vez están suscritos dentro de otras tablas, referenciadas como Tablas de Control de Archivo del DOS.

(PSP:2CH) Dirección de Segmento del Bloque de Ambiente.

Este campo mantiene la direccion de segmento del Bloque de Ambiente que el DOS establece para cada programa al ser ejecutado. Al ser cargado un programa y ejecutarse la función **EXEC**, el sistema genera una estructura de datos conocida como "Bloque de Ambiente" del *programa* (*Environment Block*) que es asignada en memoria; y que *puede* ser desasignada por una tarea si esta no requiere de las colocaciones de ambiente. El sistema coloca el apuntador al segmento de este bloque en el PSP con *offset* 002CH. El Bloque de Ambiente mantiene cierta información requerida por el Intérprete de Comandos del sistema (**COMMAND.COM**) y puede tambien contener información usada por programas transitorios. El Bloque de Ambiente no tiene efectos sobre la propia operación del sistema operativo, y su tamaño máximo es de 32 kb. Bajo condiciones normales, el Bloque de Ambiente deberá contener al menos tres *strings*: "**COMSPEC** = (variable)", "**PATH** = (variable)" y "**PROMPT** = (variable)".

Estos *strings* son colocados dentro del Bloque durante la inicialización del sistema, al momento de la interpretación de los archivos **CONFIG.SYS** y **AUTOEXEC.BAT**; y facilitan al **COMMAND.COM** la siguiente información: colocación de su archivo ejecutable, donde encontrar los comandos externos ejecutables, y el formato del *prompt* de usuario respectivamente.

(PSP:2EH) Direccion de Pila durante llamadas a funciones del DOS.

Cuando un programa llama a una función del DOS, el sistema salva en este campo el segmento de pila (*stack segment*) y los registros de apuntador del programa. Posteriormente, el DOS conmuta hacia su propio *stack* para la ejecución de la función. Antes de regresar el control al programa, el sistema recupera los valores salvados en este campo para restablecer su segmento de pila y sus registros de apuntador.

(PSP:32H) Tamano de la Tabla de Manejadores de Archivo

En este campo se guarda una variable entera que indica el número de entradas permitidas en la tabla de manejadores de archivo. Esta variable es conocida como "contador de entradas" y normalmente tiene un valor de 20. Cabe aclarar que este campo no tiene valor o uso en versiones del DOS anteriores a la v3.0.

(PSP:34H) Direccion de la Tabla de Manejadores de Archivo.

Este campo mantiene un apuntador de tipo **long** a la tabla de manejadores de archivo. Su valor de segmento usualmente coincide con el de la direccion de segmento del PSP, y su *offset*, por lo general, es 18H. Este campo no tiene valor o uso en versiones anteriores a la v3.0 del DOS.

El fin de los dos campos anteriores (tamaño y dirección de la tabla de manejadores de archivo) es el de permitir a un programa extender el numero maximo permitido de archivos concurrentemente abiertos. Esto se puede llevar a cabo de la siguiente manera: Asignando una nueva tabla de mayor tamaño. Cambiando el contador de entradas de la tabla de manejadores al valor requerido, para ello bastará que modifiquemos el valor asignado a la variable **FILES=** del archivo **CONFIG.SYS**, ya que durante la etapa de inicialización del sistema el campo del PSP que contiene el contador de entradas de la tabla tomará su valor según la variable **FILES**.

También se deberá modificar el valor del campo que mantiene el apuntador a la tabla, al valor de la dirección de la nueva tabla. Y copiando los 20 valores de la vieja tabla dentro de la nueva.

(PSP:5CH) Bloque de Control de Archivo # 1.

El DOS construye este campo, cuando sobre la Linea de Comandos aparece el nombre de un archivo como primer parámetro. En el Bloque de Control de Archivo, comunmente conocido como FCB (*File Control Block*), se construyen varias funciones de archivo que el programador puede controlar directamente. Las funciones del FCB permiten al programador crear, abrir, cerrar y borrar archivos; así como leer o escribir registros de cualquier tamaño dentro de un determinado archivo. No obstante, estas funciones no soportan la estructura de archivos jerárquica, introducida a partir de la versión 2.0 del MS-DOS, lo cual significa que sólo pueden ser usadas para accesar archivos del subdirectorio corriente.
Existe otra alternativa para manejar archivos que supera la limitante de las funciones del FCB, ésta se basa en un conjunto de funciones conocidas como "*handle functions*". En el presente, ambos métodos siguen siendo empleados.

(PSP:6CH) Bloque de Control de Archivo # 2.

El DOS construye este campo cuando, sobre la Linea de Comandos aparece el nombre de un archivo, como segundo parámetro. Este campo funciona de la misma forma que el anterior.

Los dos campos anteriores son proporcionados por el DOS para soportar programas convertidos del ambiente de CP/M.

(PSP:80H) Cola de Comandos / Area de Transferencia de Disco.

Este campo también sirve para soportar programas del ambiente de CP/M. Al escribir sobre la Linea de Comandos el nombre de un programa (para su ejecución), se lleva a cabo un analisis sintáctico a partir del segundo caracter encontrado después del nombre del programa. Esto, cuando se haga uso de la facilidad que proporciona el PSP de permitir pasar información, a un programa determinado, en forma de parámetros. La especificación de dichos parámetros se lleva a cabo en Linea de Comandos al momento de invocar a nuestro programa. Por ejemplo:

```
PROG1 PARAM1 PARAM2
```

Los caracteres que generan cada uno de los parámetros son colocados por el DOS en el PSP con *offset* 81H. La longitud de la cadena que forma el parametro es almacenada en el PSP con *offset* 80H. Ademas, debido a que los parámetros a menudo representan nombres de archivos, éstos son formateados dentro de los Bloques de Control de Archivo encontrados en el PSP, *offsets*: 5CH y 6CH. Una vez que el programa ya está en marcha, este campo se convierte en lo que conocemos como "Area de Transferencia de Disco" (DTA), que es un *buffer* de lectura y escritura para archivos abiertos bajo las funciones del DOS del FCB. Este campo también es utilizado por las funciones del DOS que manipulan directorios de archivo.

INTERRUPCIONES

El estudio de las interrupciones bajo el DOS constituye una parte importante de esta tesis, por lo que dicho tema no puede ser tratado con displicencia. No podemos hablar de TSRs sin hacerlo de las interrupciones. Ambos temas van de la mano pues un TSR depende de ciertas interrupciones para funcionar. En primera instancia mencionemos el caso de la activación del TSR. Sabemos que un tipo muy común de programas TSR requiere de una *hotkey* para su activación y aparición en pantalla. Dicho programa, para poder detectar la *hotkey*, debe proveer un mecanismo que pueda "interceptar" la interrupción de teclado y comparar los valores de tecla sensados con el patrón que forma la *hotkey*. Existen muchos otros ejemplos que ilustran el manejo de interrupciones por parte del programa TSR, pero no concierne tocar esos puntos por el momento. Solo baste saber que tanto para su activación, como para su ejecución el TSR hace uso de múltiples interrupciones. Antes de terminar esta breve introducción, queremos hacer notar que si se quiere, el TSR puede ser visto como una interrupción (desde un punto de vista particular): en primer lugar, al momento de su activación, el TSR suspende un programa, tal como una interrupción detiene un proceso. En segundo lugar, al darse una interrupción se salva el contexto del programa interrumpido para tener una referencia y poder regresar a él al término de la misma; tal proceso debe seguir un TSR para que, cuando el usuario quiera desactivarlo momentáneamente pueda regresar el control al proceso que suspendió y en el punto que se haya quedado.

Por otra parte, y para terminar; de muchas maneras, el concepto de programas residentes está ligado con el manejo de interrupciones de la IBM PC. Las interrupciones de *hardware* y *software* constituyen el mecanismo principal de control y comunicación entre un programa de aplicación residente y el sistema operativo.

Así, entendiendo la forma en que trabajan las interrupciones en la PC, habremos cubierto una parte importante del camino para el diseño de aplicaciones residentes en memoria.

■ QUE ES UNA INTERRUPCION ?

Una interrupción es una señal de *hardware* que le indica al procesador que un evento, que requiere atención especial, ha sucedido. El procesador (CPU) suspende momentáneamente la tarea que le ocupaba para poder atender la interrupción.

Sin el sistema de interrupciones de la PC, el procesador tendría que estar constantemente verificando la ocurrencia de eventos externos. Con el sistema de interrupciones, el procesador puede atender una tarea determinada y a la vez responder a un evento tan pronto ocurra.

A continuación se muestra la estructura general del proceso de interrupción en la PC, misma que un programador debe tomar en cuenta para hacer sus propias interrupciones:

- Salvar todo lo que el procesador no salvo de manera automática cuando ocurrió la interrupción.
- Bloquear cualquier interrupción que pueda interferir con la operación del manejador (cada interrupción tiene asociado un manejador, que es el programa ejecutor de la propia interrupción. De esto se hablará más adelante en el tema de "vectores de interrupción").
- Habilitar las interrupciones que pueden desempeñarse sin problema durante la operación del manejador.
- Manejar la interrupción.
- Restablecer los registros del procesador salvados en el primer paso.
- Rehabilitar las interrupciones.
- Regresar al procesamiento normal.

■ CLASIFICACION DE LAS INTERRUPCIONES.

Las interrupciones de la familia de microprocesadores 8088, se pueden clasificar en tres grupos basicos:

a) Interrupciones de *Hardware* Interno.
b) Interrupciones de *Hardware* Externo.
c) Interrupciones de *Software*.

a) Interrupciones de *Hardware* Interno.
Estas interrupciones se encuentran alambradas en el sistema, y manejan situaciones especiales en el mismo, tales como "error por division entre cero", u otras condiciones de error. Este tipo de interrupciones son generadas en el CPU.

b) Interrupciones de *Hardware* Externo.
Estas interrupciones pueden ser de dos tipos: mascarables o no mascarables.

> **Mascarables:** Son interrupciones generadas por dispositivos externos fisicamente conectados al microprocesador de la IBM PC, tales como el teclado, unidades de disco, reloj, etc. Estos dispositivos se encuentran tambien conectados al "chip" 8259A (Controlador Programable de Interrupciones -PIC-) que prioritiza y controla las interrupciones mediante ***software***. El 8259 puede ser mascarado individualmente; es decir, las interrupciones mascarables pueden ser deshabilitadas y el orden de las mismas modificado.
> Las señales que denotan este tipo de interrupciones son conocidas bajo las siglas **"INTR"**. En muchos casos, los fabricantes de computadoras personales establecen los niveles de interrupción, para los diferentes dispositivos, de manera fija e inamovible; sin embargo, otros proporcionan *switches* o *jumpers* para permitir cambiar dichos niveles.

No Mascarables: Estas interrupciones son reconocidas bajo la abreviación **"NMI"** y tienen la característica de no poder ser deshabilitadas bajo ninguna circunstancia, a diferencia de las anteriores. El microprocesador tiene reservado un pin específico para señales **NMI**.
La Interrupción No Mascarable (**NMI**) es usada para señalizar situaciones "catastróficas" en el sistema, tales como falla en memoria o pérdida inminente de energía. En la PC y compatibles, esta interrupción es activada al ocurrir un error de paridad en memoria; es decir al haberse detectado un problema grave en la memoria del sistema. El manejador estándar para esta interrupción escribe el mensaje siguiente en la pantalla: *"Memory Parity Error"*, e inmediatamente detiene el trabajo de la computadora. La interrupción **NMI** es controlada por el manejador apuntado por el vector de interrupción 02H. Si el programador, por algún motivo quisiera cambiar el manejador por otro, para evitar que al ocurrir la interrupción, el sistema pare; deberá simplemente cambiar la dirección apuntada por el vector de interrupción, para que apunte a un manejador que omita cualquier tipo de acción. Pero, por supuesto esto no es recomendable.

NOTA:

Las interrupciones de *hardware* son raramente utilizadas por el usuario de la PC o por un programador de aplicaciones, a menos que se requiera de un control estricto del *hardware*. Sin embargo, para nuestro caso, será indispensable manejar dicho tipo de interrupciones. Ya que, no obstante en su mayoría sólo conciernen a los diseñadores de componentes y programadores de sistemas; también son utilizadas, aunque en menor grado, por los diseñadores de aplicaciones residentes en memoria; que por lo general, tratan mínimamente con 2 de ellas: con la interrupción de teclado y la de *timer*.

c) **Interrupciones de *Software*.**

La familia del microprocesador 8088 reconoce 256 interrupciones distintas. Las cuales pueden ser invocadas por un programa mediante la instrucción especial **"INT"**. A las interrupciones generadas de esta manera se les conoce como interrupciones de *software*. Estas son tratadas de manera muy parecida a las de *hardware*, aunque su estudio no redunde en un conocimiento de los chips de la computadora como el de las anteriores.

En otras palabras, a pesar de que ambos tipos de interrupciones son manejadas de la misma manera, provienen de fuentes distintas. Las interrupciones de *software* deben ser vistas como peticiones hechas al sistema operativo para obtener algun servicio, y las otras como indicaciones de alguna condición determinada del *hardware*.

Las interrupciones de *software* tienen 3 niveles: interrupciones del BIOS, interrupciones del DOS e interrupciones de programas de aplicacion.

Interrupciones del BIOS: IBM dispuso ciertas interrupciones como único medio para tener acceso al *hardware* de la computadora. Estas son las interrupciones del BIOS.

Dichas interrupciones, en relación con las del DOS, son más limitadas en el sentido de que no abarcan el cumulo de funciones que éstas ultimas ofrecen al programador. Sin embargo no fueron diseñadas con este fin. Las funciones del BIOS son la plataforma de las del DOS. Algunos programadores que por razones de rapidez, optan por elaborar programas que manipulen el *hardware* directamente, evitando las funciones del BIOS, corren el riesgo de elaborar programas no portables; es decir, no pueden garantizar su funcionamiento sobre computadoras compatibles con IBM.

Las interrupciones del BIOS proporcionan un conjunto básico de operaciones para utilizar los servicios de bajo-nivel facilitados por la IBM PC; tales como entrada de teclado, salida de despliegue y E/S de disco (aunque ésta última de manera rudimentaria).

Interrupciones del DOS: En la realización del sistema operativo MS-DOS, *Microsoft* diseño ciertas interrupciones como medio por el cual los programadores pudieran tener acceso a los recursos del sistema. Éstas, son las interrupciones del DOS. La mayoría de los programas hacen uso de ellas. Las interrupciones del DOS son la interfaz de interrupción vía *software* que ofrece el sistema.

Interrupciones de Programas de Aplicación: El DOS proporciona la capacidad de crear nuevas interrupciones, para lo cual reserva ciertos vectores de interrupción (60H - 67H). Cada vector contiene la dirección de inicio de la nueva rutina de interrupción. Estas interrupciones son generadas al ejecutarse la función INT <num. interrupción> (al igual que las interrupciones del sistema). La diferencia radica en que son interrupciones nuevas escritas en su totalidad por programadores, o bien interrupciones nuevas sí, pero basadas en las originales, agregando sólo determinadas modificaciones.

Tanto los servicios del DOS como los del BIOS son invocados, como ya se dijo, mediante interrupciones de *software*. Sin embargo, la ejecución de dichas interrupciones depende, en muchos casos, del lenguaje de programación empleado.

Algunas interrupciones tienen asociadas varias funciones y subfunciones (como es el caso de la Int 21H). Así, para llevar a cabo una operación determinada se debe cargar el número de función deseado en el registro AH y el de subfunción en el registro AL, para posteriormente ejecutar la instrucción **INT "<no. de interrup.>"**. En adición a la especificación de la función y subfunción, para solicitar un servicio del DOS por lo general también se requiere de parámetros, según el tipo de función, proporcionados a través de otros registros del microprocesador. Su uso varía dependiendo de las características del servicio del DOS solicitado, y en algunos casos de la versión del mismo sistema operativo.

A continuación se muestra la secuencia general para invocar un servicio del DOS exitosamente:

1.- Cargar los registros necesarios con los parámetros apropiados, según el servicio demandado.
2.- Si la interrupcion del DOS solicitada es empleada por varias funciones, cargar el registro AH con el número de función adecuado.
3.- Cargar el registro AL con el número de subfunción respectivo (si la interrupción lo requiere).
4.- Invocar la interrupción del DOS.
5.- Examinar cualesquiera valores regresados para su validación y uso.

NOTA:

Las interrupciones de *software* son de vital importancia para los diseñadores de aplicaciones que utilicen lenguaje ensamblador o algún lenguaje de alto nivel; debido a que constituyen el puente entre el código de las aplicaciones y el sistema operativo de la PC. De entre las interrupciones de *software* más populares, se encuentra la Función Universal del DOS accesada mediante la instruccion INT 21H. Esta, es una interrupción de propósito general que permite al programador ejecutar cualquier operación del DOS directamente. Nosotros emplearemos mucho esta interrupción sobre todo para llevar a cabo el manejo de archivos.

En si, las interrupciones de *software* son extensamente utilizadas por los programadores de utilerías residentes. Adelantemos que, en esta tesis se hara uso de las interrupciones del BIOS para controlar todo lo que respecta al despliegue de video, y las interrupciones del DOS para las operaciones restantes. En capítulos posteriores podremos conocer las principales interrupciones que un TSR emplea para poder activarse, funcionar y validar ciertas condiciones.

VECTORES DE INTERRUPCION.

Atendiendo a la necesidad de exponer los conceptos básicos involucrados en el diseño de nuevas Rutinas de Interrupción para soportar TSR's, hemos decidido tratar este tema por separado.

En la arquitectura de la PC existen 256 interrupciones, numeradas del 0H al FFH. De todo el conjunto de interrupciones, existen cierto grupo definido para el uso exclusivo del procesador (por ejemplo la interrupcion 0H "Int. de División entre cero"), otro grupo para invocar funciones del ROM-BIOS, y un tercer grupo para solicitar los servicios del DOS. Ver tabla de vectores de interrupción fig. 1.8.

Cabe aclarar que existe otro grupo de interrupciones que no es clasificado juntamente con los anteriores, por no formar parte de las interrupciones originales de la PC, y por considerarse implícito en el grupo de interrupciones del DOS. Este grupo esta formado por las interrupciones llamadas de "usuario", numeradas desde el valor 60H al 67H (que aparecen en la fig. 1.8 como interrupciones "no usadas") y que, como su nombre lo indica, están reservadas para ofrecer al programador la oportunidad de crear Rutinas de Servicio de Interrupcion (ISR's) de dispositivos propias.

Ahora, dentro del sistema, cada interrupción está representada por un vector de direcciones de 4 *bytes*, conocido como Vector de Interrupcion. Los 256 vectores asociados a cada una de las interrupciones se encuentran en la memoria baja de RAM, ocupando 1024 *bytes* y formando lo que conocemos como Tabla de Vectores de Interrupción. Las localidades de memoria absoluta, que el sistema reserva para la Tabla, van de la dirección 0000:0000H a la 0000:03FFH.

Cada vector de interrupcion contiene un apuntador al manejador asociado a la interrupcion correspondiente. Este apuntador esta dado por una direccion de segmento (mantenida en la segunda palabra de cada vector de interrupción), y un *offset* (encontrado en los dos primeros *bytes* de cada vector de interrupción).

FIG.1.8 TABLA DE VECTORES DE INTERRUPCION

NUMERO	DIRECCION	FUNCION
0H	000-003H	Division Entre Cero
1H	004-007H	Paso sencillo
2H	008-00BH	No Mascarable
3H	00C-00FH	Punto Ruptura
4H	010-013H	Sobreflujo
5H	014-017H	Impresión Pantalla
6H	018-01BH	Reservado
7H	01C-01FH	Reservado
8H	020-023H	Reloj (18.2/seg)
9H	024-027H	Pulso de Tecla
AH	028-02BH	Reservado
BH	02C-02FH	Puerto 1 RS-232
CH	030-033H	Puerto 0 RS-232
DH	034-037H	Disco Duro
EH	038-03BH	Disco Flexible
FH	03C-03FH	Reservado
10H	040-043H	Llamado E/S de Video
11H	044-047H	Llamada de Verific. de Equipo
12H	048-04BH	Llamada de Verific. de Memoria
13H	04C-04FH	Llamada E/S de Disco Flexible
14H	050-053H	Llamada E/S de Puerto Serie
15H	054-057H	Llamada E/S de Cassette
16H	058-05BH	Llamada E/S de Teclado
17H	05C-05FH	Llamada E/S de Impresora
18H	060-063H	Codigo de Entrada al ROM Basic
19H	064-067H	Inicializador (*Bootstrap loader*)
1AH	068-06BH	Llamada a hora del dia
1BH	06C-06FH	Obtiene Control sobre *BREAK*
1CH	070-073H	Obtiene Control sobre Reloj
1DH	074-077H	Tabla de Inicializacion de Video
1EH	078-07BH	Tabla de Param. de Disco Flexible
1FH	07C-07FH	Tabla de Caracteres Graficos
20H	080-083H	Terminación de Programa del DOS
21H	084-087H	Funcion Universal del DOS
22H	088-08BH	Direccion de Terminacion del DOS
23H	08C-08FH	*Ctrl-Break* del DOS
24H	090-093H	Vector de Error Fatal del DOS
25H	094-097H	Lectura de Disco Abs. del DOS
26H	098-09BH	Escritura de Disco Abs. del DOS
27H	09C-09FH	Función TSR del DOS
28H-3FH	0A0-0FFH	Reservados para el DOS
40H-7FH	100-1FFH	No usados
80H-F0H	200-3C3H	Reservados para Basic
F1H-FFH	3C4-3FFH	No usados

Los manejadores de interrupción, también conocidos como Rutinas de Servicio de Interrupcion (*Interrupt Service Rutine* "ISR's"), no son otra cosa que los programas ejecutados al ocurrir sus interrupciones asociadas. O en otras palabras, el código que origina la respuesta del sistema a la condición de interrupción presente.

■ QUE SUCEDE AL OCURRIR UNA INTERRUPCION ?

Cuando el sistema advierte la ocurrencia de una interrupción (ya sea de *hardware* o de *software*), el procesador responde a ella primeramente introduciendo en el *stack* los siguientes registros: FR (Registro de Banderas), IP (Apuntador de instrucción) y CS (Registro de Segmento de Código). Los registros CS:IP, mantienen la dirección de regreso al programa interrumpido. Posteriormente, deshabilita todas las interrupciones para garantizar que durante la ejecución de la ISR de la interrupción registrada, no hayan interrupciones de ningún tipo. A continuación, el procesador busca en el *bus* del sistema un numero de 8 *bits*, correspondiente al vector de interrupción proporcionado por el dispositivo que maneja la interrupción en cuestión.

El valor leído es comparado con las entradas o elementos de la Tabla de Vectores de Interrupcion hasta ser encontrado. Para ello el procesador simplemente multiplica por cuatro el valor del vector de interrupción leído; pues el producto directamente da como resultado la direccion de desplazamiento (*offset*) requerido para localizar dicho vector en la Tabla. La direccion de segmento del inicio de la Tabla es de antemano conocida: 0000H.

Una vez localizado el vector dentro de la Tabla, se lee su contenido (que es el apuntador a la ISR requerida para procesar la interrupción), y es cargado en los registros CS:IP. Finalmente el control del procesador es transferido a la dirección (segmento-*offset*) apuntada por los registros anteriores (CS:IP), para de esta manera dar inicio propiamente al procesamiento del manejador.

Ahora, durante el proceso del manejador de interrupción, ocurre lo siguiente:

- El primer paso seguido por la mayoría de los manejadores es rehabilitar las interrupciones, de manera que las de mayor prioridad puedan ser atendidas en primer orden.
- A continuación, y antes de iniciarse la ejecución propia del cuerpo de la ISR, son salvados los registros de máquina involucrados en el proceso.
- Durante la ejecución de la ISR; ciertos dispositivos requieren del reconocimiento de una señal especial (*acknowledge* "ack"), que les es enviada desde el procesador para poder determinar si la interrupción que originaron, está siendo ya atendida.

  Nota: Por su parte, los manejadores de interrupción, generalmente deben ser escritos cuidando 2 aspectos fundamentales: la rapidez y robustez. Por esta razón, en su mayoría, se encuentran escritos en lenguaje ensamblador. Dicho lenguaje ofrece al programador la eliminación de posibilidades de *overhead* y asegura rapidez en la ejecución. Otro lenguaje comunmente usado para este fin, es el lenguaje "C", que proporciona la ventaja de la simplicidad. En este trabajo nosotros haremos uso de ambos lenguajes: del lenguaje ensamblador, para la escritura del *spooler* de impresión; y del lenguaje C, para el diseño de las aplicaciones residentes, incluyendo manejo de ambiente de ventanas.
- Para finalizar su labor, el manejador de interrupción realiza los siguientes pasos: Si la interrupción ocurrida fue de *hardware* externo (alguna de las controladas por el 8259-A PIC), el procesador debe enviar una señal de fin de interrupción hacia éste. Posteriormente se restauran los registros de máquina (salvados antes de iniciar el proceso de la ISR) a sus valores originales.

Finalmente se ejecuta la instrucción de retorno de interrupción (IRET), que saca de la pila los valores del registro de banderas, el CS y el IP, restableciéndolos a los valores que mantenían antes de la ocurrencia de la interrupción, y como recordaremos los registros CS:IP conservaban la dirección de regreso al programa interrumpido. Así, el procesamiento del mismo es reanudado exactamente en el punto de interrupción.

■ ENCADENAMIENTO Y LIGADO DE INTERRUPCIONES.

Como ya se había mencionado, los TSR se valen de ciertas interrupciones para funcionar; empezando por la interrupción de teclado (en el caso de que el programa requiera de una *hotkey* para su activación) y continuando con la *interrupción de timer* (en caso de que el programa valide el problema de la reentrancia, o en el caso de tratarse de alguna aplicación que requiera de dicha interrupción -como es el caso del conocido programa que despliega la hora y fecha corrientes-). En fin, de acuerdo a la orientación de la utilería, será el tipo y número de interrupciones empleadas por los diferentes programas residentes.

Ahora bien, un programa residente, para poder funcionar sin causar estragos en el sistema, forzosamente deberá ligarse y encadenarse a los vectores de interrupción involucrados. Pero, para entender la naturaleza de este problema, adentremos en los conceptos de encadenamiento y ligado de *interrupciones*.

Cuando un programa residente se instala a sí mismo, deberá colocar, en los vectores de interrupción que empleará, una dirección que deberá apuntar a un código propio. Tomada esta acción, se puede decir que el programa TSR ha llevado a cabo la fase de ligado. Como vemos, en esta etapa el TSR modifica el contenido presente de los vectores de interrupción para dejarlos apuntando a las nuevas rutinas substitutas.

Para introducir el concepto de encadenamiento, analizaremos una situación concreta que lo ilustra plenamente: Supóngase que se cargan en memoria varios programas TSR, cada uno de los cuales, llegado su turno, se ligaron a la interrupcion de teclado (pues así lo requerían, por depender de una *hotkey* para su activación). Una vez culminada la fase de carga de los programas, el vector de interrupcion de teclado quedará apuntando a la ISR del último en haber sido cargado; así, al detectarse cualquier pulso de tecla, será esta rutina la encargada de atenderlo. Si dicho pulso de tecla corresponde al que la ISR espera para activar su utileria, no habrá ningún problema; pero en caso contrario, lo desconocerá e impedirá la activación del TSR correcto, al no tener manera de ceder el control a la ISR indicada.

El problema se solucionaría si los programas TSR además de ligar interrupciones salvaran las direcciones previas de los vectores de interrupción. Así, en un momento determinado, al mantener la pista de las rutinas antecesoras, todos los TSR puedan *transferir* el control de la interrupción de teclado a la ISR del programa correcto. En este caso se dice que los programas TSR han realizado el procedimiento de encadenamiento.

Las consecuencias dimanentes de omitir el proceso de encadenamiento son, *por lo general*, fatales (caida del sistema). Por ejemplo, si un programa se liga a la interrupción del *timer* sin prever su uso para el resto del sistema, los demás procesos manejados por eventos del *timer* quedarán deshabilitados mientras el programa esté instalado y mantenga la "ligadura". Caso en el que el reloj del sistema podría parar.

En nuestro cometido por diseñar aplicaciones residentes robustas, requeriremos ligar nuestros programas principalmente a las siguientes interrupciones: 09H de teclado, 1CH de *timer*, 34H de estado del DOS, 28H de DOSOK, 13H de disco, 24H de error crítico y 23H del evento Ctrl-Break. Mismas que (a excepción de las 2 últimas) requeriremos encadenar a los otros procesos que las necesiten.

Las interrupciones deben ser encadenadas en el código de inicialización del programa. Para encadenar una interrupción, primero hay que leer la dirección almacenada en el vector de interrupción. La librería de funciones de Turbo C incluye la función **getvect** para realizar dicha lectura. A continuación el valor leído debe ser guardado en una variable para que, al momento de presentarse un caso no contemplado por la nueva ISR, ésta pueda enviar el control de la interrupción en cuestión a la vieja ISR apuntada por el valor previamente salvado. Dicha ISR puede corresponder a otro TSR cargado con anterioridad, o ser la rutina original del sistema.

Todo programa TSR debe encadenar las interrupciones que se vean implicadas en su procesamiento. Así, se estará asegurando que aún teniendo varios TSRs cargados en el sistema, al ocurrir cierta interrupción requerida por mas de uno de ellos, el control de dicha interrupción será pasado de una rutina a otra, a lo largo de una cadena (empezando por la rutina del último programa TSR cargado y terminando con la del manejador original del sistema) hasta llegar a la ISR que tenga contemplado su caso.

Ahora, para llevar a cabo la fase de ligado, nuestro TSR deberá escribir la dirección de la nueva ISR dentro del vector de interrupción, para lo cual haremos uso de la función **setvect** de Turbo C.

En el capítulo "Procedimiento General para la Construcción de Utilerías Residentes en Memoria" Sección: "Cómo Remover un TSR de memoria", encontraremos ilustrados los conceptos manejados en este apartado.

LOS PROGRAMAS RESIDENTES EN MEMORIA

(TSRs)

CAPITULO 2

INTRODUCCION

Es común que los usuarios exigentes del sistema PC intenten obtener el mayor provecho posible de su máquina; para lograr esto existen varias alternativas. Una de ellas se basa en la utilización de paquetes que cubran las debilidades del MS-DOS; otra, consiste en la creación de rutinas propias que puedan llevar a cabo funciones que el DOS por si mismo no puede realizar; tales como el manejo de múltiples procesos a un tiempo. A pesar de que el DOS haya sido diseñado para ejecutar sólo un programa en memoria, existen formas de poder simular el procesamiento de multiples tareas; estas, se fundamentan en ciertas técnicas basadas en el *hardware* y la estructura de interrupciones del sistema operativo. Al referirnos a estas técnicas, en concreto estamos hablando de los programas de acronimo TSR (*Terminate and Stay Resident*) también conocidos como programas residentes. Los TSR son programas sofisticados en cuanto a su escritura. En el presente trabajo, se sugieren una serie de normas especificas para evitar al programador, en lo posible, todo tipo de problema en las fases de diseño, escritura, ejecución y uso del TSR.

TSR's
APECTOS GENERALES

■ QUE ES UN PROGRAMA TSR ?

Un programa TSR es simplemente código especial que al ser ejecutado desde la línea de comandos del MS-DOS, o internamente por otro programa, se queda residente en memoria o, en otras palabras, se queda "dormido" hasta ser activado por un evento.

Los programas TSR constituyen mejoraciones naturales del sistema operativo MS-DOS: creados para hacer más asequible el ambiente de una sola tarea al usuario de la PC. Hacer TSRs significa extender el DOS de manera que su arquitectua incluya programas que puedan permanecer residentes permanentemente en memoria.

Un programa residente es aquél que al ser ejecutado por vez primera, se establece en la memoria de usuario quedando latente en el sistema, sin dar indicio de su existencia hasta que un evento genere su activación.

Algunos de los tipos más populares de programas TSR son: el de mejoración de teclado, el de accesorios de escritorio, y el *spooler* de impresion. Los programas de optimización del teclado, tales como *Prokey* y *Superkey* (algunos de los primeros de su tipo) permiten al usuario asignar secuencias de caracteres a una "tecla de función" (ALT-<tecla>, F1, etc.). Los accesorios de escritorio tales como *SideKick* y *Homebase* proporcionan al usuario algunas de las siguientes facilidades: *notepads*, calculadoras, calendarios, teléfono automático, entre otras. Y por último, el programa *spooler* de impresión, en su acepción más sencilla, permite al usuario trabajar en la computadora y a la vez mandar a imprimir una lista de archivos "encolados".

Otros programas TSR incluyen: checadores de ortografía, extensiones del procesador de comandos del DOS, ayudas de *debug*, relojes y alarmas entre otros.

Los programas residentes mas populares han sido introducidos por vendedores comerciales y otros no tan elaborados, por algunas fuentes de *software* de dominio publico.

El enfoque original que se les dio a los programas TSR fue el de ISRs que constituían la vía para poder aumentar nuevas funciones al DOS. Es decir, los primeros programas residentes eran empleados sólo para agregar ciertas características al sistema según las necesidades particulares del usuario; la característica de estos programas era su capacidad de inicializarse por sí mismos y de ligarse dentro de la estructura de interrupciones del sistema. Estos primeros programas constituyen lo que en la actualidad se conoce como programas residentes pasivos. A continuación se habla de los tipos de TSR existentes.

■ CLASIFICACION DE PROGRAMAS TSR

Los programas TSR entran en 2 variedades: Rutinas de Servicio de Interrupción o TSRs pasivos y Utilerías Residentes en Memoria o TSRs activos.

a) Programas Residentes Inactivos o Pasivos.
Reciben también el acrónimo de ISRs, debido a que, tal como las ISRs del sistema, responden sólo cuando un programa hace un llamado a la interrupción que éstas representan. Al sensar la interrupción en cuestión, el TSR pasivo ejecuta una función específica, similar a una subrutina, para luego regresar el control al programa que originó la interrupción.
Sabemos que una ISR puede responder tanto a una interrupción generada por un dispositivo de *hardware* como a un programa; sin embargo está orientada a soportar dispositivos de *hardware*. Ejemplos: Las ISRs

que responden a la interrupción de *timer* (relojes, alarmas, etc), las ISRs para manejar el *mouse* (por ejemplo MOUSE.COM de Microsoft) que operan sobre interrupciones de *hardware* basadas en el movimiento del ratón, etc.

Un TSR inactivo trabaja en lo que podríamos llamar un ambiente "benigno", debido a que se ejecuta hasta ser llamado por otro programa. Sabemos que en el sistema PC sólo una operación puede estar en progreso a la vez, condición que no es violada por los TSRs pasivos, ya que por el hecho de ser llamados por otros procesos, se tiene la seguridad de que ninguna operación del DOS estaba en progreso; por lo que, un TSR inactivo puede hacer uso de las facilidades del DOS sin problema.

Ejemplos de TSRs pasivos, lo constituyen aquellos programas que responden unicamente cuando otro programa les pasa una petición especifica de servicio. Este tipo de programas son fáciles de escribir debido a que no requieren de "trucos" especiales de codificación (a diferencia de los activos).

b) Programas Residentes Activos.

Cuando los TSR fueron ligados a la interrupción de teclado, dejaron de ser programas pasivos. Con la nueva modificación los programas podían "decidir sobre sí mismos" respecto al momento de su actuación.

Las utilerías residentes en memoria son ISRs que comunmente no involucran manejo de *hardware* especial. Responden a un pulso de tecla o *hotkey* definida, lo que implica que el procesamiento del programa se efectuará sólo hasta detectar la petición de usuario.

Los TSR activos determinan y conservan las condiciones prevalecientes en el sistema previas al momento de su activación o interrupción, para que al momento de su finalización vuelvan a colocar el sistema a su estado original, como si nada hubiera pasado.

Otra característica de los programas residentes activos es la utilización de técnicas de ventana para

comunicarse con el usuario, debido a la propia naturaleza *pop-up* de los mismos.

Los TSR activos son el tipo mas comun de programas residentes. Ejemplo de ellos: todas las utilerias *pop-up* conocidas tales como *Sidekick*.

Los programas de esta clase suelen ser identificados por varios nombres, entre ellos: TSR *memory resident utility*, *desktop accessory*, *pop-up*, etc.

Los TSRs activos dan al usuario la impresion de *multitasking* en la PC. Decimos impresión porque en realidad solo simulan el ambiente multitarea, haciendo pensar al usuario que mas de una operacion esta sucediendo simultaneamente. Para lograr dar esta ilusión de operación simultánea, el código de los TSR tiene que incluir una serie de pasos adicionales (en relación a los TSRs pasivos) que vienen a complicar su diseño y escritura.

Por ejemplo, todo TSR activo robusto tiene que prever la caracteristica de "No Reentrancia" del DOS. Como sabemos el DOS no es reentrante, por lo que el TSR debe evitar, a toda costa, "romper" o interrumpir la secuencia de una rutina interna del DOS. Si el DOS está procesando algo (un acceso a disco por ejemplo) en el momento en el que un TSR, que a su vez involucre escritura en disco, sea activado se provocaria un serio problema en el sistema, sin mencionar los posibles daños causados al disco. El problema de la reentrancia será estudiado en capitulos posteriores. En sí, todo TSR activo debe "vigilar" tanto lo que hace el DOS como los usuarios, hecho que viene a incrementar la complejidad de este tipo de programas.

■ ACTIVACION DE UN TSR

Un TSR puede ser activado por 2 tipos de eventos: los externos y los internos. Los primeros son privativos de los programas TSR activos, y los segundos de los TSRs pasivos. Veamos en qué consiste cada uno de ellos.

a) Internos.
Son aquellos eventos basados en tiempo, o eventos originados por interrupciones generadas internamente por *otro* programa.

b) Externos.
Se generan al sensar la combinacion de teclas establecida para activar la utileria. A esta combinación de teclas nos referimos cuando hablamos de la *hotkey* (tecla caliente).

La activacion de un TSR resultante de un evento externo, provocara la suspension momentánea del procesamiento del programa corriente en el sistema, el tiempo que dure la ejecución del TSR. Al termino del TSR el programa interrumpido reanuda su labor. El proceso interrumpido puede ser un programa transitorio, otro programa residente, o bien el DOS mismo.

■ FUNCIONES DE RESIDENCIA

El DOS incluye 2 funciones que permiten a un programa declararse así mismo residente, y aunque entre ambas sólo existan diferencias menores, en un momento determinado esas diferencias pueden ser decisivas. Estas funciones son:

a) La función 31H Int. 21H del DOS.
Esta función provoca la terminacion del programa corriente permitiendole permanecer residente. La memoria ocupada por este programa será respetada por el DOS.

b) La Int. 27H lleva a cabo la misma función, pero restringe la longitud del programa residente a un máximo de 64K.

La intención original de Microsoft al poner estas funciones a disposicion de los usuarios, no fue la de soportar programas de utileria sino más bien la de permitir a los desarrolladores de sistemas escribir ISRs para manejar los dispositivos de *hardware* de E/S más usuales tales como el *mouse*, tableta digitalizadora, *joystick*, etc; dispositivos que no figuran como parte estándar del sistema. No obstante, podemos dar muchas más aplicaciones a las funciones de RESIDENCIA. El ambiente generado por las funciones TSR del DOS, puede soportar otro tipo de programas no necesariamente asociados con los dispositivos de *hardware* , pero que permiten extender la interfaz con el usuario (es decir, programas residentes activos).

Pero conozcamos el contexto para emplear las funciones de residencia: Cuando un TSR es corrido por primera vez, inicializa o construye sus tablas de memoria y se prepara para su ejecución posterior ligandose a una interrupción del DOS. Hecho esto, el programa determina la cantidad de memoria que requerirá para permanecer en memoria. Posteriormente coloca el valor 31H en el registro AH, el código de regreso en AL y el numero de párrafos (bloques de 16 *bytes*) que se asignarán al TSR en el registro DX. Cuando el programa queda instalado, el área de TPA es reducida y el código de salida es pasado de regreso al proceso padre. El procedimiento suena simple, pero ¿lo es?. Puede resultar simple en le caso de TSRs extremadamente simples. Pero de ninguna manera podrá ser así, si se intenta escribir una utileria que involucre facilidades de reloj, calculadora, editor de texto, etc.

■ COMO CARGAR UN TSR EN MEMORIA?

Los programas residentes se ejecutan desde el *prompt* del procesador de comandos, tal como si fueran programas transitorios normales de extension .EXE o .COM. La diferencia estriba en que el TSR empleará esta etapa para instalar su código de inicialización, retardando su activacion hasta sensar la *hotkey* o detectar el evento correspondiente.

De hecho, ni el DOS ni el procesador de comandos, en particular, tienen manera de saber con que tipo de programas se está tratando, no se sabe si llegarán a estar residentes en memoria o no. Y no es sino hasta la finalización de los mismos, cuando se puede definir dicha situación. Todo programa TSR, durante su primera ejecucion, lleva a cabo poco antes de terminar alguna de las funciones TSR del DOS.

Cuando un TSR es corrido por primera vez, inicializa o construye sus tablas de memoria y se prepara para su ejecución posterior ligandose a una interrupción del DOS. Hecho esto, el programa determina la cantidad de memoria que requerirá para permanecer en memoria. Posteriormente coloca el valor 31H en el registro AH, el código de regreso en AL, y el número de párrafos (bloques de 16 bytes) que se deberan reservar para el TSR en el registro DX. Cuando el programa queda instalado, el sistema reduce el área de TPA incrementando la dirección baja de la misma, justamente encima del programa residente. Finalmente, el código de salida es pasado de regreso al proceso padre.

El procedimiento suena simple, pero ¿lo es?. Puede resultar simple en le caso de TSRs extremadamente simples. Pero de ninguna manera podrá ser así, si se intenta escribir una utileria que involucre facilidades de reloj, calculadora, editor de texto, etc.

■ QUE PROGRAMAS PUEDEN SER RESIDENTES ?

Según la experiencia tenida, algunos de los programadores que inician su faceta como diseñadores de programas residentes empiezan a desarrollar una tendencia por querer hacer residente casi todo programa, les es atractivo tener todas sus aplicaciones disponibles al solo contacto de una tecla. Por ello es necesario aclarar que no todo puede o debe ser residente.

Recordemos, en primera instancia, que en tanto se agreguen programas residentes, el tamaño del TPA se verá decrementado, pudiendo llegar a un punto en el que no exista memoria suficiente ni para correr los programas transitorios normales que el usuario acostumbraba.

Se llegaría al extremo de encontrarse "ricos" en cuanto a utilerías *pop-up* se refiere, pero incapacitados hasta para escribir un "memo" en un procesador de palabras.

Por otro lado, debemos recordar también que el sólo hecho de manejar TSRs implica estar forzando los límites de diseño del DOS. Por lo que pueden tener lugar problemas de coexistencia entre TSRs. No es una sorpresa que algunos programas TSR provenientes de diferentes fuentes no puedan coexistir pacíficamente. La respuesta se basa en el hecho de que, en el caso de tener varios TSRs en el sistema, algunos de ellos requieren, para su buen funcionamiento, ser cargados en memoria al final de todos. Estos programas buscan ser los primeros en la cadena constituida por los demás programas que también han ligado a sí las interrupciones que los primeros manejan. Así, si pretendiéramos introducir al sistema más de un programa de estas características se podría ocasionar la caída del sistema. *Sidekick* es un ejemplo de estos programas; sin embargo, sus diseñadores han validado la posibilidad de que se presente el caso antes mencionado. *Sidekick* "roba" la interrupción de *timer* a cualquier programa cargado después de él.

Estos problemas han sido el resultado lógico de la pretensión por extralimitar la naturaleza del DOS. Los desarrolladores de TSRs, han tratado de llegar a una definición de programa residente de "buen comportamiento", implantando estándares. Y aunque hoy día conocemos ya muchos de los aspectos a considerar para la realización de TSRs robustos, no se ha llegado todavía a un acuerdo general.

Existen algunas alternativas viables para eliminar, en lo posible, el problema de coexistencia entre TSRs; una de ellas consiste en la creación de un programa manejador que lleve la función de arbitraje; es decir, de decidir a que programas pasar qué interrupción y en qué momento. *Borland*, fue la primera compañía que puso en práctica este método, promoviendo un manejador de ambiente TSR llamado *Sidekick* Plus.

Realizar un análisis exhaustivo de todos los problemas y cuestiones inmersas en este tema, nos podría desviar del punto al que queremos llegar en esta discusión. Hemos empezado a vislumbrar la existencia de una serie de trabas y obstáculos para la implementación de programas residentes. En concreto, podría pensarse que el realizar utilerías TSR conlleva tantos puntos que cuidar que, en realidad su creación no se ve ampliamente justificada. Queremos aclarar que esto puede resultar cierto, en caso de no tener claros todos los conceptos necesarios. Sin embargo si hacer TSRs no valiera la pena, compañías como *Borland* o Microsoft no hubieran perdido su tiempo, dedicando a este estudio su esfuerzo y recursos. Precisamente, uno de los objetivos de esta tesis es el de presentar, de una manera simple y "digerida", una metodología para la fácil creación de programas residentes en memoria. Se puede lograr hacer TSRs robustos, poderoso, útiles y prácticos. La clave consiste en que el programador encargado de su creación tenga en cuenta ciertos aspectos básicos como son: requerimientos de los usuarios y ambiente en el que se desenvolverá el TSR.

Retomando el tema original: "¿que puede ser residente y qué no?", presentamos a continuación las primeras pautas que pueden ayudar al programador de TSRs a poder decidir si conviene que su utilería sea convertida en un programa TSR o sea un programa transitorio normal:

- Tamaño de la Utilería.
  Es necesario cuidar el espacio de memoria del TPA. Debemos tener presente que un programa TSR grande no puede existir a expensas de programas transitorios que el usuario acostumbra usar. Si el programa no puede ser compilado y ligado bajo el modelo de memoria *tiny*; es decir, si su código de datos y de *stack* sobrepasan los 64K, entonces se debe contemplar la posibilidad de que no convenga hacer residente dicho programa. Cabe mencionar que en algunos casos son admisibles los programas compilados y ligados bajo el modelo de memoria *small*.

- Frecuencia de Uso de la Utilería.
  Es obvio que no podrán ser residentes programas de uso infrecuente. Y llamamos de uso infrecuente a aquellos que son corridos una o dos veces al dia.
  Un parámetro de comparación aceptable es: hacer residentes aquellos programas que son utilizados aproximadamente cada hora. Por ello es que decimos que para la creación de utilerías residentes, el programador debe conocer perfectamente los requerimientos del usuario.
  No se debe sobrecargar al sistema con programas TSR de uso esporádico.

- Grado de Disposición de Recursos de la Utilería.
  Si el programa que se desea hacer utilería *pop-up* requiere asignar demasiada memoria, o si la memoria del sistema es limitada para las funciones que realizara, podría correrse el riesgo de degradar su ejecución o funcionamiento, ese programa no debera ser residente. Por ejemplo, si un programa de estas características interrumpe la ejecucion de un archivo .COM; debido a que el DOS asume que el programa corriente (el archivo .COM) tiene toda la memoria, el espacio del que podrá disponer el TSR será muy limitado.

- Ambiente de desempeno.
  Es recomendable probar la ejecución del futuro TSR en conjunto con los programas que el usuario ocupa comunmente. Si el programa corre sin problema, podemos afirmar que es un buen candidato para hacerlo residente.
  Algunos ejemplos de programas residentes utiles:
  - Un diccionario especializado, en un ambiente de procesador de palabras.
  - Un verificador de sintaxis, al editar un programa bajo cualquier lenguaje.
  - Una calculadora o un *Notepad*. Estas aplicaciones son importantes en cualquier tipo de ambiente.

- Un reloj o alarma, tambien en cualquier ambiente.
- Una utileria que ponga a disposicion del usuario la facilidad de ejecutar cualesquiera comandos del DOS, de manera simple, rapida y estando en cualquier tipo de aplicación.

- **Tiempo** de Ejecucion de la Utileria.
  Al hablar de un TSR activo, inmediatamente imaginamos un programa de rapida ejecucion. La propia naturaleza *pop-up* de los mismos, llevan a esta deducción. Un programa *pop-up* sera llamado para realizar una tarea secundaria, mientras el usuario se encuentre trabajando sobre una tarea principal. Y no sera lógico que la tarea secundaria sea tal que requiera mas tiempo de procesamiento que el esperado por el usuario.
  Recordemos que hace relativamente pocos años, los usuarios estaban satisfechos con una Base de Datos "*Query*" que tardaba en arrojar los resultados toda una noche. Ahora, los usuarios se molestan si por ejemplo estando trabajando con una hoja de cálculo, para poder utilizar un programa de *modem* desde el DOS, deben salvar su trabajo y salir de la aplicación presente.

Hemos dado a conocer en este capitulo, de manera general, lo que son los programas residentes. En el siguiente capitulo analizaremos en profundo todos los aspectos, problemas y alternativas de solucion, concernientes a la creacion de lo que hemos definido como utileria residente "de buen comportamiento".

PROBLEMAS INVOLUCRADOS EN LA CONSTRUCCION DE APLICACIONES RESIDENTES PODEROSAS

CAPITULO 3

INTRODUCCION

La presente discusion se llevará a cabo evocando aquellos aspectos que hacen del DOS un sistema operativo limitado para manejar un ambiente de tareas multiples; ya que son precisamente las limitantes del DOS, las que darán la pauta para introducirnos en la problemática inmersa en el diseño de programas TSR poderosos.

Llamamos TSRs poderosos a todo programa residente capaz de hacer uso de todos los recursos que el sistema ofrece y lo suficientemente robusto para poder convivir en armonía con otros, a pesar de los inconvenientes de diseño del DOS.

Sabemos que el DOS es en esencia un archivo jerárquico y servidor de dispositivos de unidad-de-registro, que soporta un sólo usuario y una sóla tarea. Hecho del que dimanan muchas de las dificultades a las que nos enfrentaremos en este capitulo.

Sin embargo queremos aclarar que, en principio, existe un problema medular conocido con el nombre de "Reentrancia"; y del que se desprenden muchos otros que serán tratados a continuación.

Pero antes de entrar de lleno a este tema, hemos considerado conveniente hablar, en primera instancia, del problema de la "Restauración de la pantalla" al término del TSR. Son muchos los programadores que al desarrollar utilerías *pop-up* pasan por alto este aspecto provocando desconcierto en el usuario, que no es advertido de las posibles consecuencias que se pueden suscitar al hacer interactuar su utilería con determinados paquetes no compatibles, en cuanto a video con ella.

En si podemos considerar como "problema" a cada una de las etapas implicadas en el desarrollo de una utilería residente (en el sentido de que el programador desconozca por completo el procedimiento); no obstante, por ello debemos aclarar que este capítulo hace referencia a aquellos problemas que son considerados "típicos" y privativos de los programas residentes en memoria, no así a otros que podemos considerar más generales.

EL PROBLEMA DE DESPLIEGUE DE LOS PROGRAMAS "POP-UP"

El manejo del despliegue constituye uno de los primeros problemas que se deben encarar en la realización de programas *pop-up*.

Cuando un *pop-up* es invocado, este debe salvar en su propia memoria el contenido de la zona de despliegue, o la parte de ésta, que ocupara al desplegarse en pantalla: para que al terminar pueda *restaurar* la pantalla a su estado original volviendo a escribir en la memoria de video la parte que salvó.

Esto en realidad es tarea simple para un programador de sistemas, siempre que el modo de video manejado sea el mismo invariablemente. Sin embargo, debemos tener en cuenta que algunos programas como "Microsoft Word" trabajan en modo "gráficos"; de manera que si tuvieramos que activar nuestra utileria estando en este ambiente, tendriamos que restaurar parte de la pantalla en modo texto y parte en modo gráficos. Pero veamos todos los inconvenientes que esto implica.

Primero, el area de despliegue en modo "gráficos" tiene de 4 a 7 veces más datos por caracter (dependiendo del adaptador de video), lo cual implica que el programa residente tenga que reservar más espacio para mantener toda la información de la zona en cuestión; con lo que estaremos restringiendo su memoria. Además de que tambien deberá leer y restablecer esos datos en términos de puntos y no de caracteres.

Suponiendo que lo anterior fuera llevado a cabo, lo que seguiria seria desplegar el TSR en pantalla. Sin embargo, antes de proceder a hacerlo, debemos tener presente que el *pop-up* no puede desplegarse directamente en la pantalla de "gráficos" sin antes convertir todos sus caracteres a puntos.

Si bien podría nuestro *pop-up* dejarle este trabajo al BIOS, debemos considerar que éste es muy lento y lo que el usuario percibiría no sería un *pop-up* sino un despliegue en "cámara lenta" punto por punto, que podría tardar demasiado. Por lo que este camino queda descartado.

Se nos puede ocurrir que otra salida estaría en que el programa residente pudiera conmutar el modo de video; es decir, que cambiara el despliegue a modo de "caracteres" antes de desplegarse. Y una vez hecho esto, realizar su trabajo para que finalmente cuando haya terminado, restaure la pantalla volviendo a conmutar al modo de "gráficos".

Para poder actuar de esta manera existen 2 alternativas:

La primera y más segura, consiste en cambiar el modo de video mediante el BIOS, pero hay que tener presente que el BIOS de la PC borra el contenido de la memoria de video al cambiar los modos de video. Ello significa que el programa residente tendría que salvar todo el contenido de la memoria de video (16 *Kbytes*) en el caso del adaptador de Gráficos/Color "CGA"). Lo cual se viene a traducir en un desperdicio de memoria.

La segunda opción estriba en que el programa residente cambie el modo de video directamente accesando al controlador de video 6845 (en caso de contar con un Adaptador de Gráficos/Color "CGA"). De este modo el programa no tendría que salvar más memoria de video que la utilizada para su propio despliegue. Esto es lo que *SideKick* hace cuando aparece en una zona de despliegue de gráficos, y lo que da explicación al porqué en el resto de la pantalla (excepto la parte para la ventana de *Sidekick*) subitamente aparecen caracteres aleatorios. Pese a ello, el método trabaja, pero tiene una limitante: sólo trabajará con un adaptador de despliegue CGA. El Adaptador de Gráficos Mejorado (EGA) no maneja el controlador de video 6845. Por lo que si un programa residente emplea el método de la conmutación del tipo de video para restaurar las pantallas, y es probado sobre un adaptador EGA, el programa *pop-up* no podrá accesar nunca al 6845, provocando la pérdida del control de video, y originando la aparición de "basura" en pantalla.

Con la tarjeta de gráficos Hercules también se tienen problemas. No hay manera de que el programa *pop-up* pueda determinar el modo de video empleado por la tarjeta, si gráficos o texto. Además cabe señalar que aún en modo texto, el *pop-up* no está libre de problemas.

Veamos el caso de que otro programa tomara el dominio del *software* del controlador de video 6845 directamente. Supongamos que un programa hiciera uso de paginación de video programando diferentes direcciones de inicio en los registros del controlador de video; pasaría que, como los registros del Controlador de Video son de "sólo escritura", nuestro programa *pop-up* no tendría manera de determinar que parte de la memoria de video está siendo desplegada actualmente, lo cual provocaría la posibilidad de que el *pop-up* se desplegara en un área de memoria de video no visible. Desde la perspectiva del usuario sólo parecerá que la *hotkey* no tuvo efecto, pero no será así ya que como en realidad el *pop-up* está corriendo, el programa inferior (el programa interrumpido) quedará suspendido y el usuario no se explicará porqué dejó de trabajar. El programa residente nunca podrá evitar estos problemas ya que aún al colocar el controlador de video a un estado "normal", éste nunca podrá retornar la pantalla (o zona de despliegue) a su estado anterior, ya que la información es simplemente inaccesible.

El fin de esta discusión es mostrar al programador todas las alternativas hasta ahora encontradas, sus "pros" y "contras". Nuestra intención es poner en evidencia todos los inconvenientes relativos al despliegue y desaparición en pantalla de la utilería.

Como el lector se habrá percatado, en el camino para solucionar el problema planteado aquí, se encontrarán más obstáculos que alternativas viables. Esto, ha sido motivo de polémica entre los diseñadores de *software*, sin embargo, la experiencia ha demostrado que vale la pena pagar este pequeño precio y optar por alguna alternativa.

La experiencia ha demostrado que los problemas en realidad se puede aminorar considerablemente y en la mayoría de los casos hasta desaparecer, si el programador es cauto y antes de diseñar su utilería residente analiza el tipo de usuarios al que lo va a dirigir y el ambiente en el que trabajará.

Con todo, consideramos que la mejor opción está en hacer uso (hasta donde sea posible) de las rutinas del BIOS; debido a que, a diferencia de la alternativa de accesar directamente al controlador de video, ayuda a mantener la compatibilidad futura con los nuevos tipos de adaptadores de video.

EL PROBLEMA DE ACTUALIZACION DEL CONTEXTO

Cuando un TSR es ejecutado por primera vez, goza de todos los recursos del sistema. Pero una vez que se ha declarado residente y terminado, los recursos serán otorgados al programa transitorio corriente del sistema, o al procesador de comandos del DOS, en caso de no existir dicho programa. Así el TSR quedará en espera de un pulso de tecla para proceder a su activación Cuando la *hotkey* sea sensada, el *pop-up* aparecerá en pantalla e iniciará su ejecucion; sin embargo, todo lo que haga se asumirá como parte de la ejecución del proceso interrumpido, debido a que a éste todavía pertenecen todos los recursos del sistema. En pocas palabras, el TSR se convierte en un "parásito", y las implicaciones que esto trae consigo pueden llegar a ser muy graves (entenderemos mejor el impacto de estas consecuencias después de haber dado lectura al apartado siguiente que habla del problema de no reentrancia del DOS).

Pero volviendo al tema, hemos llegado al punto que queríamos tocar: el problema que estamos analizando deriva del hecho de que el contexto que el sistema reconoce es el del programa interrumpido debiendo ser, después de haber sensado la *hotkey*, el del TSR. Esto, debido a que los apuntadores a los recursos del sistema siguen "apuntando" a los registros del programa interrumpido. Para que el ambiente TSR funcione en el sistema, será necesario intercambiar el contenido de los apuntadores antes mencionados a las direcciones correspondientes del programa residente.

A esto se le conoce como conmutacion de contexto "*context switching*" en un ambiente *multitasking*, fase indispensable en la realización de utilerías residentes en memoria. La conmutación deberá ser llevada a cabo por el TSR y se dividirá en dos etapas: la primera consistirá en salvar el estado de los registros del programa interrumpido y colocar los propios como corrientes para que el sistema reconozca al TSR como tarea presente. Y la segunda estriba en devolver los registros a su estado original al término del TSR.

La conmutación de contexto puede verse como un complemento de varios tipos de conmutación que a menudo suelen ser tratados por separado éstos son: la conmutacion de pila, la conmutación de PSPs y la conmutación de DTA.

■ CONMUTACION DE STACK's.

En el momento de la ejecución de un programa el sistema construye un *stack* para dicho programa. Todo programa requiere y de hecho tiene un *stack* para su uso particular, incluyendo los programas residentes. Cuando el TSR interrumpe a otro programa, el segmento-de-*stack* y el apuntador-de-*stack* apuntan a la pila del programa interrumpido; por lo tanto todas las operaciones que el TSR lleve a cabo serán efectuadas sobre la pila del programa interrumpido y no sobre la propia.

Esto en si no constituye un error, de hecho muchos programas en lenguaje ensamblador justamente trabajan así. Sin embargo se tiene una limitante, los programas que empleen este método deben rigurosamente restringir el uso de espacio en pila. Esto obedece a que, en primer lugar, no tenemos la manera de saber "cuánto" *stack* podrá proporcionar el programa interrumpido y en segundo lugar, para dicho programa interrumpido el DOS sólo puede garantizar un espacio de pila suficiente sólo para los registros de máquina.

Nosotros emplearemos el lenguaje C para la construcción de un manejador de TSRs; este lenguaje utiliza el *stack* de manera intensiva, por lo que no nos conviene trabajar sobre el *stack* del programa interrumpido. Para nuestro caso resulta indispensable optar por el método de conmutación de pila; sin embargo, independientemente de la situación en la que el programador se encuentre, este método sera siempre el más recomendable. En palabras usadas por los programadores, el metodo consiste en que "el TSR conmute hacia su propio *stack* en el momento de interrumpir otro proceso".

Que el TSR conmute hacia su propio *stack* significa que deberá salvar el registro de segmento-de-*stack* y todos los registros de apuntador, antes de intercambiar su valor por la dirección de sí mismo (esta tarea es llevada a cabo durante la carga del TSR en memoria); y regresar a estos registros los valores salvados, antes de que el TSR regrese al programa interrumpido (acción tomada durante las ejecuciones subsecuentes).

El lector se podrá preguntar qué sucedera si el TSR fuera reentrante; es decir que se pudiera interrumpir así mismo. Evidentemente, en este caso la conmutación puede fallar. Si el *stack* es conmutado una segunda vez, se sobreescribirá sobre el área salvada para el registro de *stack*. En el diseño y escritura de nuestro TSR deberemos contemplar este caso, nuestros programas serán no reentrantes. En la realidad, esto no llega ni siquiera a constituir una restricción, el concepto de programas reentrantes más bien es manejado en el ambiente de programas transitorios; el que una utilería residente fuera reentrante implicaría por ejemplo interrumpir una calculadora para correr otra aplicación, lo cual ni es frecuente, ni lógico en la mayoría de los casos.

Para evitar problemas debemos prever que nuestros TSRs no permitan llamadas a sí mismos. Y aunque se pueda pensar que al usuario nunca se le ocurriría actuar de esta manera, inconscientemente sí lo puede hacer, por ejemplo al volver a presionar por error la *hotkey* en el momento de ejecución de la utilería. Para prevenir a un TSR de ser reentrante, recomendamos "prender" una bandera cuando este sea desplegado y "apagarla" hasta que sea finalizado. Así, si la *hotkey* vuelve a ser sensada se encontrará que la bandera está prendida indicando ignorar la nueva activación requerida.

■ CONMUTACIÓN DE PSP's.

En el capítulo I, sección "El Prefijo de Segmento de Programa; explicamos de manera general, el porqué la necesidad de realizar la conmutación de PSPs. Ahora veamos un poco más a detalle estos conceptos, y conozcamos el método que emplearemos para realizar dicha conmutación.

Partamos del siguiente hecho: Cada programa cuenta con un PSP, pero para el DOS el único PSP real es aquél que corresponde al ultimo programa ejecutado; es decir al proceso corriente. Un caso que ejemplifica lo anterior es el que se presenta cuando un determinado programa genera la ejecución de otros (éstos últimos llamados programas hijos - para nuestro caso pensemos en TSRs).

Al ejecutarse cada uno de los procesos hijos, el DOS les proporciona un PSP de manera individual; pero como el sistema sólo puede reconocer una tarea activa a la vez, el unico PSP presente será el del proceso padre. De esta manera, todo lo que realice el hijo será adjudicado al padre; así, si el hijo lleva a cabo alguna operación de archivo (es decir, que implique la utilización de la Tabla de Manejadores de Archivo asociada a todo programa en el PSP, -pues éste es el caso que aquí nos interesa analizar-), el DOS utilizara para tal fin el PSP del padre y manejará todo desde la perspectiva del único programa que reconoce como corriente en el sistema. Veamos a continuación las implicaciones de todo esto.

De entre todos los campos del PSP, el de mayor interés para la presente discusión lo constituye el arreglo de 20 manejadores de archivo. Este número es debido a que a cada programa le es permitido abrir hasta 20 archivos a la vez. A cada archivo le es asignado uno de los manejadores suscritos en este arreglo, o en otras palabras, al crear un archivo le es asociada una entrada de la Tabla de Manejadores de Archivo. Las primeras 5 entradas están asignadas a los dispositivos lógicos **stdin**, **stdout**, **stderr**, **stdaux** y **stdprn** cuando el programa es ejecutado por primera vez. En tanto un programa abra archivos, los nuevos manejadores serán almacenados en la Tabla. Las entradas del arreglo que permanecen sin uso contienen un valor de "-1". El programa direcciona los archivos como si estuvieran suscritos (con índices) dentro del arreglo.

Supongamos que durante la carga en memoria de un TSR, éste abre un archivo antes de declararse así mismo residente y terminado. El sistema reaccionará almacenando el manejador para ese archivo en el PSP del TSR.

Posteriormente, considerese que se ejecuta un programa, dentro del cual el usuario activa el TSR. El DOS, no teniendo la capacidad de distinguir el nuevo programa, asumirá que el programa interrumpido está todavía corriendo.

Ahora, supongase que el TSR hace una llamada al DOS en referencia al archivo que abrio cuando fue ejecutado por primera vez; el sistema, atendiendo la llamada, irá a buscar el manejador para dicho archivo en el PSP del programa interrumpido y no en el del TSR. Esta situación puede crear graves problemas. Primero, si en el arreglo de manejadores de archivo, correspondiente al PSP del programa interrumpido, existiera una entrada con relación al manejador que estamos buscando, estaremos referenciando un archivo equivocado. Supongamos que se desea realizar una operación de escritura sobre el archivo, al encontrar que sí existe manejador para el mismo, la operación se realizará sobre el archivo del programa interrumpido correspondiente a esa entrada y no sobre el del TSR. Segundo, si se encuentra que no existe la entrada respectiva al manejador buscado, el DOS asumirá que se está haciendo referencia a un archivo no abierto.

A continuación expondremos algunas de las soluciones posibles para resolver el problema y posteriormente dejaremos ver los "pros" y "contras" que de cada una dimanan:

- Primera Solución: No abrir archivos al momento de la inicialización del TSR. Si se requiere de la apertura de archivos, hacerlo en el momento que el TSR interrumpa a otro programa.

- Segunda Solución: No hacer uso de los manejadores de archivo de la versión 2.0 del DOS y posteriores. En su lugar, emplear el Bloque de Control de Archivos (FCB) de las primeras versiones. De esta manera se evitaría el uso del arreglo de manejadores del PSP ya que las funciones del FCB no requieren de éste. Las tablas del FCB son mantenidas en el espacio de datos de los programas que se sirven de ellas.

- Tercera Solución: Conmutar los apuntadores al PSP del DOS cuando el TSR interrumpe y volverlos a conmutar (restableciendolos a su valor anterior) cuando termine.

Respecto a la primera alternativa: El método en sí implica que el TSR utilizará, para su manejo interno de archivos, las entradas del arreglo del PSP del programa interrumpido. Sin embargo, esto acarrea 2 desventajas: Primera, el hecho de abrir y cerrar archivos al momento del despliegue podría incrementar el tiempo de ejecución de la utilería. Segunda, nadie puede asegurar que el PSP del programa interrumpido cuente con el espacio suficiente para poder controlar todos los manejadores del TSR. Luego, aunque el espacio sea suficiente el TSR podría saturar el número de entradas disponibles e impedir que el programa interrumpido pueda tener la opción de abrir más archivos.

Con relación a la segunda alternativa: El usar funciones del FCB trae consigo 2 desventajas: Primera, los archivos creados por dichas funciones deben estar en el subdirectorio corrientemente activo del sistema al momento de abrirlo. Es decir, para la apertura de un archivo no es permisible trazar trayectorias de subdirectorios del DOS para especificar el archivo que se desea abrir. Segunda, todas las operaciones de archivo de la librería estándar de Turbo C (que será el que nosotros emplearemos), asumen el uso de manejadores de archivo, por resultar más flexible. Por lo tanto, para usar las funciones del FCB, se deben desarrollar funciones equivalentes para reemplazar las funciones estándar siguientes: **open, close, read, write, fopen, fclose, fget, fput, fprint**, etc.

En cuanto a la tercera alternativa, la desventaja radica en que no existen funciones documentadas del DOS para modificar la dirección que el DOS reserva para almacenar el PSP del proceso corriente. La única función documentada que se conoce, hace referencia a esta materia es la función 62H de la Interrupción 21H. La cual permite leer el PSP actual en el sistema. Esta función no es de utilidad para nuestro caso, ya que lo que

requerimos no consiste en leer únicamente sino en modificar el PSP. Además dicha función se encuentra únicamente disponible con versiones posteriores a la v3.0 del DOS inclusive, hecho que se traduce en una limitante más.

Solución Definitiva: Tal panorama sugiere descartar todas las alternativas; sin embargo, gracias a las investigaciones hechas por algunos escritores de *software* conocidos, las posibles desventajas de la ultima alternativa desaparecen.

Producto de las investigaciones antes mencionadas, son algunas publicaciones que descubren el desempeño de algunas funciones no documentadas para cambiar la dirección de PSP del DOS y cómo leerla. Dos funciones de la interrupción 21H no documentadas (la 50H y 51H) colocan y obtienen la dirección del PSP respectivamente. La dirección del PSP es conocida como "Proceso ID" (PID), de aquí que las 2 funciones anteriores sean identificadas bajo los nombres de SetPID y GetPID. Sin embargo dichas funciones sólo pueden trabajar confiablemente en versiones del DOS posteriores a la v3.0 inclusive. No son confiables para las versiones anteriores, debido al hecho de que comparten una pila con otras funciones del DOS, pudiendo estas últimas ser interrumpidas por un TSR.

En otras palabras, si el TSR hiciera uso de las funciones 50H y 51H y al momento de su activación interrumpiera alguna de las funciones antes mencionadas, el sistema podría fallar.

Entonces la solución consiste en buscar un método que, haciendo uso de las funciones 50H y 51H, añada otras funciones para lograr un desempeño correcto en todas las versiones del DOS.

Para llevar a cabo lo anterior se pueden seguir varios caminos. Uno de éstos consiste en determinar a través de la experimentación, la direccion donde el DOS almacena el PID para cada version, para luego consignar los resultados en una tabla. Por su puesto esta alternativa resulta de confiabilidad vacilante. Otro camino, el mejor hasta ahora encontrado, consiste en determinar la dirección de PID al tiempo de corrida (cuando el TSR es cargado). El procedimiento que este método involucra será detallado en el capitulo "Procedimiento General para la Construcción de Utilerías Residentes en Memoria" CUADRO G.2.

■ CONMUTACION DE DTA.

El DTA "Area de Transferencia de Disco", es un *buffer* de 128 *bytes* almacenado en el PSP y representa el área en la que el DOS lee y escribe cualesquiera archivos de disco abiertos mediante las funciones del FCB. El DOS tambien utiliza esta área para funciones que tratan con búsquedas en directorios.

Al establecerse el PSP para cada programa, intrinsecamente también le es asignado un espacio para realizar operaciones en disco, este espacio es apuntado por el DTA. Todo TSR que haga uso de funciones del DOS que escriban hacia este espacio en memoria, debe salvar la dirección de DTA del programa interrumpido para establecer la propia. Luego, al regresar al programa interrumpido, el TSR debe restablecer la dirección del DTA a la del programa interrumpido.

Es obvio que debemos actuar de esta manera para evitar que el TSR trabaje sobre el area de DTA del programa interrumpido. Si el TSR no realiza manejo de archivos, no habrá problema, pero en caso contrario podrá modificar el DTA del programa interrumpido. Si se trabaja bajo el lenguaje C, que es nuestro caso, este aspecto cobra importancia, ya que no se puede tener la seguridad de que el TSR llegue a utilizar el área de DTA; recordemos que las acciones de las funciones de la librería de Turbo C, permanecen transparentes al programador, y puede darse el caso que implícitamente puedan hacer uso del área de DTA.

La librería de Turbo C incluye funciones que leen y colocan la direccion del DTA. Estas son: **Getdta**, que devuelve la dirección corriente del DTA; y **Setdta**, que cambia la dirección del DTA. Las funciones deben seguir el formato:

```
# include < dos.h >
char far *dta;
dta = getdta();
setdta(dta);
```

EL PROBLEMA DE LA REENTRANCIA

Una de las principales limitaciones del DOS que se erige como obstáculo para simular un ambiente *multitasking* en la PC, radica en que sus servicios no son reentrantes. Si, debido al diseño de un sólo usuario y una sóla tarea, los servicios del DOS no pueden ser interrumpidos por un segundo programa para luego hacer uso de ellos.

El que el MS-DOS (así como el PC-DOS) no sean reentrantes significa que no se pueden hacer llamadas a funciones de la interrupción 21H del DOS, mientras otra esté en progreso. El porqué de esto radica en que el DOS conmuta a una pila interna. En esta sección conoceremos la manera en la que esta situación afecta al establecimiento de programas residentes. Pero cabe aclarar que aún cuando sea superado el problema de la reentrancia, debemos tener en cuenta otros aspectos para validar el perfecto funcionamiento del programa TSR; uno de ellos: el reconocimiento (por parte del sistema) del PSP corriente. Durante algunas llamadas al DOS, este requiere almacenar información acerca de archivos abiertos recientemente en el PSP del programa que esta realizando la llamada al DOS. El DOS usa la dirección del PSP corriente (que mantiene almacenada internamente) para determinar cuál, de entre todos los programas cargados, es el que posiblemente está haciendo la llamada a sus servicios. Por ello, si un programa residente pretendiera abrir un archivo sin comunicarle previamente al DOS que él es el autor de la llamada (pasándole su dirección de PSP), la información de la apertura del archivo sería almacenada en el PSP del programa de aplicación interrumpido; así, cuando dicho programa terminara, el DOS cerraría el archivo que el programa residente hubo abierto. Existen más consecuencias que analizar a este respecto, para mayor referencia véase el tema del PSP tratado en el capítulo I. En esta parte solo nos dedicaremos al estudio de causas, consecuencias y posibles soluciones al problema de la reentrancia y a los que de este derivan.

En terminos simples, podemos afirmar que bajo el DOS, las llamadas a interrupciones dentro de interrupciones no son permitidas, o bien, que un programa residente no puede hacer llamadas al DOS mientras un programa de aplicación esté tambien haciendo alguna llamada a este. Esto es lo que comunmente se conoce como "Problema de la Reentrancia".

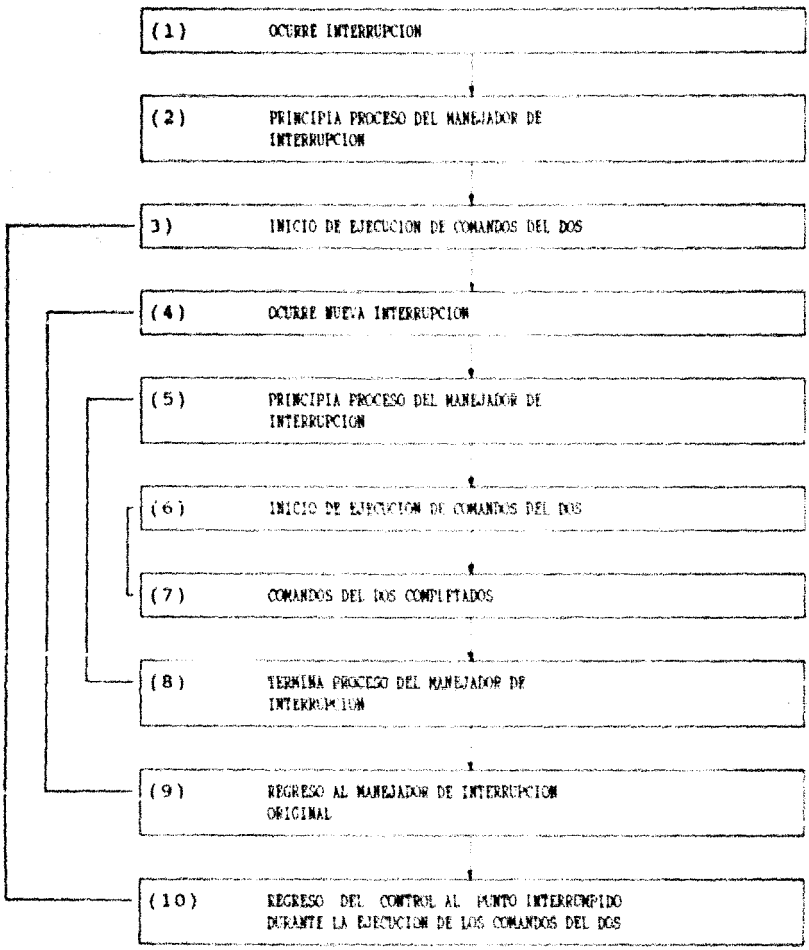
Para dar una explicacion más clara a este problema, consideremos el siguiente ejemplo: Supongase que hemos desarrollado un manejador de interrupción (ISR) que se ha instalado como una utileria TSR. Y en el momento en que la interrupción esta siendo procesada por nuestro "software" (y antes de que su proceso sea completado) es sensada una nueva interrupción de la misma naturaleza. Los eventos resultantes se ilustran en la figura 3.1.

Al ocurrir la primera interrupcion, paso no.(1), da inicio la ejecucion de la ISR asociada a ésta (2), dentro de la cual se llevan a cabo llamadas a funciones del DOS. En este momento el DOS almacena en su pila, todos los registros involucrados en la realización de la funcion, así como la direccion de regreso al programa interrumpido justo en el punto en el que fue suspendido.

Posteriormente, los comandos del DOS empiezan a ejecutarse (3) hasta que es sensada la ocurrencia de la nueva interrupción (4). Esto ocasiona el reinicio del proceso de la ISR (5), y así una vez más la pila del DOS es vuelta a llenar, pero ahora con la dirección de regreso al punto en el que ocurrió la segunda interrupcion, y que corresponde a una parte intermedia del código de ejecución de los comandos del DOS.

Aunque ya se puede inferir, cabe resaltar que la pila empleada por el DOS durante la primera interrupción es la misma que la empleada para la segunda interrupcion, ya que ambas interrupciones son del mismo tipo. Recordemos que para realizar sus funciones, el DOS hace uso de 2 pilas. Una de ellas empleada exclusivamente para funciones bajas (0H-12H) y la otra para funciones altas (12H-en adelante).

FIG. 3.1 ILUSTRACION DEL PROBLEMA DE REENTRANCIA DEL DOS.



A continuación, se ejecutan los comandos del DOS (6), y una vez completados termina también el proceso de la ISR ejecutándose la instrucción que conocemos como regreso de Interrupción (IRET) (9), para dar por terminado el manejo de la segunda interrupción.

Finalmente, el control es devuelto a la dirección de regreso presente en la pila del DOS (10), dirección correspondiente a algún paso intermedio en la ejecución de los comandos para procesar la primera interrupción y misma que seguirá permaneciendo en la pila del DOS como dirección de regreso corriente. Por lo tanto, jamás se podrá retornar el control al proceso original en el que se suscitó la primera interrupción. Y como es de esperar, esta situación se torna en un grave problema.

Después de haber expuesto el problema, el programador caerá en la cuenta que la solución al mismo consiste en cuidar que mientras se esté ejecutando una función del DOS, no se permita ningún tipo de interrupción. Esto, para nuestro caso se viene a traducir en evitar la activación del TSR siempre que una función insegura esté en progreso. Recordemos que nuestro TSR será concebido por el sistema operativo como una Rutina de Servicio de Interrupción, debido a su incapacidad de inferir la existencia de más de un proceso corriendo concurrentemente. Al igual que una ISR, el programa residente se activara al darse una interrupción (que para nuestro caso será la Int. 09 de teclado, al sensar la *hotkey*), posteriormente llevará a cabo su trabajo valiéndose de las funciones del DOS y finalmente regresará al punto de suspensión del programa interrumpido.

En resumen una primera solución, que con los elementos planteados el lector ya puede inferir, radica en proveer al TSR de la capacidad de detectar el estado de seguridad del DOS y de sensar la *hotkey* para que, en el caso de tener al DOS fuera de peligro, proceder a la activación del TSR y en caso contrario retardar su activación hasta que el sistema esté fuera de funciones inseguras.

Precisamente de este planteamiento dimanan tres de los primeros problemas que el programador de TSRs debe enfrentar: ¿Cómo detectar la *hot key*? ¿Cómo activar el TSR? y ¿Cómo determinar cuándo el TSR puede ser activado y cuándo no?. De estos tres primeros problemas se desprenden otros más que se irán mencionando oportunamente a lo largo de este capítulo.

COMO DETECTAR LA *HOTKEY* ?

Todo programa residente que sea activado mediante una combinación de teclas y/o que requiera procesar de manera especial alguna entrada de teclado, deberá ligarse y encadenarse a la interrupción de teclado previa. Al ligarse a dicha interrupción, el programa permanece vigilante al suceso de la presión de la *hotkey* para activar la utilería; y al encadenarse, permite la continuidad de los servicios del teclado del ROM-BIOS para cualesquiera otros programas cargados con posterioridad.

Pero veamos mas a fondo la interrupción de teclado para tener una idea clara de cómo la *hotkey* es detectada.

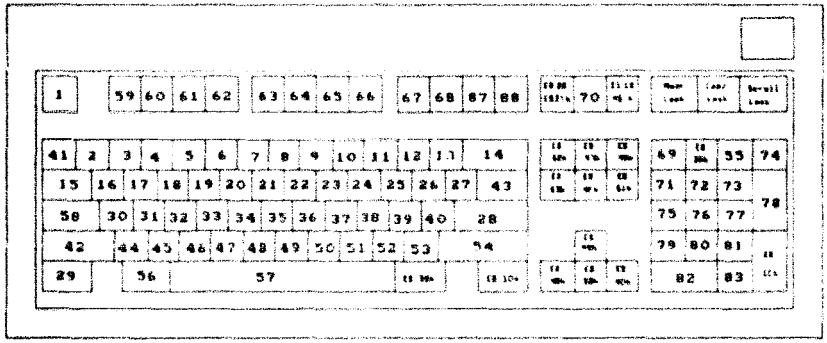
Nota: la interrupción de teclado que manejaremos será la interrupción de *hardware* 09H por las razones que al final de este apartado se exponen.

■ EL TECLADO.

Al ser presionada una tecla, se *genera una interrupción* de *software* y el microprocesador del teclado genera un código de exploración (numero de 8 bits) correspondiente a la tecla presionada que, inmediatamente envia por los cables del teclado a la tarjeta de la PC y es almacenado en el puerto de entrada de datos de teclado (Puerto "PA" del 8255).

El código de exploración, representado en un *byte*, será un numero entre 1 y el numero total de teclas (*generalmente* 83); este código identifica de manera única al caracter pulsado. El bit de mayor orden (bit 7) del codigo de exploración indica si la tecla fue presionada o liberada. Será "0" en el caso de que la tecla haya sido oprimida y "1" en caso de haber sido liberada. La fig. 3.2 muestra los códigos de exploración respectivos para el teclado tipo PC-AT.

FIG. 3.2 CODIGOS DE EXPLORACION DEL TECLADO



A continuación, se inicia el procesamiento de la ISR de teclado apuntada por el vector de interrupcion 09H. La ISR lee el contenido del Puerto de Entrada de Datos del teclado y a partir del valor obtenido determina las teclas que fueron presionadas y emprende su procesamiento. Cabe observar que el *software* del teclado de la PC no genera el valor ASCII que todo lenguaje de programacion de alto nivel obtiene al realizar una lectura de consola. Es la ISR de teclado del ROM-BIOS la encargada de traducir los códigos de exploracion a sus valores ASCII correspondientes y de insertarlos en el *buffer* de teclado. Los programas que leen entradas de teclado, incluido el DOS, obtienen los datos de este "*buffer*".

El ROM-BIOS además, mantiene un *byte*-de-estado, también conocido como máscara de estado, encontrado en la localidad de memoria 0:417. Este *byte* está compuesto por 8 *bits* que representan cada una de las teclas especiales (también conocidas como teclas *toggle/shift* - que cambian el estado del teclado-). Al presionar cualquiera de ellas se prende el bit correspondiente de la máscara, y al ser presionada de nuevo se reinicializa (ver fig.3.3).

FIG. 3.3 MASCARA DE ESTADO.

TECLA	VALOR
INS	128
CAPS LOCK	64
NUM LOCK	32
SCROLL LOCK	16
ALT	8
CTRL	4
LEFT SHIFT	2
RIGHT SHIFT	1

Como la *hotkey* esta compuesta por la combinación de teclas (una tecla especial mas cualquier otra que tenga un código de exploración válido) para llevar a efecto la detección de la *hotkey*, nuestro programa residente deberá contar con una ISR de teclado propia, que pueda sensar la *hotkey* comparando el código de exploracion de la segunda tecla de la combinación con el esperado y leyendo el *byte* de estado para determinar si su valor corresponde al de la tecla especial buscada.

■ LA ISR DE TECLADO.

Las acciones en orden cronológico que debe incluir nuestra ISR de teclado son, en general, las siguientes:

Dado el evento de haber presionado una o un grupo de teclas, la ISR deberá determinar si dicho pulso corresponde a la *hotkey*, para ello lee tanto la máscara de estado como el código de exploración. Inmediatamente procede a enviar una señal de reconocimiento al teclado y otra de fin de interrupción al 8259 (el porqué de estas acciones será explicado en el apartado siguiente, con el fin de no desviar la secuencia de ideas aquí presentadas). Posteriormente verifica si el TSR no esta activado en ese momento; así, si ambas condiciones son afirmativas el TSR podrá ser activado. No obstante, nuestra rutina sólo se limitará a prender una bandera (llamada *hotkey*) indicando que el usuario desea activar el TSR en ese momento, pero no procedera a hacerlo ya que, como se justifica a continuación, la ISR del *timer* es la más adecuada para realizar esta tarea.

La primera razón obedece a que el evento de presionar la *hotkey* es independiente de lo que esté haciendo el DOS. El *software* del teclado trabaja de manera independiente, pudiendo sensar la presión de teclas asíncronamente, por lo que no podemos tener la ligereza de considerar improbable que este evento pueda ocurrir al estarse ejecutando algún comando del DOS.

Durante este estudio hemos podido constatar que son muchos los programadores que optan por proceder a la activación de su utilería desde la ISR de teclado, inmediatamente después de haber sensado la *hotkey*. Por supuesto que para lograrlo sin problema, han incluido dentro de su ISR la verificación del estado de seguridad del DOS. Pero nosotros no optamos por esta alternativa debido a que sólo si se encuentra que el DOS está a salvo, no existirá inconveniente para proceder a la activación del programa; pero si el DOS está ocupado, la ISR ignorará la *hotkey*. Esto puede descontrolar al usuario ya que lo que él espera es ver desplegada en pantalla su utilería residente de inmediato.

Por este motivo optamos por activar el TSR en nuestra Rutina de Servicio de Interrupción del *timer*, de manera que una vez sensada la *hotkey* no pase inadvertida.

Finalmente, cuando nuestra ISR de teclado haya sensado cualquier otro pulso de tecla diferente a la *hotkey*, cederá el control de la misma a la ISR de teclado previa, es decir llevará a cabo lo que hemos estado manejando como "encadenamiento".

■ EL 8255 Y EL 8259.

El objeto de este apartado no es el de profundizar en el funcionamiento y composición de estos "chips" de soporte, ya que ello no constituye parte del tema de esta tesis. Más bien, esta parte está dedicada a esclarecer todas las dudas que el lector programador se pudiera plantear al emprender el diseño de su Rutina de Servicio de Interrupción de teclado y todo lo que a ella concierna.

En el punto anterior, habíamos puesto en evidencia la necesidad de enviar una señal de fin de interrupción al 8259 siempre que nos dispongamos a escribir una ISR de teclado propia. A continuación veremos la justificación de ello y el papel que, al respecto, juegan el 8259 y el 8255.

El Controlador Programable de Interrupciones (8259) -PIC-

Cuando en el sistema ocurren eventos exactamente en el mismo instante, el 8259 "Controlador Programable de Interrupciones" las pasa al procesador 8088 en un orden de prioridades. La prioridad más alta corresponde a la fuente de interrupción "0" y la más baja a la fuente de interrupción "7".

El 8259 presenta la información al 8088 bajo un código especial para cada tipo de interrupción y único a cada fuente. Esto es lo que nos permite asignar una ISR única a las diferentes fuentes de interrupción. Por otro lado, a cada fuente de interrupción le corresponde una de las 8 señales de entrada del 8259, tales señales se encuentran alambradas sobre el "bus de control", de manera que cualquier dispositivo ligado al sistema del *bus* puede accesar a este mecanismo de interrupción. Las 8 señales en el bus son conocidas bajo los símbolos IRQ0 - IRQ7. (Ver fig.3.4).

FIG. 3.4 FUENTES DE INTERRUPCION.

ENTRADAS AL 8259	TIPO DE CODIGO	DISPOSITIVO
IRQ0	08H	TIMER
IRQ1	09H	TECLADO
IRQ2	0AH	INTERFAZ DE GRAFICOS COLOR
IRQ3	0BH	NO USADO
IRQ4	0CH	INTERFAZ SERIE (RS-232)
IRQ5	0DH	NO USADO
IRQ6	0EH	DISKETTE
IRQ7	0FH	IMPRESORA

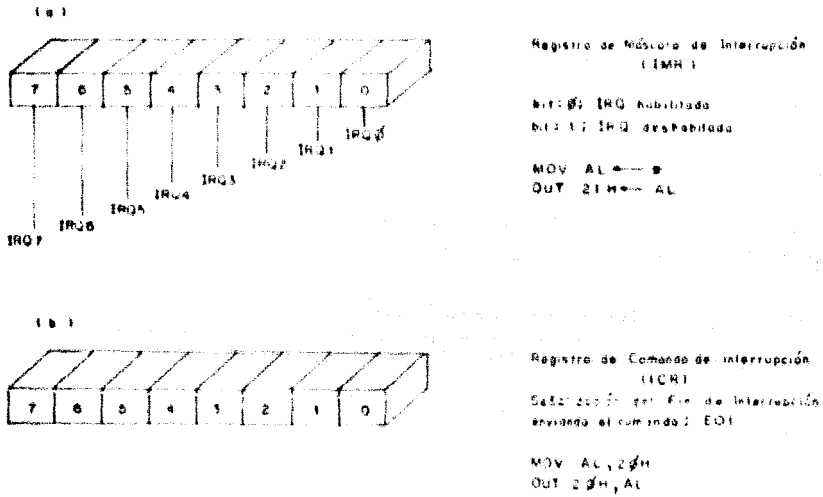
Una vez procesada la interrupción de mayor prioridad y antes de ceder el control a la siguiente, el 8088 envía una señal de fin de interrupción al 8259 para que este pueda dar por terminado el Servicio de Interrupción que se acaba de procesar. Esto debe tenerse presente siempre que se escriba una Rutina de Servicio de Interrupción para tomar el control del teclado.

En sí desde el punto de vista del "desarrollador" de *software*, la programación del 8259 consiste de 2 acciones básicas:

- La primera acción descansa en habilitar o deshabilitar cualquier fuente de interrupción. Para lo cual se debe modificar el estado del Registro de Máscara de Interrupción o IMR. El IMR es un registro de 1 *byte* al que se puede accesar vía el puerto de E/S 21H.
  Cada bit del IMR corresponde a una fuente de interrupción según el número del mismo bit, (Por ejemplo bit0-IRQ0, bit1-IRQ1, etc). Para habilitar una fuente de interrupción determinada deberemos poner a "0" su bit correspondiente en el IMR, así se generará una señal sobre la entrada respectiva al 8259 y finalmente se enviara la interrupción al 8088.
  Para deshabilitar una fuente de interrupción el bit asociado a ésta debe ser puesto en "1", con lo que se impedirá la generación de una interrupción del tipo en cuestión. (Ver fig. 3.5a).

- La segunda, que es la de nuestro interés, radica en llevar a cabo la señalización del fin de una Rutina de Servicio de Interrupción. Para ello, el programador deberá enviar el comando "*End Of Interrupt*" (EOI), reconocido por el sistema en su representación hexadecimal "20H", hacia el Registro de Comandos de Interrupción (ICR) del 8259. (Ver fig. 3.5b).
  Para evitar confusiones conviene aclarar que coincidentemente este registro (ICR de 1 *byte*) es accesado mediante el puerto "20H".

FIG. 3.5 EL "IMR" Y EL "ICR".



Interfaz de Periféricos Programable (8255)

El 8255 es un "chip" de interfaz de E/S de propósito general que puede ser configurado de diversas maneras. Soporta una gran variedad de dispositivos y señales. Entre ellos podemos mencionar el teclado, "*speaker*", *switches* de configuración, etc.

El "chip" cuenta con 3 puertos llamados PA, PB y PC que son mapeados hacia las direcciones de E/S 60H, 61H y 62H respectivamente. Ademas, incluye un Registro de Comandos de un *byte* accesado mediante la dirección de E/S 63H.

Para inicializar este "chip", el BIOS envia el valor "99H" al Registro de Comandos, y al acto éste procede a configurar el 8255. La configuración consiste en establecer los puertos PA y PC como puertos de Entrada y el puerto PB como puerto de Entrada/Salida. Cada puerto consta de 1 *byte*. La función de los puertos PA y PC dependen del estado de los bits 7 y 2 del puerto PB respectivamente.

Adentrar en este tema no compete a este estudio; sólo resaltemos que, cuando el bit 7 del puerto PB está en "0", el puerto PA mantiene el código de exploración de la tecla sensada por el *hardware* del teclado.

Dentro del teclado existe un microprocesador que explora y detecta cualquier cambio en el estado de las teclas; éste recibe todo su "poder" y señales de reloj de la tarjeta del sistema.

Ahora, si por algún motivo queremos mantener al teclado inoperante, debemos entonces deshabilitar la señal de reloj. Para ello debemos proceder a colocar el bit 6 del puerto PB a "0". Si, por otro lado, requerimos enviar una señal de reconocimiento (*ack*) al teclado, debemos colocar el bit 7 del puerto PB a "1". Luego, para asegurar que el teclado quede apropiadamente habilitado, deberemos colocar los bits 7 y 6 del puerto PB a "0" y "1" respectivamente. Sólo de esta manera (encontrándose el puerto PB en este estado), el teclado podrá generar la señal de interrupción IRQ1 siempre que cualquier tecla haya sido presionada o liberada. Al darse la interrupción IRQ1, el microprocesador del teclado transmitirá el código de exploración a la tarjeta del sistema y ésta esperará a que la señal de "*ack*" sea regresada.

Conclusión.

Toda la información contenida en este apartado referente al 8255 y al 8259, ha sido presentada con el fin de ubicar al programador en un contexto general, de proporcionarle todas las herramientas que requiere para entender el porqué de todos los aspectos involucrados en la realización de su propio soporte de teclado.

En nuestro cometido por diseñar una nueva ISR de teclado para nuestro TSR, es muy importante que tengamos en consideración todo lo anteriormente expuesto ya que, precisamente al tomar el completo control del teclado, nuestra labor será la de imitar el comportamiento de la ISR de teclado del sistema.

A manera de conclusión podemos decir que nuestro software de soporte de teclado necesariamente deberá incluir 3 acciones indispensables independientemente de las otras muy particulares que realice. Éstas son:

- Obtener el código de exploración transmitido a la tarjeta del sistema, leyendo el puerto "PA" del 8255.
- Enviar una señal de reconocimiento (*acknowledge*) al teclado colocando momentáneamente el bit 7 del puerto "PB" en "1".
- Enviar señal de fin de interrupción (20H) hacia el Registro de Comandos de Interrupción del 8259 (accesando al puerto 20H).

■ LA INTERRUPCION 16H o la INTERRUPCION 09H ?

Existen 2 interrupciones disponibles en el sistema PC para leer pulsos de tecla de la consola. Éstas son, la INT 09H y la INT 16H. Aunque el último fin de ambas, sea el mismo, su uso no puede ser indistinto debido a su naturaleza. La INT 16H es una interrupción de *software* y la INT 09H es de *hardware*.

Cuando se presiona una tecla, se genera la interrupción 09H, e inmediatamente se ejecuta la rutina del BIOS respectiva. EL manejador de la interrupción, almacena el pulso de tecla en un *buffer*; mismo que funge como interfaz entre la interrupción 09H y la 16H. El manejador de la INT 16H (que es otra rutina de BIOS), al detectar la existencia de algun elemento en el *buffer* antes mencionado, simplemente saca el siguiente pulso de tecla del *buffer* para entregarlo al DOS.

Un TSR activo, al requerir ligarse a una interrupción de teclado para su activación, debe hacer uso de cualesquiera de las 2 interrupciones antes mencionadas; pero, como ya habíamos adelantado en apartados anteriores, es recomendable que esta interrupción sea la de *hardware* 09H. Veamos porqué:

En primer lugar, mediante la interrupción 09H un programa puede manejar códigos de exploración, y haciendo uso de la interrupción 16H sólo puede manejar valores ASCII. Recordemos que para que un TSR pueda detectar la *hotkey*, deberá comparar los códigos de exploración de las teclas pulsadas, con los que corresponden a dicha *hotkey*.

En segundo lugar, sabemos que, en su inmensa mayoría, los programas de aplicación utilizan la interrupción 16H para la obtención de pulsos de tecla. Supóngase que se está trabajando con alguno de estos programas (por ejemplo un procesador de palabra), y que se tiene instalado también un programa residente que a su vez hace uso de la misma interrupción de teclado. Si por algún motivo, la combinación de teclas que forma la *hotkey* coincidiera con alguna otra combinación empleada por el procesador para realizar una función determinada, por ejemplo la función de salvar archivos; en lugar de salvar, se activará el TSR. En pocas palabras, el hecho de emplear la misma interrupción, puede dar origen a problemas ocasionando daños irreversibles.

Para evitar este tipo de situaciones, es recomendable que en el diseño de programas residentes, se opte por la utilización de la interrupción 09H; cuidando además, que la combinación de teclas que formarán la *hotkey* sea original, *aminorando así* la probabilidad de que otros programas residentes la utilicen. De esta manera estaremos otorgando a nuestra utilería, la capacidad de trabajar en cualquier ambiente.

Antes de pasar al siguiente tema, sólo cabe hacer una última aclaración: el hecho de emplear la interrupción de *hardware* 09H en el diseño de TSRs, implica el tener que trabajar a "nivel sistema" y no a "nivel aplicación" como sucedería con la interrupción 16H; por lo que se deben llevar a cabo ciertas acciones adicionales, como la rehabilitación apropiada del controlador de interrupciones 8259, y las demás ya expuestas en el punto anterior.

COMO ACTIVAR EL TSR ?

La respuesta directa a esta incógnita es: llamando a nuestro archivo ejecutable sólo hasta que se haya verificado que al momento de sensar la *hotkey* la bandera "BUSY" se encuentre apagada. Prueba que sólo puede ser realizada con el auxilio de la interrupción de *timer*; debido a que el acto de presionar las teclas que forman la *hotkey* constituye un evento asincrono (no predecible); por lo que se requiere que nuestro TSR se encuentre monitoreando constantemente la bandera "BUSY", cosa que sólo es posible ligando nuestra utileria a la interrupcion de *timer*, para ordenar que en cada tick de reloj se lleve a cabo esta verificación. Por lo tanto, nuestro TSR se deberá ligar y encadenar también a interrupción de *timer*.

Recordemos que el reloj del sistema interrumpe a la PC 18.2 veces por segundo, y los intervalos entre la generación de cada interrupción es lo que conocemos como ciclos de reloj. Pues bien, debemos crear una nueva ISR de *timer* para que además de ejecutar sus funciones, esté verificando en cada ciclo, si el DOS se encuentra en el estado de seguridad que requerimos.

LA ISR DE *TIMER*.

El cuerpo de la ISR de *timer* de nuestro TSR debe ser, en general, el siguiente:

Primeramente, debemos ligar nuestra ISR a la interrupción de *timer* Int 1CH, con lo cual se activará con cada "tick" del reloj del sistema, independientemente de cualquier evento externo.

Posteriormente nuestra ISR deberá encadenar la Int 1CH. En seguida deberá verificar si la bandera *hotkey* está encendida, y si el DOS no está llevando a cabo alguna función; caso en el que deberá enviar una señal al 8259 para indicar -Fin de Interrupción-, asi como apagar la bandera *hotkey* y finalmente activar la utileria llamando a la función que la ejecuta.

Los pasos concretos correspondientes a la activación de la utileria son tratados en el capitulo siguiente.

COMO EVITAR QUE MI TSR INTERRUMPA EL CURSO NORMAL DE UNA FUNCION DEL DOS ?

A) EVITANDO LLAMADAS A FUNCIONES ALTAS DEL DOS.

Si el problema al interrumpir la ejecución de una función del DOS radica en el hecho de que el TSR haga llamadas al DOS, entonces la primera medida que se nos ocurriría consistiría en diseñar TSRs que se valieran de otros recursos para realizar sus funciones, excluyendo la posibilidad de utilizar los servicios del DOS.

Ante esto, encontramos que las funciones bajas del DOS (que manejan la Entrada/Salida de consola y pantalla) pueden ser reemplazadas por las funciones del BIOS. Medida que lejos de traer inconvenientes redituará en beneficios, ya que en muchos casos el BIOS ofrece mejor funcionamiento.

Sin embargo, encontramos que para las funciones altas del DOS (que incluyen manejo de archivos) no existen sustitutos. Por lo que nuestros TSRs al prescindir de ellas, no podrían abarcar muchas aplicaciones interesantes, convirtiéndose en utilerías limitadas que si bien tienen la capacidad de resolver el problema originalmente planteado, no la tienen para ser consideradas como TSRs poderosos.

Hasta este punto, contaríamos con la capacidad de construir utilerías residentes que solamente hicieran uso de la consola y de la pantalla tales como" calculadoras, relojes, alarmas, las que emplean todo tipo de despliegue de video (tablas matemáticas, formularios, recordatorios), etc. Sin embargo la limitante que implica el tener que sacrificar el uso de archivos puede ser grande.

B) HACIENDO USO DE LA BANDERA "BUSY" DEL DOS.

Una mejor opción de resolver el problema es: proveer al TSR de la capacidad de detectar si alguna función del DOS está en progreso al momento de haber sensado la *hotkey*.

En el inciso anterior vimos que las funciones bajas del DOS pueden ser reemplazadas por las del BIOS. Esto será retomado aquí por resultar ventajoso, por lo que las funciones del DOS referidas en el párrafo anterior se reducen a las funciones altas, también conocidas como "inseguras" por los programadores.

Ahora, para averiguar si el DOS está llevando a cabo una función insegura, contamos con una bandera que el DOS mantiene en cierta localidad y que es prendida al acto de dar inicio la ejecución de alguna de ellas y apagada al completarse dicha función. Esta bandera es conocida como "*BUSY DOS FLAG*" (Bandera de Ocupación del DOS). La manera de acceder a ella es mediante la Int 21H función 34H, la cual regresa la dirección de Segmento del DOS y un *offset* hacia la bandera *BUSY*. La forma de interpretar el estado de la bandera es la siguiente: si la bandera *BUSY* se encuentra prendida, el DOS no se encuentra en un estado de seguridad.

Nuestro TSR solo deberá permitir su activación cuando habiendo detectado la presión de las teclas que conforman la *hotkey*, la bandera *BUSY* sea igual a "0". Siendo éste el caso, desde la misma ISR de *timer* se procederá a la activación de la utilería.

Por otro lado, si la bandera *BUSY* se encuentra prendida y la *hotkey* sensada, el TSR deberá retardar su activación hasta la terminación de la función del DOS en progreso.

Este es el momento de plantear una nueva situación: tal parece que todo problema hubiera sido resuelto, pero no hemos tomado en cuenta el caso en el que a pesar de que la bandera *BUSY* se encuentre prendida, las funciones inseguras del DOS no están siendo ocupadas. Esto ocurre cuando el Procesador de Comandos está en espera de instrucciones (hecho que se convierte en otro nuevo problema a tratar).

COMO DETECTAR CUANDO EL DOS ESTÉ SEGURO PARA USAR FUNCIONES ALTAS ENCONTRANDOSE EN UN ESTADO "OCUPADO" ?

Es necesario señalar que aún no haciendo uso de funciones altas, al DOS se le atribuye el estado de "ocupado" durante aquellos periodos en los que el sistema está en espera de un "pulso de tecla" ("keystroke"); periodos que no representan ningún peligro y en los que, para nuestros fines, el sistema se debe considerar en un estado de "seguridad".

Internamente la bandera *BUSY* es prendida no sólo en los momentos en los que el DOS realiza funciones (01H-0CH) de la interrupción 21H, sino tambien cuando está en espera de una entrada de consola. Caso en el que a pesar de que el DOS sea considerado "ocupado", el sistema se encuentra en condiciones de poder utilizar funciones altas y por tanto de permitir la activación del TSR.

Ahora, para detectar esta situación, existe una interrupción que permanece reservada oficialmente por *Microsoft* e IBM. Ésta es la interrupción 28H conocida bajo el nombre de DOSOK (simbolizando DOS Ok!) también conocida como "Interrupción de Ocupación de Teclado". Una descripción general de ésta es la siguiente:

> Descrip. Int.28H: Interrupcion de *software* regularmente llamada durante el "*loop* de poleo" de la cola de E/S del DOS para comunicarle a un programa TSR si las condiciones prevalecientes en el sistema son favorables para que éste pueda hacer uso de operaciones de archivo o de funciones posteriores a la 0CH de la Int 21H. Nota: Estos datos son producto de un compendio de información técnica no oficial.

Antes de prosequir, cabe mencionar que parte del "misterio" del famoso programa PRINT.COM ("*spooler* de impresión") de Microsoft, radica en el uso de esta interrupción, para todos desconocida en ese entonces.

El completo propósito de la Int 28H es "decirle" a los TSRs (específicamente al programa PRINT.COM, desde el punto de vista de Microsoft) si pueden hacer uso o no de las funciones altas del DOS.

Retomando el tema, la solución a nuestro problema inicial salta a la vista: el programa TSR deberá monitorear la interrupción 28H, para que al momento de detectar en el sistema un estado de espera de un pulso-de-tecla, intervenga la nueva ISR DOSOK y proceda a activar el TSR, siempre que la *hotkey* haya sido presionada.

■ LA ISR DE OCUPACION DE TECLADO (DOSOK).

En primer término debemos instalar nuestra propia ISR DOSOK ligándola al vector de interrupcion 28H. De esta manera, la ISR se activará siempre que el sistema este en espera de una entrada de consola.

Cabe aquí anotar que cuando un programa no está ligado al vector de Int 28H, este simplemente apuntara a una instrucción de máquina de regreso de interrupcion (IRET).

La acción inicial de la nueva rutina será la de encadenarse a la Int 28H para hacer de ésta, una interrupcion general a todos los programas. Posteriormente verificará si tanto la bandera *hotkey* como la bandera *BUSY* están prendidas, ya que esto significaría tener el caso de haber sensado la *hotkey* estando el DOS en espera de un "pulso de tecla"; y por lo tanto el TSR puede ser desplegado. Será la ISR DOSOK la que en este caso, se ocupe de dicha tarea. Para ello, primero deberá enviar una señal de Fin de Interrupcion al 8259 y apagar la bandera *hotkey*. En el caso de que la bandera *hotkey* se encontrara apagada, la ISR sólo deberá regresar el control al programa que llamó a la interrupción DOSOK.

Para permitir a un TSR interrumpir a otro, cada TSR deberá llamar a la interrupción DOSOK en el momento que esté esperando leer algo del teclado; ya que en esos instantes el sistema está en condiciones de poder atender a cada TSR. Sabemos que la función **getchar** de Turbo C, hace implícitamente un llamado a la interrupción 28H. Por lo tanto, para lecturas y cualquier entrada de teclado nuestro TSR hará uso de esta función.

COMO EVITAR INTERRUPCIONES A OPERACIONES EN DISCO ?

En el diseno de nuestro TSR, debemos tener en cuenta el caso de que un usuario active la utileria residente, encontrándose el sistema llevando a cabo una operación de disco. Ya que como una operación de disco no puede ser, por ningún motivo interrumpida, nuestro TSR deberá, en el mejor de los casos, retardar su activación hasta que la operación sea completada. Decimos en el mejor de los casos, debido a que muchos programadores optan por ignorar la ocurrencia de la *hotkey*, en lugar suspender la activación del TSR momentáneamente hasta asegurarse de que dicha acción no provocará problemas en el sistema.

Si a un TSR le fuera permitido interrumpir, mientras una operación de lectura, escritura o busqueda en disco estuviera en progreso, el tiempo latente utilizado por el programa TSR podría generar errores de disco para el programa interrumpido. Es decir, si al activarse un TSR se interrumpiera una operación de búsqueda-lectura o de búsqueda-escritura; particularmente después de la busqueda y antes de la lectura/escritura, el uso del sistema de disco del TSR originaria el mal funcionamiento del procesamiento del disco del programa interrumpido.

Para evitar esta situación, todo programa residente se debe ligar por si mismo al vector de interrupción de procesamiento de disco del ROM-BIOS, correspondiente a la interrupción 13H. Una vez teniendo el control de esta interrupción, cada vez que el TSR detecte su ocurrencia, se abstendrá de activarse y colocará una bandera indicando que la *hotkey* ha sido presionada para que, posteriormente dicha bandera sea probada por la ISR del *timer* o la ISR DOSOK.

COMO TRATAR EL CASO DE UN ERROR CRITICO ?

Es de suma importancia, en la construcción de programas residentes, no pasar por alto la posibilidad de que al activarse el TSR e iniciar su trabajo, detecte un error crítico en el sistema, Veamos porqué:

En principio, cuando el DOS encuentra un estado de error crítico sensa una condición de "no listo" y genera la interrupción respectiva, inmediatamente el error es manejado por una ISR en el procesador de comandos que responde desplegando en pantalla el, tan común, mensaje "abort, retry or ignore". Según la opción elegida, la ISR envía de vuelta al DOS un valor que especifica lo que el DOS deberá hacer respecto al error.

Ahora, cuando el sistema no maneja programas residentes, no habrá ningún problema: se procesará la instrucción dada por el usuario, tal como conocemos. Pero cuando se trabaja en un ambiente de TSRs el panorama cambia.

Supongamos que un TSR interrumpe a un programa transitorio e intenta realizar una lectura de disco flexible encontrandose la puerta del *drive* abierta (caso de un error crítico). El sistema genera una interrupción 24H y el procesador de comandos del DOS se hace cargo de la situación enviando a pantalla el mensaje que indica al usuario si desea abortar, reintentar o ignorar el error. Supongamos que el usuario teclea "A". En consecuencia, el DOS intentará abortar el programa corriente. Pero cuál es el programa corriente ?. Estudiemos las consecuencias de las 2 alternativas:

- Si el TSR intercambió previamente las direcciones de PSP, el sistema intentará abortar el TSR; caso en el que, debido a la incapacidad del DOS de reconocer la existencia de un programa transitorio en una condición

do suspendido, nunca se devolverá el control al programa interrumpido aún cuando siga estando asignado en memoria.

- Si el TSR no intercambió las direcciones de PSP, el DOS abortará, sin más, al programa transitorio ignorando completamente que el TSR había sido activado.

La solución radica en ligar todo programa residente al vector de interrupción 24H, aún cuando no se desee manejar los errores relativos a dicha interrupción. El vector debe ser ligado al momento de haber sensado la *hotkey*, y debe ser regresado a su valor anterior, al termino del TSR y antes de regresar el control al programa interrumpido. Una vez ligado el TSR al vector de interrupción 24H, será capaz de detectar cuando una interrupción de tal género ocurra. Caso en el que el TSR deberá ignorar el error, para evitar los problemas antes mencionados; asumiendo que el usuario es capaz de detectar el error. La interrupción de error crítico no será encadenada. Hecho que implica lo siguiente:

- Heredar los errores a otro programa TSR, activado con posterioridad al nuestro.

- Haber heredado los errores de otro programa activado antes que el nuestro, al ordenar que ignore dichos errores, también los ignorará con relación al primero.

Esto se solucionaría escribiendo una ISR manejadora de errores críticos inteligente que abarcara todo el sistema, alternativa poco convincente

La experiencia ha mostrado que muy raramente los casos de error crítico se presentan, y cuando así es, el usuario tiene la capacidad de eliminarlos. El método aquí sugerido, es el empleado por la mayoría de los diseñadores de aplicaciones residentes.

COMO MANEJAR UN EVENTO CTRL-BREAK ?

Sabemos que la interrupción **Ctrl-Break** provoca la terminación inmediata de un programa, pero si este programa es un TSR existirán serios problemas, debido a que la terminación de un programa residente no es igual a la de un programa transitorio normal, por involucrar un mayor número de acciones.

Cuando el usuario presiona las teclas **Ctrl-Break**, se genera la interrupción 23H y es desplegado en pantalla el token " ^C " en la posición corriente del cursor. El manejador correspondiente a esta interrupción provoca la terminación inmediata del programa corriente.

Por otro lado, sabemos que la terminación de un TSR no involucra el mismo procedimiento que la terminación de un programa transitorio. Para terminar un programa residente se debe tener en cuenta que éste tiene ligados ciertos vectores de interrupción y eliminarlo sin restablecer dichos vectores equivaldría a dejar deshabilitadas las interrupciones que éste manejaba para otros programas cargados con posterioridad a él.

Por ello, todo programa TSR debe cuidar que nunca ocurra una interrupción Ctrl-Break durante su proceso. Para ello deberá realizar lo siguiente:

Cuando el TSR sea activado, deberá averiguar el estado corriente (habilitado/deshabilitado) del proceso Ctrl-Break. Para ello, nos podemos valer de la función 33H del DOS, que permite a un programa leer y cambiar el estado del proceso Ctrl-Break. Una vez conocido su estado, en caso de encontrarse habilitado, el TSR deberá cambiarlo al estado inverso. Cuando el programa residente este listo para regresar al proceso interrumpido, deberá colocar el estado del proceso Ctrl-Break a su condición anterior. Notemos que no es necesario ligar el programa al vector de interrupción 23H.

Procediendo de la manera anteriormente descrita, si se da el caso de que el usuario presione las tecla Ctrl-Break cuando el TSR esté corriendo, el programa interrumpido será terminado inmediatamente después de que el programa residente regrese al éste. Si todos los programas TSR usan la misma lógica, estaremos validando que únicamente los programas transitorios sean terminados por la interrupcion Ctrl-Break.

En este trabajo, tambien hemos considerado el caso de que el usuario desee terminar un programa TSR. El método preciso para la remoción de un programa residente, es expuesto en el capítulo siguiente.

PROCEDIMIENTO GENERAL PARA LA CONSTRUCCION DE UTILERIAS RESIDENTES EN MEMORIA

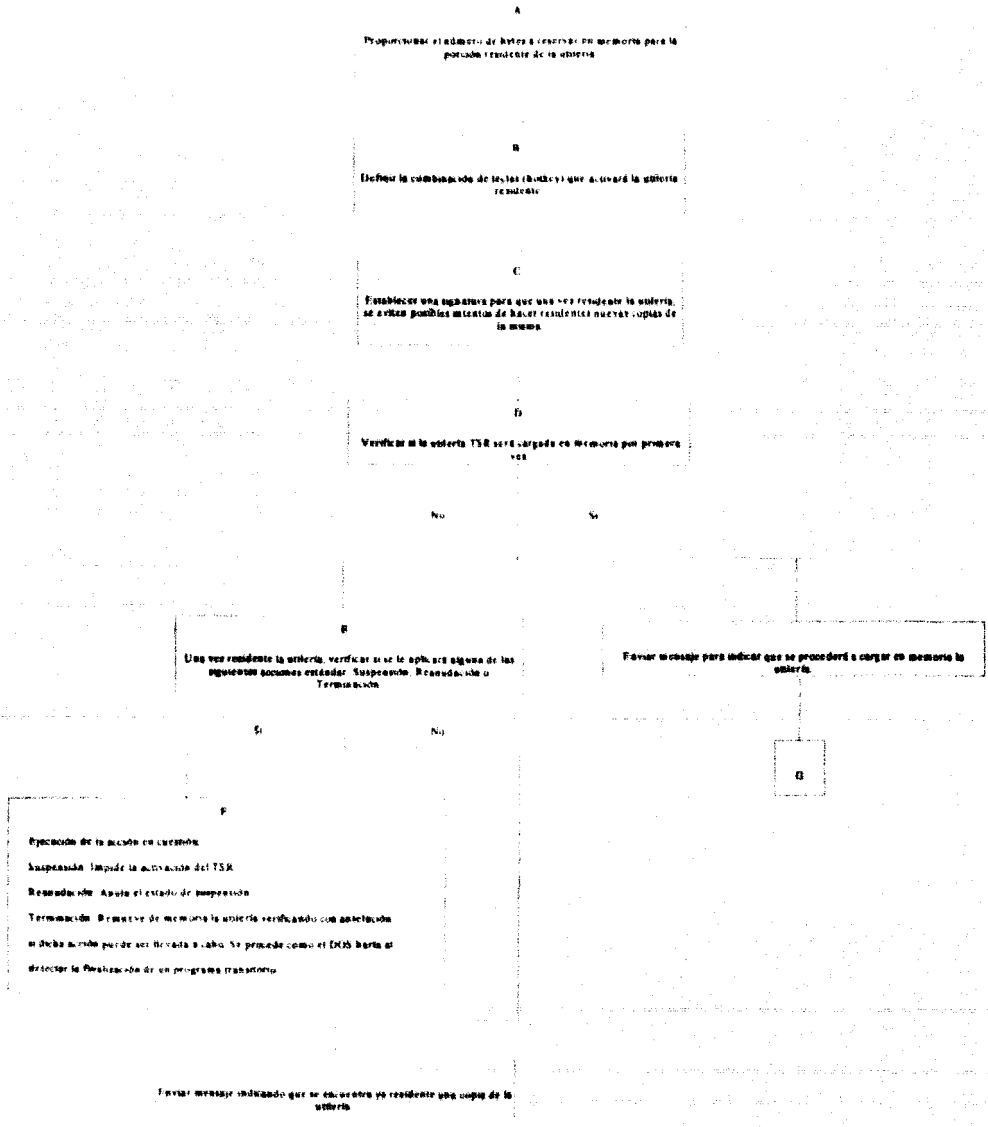
CAPITULO 4

INTRODUCCION

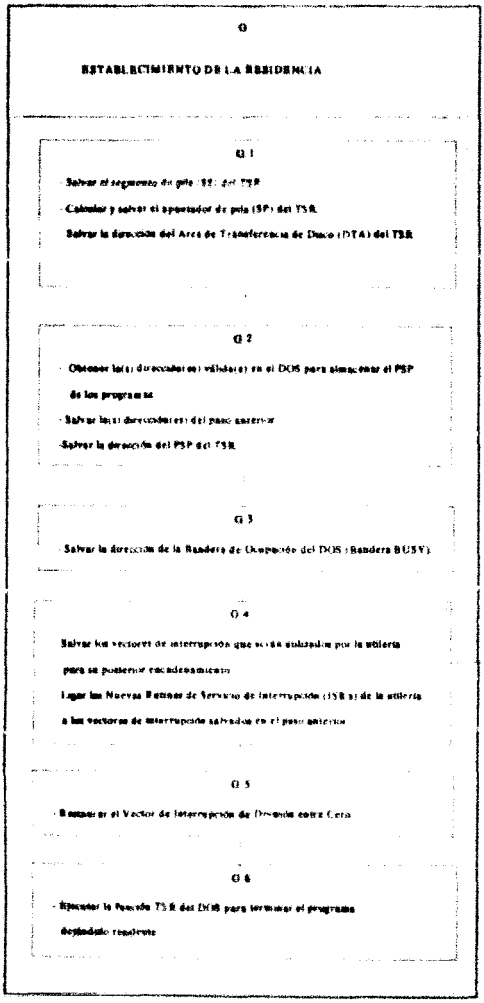
Este capítulo está centrado en la exposición de un método general que puede servir de guía para el programador de aplicaciones residentes en memoria robustas. Para ello, hemos resumido todos los pasos implicados en un esquema general (ilustrado en la fig. 4.1). Dicho esquema muestra cada una de las etapas del método a manera de diagrama de flujo, lo cual facilita al lector la visualización y comprensión del orden en el que cada uno de los eventos se debe suceder.

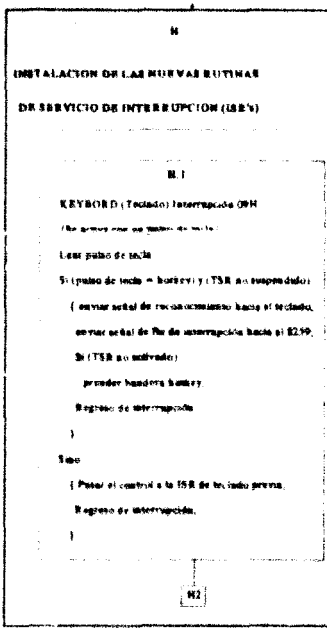
Como se puede apreciar en la fig. 4.1, cada etapa es identificada por una letra. Esta Referencia servirá para ir desglosando a lo largo de este capítulo las particularidades de cada una de ellas.

FIG. 4.1 ESQUEMA GENERAL PARA LA IMPLEMENTACION DE UTILIERIAS RESIDENTES EN MEMORIA

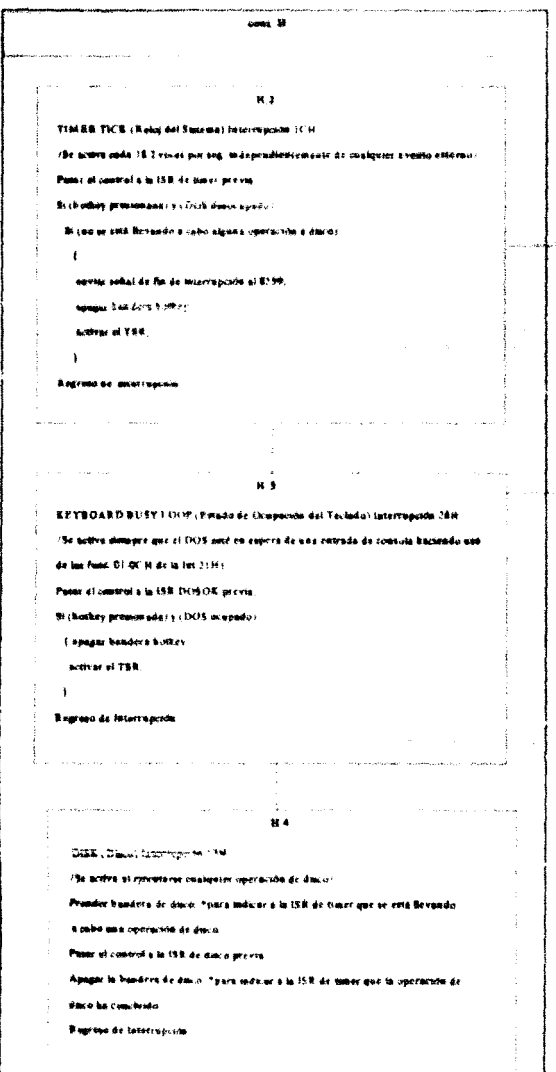


CONT. FIG. 4.1 ESQUEMA GENERAL PARA LA IMPLEMENTACION DE UTILIERIAS RESIDENTES EN MEMORIA





CONT. FIG. 4.1 ESQUEMA GENERAL PARA LA IMPLEMENTACION DE UTILIERIAS RESIDENTES EN MEMORIA



CUADRO A
TAMAÑO DEL PROGRAMA

Un paso preliminar en el diseño de una utilería residente es la especificación al DOS del tamaño de nuestro TSR. Es necesario proporcionar esta información para que el DOS pueda determinar la cantidad de *bytes* a reservar.

Ahora, cómo conocer el tamaño del programa? Como se discutió en el capitulo II, una salida fácil sería declarar el programa de 64 K. Es decir, siguiendo las recomendaciones propuestas con antelación para la construcción de programas residentes, proporcionar al sistema el tamaño máximo admisible (64K), que son ocupados por el área de datos y de pila al emplear el modelo de memoria *tiny*.

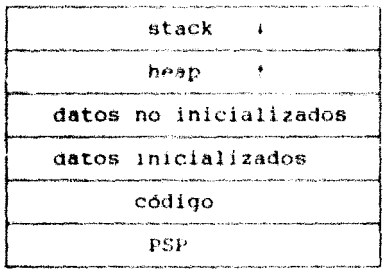
Sin embargo adoptar esta alternativa no es recomendable, ya que si el programa de utilería requiere de menor cantidad de memoria, esto se traduciría en un consumo innecesario de la misma. El desperdicio de memoria constituye un serio problema en el ambiente de TSR's. Recordemos que cada programa cargado aminora el área de TPA.

Es evidente entonces, que lo óptimo será proporcionar al DOS el tamaño exacto del programa. Sabemos que el procedimiento para calcular el tamaño de un programa varía de acuerdo al lenguaje que se emplee. Por ejemplo, en lenguaje ensamblador resulta sumamente sencillo el cálculo del tamaño del programa; sin embargo para la construcción del manejador de programas residentes, hemos elegido utilizar el lenguaje C (específicamente Turbo C), las justificaciones de tal determinación son referidas en la Introducción del presente trabajo. El cálculo del tamaño de un programa en lenguaje C no es tan simple; para tal efecto se debe analizar la forma en la que el compilador de Turbo C construye el programa.

■ CÁLCULO DEL TAMAÑO DE UN PROGRAMA EN TURBO C.

Siguiendo las pautas para la construcción de programas TSR, después de compilar el programa bajo el modelo de memoria *tiny*, resultará el mapa de memoria siguiente:

FIG. 4.2 MAPA DE MEMORIA BAJO EL MODELO DE MEMORIA *TINY*



- El PSP "*Program Segment Prefix*": Es un área de control de 256 *bytes* que se encuentra al inicio de cada programa en memoria; consta de varios campos que el DOS utiliza para manejar el ambiente de procesamiento del programa. (En el capítulo I se trata este tema a detalle.)

- El Area de Código: Contiene las instrucciones en lenguaje de máquina que ejecutan las funciones del programa.

- El Area de Datos: Se divide en dos partes, la primera es el área en la que se encuentran las variables de datos tanto estáticas como externas, que son inicializadas al ser declaradas en el programa; la segunda parte representa el área que contiene las variables de datos estáticas y externas no inicializadas.

- El Area de Heap: Es el bloque de memoria en el que se realizan las asignaciones dinámicas de memoria de nuestro programa. El área de *heap* crece hacia arriba (ver fig. 4.2).

- El Area de Pila: Es el bloque en el que se llevan a cabo todas las operaciones de pila de nuestro programa. El área de pila crece hacia abajo.(ver fig. 4.2).

Cuando el programa es ejecutado por primera vez, el apuntador de pila se encuentra en el tope de la misma, es decir, en la marca de los 64K establecida por un programa basado en el modelo de memoria *tiny*.

Como podemos apreciar, el tamaño del programa está dado por la suma del tamaño de cada uno de sus bloques. En el caso de querer manejar un tamaño fijo de 64K para un programa, no será necesario conocer el tamaño de sus bloques, pero para nuestros fines sí lo será, ya que lo que buscamos es ahorro de espacio en memoria. Así, nuestro problema se traduce en lograr establecer un apuntador de pila en una dirección que señale el límite de la memoria ocupada por nuestro programa, debajo de la marca de los 64K.

A continuación calcularemos cada una de las áreas de memoria del programa.

Tamaño del PSP.

El área para el "Prefijo de Segmento de Programa" siempre es fija de 256 *bytes*.

Tamaño del Area de Código y del Area de Datos.

Para calcular estas áreas, sugerimos hacer uso del programa TLINK de Turbo C, que genera un archivo denominado "MAP" el cual contiene información útil para este propósito. El mapa resultante de este archivo es el siguiente:

FIG. 4.1 ARCHIVO "MAP" DE TURBO C

start	stop	length	name	class
00000H	010BAH	010BBH	_TEXT	CODE
010C0H	013D8H	00319H	_DATA	DATA
013DAH	013DDH	00004H	_EMUSEG	DATA
013DEH	013DFH	00002H	_CVTSEG	DATA
013E0H	013E5H	00006H	_SCNSEG	DATA
013E6H	014EDH	00108H	_BSS	BSS
014EEH	014EEH	00000H	_BSSEND	BSSEND

- En la columna **class** podemos identificar los diferentes tipos de segmento: código (CODE), variables de datos inicializadas (DATA), y variables de datos no inicializadas (BSS).

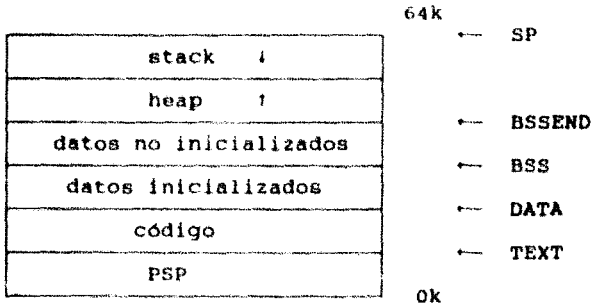
- En la columna **name** se encuentran los apuntadores al inicio de cada uno de los segmentos anteriormente mencionados, considerando el mismo orden: _TEXT, _DATA, _BSS, _BSSEND. Los apuntadores EMUSEG, _CVTSEG y _SCNSEG, también se encuentran dentro del segmento de variables de Datos Inicializados (DATA). Observar que la dirección apuntada por _BSSEND corresponde también al inicio del área de *heap*.

- Las columnas **start** y **stop** contienen el tamaño, en valor hexadecimal, del inicio y final del segmento indicado en la columna **class**.

- La columna **lenght** mantiene el tamaño de cada segmento. El tamaño de un programa que no hiciera uso del *stack* ni del *heap*, se obtendría tan sólo por la suma de los valores contenidos en esta columna, más 256 *bytes* correspondientes al PSP.

Con la información proporcionada por el archivo MAP, el mapa de memoria puede verse como se muestra a continuación:

FIG. 4.4 MAPA DE MEMORIA INDICANDO APUNTADORES



Tamaño del Area de *Heap*.

Como es en el área de *heap* donde se llevan a cabo las asignaciones de memoria del programa, su tamaño dependerá de la cantidad de memoria que el programa asigne; por tanto, debemos buscar algún método para determinar esta información.

En sí, se debe examinar el uso de las funciones de asignación de memoria de Turbo C. Por ejemplo, procediendo de esta manera en nuestro caso, primeramente podemos identificar que las funciones de ventana requieren de asignación de memoria para salvar el contenido previo del área en pantalla sobre la que serán desplegadas. Ahora, para calcular la cantidad de memoria que consumen todas las ventanas (y así tener una aproximación del tamaño del área de *heap*), se tendría que hacer una estimación del tamaño y número de ventanas que serán concurrentemente establecidas, teniendo en cuenta que cada ventana asigna un área de memoria para su "*buffer*-de-salvado" cuyas dimensiones corresponden a la altura de la ventana multiplicada por su ancho y el resultado multiplicado por 2. (Para mayor referencia ver los capítulos V y VI, dedicados al Manejo de Ventanas.)

También se debe tener en cuenta que las operaciones de ocultar o destruir una ventana provocan la liberación de memoria que ocupaba su "*buffer*-de-salvado" en el área de *heap*. Una vez considerado todo lo anterior, se debe proceder a la estimación de las demás demandas de área de *heap* del programa, mediante una revisión de la utilización de las funciones de Turbo C que involucren asignación de memoria.

De esta manera, una vez obtenido el tamaño del área de *heap*, podemos conocer el tope del mismo y por lo tanto la dirección del fondo de la pila.

Hemos presentado un método para la delimitación del *heap*, sin embargo en la parte final del punto siguiente presentaremos otra opción que no implica tanta complejidad y que servirá para la obtención tanto del área de *stack*, como de la de *heap*.

Tamaño del Area de Pila.

Como se puede apreciar en la figura 4.4, la pila se encuentra sobre el *heap* y crece hacia abajo. Cuando el programa es ejecutado por primera vez, el apuntador de pila se encuentra en el tope de la misma, es decir, en la marca de los 64K.

Por otro lado, sabemos que la cantidad de pila requerida por un programa dependerá de la frecuencia con la que se hagan llamadas a funciones anidadas, así como de la cantidad de datos locales usados por dichas funciones. Esto obedece al hecho de que con cada llamada a función se salvan algunos registros y parámetros que la función empleará. Ademas, si la función tiene variables automáticas, tambien serán introducidas a la pila y por tanto el apuntador de pila recorrido hacia abajo. Entonces una función que sea llamada recursivamente y que tenga muchos parámetros y variables automáticas, requerira de una cantidad considerable de espacio de pila. Si por este camino logramos obtener el tamaño del área de pila, solo restaría agregar esta cantidad a partir del tope del área de *heap* (o fondo del área de pila, obtenido en el punto anterior), para así obtener la dirección del tope de la pila, misma que delimitaría el fin de la memoria ocupada por nuestro programa.

Como podemos observar, el proceso para la estimación del tamaño del area de pila, por este metodo, resultaría dificultoso y tardado; ya que conlleva un análisis minucioso de cada función utilizada en el programa.

Así, tras un estudio para encontrar un método para solucionar este problema se cae en la cuenta que la misma naturaleza dinámica del *heap* y del *stack* sugieren al programador determinar el tamaño del programa mediante prueba y error. Este método que a continuación desglosaremos, resultó ser el mejor por razones de rapidez y confiabilidad. El método consiste de los siguientes pasos:

1.- Compilar el programa bajo el modelo de memoria *tiny* de Turbo C. Con lo cual el sistema le asignará un espacio en memoria de 64K.

2.- Proceder a mover el tope de la pila hacia abajo; es decir, por debajo de la marca de los 64K. Por cada movimiento experimental se forzará al programa a emplear la menor parte de *heap* y de *stack* posible.

3.- Continuar los movimientos experimentales hasta que nuestro programa se empiece a comportar de forma impredecible o provoque la "caida" del sistema. Esto indicará que el tope de pila deberá estar en una posición inferior a la presente.

4.- Retroceder el tope de la pila a las posiciones necesarias, hasta establecerlo en un nivel de seguridad (en el que no se presenten los problemas anteriormente mencionados.)

5.- Por último, verificar que el programa trabaje de manera absoluta para la posición de tope de pila seleccionada. Medida con la cual comprobaremos que efectivamente el tope de pila se encuentre en la posición adecuada y no provoque el traslape de las áreas de pila y de *heap*.

CUADRO B
DEFINICION DE LA *HOTKEY*

El paso siguiente es definir la combinación de teclas (*hotkey*) que activará la utilería residente.

Como ya se mencionó en secciones anteriores, un programa TSR puede ser activado por eventos internos o externos; de éstos últimos corresponde hablar en esta parte.

Los eventos externos, son los que se valen de una combinación de teclas (también conocida como "pulso de tecla") o *hotkey* para activar el TSR. Por lo general la *hotkey* está compuesta por una tecla especial (Ctrl, Alt, Shift, Insert, Caps Lock, Scroll Lock) y cualquier otra que tenga un código de exploración válido

■ DETECCION DE LA *HOTKEY*

El proceso que el DOS realiza para la detección e identificación de la *hotkey* es el siguiente: Sabemos que cuando una tecla es presionada, se genera una interrupción de *hardware* que es atendida por la Rutina de Servicio de Interrupción 09H; ésta procede a leer el puerto de entrada de datos de teclado que regresa el código de exploración de la tecla presionada. En el caso de que la tecla presionada haya sido alguna de las siguientes: INS, CAPSLOCK, NUMLOCK, SCROLL LOCK, ALT, CTRL, LEFT SHIFT o RIGTH SHIFT; el DOS actualiza el valor de la máscara de estado del teclado para dar a conocer al programador la tecla especial presionada, cuando éste lo requiera.

Todo lo referente a la interrupcion de teclado y el mecanismo de sensado de la *hotkey* fue ya mencionado en el capítulo III. Sólo recordemos que la máscara de estado del teclado consiste de un *byte* (ver fig. 4.5) que el DOS reserva en la dirección 0:417 para conocer el estado de las teclas especiales antes enumeradas.

FIG. 4.5 MASCARA DE ESTADO DEL TECLADO

INS LOCK	CAPS LOCK	NUM LOCK	SCROLL	ALT	CTRL	LEFT SHIFT	RIGHT SHIFT

La máscara involucra las 8 teclas especiales empleadas para cambiar el estado del teclado, cada una de las cuales corresponde a un bit. Un bit encendido (colocado en "1") indica que la correspondiente tecla está siendo presionada.

De lo anterior dimana que para determinar si la *hotkey* ha sido presionada deberemos:

- Verificar que el estado de la máscara sea tal que corresponda al resultante de haber presionado la tecla especial de la *hotkey*.
- Confirmar que el código de exploración sensado corresponda al del segundo caracter que forma la *hotkey*.

■ RECOMENDACIONES PARA LA SELECCION DE LA *HOTKEY*.

Seleccionar una combinación de teclas que no sea de uso común, es decir, que sea improbable interfiera con otras teclas de programas.

Una buena política es evitar definir como *hotkey* teclas de función como F1, F2, etc., así como combinaciones del tipo Alt-"letra", o del tipo Ctrl-"letra".

Esto debido a que, por un lado, la inmensa mayoría de los programadores han adoptado estos estándares y por otro, numerosos programas involucran teclas de función en su procesamiento.

La combinación elegida por nosotros, para activar el manejador de TSR's fue "Alt-Punto".

CUADRO C
ESTABLECIMIENTO DE LA SIGNATURA

Por su diseño monousuario, monotarea; al DOS le resulta ajeno el ambiente de programas residentes. El DOS solo "entiende" el concepto de programas transitorios; programas que residen en el TPA mientras se estén ejecutando y que indican al DOS su finalización para que éste libere la memoria ocupada en el TPA y recupere los recursos asignados al programa en cuestión.

Hemos mencionado ya en capitulos previos, que el DOS cuenta con funciones para dejar programas residentes en memoria; sin embargo, no tiene manera de saber que un programa quedará residente hasta que éste finalice su ejecución. Tampoco puede conocer el nombre del programa residente, aun cuando la información necesaria para ello se encuentra disponible (como veremos mas adelante).

Por lo tanto, si un programa residente se ejecuta varias veces (desde la Linea de Comandos del DOS), permanecerán residentes en memoria tantas copias del mismo como ejecuciones se hayan ordenado. Por lo que si este problema no es considerado, se tendria un desperdicio de memoria que arrastraria serios problemas y no existiria justificación para la utilización de programas residentes

Como vemos, es necesario que nuestro programa residente pueda determinar si alguna copia de si mismo está ya residente en memoria.

NUESTRO PROGRAMA ESTA YA RESIDENTE ?

La manera más simple para que el TSR pueda determinar si una copia de sí mismo está ya residente, es mediante la utilización de uno de los vectores de interrupción disponibles al programador, que van de la dirección 60H a la 67H, utilizándolo como vehiculo de comunicación entre el programa y su copia residente.

Para ello, consideramos conveniente exponer 2 alternativas:

1.- **Selección arbitraria del vector de interrupción de usuario.**

La mayoría de los programas publicados en revistas y libros, así como algunos paquetes, hacen uso de éste método que consiste de lo siguiente: Seleccionar arbitrariamente uno de los vectores de interrupción de usuario. Luego, durante la primera ejecución del TSR, examinar el vector seleccionado para determinar si su contenido es un valor nulo o bien un apuntador a una rutina. En el primer caso, se asume que la utilería no está todavía residente, por lo que se procede a ligarla al vector de interrupción seleccionado para que posteriormente se declare así mismo residente. En el segundo caso, cuando existe un apuntador al vector de interrupción; el TSR ejecutará la Rutina de Interrupción asociada a dicho vector y aguardará por un valor de regreso; si dicho valor no es el esperado, se colegirá que la utilería no está todavía residente, por lo que para tal efecto se deberá proceder de igual manera que en el primer caso.

Inconvenientes del método:

El hecho de que la selección del vector se realice de forma arbitraria, constituye el principal problema, ya que no se tiene la seguridad de que otro programa no seleccionará también el mismo vector. Recordemos que el diseño del DOS, enfocado a una sóla tarea, permite a ésta el tener a su disposición cualesquiera de los vectores disponibles sin importar que hubiesen estado ligados a otro programa.

2.- **Selección del vector de int. cotejando signaturas.**

Este segundo método constituye una mejor alternativa. Consiste en tener el vector de interrupción previamente ligado a la utilería y apuntando a una signatura en memoria.

La signatura consta de una cadena de caracteres que deberán ser únicos al programa y que se encuentra localizada en el área de datos.
A diferencia del método anterior, al ser cargado el programa por primera vez, explorará los vectores de interrupción desde la dirección 60H hasta la 67H; y si cualesquiera de estos vectores apunta a la signatura, se deducirá que una copia del programa está ya residente. En caso contrario, el programa se apropiará de un vector de interrupción de contenido nulo y colocará el apuntador a la signatura.

Inconvenientes del método: Desafortunadamente no existe una manera infalible de evitar que otros programas se apropien del vector de interrupción seleccionado; sin embargo, éste es el mejor método conocido y utilizado por programas tales como *SideKick* y mismo que emplearemos en este trabajo.

CUADRO D
VERIFICACION DE LA PRIMERA CARGA EN MEMORIA

En este punto las acciones a tomar son las siguientes:

a) Si la utilería todavía no ha sido cargada en memoria por vez primera, proceder a hacerlo. (La metodología para ello es mostrada en el cuadro "G", pag.26).

b) Si la utilería ya ha sido cargada en memoria por primera vez, el programa debe tener en cuenta lo siguiente:

- Contemplar la posibilidad de la existencia de nuevos intentos por volver a cargar la utilería.
  Que es precisamente el caso contemplado en el cuadro anterior (cuadro "C"), en donde se hizo notar la necesidad de que el programa fuera capaz de detectar, mediante el uso de la signatura, si una copia de sí mismo está ya residente para evitar que cada vez que el usuario ejecute el programa, una nueva copia de sí mismo quede residente también.

- Determinar si el intento subsecuente por cargar la utilería fue provocado para establecer comunicación entre esta y la copia residente de la misma. (Caso que será tratado en el siguiente cuadro).

CUADRO E
VERIFICACION DE ESTABLECIMIENTO DE COMUNICACION

En el cuadro "C", se dio la pauta para introducir el concepto de comunicación entre el TSR y su copia residente. Una vez que el programa ha encontrado un vector de interrupción de contenido nulo y se ha apropiado de éste, será empleado con una doble finalidad: primera, para la determinación de la residencia (como fue explicado en el cuadro "C"); segunda, para establecer la comunicación antes mencionada con el fin de cambiar el estado del TSR.

La comunicación involucra lo siguiente: la primera vez que el programa es ejecutado se establece una copia de sí mismo en memoria (conocida como copia residente), luego cualquier ejecución subsecuente de dicho programa da por resultado una segunda copia (llamada copia transitoria) que podrá emplearse para pasar parametros a traves del vector de comunicaciones a la copia residente. Ésta reconocerá el parámetro pasado y procederá a modificar el estado actual del TSR por cualquiera de los estados existentes (suspensión, reanudación o terminación) según el mismo parámetro.

A manera de esclarecer estos conceptos e ilustrar la metodologia, se muestran los siguientes puntos:

1.- La copia transitoria verifica la existencia de una copia residente de la misma. Esto se lleva a cabo explorando los vectores de interrupción disponibles al programador hasta encontrar la signatura que identifica a ambas copias.
2.- Una vez confirmada la existencia de la copia residente, ésta regresa a la copia transitoria el número del vector de interrupción en el que se encontró la signatura. Dicho vector es llamado "vector de interrupción de comunicaciones".

3.- Se inspecciona si la copia transitoria viene acompanada de algún parametro valido (Es decir si al volver a ejecutar la utileria desde la Linea de Comandos del DOS, se incluyó alguno de los parámetros permitidos por el programa). En caso afirmativo: La copia residente deberá tomar las acciones pertinentes, a través del vector de comunicaciones, para ejecutar dicho parámetro (caso comprendido en el cuadro "G").
En caso negativo: De no existir parámetros o de ser inválidos, se desplegará en pantalla un mensaje para indicar que ya existía una copia residente del programa.

CUADRO F
EJECUCION DEL PARAMETRO

Las tres acciones a tomar son "Suspensión", "Reanudación" y "Terminación". El efecto de cada una de ellas es mencionado a continuación:

- **SUSPENSION.**

Accion derivada del parámetro "WAIT".

El suspender un programa TSR no implica quitarlo de memoria, sino de ordenarle desactivarse momentáneamente. Esto se lleva a cabo mediante la utilización de una bandera, que estando encendida indica al programa que ignore la *hotkey*.

- **REANUDACION.**

Acción derivada del parametro "RESTART".

El comando de reanudación tiene la función de limpiar la bandera encendida al utilizar el comando "WAIT". Es decir, anula el estado de suspensión indicándole al programa que la utilería puede volverse a activar.

- **TERMINACION.**

Acción derivada del parametro "QUIT".

El DOS cuenta con un comando para llevar a cabo la terminación de un programa (éste es el comando "QUIT"), sin embargo para terminar nuestro TSR no podemos hacer uso de él, por el simple hecho de que el diseño del DOS sólo se ajusta al ambiente de una sola tarea. No obstante, tampoco podemos cerrar la posibilidad de que el usuario pueda desinstalar el TSR cuando así convenga. Por tanto, debemos contar con un comando de terminación propio que también incluya las funciones que el DOS realiza para terminar un programa transitorio, además de llevar a cabo las pertinentes para terminar un programa residente.

Recordemos que el DOS desconoce todo lo referente a un programa TSR. Así, cuando nuestro programa se haya declarado residente, el DOS sólo lo habrá considerado terminado, olvidándose de su existencia. Concluyendo entonces, que antes de terminar un programa TSR debemos tener presente las diversas situaciones que un ambiente de programas multiples trae consigo, ademas de llevar a término las acciones que el DOS habría realizado en el caso de que el programa hubiera sido transitorio.

Pasos Involucrados en la Terminación de un Programa TSR.

1.- Instruir al programa para terminar por si mismo.
(Acción realizada por la Copia Transitoria).

1a) Para ello, podría ser usada una *hotkey* distinta a la de activación, expreso para indicarle al programa que termine. Sin embargo, no es práctico manejar tantas *hotkey* (mencionamos este caso a manera de descubrir esta posibilidad empleada por algunos programadores).

1b) Otra opción (que será la que emplearemos) consiste en utilizar el vector de interrupción de comunicaciones, enviando por su conducto un comando de terminación a la copia residente de la utileria. (Tal como se expuso en el cuadro "E").

2.- Restablecer los vectores de interrupción involucrados a su contenido previo.
(Acción llevada a cabo por la Copia Residente)
Un paso fundamental comprendido en la acción de quitar un TSR de memoria, es restablecer a su valor anterior el contenido de todos los vectores de interrupción que hayan sido encadenados al programa.
Sin embargo es importante aclarar que sólo se permitirá llevar a cabo esta acción siempre que, encontrándonos en el caso de tener más de un TSR en memoria, los vectores de interrupción encadenados al TSR (que se desea eliminar) no coincida con los de los TSRs cargados con posterioridad.

Para dar explicación a lo anterior, considérese el siguiente caso: Supóngase que se han cargado en memoria 3 programas TSR en el orden siguiente TSR1, TRS2 y TSR3. Los 3 programas encadenan la interrupción de teclado a ellos y el usuario desea sacar de memoria al TSR2. Ignorando la consideración referida en el párrafo anterior, el curso de los acontecimientos sería el siguiente:

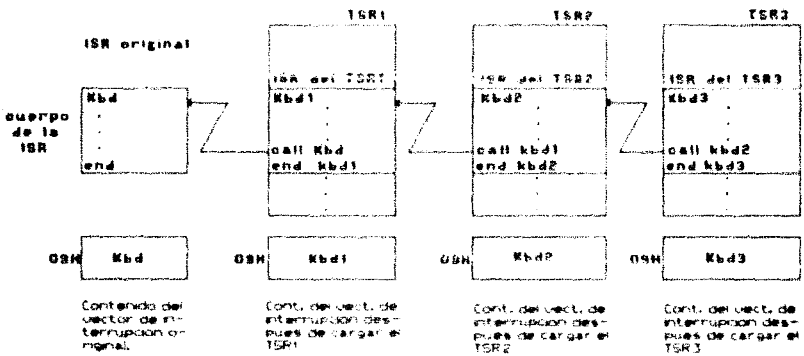
FASE 1.- Previamente a la carga de cualquier TSR, el vector de interrupción de teclado contiene la dirección original de la Rutina de Servicio de Interrupción (ISR) de teclado, (es decir, la propia del sistema) que en la figura 4.6a hemos identificado como ""kbd". Después de la carga del TSR1, el contenido del vector de interrupción de teclado ha cambiado para apuntar ahora a la dirección de la ISR de teclado del TSR1 (kbd1). Lo mismo sucede al cargar el TSR2 y el TSR3, para que finalmente el contenido del vector de interrupción de teclado corresponda a la dirección de la ISR de teclado del último programa cargado (kbd3).

FASE 2.- El proceso para eliminar de memoria el TSR2 consiste en regresar al vector de interrupción de teclado al valor que el TSR2 salvó como previo (kbd1) (ver fig.4.6b), para luego devolver al sistema el espacio de memoria que ocupaba (esto último tratado en el punto 4).
Aquí es importante que identifiquemos los problemas derivados de haber eliminado al TSR2 de memoria:

- A partir de este momento, cualquier petición de entrada de teclado será atendida por la ISR apuntada por el vector de interrupción de teclado, ésta es la ISR del TSR1, que a su vez sólo hace llamado a la ISR de teclado original; por lo que jamás se podrá acceder a la rutina del TSR3, programa que nunca podrá ser activado.

FIG. 4.6 SITUACION ANTERIOR Y POSTERIOR A LA REMOCION DE UN TSR INTERMEDIO EN MEMORIA

A. SITUACION AL MOMENTO DE LA CARGA DE LOS TSR'S

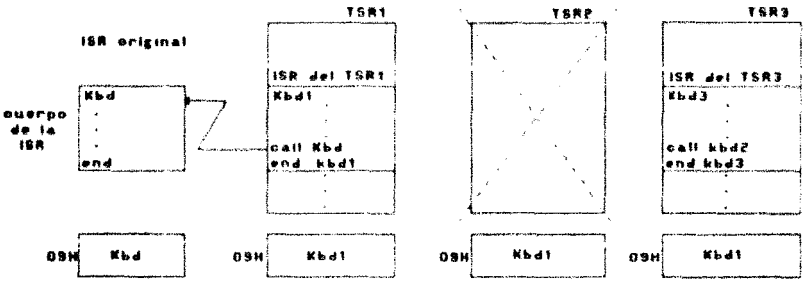


Esta fig. muestra la dependencia existente entre los TSR's al estar encadenados a la misma interrupcion. Cualquier entrada de teclado pasara desde la ISR del TSR3 hasta ser procesada por aquel que hemos contemplado su caso.

NOTA: Ver como cambia el estado del vector de interrupción del teclado.

Cont. fig 4.6

B. SITUACION POSTERIOR A LA REMOCION DE MEMORIA DEL TSR2

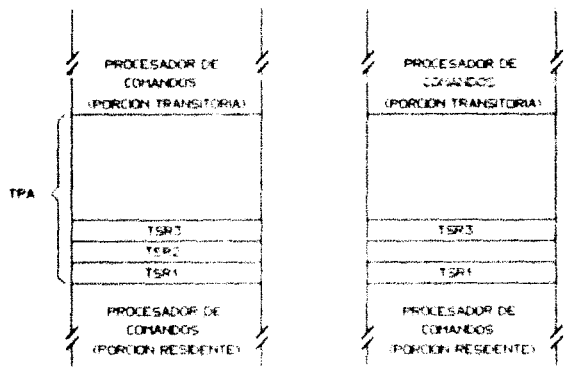


Al quitar el TSR2 de memoria, restablece el contenido del vector de int 9 a su valor previo (kbd1) se rompe la liga. kbd1 se convierte en el valor actual del vector de interrupcion de teclado. Por lo que los TSR's cargados antes del TSR2 podran funcionar, pero los que hayan sido cargados despues de este quedaran deshabilitados.

- Aún cuando se consiguiera que el vector de interrupción de teclado apuntara a la ISR del último TSR cargado (kbd3), ésta su vez pasaría el control a la rutina apuntada por kbd2 (en el caso de detectar una entrada no contemplada por el TSR3, o bien en el caso de pretender quitarlo de memoria), y como kbd2 era el apuntador a la rutina de teclado del TSR2, eliminado ya de memoria, el TSR3 pasará el control del teclado una rutina inexistente.

- Por si fuera poco, si aún con todo lográramos eliminar sin problemas al TSR2 (caso de cualquier TSR que se encuentre "rodeado" por otros TSRs en el área de TPA), se tendría el problema de una memoria fragmentada. (Ver fig. 4.7).
  Los espacios libres sólo podrían ser empleados en el caso de que algún programa requiriera asignación de memoria, pero nunca se utilizarán para cargar otros programas. El DOS no fue creado para administrar ni manejar el concepto de memoria fragmentada.

FIG. 4.7 ESQUEMA DE MEMORIA FRAGMENTADA



a) AL MOMENTO DE CARGAR LOS PROGRAMAS EN EL ORDEN: TSR1, TSR2, TSR3

b) AL MOMENTO DE SACAR DE MEMORIA EL TSR2

NOTA: AL ELIMINAR DE MEMORIA CUALQUIER TSR A EXCEPCION DEL ULTIMO EN HABER SIDO CARGADO, SE TENDRA UNA MEMORIA FRAGMENTADA CUYOS ESPACIOS NO PODRAN SER REUTILIZADOS.

SOLUCION: Antes de restablecer a su contenido previo los vectores de interrupción encadenados al TSR que se desea eliminar, se debe determinar si este todavia posee los vectores de interrupción corrientes, ya que de lo contrario podemos asumir con toda seguridad que otro programa cargado despues del nuestro nos los ha arrebatado.

Siendo asi, no podemos sacar al TSR de memoria (por los problemas mencionados en el punto anterior); por lo que la solucion consiste en desactivarlo "momentaneamente", es decir en ponerlo en un estado de suspensión.

Pero antes de continuar, cabe hacer mención de la manera en la que el TSR puede precisar si todavia posee los vectores de interrupción: debe comparar las direcciones de sus ISRs con las contenidas en los vectores de interrupción correspondientes, si cualquiera de ellas no coincide significa que algún TSR cargado con posterioridad al nuestro se ha apropiado de la interrupcion en cuestión, y se procederá como se estableció anteriormente. Pero si se ha comprobado que el TSR todavía posee todos los vectores de interrupción a los que se encadenó, se permitira proceder a sacarlo de memoria siguiendo el paso 3.

Como se puede observar, una manera inequívoca de eliminar los TSR de memoria es en el orden inverso al que fueron cargados. Sin embargo nuestro método además, valida el caso de algún intento, por parte del usuario, de quitar de memoria cualquier otro programa TSR.

3.- Cerrar los archivos abiertos por el TSR.

Debemos mencionar como antecedente que cuando un programa transitorio termina, el DOS automáticamente cierra todos los archivos abiertos por éste; esto lo lleva a cabo explorando el arreglo de manejadores de archivo contenido en el PSP del programa.

Ahora bien, cuando un programa residente termina, el DOS no se encarga de cerrar los archivos abiertos por él, puesto que para el nunca terminó y el seguir trabajando en tales condiciones podría provocar serios problemas. Veamos a continuación porqué.

El DOS mantiene una tabla de manejadores de archivo con un número fijo de entradas, cada una de las cuales representa un archivo actualmente abierto por algún programa. Las entradas de la tabla son liberadas cuando el programa cierra los archivos respectivos. Así, si al eliminar de memoria un TSR no se cierran los archivos que éste haya abierto con antelación; las entradas de la tabla correspondientes a dichos archivos nunca serán liberadas y por lo tanto no podrán ser reutilizadas por otros programas. Y como se puede colegir, se corre el riesgo de agotar las entradas a la tabla conforme más programas sean cargados en memoria, caso en el que se podría provocar la "caída" del sistema.

SOLUCION: Como ya se puede advertir, la salida al problema consiste en que la rutina de terminación de nuestro TSR incluya las acciones convenientes para cerrar los archivos abiertos por éste. Para ello consideramos de utilidad recordar que las entradas al arreglo de la tabla de manejadores de archivo en el PSP del programa, representan prácticamente archivos a nivel manejador. El utilizar dichas entradas equivale a emplear las funciones de E/S de bajo nivel "no buffereadas" (desprovistas de memoria intermedia) en lenguaje C. Por otro lado, las funciones de "C" que ejecutan el flujo de E/S utilizando memoria intermedia, mantienen sus propios *buffers* internos y apuntadores; y pueden requerir vaciar sus *buffers* en lugar de cerrarlos. Esta acción de vaciado, más que una operación del DOS, es una operacion de librería de C.

Nuestra utileria deberá ser capaz de cerrar este tipo de archivos, para lo cual tiene a su alcance la función estándar fclose de C. Por otro lado, los archivos que un TSR abra simplemente con las funciones open y creat deben ser cerradas tan sólo con la función close.

4.- Devolver al sistema el espacio ocupado por el TSR.
Evidentemente, el paso final involucrado en la terminacion de un TSR es liberar la memoria que le fue asignada. Para tomar tal acción es necesario conocer el proceso de asignación y desasignacion de memoria bajo el DOS. Por ahora, teniendo como objetivo plantear el problema en general y exponer la solución a éste, nos baste recordar que la memoria es asignada a un programa por 2 conductos: por el DOS directamente o por el programa mismo a traves de las funciones del DOS; y que en ambos casos el proceso de asignación y desasignación de memoria es el mismo.
Ahora, recordemos que para asignar memoria, el DOS designa bloques de memoria a cada programa; y para identificar qué bloques corresponden a qué programas, cada bloque mantiene una cabecera, que en realidad es otro bloque de memoria contiguo de 16 *bytes*, conocido también como bloque de control, el cual reserva un campo de 2 *bytes* para guardar el PID del proceso que tiene apropiado ese bloque de memoria. Por simplicidad y para referencias posteriores llamaremos a este campo "campo identificador" ,(los demas campos no son de nuestro interés por el momento). Cuando el campo identificador es nulo se asume que el bloque de memoria asociado a él esta libre.
SOLUCION: Ahora bien, conocido lo anterior podemos inferir la solución al problema de liberar la memoria asignada a nuestro programa. Nuestro cometido será:

a) Identificar todos los bloques de memoria asociados a nuestro programa.

Esto será llevado a cabo cotejando el valor del PID de nuestro programa con el que reside en el campo *identificador de cada bloque de control* existente.
Aquí cabe mencionar que a saber existen al menos 2 bloques de memoria asignados a nuestro programa (independientemente de los demás); el ocupado por el PSP del programa y el bloque del ambiente del programa (encontrado en la dirección señalada en el PSP con *offset* 2CH).

b) Una vez determinados los bloques asignados a nuestro programa, debemos proceder a liberarlos del área de TPA cambiando el valor del campo de PID a un valor nulo.
Para tal efecto utilizaremos la Int 21H función 49H del DOS.

Finalmente habiendo completado los pasos 1, 2, 3 y 4 podemos concluir que el TSR ha sido terminado y eliminado de memoria. Para complementar los conceptos de "Asignación de Memoria" aquí mencionados, y para ubicar al lector en un contexto general, hemos incluido en el capítulo I una sección dedicada al tratamiento exclusivo de este tema.

CUADRO G
ESTABLECIMIENTO DE LA RESIDENCIA

Esta parte esta dedicada a exponer el procedimiento seguido para hacer residente un programa, poniendo de manifiesto que para tal fin, es necesario salvar el estado corriente de ciertos parámetros antes del establecimiento de la residencia.

Previamente al establecimiento de la residencia del programa, mediante la ejecución de alguna de las funciones del DOS (mencionadas con antelación), es indispensable que el programa obtenga cierta información que en cualquier otro momento sería imposible determinar. Esto, debido a que cuando el TSR es ejecutado por primera vez, goza de todos los recursos que el DOS proporciona a un programa normal, pero después de que el programa se haya declarado así mismo residente y terminado, dichos recursos serán otorgados a otros programas, o bien al Procesador de Comandos del DOS, en caso de no existir programas cargados con anterioridad.

La información que el programa deberá salvar es la siguiente: SS (Segmento de Pila), SP (Apuntador de Pila), DTA (Apuntador al Area de Transferencia de disco), PSP del TSR, BUSY DOS FLAG (Bandera de Ocupación del DOS), y los vectores de interrupción empleados por el TSR.

En el esquema global-cuadro G encontrado al inicio de este capitulo, dividimos las acciones a tomar en 6 bloques que agrupan operaciones relacionadas entre sí, con el fin de proporcionar una explicación ordenada y especifica a cada parte; así como de justificar y explicar la utilidad de salvar la información. Veamos cada uno de estos bloques:

CUADRO G.1

En este bloque se indica salvar las direcciones de: SS, SP y DTA del TSR. Los valores de éstos parámetros serán utilizados posteriormente para llevar a término la conmutación de contexto general.

Por ahora, basta recordar que los conceptos manejados en la conmutación de contexto son de vital importancia para la construcción de TSRs poderosos; ya que de ello se desprende el complemento a la solución a problemas tales como el de la reentrancia.

En si, en este punto se llevan a cabo los pasos de lo que conocemos como conmutación de pila y de DTA, que forman parte de la conmutación de contexto general, y cuyos pasos finales se concentran en la parte de "Activación del TSR" (Cuadro "I").

Una vez conocido lo anterior, observemos que en el esquema general, se indica que un paso preliminar al de salvar el Apuntador de Pila del TSR es el cálculo del mismo. La forma de proceder para tal fin es mostrada en la fase de documentación del programa.

CUADRO G.2

En este cuadro se apunta que el paso siguiente consiste en salvar las direcciones del PSP del TSR, reservadas por el DOS para almacenar la dirección del PSP de todo programa.

La información anterior será empleada posteriormente para llevar a cabo la conmutación de PSPs, parte de la conmutación de contexto general, que al igual que en el punto anterior, será concretado en la parte de "Activación del TSR" (Cuadro "I").

Cabe aclarar que la acción de salvar las direcciones válidas mantenidas por el DOS para almacenar la dirección del PSP de los programas, no se puede llevar a cabo directamente. Tal como se puede apreciar en el esquema global, es necesario obtener con anterioridad dichas direcciones.

El proceso a seguir para tal fin se muestra a continuación, pero antes de exponerlo queremos hacer tres aclaraciones:

Primera: Emplearemos de aquí en adelante las siglas "PID" para referirnos a la dirección del PSP del programa corriente; por razones de simplicidad y para identificarnos con el argot de los diseñadores del DOS.

Segunda: Es necesario tener presente que, el DOS, a partir de la versión 3.0, reserva solo una localidad para almacenar el PID mientras que las versiones anteriores reservan dos. Así, nuestro programa deberá prever ambos casos.

Tercera: Recordar que las instrucciones involucradas en este punto, se ejecutarán al tiempo de corrida, cuando el TSR sea cargado por primera vez.

PROCEDIMIENTO PARA LA OBTENCION DE LAS DIRECCIONES DE PID

1.- Obtener el PID del TSR ejecutando la función **GetPID** del DOS.
2.- Salvar el PID del punto anterior en una variable.
3.- Explorar la memoria ocupada por el DOS hasta encontrar un valor idéntico al PID de nuestro TSR.
4.- Una vez encontrado el valor, salvar temporalmente la dirección donde fue encontrado el PID (direccion prueba), hasta cerciorarse de que dicha dirección sea válida. (punto siguiente).
5.- Verificar que la "direccion prueba" corresponda con aquella que el DOS reserva para almacenar el PID del proceso corriente.
Para ello, hemos llegado a la siguiente conjetura: Primero, si la "dirección prueba" es correcta, su contenido debe ser el PID del proceso corriente; por lo que, al ejecutar la función SetVect del DOS, para cambiar el valor del PID artificial (de muestra) conocido; el contenido de la "dirección prueba" deberá conmutar a este segundo valor, caso en el que podremos

afirmar que se ha encontrado una dirección válida del DOS para almacenar el PID. (No pasar a otro punto hasta que la prueba tenga éxito).

6.- Volver a colocar el contenido del PID original (salvado en el punto 4) en la dirección de PID encontrada.

7.- Continuar la exploración de memoria iniciada en el punto 3 hasta encontrar un segundo valor idéntico al PID de nuestro TSR. Cumplido lo anterior, proceder como se hizo anteriormente a encontrar la primera dirección de PID válida, y pasar al punto 8.
Si al explorar la memoria por completo no se encontró ningún valor idéntico al que se está cotejando (caso de manejar la versión 3.0 del DOS o posteriores), saltar este punto.

8.- Salvar la(s) dirección(es) válida(s) de PID del DOS antes de que el TSR se declare así mismo residente.

CUADRO G.3

En esta parte se señala salvar la dirección de la Bandera "BUSY", o Bandera de Ocupación del DOS. Éste es un paso importante, preliminar al establecimiento de la residencia, sin el cual no podría determinarse el instante idóneo (en cuanto a seguridad se refiere) para activar el TSR.

Recordemos que la dirección de la bandera "BUSY" debe ser salvada para permitir a la ISR del *Timer* conocer el estado de la misma y con ello, aunado a las demás consideraciones discutidas en el capítulo II, poder determinar si podrá proceder al despliegue del TSR.

Ahora, para la obtención de la dirección de la Bandera "BUSY", podemos hacer uso de la Int.21H func. 34H del DOS, que regresa en el registro ES la dirección-de-segmento del DOS y en **BX** el *offset* hacia esta bandera especial que el DOS mantiene.

Cabe mencionar que, la función 34H del DOS forma parte de las funciones NO DOCUMENTADAS de *MICROSOFT*; por ser pieza importante para el esclarecimiento de los programas residentes en memoria.

CUADRO G.4

El paso que se indica en este cuadro es salvar los vectores de interrupción que serán utilizados por la utilería y ligar las nuevas ISRs a ellos.

La acción de ligar las ISRs consiste en cambiar el contenido de los vectores de interrupción para dejarlos apuntando a las nuevas rutinas sustitutas del TSR; para lo cual es indispensable haber guardado con antelación el valor previo de cada vector de interrupción involucrado.

Si no se actuara de esta manera se rompería la cadena que debe siempre prevalecer en un ambiente de programas múltiples. La acción de salvar los vectores de interrupcion y de llamarlos en el momento adecuado, también puede es identificada bajo el patronímico de "encadenamiento". Conceptos que ya han sido tratados por separado con anterioridad.

Los vectores de interrupcion que serán salvados y encadenados son: el del *timer*, teclado, Ciclo de Ocupación de Teclado (DOSOK) y Disco.

CUADRO G.5

Este cuadro ordena restablecer el Vector de Interrupcion de División entre Cero a su contenido anterior. A continuación explicamos el porque y el cómo llevar a cabo esta acción:

Cuando es corrido un programa transitorio en Turbo C, inmediatamente se empieza a ejecutar su código de arranque (residente en el archivo C0t.obj, para programas compilados bajo

el modelo de memoria *tiny*; y en el archivo C0s.obj, para programas compilados bajo el modelo de memoria *small*), proceso durante el cual se establecen los valores iniciales del *stack* y del *heap* para luego invocar a la función principal (que involucra todo el código) del programa de aplicación.

Sin embargo, es de suma importancia hacer notar que el código de arranque incluye una ISR de División-entre-Cero, la cuál es ligada al vector de interrupción de División-entre-cero, antes de que la función principal del programa de aplicación sea invocada. Es decir, se coloca una nueva ISR de División-entre-Cero para atender todas las peticiones hechas por el sistema. Al regresar la función principal al código de arranque, el Vector de interrupción de División entre Cero es restablecido a su contenido previo. Cabe aclarar que cuando decimos que la función principal regresa al código de arranque, significa en un programa transitorio que el código del programa de aplicación ha sido ejecutado y el programa está listo para terminar.

Notemos ahora que ocurriría al ejecutarse un TSR: Los programas residentes en memoria no terminan con el regreso de la función principal, ya que para tal fin hacen uso de las funciones TSR del DOS; por lo que al Vector de Interrupción de División entre Cero nunca le es regresado su valor anterior, dando por resultado que si en algún otro programa se genera un error de división entre cero, este será procesado por la ISR que se encuentra ligada al código de arranque del TSR, pero el TSR no desea que esto ocurra, ni tampoco el programa que genero dicho error.

Como se puede apreciar, es indispensable que nuestro programa TSR tome en consideración lo anterior e incluya la acción de restaurar el Vector de Interrupción de División entre Cero antes de llamar a la función para terminarlo y hacerlo residente. Observemos que no estamos haciendo otra cosa que proceder tal como el sistema lo hubiera hecho al tratarse de programas transitorios.

Borland junto con el software de Turbo C, proporciona el código fuente para el codigo de arranque. Dicho codigo se encuentra en 2 archivos fuente llamados **c0.asm** y **rules.asi**. Será necesario modificar el archivo **c0.asm** y ensamblarlo dos veces: una para el modelo de memoria tiny y otra para el modelo de memoria small. La variable en la que **c0.asm** salva el vector es **ZeroDivVector**, pero esta variable viene declarada como local. Por lo tanto, para que el programa TSR en C pueda restablecer el vector de interrupcion de Division-entre-cero al valor almacenado en dicha variable, se deberá modificar **ZeroDivVector** para que quede declarada como una variable de tipo externa. Para hacer esto, debemos saber que Turbo C añade un caracter de subrayado al inicio de nombres de variables externas, de modo que en todo el código de **c0.asm** donde aparezca la variable en cuestión, deberá ser modificada a _ZeroDivVector, además la definición donde esta declarada la variable **ZeroDivVector** en **c0.asm** deberá ser remplazada por esta definicion:

```
PubSym@  ZeroDivVector <dd  0>, __CDECL__
```

Una vez realizado lo anterior, ensambla el archivo 2 veces con los siguientes comandos:

```
C>masm c0,c0t /ML  /D__TINY__;
C>masm c0,c0s /ML  /D__SMALL__;
```

Estos comandos crearan 2 archivos denominados **c0t.obj** y **c0s.obj**, con los que se deben remplazar los que traia Turbo C de fabrica.

Finalmente al inicio del codigo del programa TSR en C, se debe declarar **ZeroDivVector** como un apuntador a una función de interrupción, este apuntador es el que se definió como variable externa en el codigo de arranque. Esto se hace con el fin de poder devolver a su valor original el contenido del Vector de Interrupción de division-entre-Cero, para lo que se hará uso de la función **setvect()**. Cabe aclarar que todo esto debe ser realizado antes de que el programa se declare residente.

Existe otra solución para aquellas personas que no desean utilizar la técnica antes descrita, y consiste en permitir que el TSR maneje el error de división-entre-cero la primera vez que éste ocurra. Al ocurrir dicho error en algún otro programa, la ISR del código de arranque de Turbo C desplegara un mensaje de error y terminara el programa, que es exactamente lo que hace el manejador de interrupción de división-entre-cero del DOS. Debido a que el DOS se encarga ahora de finalizar el programa, éste restablece el vector de interrupcion de división-entre-cero a su dirección original, y por lo tanto, el TSR no se hará mas cargo del problema. Este enfoque es mañoso, pero trabajara. Por supuesto que se deberán quitar todas las referencias a la variable **ZeroDivVector** en los programas TSR descritos despues en esta tesis.

CUADRO G.6

En este cuadro se indica ejecutar la función TSR del DOS para terminar el programa dejándolo residente.

En el capitulo II se hablo ya de las funciones que el DOS proporciona para terminar un programa dejándolo residente en memoria. Para la construcción de nuestro programa de utilería TSR, haremos uso de la funcion 31H del DOS.

Con este ultimo paso damos por terminado el bloque de instrucciones necesarias para el establecimiento de la residencia de nuestro programa.

CUADRO G
INSTALACION DE LAS NUEVAS RUTINAS DE SERVICIO DE INTERRUPCION

En el esquema general (fig. 4.1) se muestra a manera de pseudocodigo el contenido de cada una de las ISRs que deben ser ligadas a los vectores de interrupcion en el cuadro "G".

Toda la información relativa a las rutinas de servicio de interrupción puede encontrarse en el capitulo III. Remitirse a las secciones siguientes:

- ISR de Teclado.
- ISR de *Timer*.
- ISR de Ciclo de Ocupación de Teclado.
- Cómo evitar interrupciones a operaciones en disco.

CUADRO I
ACTIVACION DEL PROGRAMA TSR

La activacion del TSR es llevada a cabo desde la ISR de *Timer* o desde la ISR "DOSOK" tal como se puede apreciar en el esquema general. Esta sección presenta el procedimiento a seguir para activar el TSR.

Primero observemos que previa y posteriormente a la ejecución de la funcion de activación del TSR se llevan a cabo ciertas acciones particulares.

Las acciones preliminares a la activación del programa incluyen las instrucciones para salvar el contexto del proceso que será interrumpido por el TSR, así como para establecer al TSR como proceso corriente en el sistema.

Después de la activación del TSR, se incluyen las operaciones para volver o instaurar como proceso corriente al programa interrumpido, para lo cual se recupera el contexto del mismo salvado con anterioridad.

Todo este proceso constituye lo que se conoce comunmente como "conmutación de contexto", entendiendo por ello la transferencia del control del sistema de un programa a otro.

En resumen, los pasos involucrados en la activación del TSR son los siguientes:

NOTA: La activación del TSR debe obedecer tanto a la acción de haber presionado la *hotkey*, como a la de haber acreditado las pruebas necesarias.

1.- Como se puede observar en el esquema general, se inicia por prender una bandera para indicar que la activación del TSR está en proceso. Con esta acción se impide que la ISR de teclado permita un nuevo intento por presionar la "*hotkey*', validando el caso de que una "llamada reentrante" (término manejado por los diseñadores de "software") vuelva a comenzar el proceso de activación.

2.- Primera fase de la conmutación de Pila:
Se salvan los registros de pila del programa interrumpido y se establecen como corrientes los del TSR.

3.- Primera fase de la conmutación de DTA:
Se obtiene, mediante una función del DOS, la dirección del área de DTA del programa interrumpido y se salva en una variable; para luego actualizar la dirección de DTA a la del TSR.

4.- Primera fase de la conmutación de PSPs:
Consiste en salvar el PSP del programa interrumpido para actualizar las direcciones de PID del DOS al valor del PSP del TSR.

5.- A continuación se salva la dirección del Vector de Interrupción de Error Crítico y se liga una nueva ISR de Error Crítico a dicho vector; esto, debido a que durante la ejecución del TSR queremos que se ignore todo error de este tipo (como ya se había justificado).

6.- Guardar el estado original de la bandera de Ctrl-Break y deshabilitar sus interrupciones.
Es de vital importancia deshabilitar el proceso Ctrl-Break mientras el TSR se esté ejecutando. Recordemos que cuando el usuario presiona las teclas Ctrl-Break, el DOS despliega el token " ^C " en la posición corriente del cursor en la pantalla, y al acto se ejecuta la interrupción 23H que provoca la terminación inmediata del programa corriente. Esto a todas luces sería catastrofico para el TSR ya que podría tener encadenados a él ciertos vectores de interrupción manejados también por programas cargados con posterioridad, provocando así el rompimiento de la cadena.

7.- Activación del TSR. (Ejecución del prog. de utilería).

8.- Una vez dada por terminada la ejecución del TSR, se regresa a su estado original la Bandera de Ctrl-Break. Así, si por algun motivo se hubiese presionado la combinación Ctrl-Break durante la ejecución del TSR, la acción resultante de este evento se aplicará hasta el término del TSR y sobre el programa interrumpido (en el caso de ser un programa transitorio).

9.- Se restablece el Vector de Interrupción de Error Crítico al contenido salvado en el punto 5.

10.- **Fase final de la conmutación de PSPs:**
Esta fase consiste en actualizar las direcciones de PID del DOS a las del PSP del programa interrumpido, para que éste vuelva a ser reconocido por el sistema como proceso corriente.

11.- **Fase final de la conmutación de DTA:**
Se actualiza la dirección del Area de Transferencia de Disco a la del programa interrumpido. Salvada en el punto 1.

12.- **Fase final de la conmutación de Pila:**
Los registros de pila del DOS son llenados con los valores salvados en el punto 2. Con ello el SP y el SS nuevamente apuntan al programa interrumpido.

13.- **Apagar la bandera que indica TSR terminado.**
Al llevar a cabo este último punto se sientan las condiciones para volver a permitir una nueva activación del TSR.

LOS PROGRAMAS POP-UP

Y LAS VENTANAS DE VIDEO

CAPITULO 5

UN AMBIENTE INTERACTIVO

Hoy día, en el mundo del *software* uno de los aspectos que ha cobrado mayor importancia es el de procurar un ambiente del todo amigable con el usuario. Las expectativas del usuario cada vez son mayores; conforme avanza el tiempo los usuarios exigen más capacidades de los sistemas de *software*, demandando mejoras en cuanto a rapidez, eficiencia, robustez....; y en la actualidad, enfatizándose una característica más: la sencillez.

Un sistema interactivo, como su nombre lo indica, es aquél que "interactua" con el usuario; está diseñado para aleccionarlo respecto a lo que debe hacer en cada etapa y durante toda una sesión. Un sistema interactivo deberá estar basado en un lenguaje que le permita hacer buen uso del teclado y la pantalla. En palabras simples, desde la perspectiva del usuario, lo que de un paquete de aplicación se espera es: orden, presentación por pantallas, facilidad de uso, manejo de colores, disposición de mensajes de ayuda en cada etapa, corrección de errores, etc.

De las muchas técnicas, que existen en la actualidad, para la creación del ambiente de los sistemas de *software*, una de las más populares es aquella que se basa en lo que se conoce como "ventanas de video". Todos hemos tenido ocasión de conocer el ambiente de ventanas; no obstante, iniciaremos este estudio a partir de su definición, para ir profundizando en el conocimiento del tema de manera ordenada.

Las ventanas están constituidas por un área rectangular de pantalla delimitada por un borde visible, son desplegadas independientemente de las demas, y su uso puede ser de lo más variado: para desplegar menús, mensajes, información de ayuda, áreas para entrada de elementos de datos y texto, etc.

Mencionado lo anterior, no hace falta que hagamos explícita la relación existente entre los programas *pop-up* y el ambiente de ventanas. De primera instancia, la misma naturaleza de los programas *pop-up* ("aparecer-desaparecer") dan la respuesta.

Al hacer su aparicion en pantalla, un programa residente activo está intrínsecamente manejando el concepto de ventanas de video, pues está ocupando un área de la pantalla (ventana) que debe restablecer a su contenido anterior al ser desactivado. Ademas, cada utileria residente en memoria puede incluir varias facilidades, que a su vez pueden requerir el despliegue de ciertos mensajes (ventanas de ayuda).

El manejo de ventanas es indispensable en la elaboración de programas residentes activos, en esta segunda parte del trabajo, aprenderemos todos los aspectos relativos al manejo de un ambiente de ventanas; empezando desde el diseño de las mismas, su codificación y concretando con ejemplos de aplicación.

Este tema incluye varios capitulos, constituyendo la segunda parte de esta tesis; la razón que nos movió a dedicar a esta materia, todo este espacio es la siguiente: Creemos que un programador especialista en residencia, no puede permanecer al margen de las técnicas de manejo de video; pues estas constituyen pieza fundamental de los conceptos que las utilerías residentes implican. Existen en el mercado algunas librerías de funciones de ventana, pero indiscutiblemente un programador que cuente con todos los conocimientos involucrados en el desarrollo de utilerías residentes, no se verá nunca limitado para poder desarrollar programas residentes de todo tipo, tendrá más claros los conceptos que el tema de programas *pop-up* implica, y por tanto podrá ser un buen instructor de la materia; podrá tener una mejor concepcion de la relación existente entre los aspectos de *hardware* y los de *software* de la PC, y ademas podrá comprender la mecánica del nuevo concepto de presentación de ventanas, manejado por diversos sistemas, y tan ampliamente difundido.

Un programa TSR que cumple con las caracteristicas de interfaz con el usuario basada en una especie de "diálogo", y presentación de ventanas es considerado un programa de computadora interactivo. Sabemos que el medio de expresión de la computadora es un lenguaje, pero cada programa de aplicación que nosotros hagamos (con las caracteristicas antes señaladas) puede ser considerado un nuevo lenguaje o "dialecto" derivado del lenguaje al que debe su origen.

En la elaboración de este nuevo lenguaje, el programador debe analizar diferentes cuestiones como: el ambiente de programación, las estructuras de datos, los algoritmos funcionales y la comunicación con el usuario. La solidez en la interfaz de usuario será de suma importancia. Esta segunda parte, está canalizada a enseñar al programador a crear un ambiente de presentación de ventanas, instruyéndole desde el diseño de las funciones básicas generales. Funciones que podrán luego ser llamadas por cualesquiera programas para generar el ambiente interactivo de usuario.

El lenguaje en el que nos basaremos es el lenguaje C (por las razones ya mencionadas en la introducción de esta tesis), y las funciones que diseñaremos involucrarán manejo de: ventanas de video en sus diferentes variantes, menús, patrones de entrada de datos y editor de texto. Pero antes de entrar de lleno al tema convendrá repasar los antecedentes que presentaremos a continuación.

CONCEPTOS PREVIOS AL MANEJO DE VENTANAS

INTRODUCCION

Sabemos que una ventana es una area en la pantalla de la PC utilizada para un propósito específico, y que el uso de ventanas se ha convertido en un método muy popular de presentar la información al usuario de la PC, debido a la facilidad que prestan para soportar multiples tareas asincronas.

De igual manera, sabemos que un sistema con *software* de ventanas puede desplegar cada una de ellas como una presentación de datos individual e independiente. Las ventanas deben ser concebidas de tal manera que no guarden relación unas con otras, ni tengan restricciones de espacio de video. De manera que sea posible tener varias ventanas visibles, aún cuando algunas puedan abarcar gran parte de la pantalla, permitiendo la aparición de una encima de otra.

Y por último, otra caracteristica que queremos enfatizar es que al manejar un ambiente interactivo de ventanas, la acción de remover (*pop-down*) de pantalla alguna de ellas, trae implícito un mecanismo para volver a desplegar en ese espacio, la información que antes de su aparición (*pop-up*) existía. Dicho mecanismo debe prever que el área restablecida puede corresponder a otra ventana o sólo a parte de ella, a varias partes de otras ventanas, o bien a varias partes de ventanas junto con una parte de la RAM original, etc.

El punto al que queremos llegar es que, para el manejo de todas las caracteristicas antes mencionadas, el conocimiento de aspectos relativos a la memoria de video juega un papel fundamental.

Todo despliegue de video es generado en el espacio de memoria RAM, accesada directamente por el procesador mediante una técnica llamada "mapeo de memoria de video". Por ello, antes de tocar otros puntos, iniciaremos este apartado con el estudio de la constitución, operación y técnicas de acceso de la memoria de video de la PC.

■ ARQUITECTURA DE LA MEMORIA DE VIDEO.

En un inicio la tecnología de microcomputadoras estaba basada en terminales de video conectadas a los puertos serie de E/S. Con el tiempo dicha tecnología se fue desarrollando hasta llegar a lo que conocemos ahora como PC, en donde la pantalla de video es una parte integral de la misma. La IBM PC usa una arquitectura de *hardware* que incluye la "electrónica" del video. Los despliegues de pantalla son construidos en base a la memoria de refresco de video; haciendo uso de una porción de la memoria de acceso aleatorio (RAM). La RAM de video puede ser leída y escrita por el procesador y, por ende, accesada por los programas.

Aspectos como localización y características del dispositivo de video RAM, se encuentran estandarizados en toda la línea de productos PC. Es decir, la compatibilidad está garantizada pues todos los *clones* de la PC tienen la misma configuración.

En la vasta línea de PC's podemos encontrar diferentes tipos de arquitectura de video. Las arquitecturas mas comunes son: Adaptador Monocromático (MA), Adaptador de Gráficos Color (CGA), Adaptador de Gráficos Mejorado (EGA) y Arreglo de Gráficos de Video (VGA). Cada una de ellas utiliza un tipo específico de monitor y de tarjeta de video.

- El MA soporta un monitor de despliegue monocromático y sólo es capaz de manejar modo texto.

- El CGA opera un monitor a color y puede soportar modo texto y modo gráficos. En modo texto puede manejar hasta 8 colores para ambos planos (fondo -segundo plano- y frente -primer plano-). Exclusivamente para el primer plano puede manejar 2 niveles de intensidad. En modo gráficos soporta "gráficos" de baja resolución (640 x 200 *pixeles*) en un sólo color.

- El EGA puede ser usado en el mismo modo texto a color como el CGA, con la ventaja de poder *soportar gráficos de colores* múltiples en alta resolución (640 x 350).

- El VGA puede ser visto como una extensión del EGA, con una resolución de 640 x 480 *pixeles*. El VGA puede emular tanto al CGA como al EGA. Sin embargo, la mejora más importante incluida en este tipo de despliegue es la utilización de un monitor analógico en lugar del tradicional monitor digital. Al trabajar con señales analógicas, este nuevo sistema de video puede desplegar paletas de 256 colores de los 262,144 disponibles.

El *software* empleado en esta tesis, trabaja en modo texto sobre cualesquiera de los dispositivos de video antes mencionados. A continuación hablaremos de la organización de la memoria RAM de video, pero antes queremos aclarar que aunque la discusión haga referencia únicamente al dispositivo de video CGA, también es aplicable al EGA y al VGA.

La RAM de video está organizada en un arreglo de caracteres bidimensional (columnas y renglones) que permanece contiguo en memoria. El arreglo está formado por 25 renglones de 80 caracteres. La memoria de video reserva una palabra (16 bits) para cada posición de caracter en la pantalla; el byte menos significativo o bajo contiene el valor ASCII del caracter y byte alto, el atributo de video del mismo.

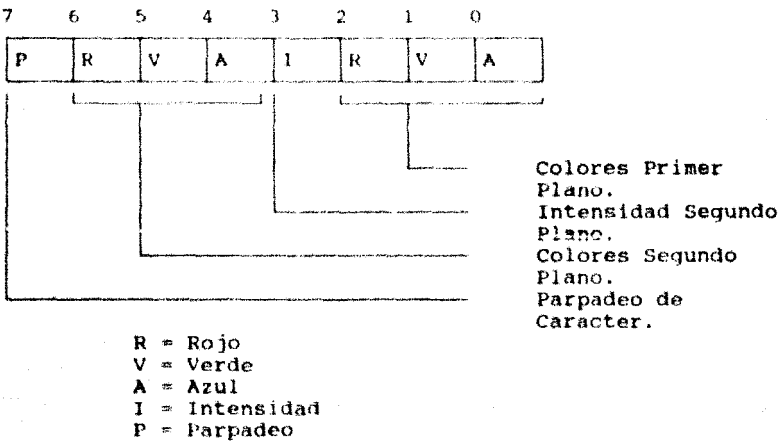
El MA tiene una página RAM de video; el CGA puede tener de 4 a 8 páginas. Para mantener la compatibilidad, nuestro *software* utilizará sólo la primera página de memoria del dispositivo de video en cuestión.

La memoria de video está localizada en la región superior del espacio de direccionamiento del procesador. Esta característica obedece a que los diseñadores de la PC, al tener que enfrentar un máximo de memoria direccionable de 1 Mb., decidieron colocar la RAM de video y la memoria de sólo lectura (ROM) del Sistema Básico de E/S (BIOS) en los límites superiores de este espacio de direccionamiento. Luego, para permitir la coexistencia del MA y el CGA en la misma máquina, los diseñadores asignaron diferentes direcciones de segmento para las 2 memorias de video. La memoria

monocromática principia en el segmento B000H, y la memoria de CGA inicia en el segmento B800H. Cualquier programa puede determinar cuál de los dos tipos de despliegue está en uso corriente, llamando a una función del ROM-BIOS. El programa que hace la llamada puede así ajustarse a la dirección de segmento de la RAM de video apropiada.

Pero retomemos el concepto de representación de caracteres mediante los bytes de atributo y de valor ASCII. Respecto a éste último, todo está claro basta remitirnos a una tabla para conocer los valores ASCII de cada caracter. Respecto al byte de atributo de video, consta de 2 campos de 3 bits para controlar el color, y 2 bits para las características del caracter. Los campos contienen la información para asignar el color al primer y segundo plano del caracter. Los bits de características controlan la intensidad alta/baja para el primer plano, y el parpadeo del caracter al momento del despliegue. La figura 5.1 muestra la configuración del byte de atributo.

FIG. 5.1 EL BYTE DE ATRIBUTO DE VIDEO



Los componentes rojo, verde y *azul* pueden ser combinados para proporcionar 8 colocaciones de color distintas que despliegan los colores blanco, rojo, verde, azul, *magenta*, *cyan*, amarillo y negro. Al añadir el bit de intensidad a los colores del primer plano se generan 8 valores más. El significado de los valores que involucran "color" en el byte de atributo de video, varía "levemente" cuando se usa el MA. Debido el MA no soporta colores, unicamente tendran sentido las combinaciones blanco sobre negro, negro sobre blanco y azul sobre negro (combinación que genera caracteres subrayados), característica no disponible en el CGA. Otras combinaciones pueden producir negro sobre negro o blanco sobre blanco. En cuanto a los valores de intensidad y parpadeo no existe diferencia, el MA soporta los mismos atributos, en este sentido, que el CGA.

■ ALMACENAMIENTO Y DESPLIEGUE DE DATOS DE VIDEO

El sistema de despliegue de la PC está basado en el chip Motorola 6845 Controlador de Tubo de Rayos Catódicos (CRTC). Los sistemas EGA y VGA utilizan chips basados en este diseño. El CRTC permite el manejo de muchas tareas de despliegue importantes, facilitando el trabajo a los programadores; entre estas tareas podemos mencionar:

- Detección de señales de lápiz optico.
- Control del incremento del contador de dirección del *buffer* de video.
- Sincronización despliegue y tiempos (pulsos de reloj).
- Selección del *buffer* de video.
- Determinación del tamaño y posición del cursor

El diseño del sistema de despliegue es conceptualmente simple. El despliegue de la PC es llevado a cabo por un mecanismo de mapeo de memoria, despliegue, que simplemente puede ser concebido como un reflejo de lo que está en la memoria (de video) de la computadora.

Existe un *buffer* de memoria que almacena la información que aparece en el despliegue. La dirección de inicio del *buffer* de memoria y su longitud varían, dependiendo del tipo de despliegue de video en uso, del modo de despliegue corriente y de la cantidad de la memoria asignada para el despliegue.

Los adaptadores de despliegue generalmente, manejan de 4K a 256K de memoria. Sin embargo, los datos requeridos para definir una pantalla de despliegue suelen ocupar un espacio de memoria menos significativo; razón por la cual algunos adaptadores pueden controlar más de una pantalla de despliegue. Notar que decimos pantalla de despliegue y no monitor de despliegue. Las pantallas de despliegue, o páginas, son la representación en memoria de lo que aparece en la pantalla.

El número de páginas de despliegue disponibles para modo texto, en todo tipo de adaptador de video se obtiene como resultado de multiplicar el total de bytes incluidos en una pantalla por 2. Con 80 caracteres de texto por línea, el resultado sería: 2 x 80 x 25 = 4000 bytes, o aproximadamente 4K. Si se usa el adaptador para 40 caracteres de texto por línea (2 x 40 x 25), cada pantalla ocuparía 2000 bytes o aproximadamente un espacio de 2K. Utilizando estos cálculos, podemos fácilmente saber porqué el CGA puede obtener 8 páginas de despliegue de un espacio de *buffer* de 16K.

El tamaño del *buffer* de la tarjeta EGA varía debido a que dicho adaptador puede tener 64K, 128K, o 256K de memoria. Memoria en la que se mantiene la información de las imágenes de pantalla y los patrones (fuentes o tipos de letra) hasta para 1024 caracteres de despliegue. Los cálculos hechos en el párrafo anterior, pueden ayudarnos a determinar el número de páginas de despliegue disponibles.

El EGA, el MCGA, y el VGA tienen 2 direcciones diferentes de inicio de *buffer* de gráficos. Por tanto, para prever la compatibilidad, el programador puede basarse en la dirección del adaptador CGA, pues los adaptadores primeramente mencionados pueden emular a éste último. La dirección de segmento del CGA es B800H y A000H la dirección de inicio de segmento de su *buffer* de gráficos original.

El EGA, el MCGA y el VGA tienen 2 direcciones diferentes de inicio de *buffer* de graficos: la A000H (original) y la B800H. Con esta última, dichos adaptadores pueden emular al CGA.

El chip CRTC, explora el área de memoria de despliegue de forma independiente a la operación del sistema de la computadora, y, basado en la información ahí almacenada, actualiza el despliegue de video. El despliegue de pantalla existente es producido por un haz de electrones que "enciende" o "apaga" pequeños puntos en pantalla (denominados elementos de imagen o *pixeles*) al tiempo que explora cada línea de la pantalla. Dicho haz de electrones traza una trayectoria de izquierda a derecha y de arriba hacia abajo sobre la pantalla completa.

El Problema de la Nieve en Pantalla.

Para proporcionar una imagen estable, se realiza un refresco de memoria de video periódicamente. Esto es, el haz de electrones hace un ciclo completo sobre toda la pantalla a razón de 60 veces por segundo de la manera siguiente: después de haber recorrido una línea completa, el haz se traslada del extremo derecho al extremo izquierdo de la pantalla, quedando al inicio de la línea siguiente. El lapso de tiempo transcurrido durante este traslado es llamado "intervalo de retrazo (volver a trazar) horizontal" o "barrido horizontal" (HRI). Después de haber recorrido todas las líneas, se dice que el haz ha completado un ciclo, y deberá iniciar otro; para ello el rayo de electrones debe ahora, trasladarse desde la posición derecha más baja de la pantalla a la esquina superior izquierda de la misma. Este movimiento es llamado "intervalo de retrazo vertical" (VRI) o "barrido vertical". Obviamente, durante ambos intervalos, el haz es "apagado", lo cual significa que el voltaje del canón de electrones es bajo y por ende el CRT no puede desplegar pixeles en pantalla. Momento en el que se tiene la seguridad de que el controlador del CRT no está accesando a la memoria de video.

Las personas que realizan programas que escriben directamente en la memoria de video deben conocer la manera en la que el adaptador que estén manejando, utiliza la memoria de despliegue, y así poder programar efectivamente los procesos HRI y VRI. La memoria asignada para el despliegue es en realidad una memoria dual especial; al tiempo que el procesador escribe valores en ella, el CRTC los lee. Esta característica precisamente es la que da origen al efecto no deseado conocido como "nieve" en pantalla. O dicho en otras palabras, el efecto de "nieve" surge como resultado del propio diseño de la arquitectura del hardware del dispositivo de despliegue, al permitir que tanto el procesador como el controlador del CRT realicen accesos a la memoria de video al mismo tiempo. Pero analicemos mejor el problema.

Un ejemplo que ilustra el problema del efecto de nieve, se tienen cuando el microprocesador se encuentre modificando un valor en una localidad de memoria determinada, al mismo tiempo que el CRTC está leyendo ese mismo valor, el resultado será: aparicion de una especie de "llovizna" en pantalla (nieve), que provoca distorsion. (Problema de importancia únicamente manejando modo texto).

Para eliminar el efecto de nieve deberemos proceder de la manera siguiente: toda operación de escritura sobre la memoria de video deberá realizarse unicamente durante los periodos HRI y VRI. Porqué? porque estos lapsos de tiempo constituyen "periodos de seguridad" en los que se sabe, el CRTC no se encuentra accesando a la memoria de video, permitiendo al procesador hacerlo sin interferencia. Los procesos HRI y VRI revisten mayor importancia cuando se trabaja con un adaptador CGA. Ya que podría decirse que en otros tipos de adaptadores el problema de "nieve" en pantalla no existe. De esta manera el problema ahora se traduce en identificar los ciclos HRI y VRI.

Para poder determinar la existencia de una condición HRI o VRI, se debe examinar continuamente el registro de estado del CRTC en el puerto de E/S 3DAH. El bit 0, de dicho registro, indica si existe una condicion HRI, y el bit 3 refleja la misma

información pero para la condición VRI. Cuando cualesquiera de los bits anteriores está encendido (colocado en 1) se debe asumir que el intervalo de retrazo respectivo inicia, y cuando está apagado (colocado en 0) debemos interpretar que el ciclo, en cuestión, se ha completado. Por otro lado, debido a que los intervalos HRI ocurren mucho más a menudo y son más fáciles de detectar cuando se esta programando, que los intervalos VRI; la mayoría de las rutinas dirigidas a la memoria de pantalla prueban únicamente la condición HRI. Así, la regla es que mientras el bit HRI permanezca encendido, el programador podrá colocar tantos caracteres como sea posible en la memoria de despliegue, teniendo la seguridad de que no se provocará ningún tipo de interferencia o "nieve" en pantalla.

Como ya se había mencionado, las pautas para evitar la interferencia en pantalla se aplican primeramente al CGA, y únicamente cuando se realice una operación de escritura en la memoria de video. Las condiciones HRI y VRI no requieren ser probadas durante una operación de lectura de la memoria de video, pues dicha operación no provoca interferencia.

Por último, sólo restaría hacer una última aclaración: Si se intentara realizar operaciones de escritura en la memoria de video siguiendo las reglas aquí expuestas basándose en un lenguaje de alto nivel (por ejemplo lenguaje C); resultará que, debido a la limitación de rapidez en la ejecución de dicho lenguaje, el tiempo éste requiere para escribir un caracter en la memoria de video, excederá al intervalo HRI, y por lo tanto la "nieve" persistirá.

La solución a esta última cuestión se basa en la utilización de rutinas en lenguaje ensamblador. Las funciones de este lenguaje pueden esperar la sucesión de los períodos de retrazo y tener suficiente tiempo durante ellos, para escribir un caracter, lo cual es posible debido a que el código en lenguaje ensamblador se ejecuta lo suficientemente rápido para tratar con las estrictas restricciones que el manejo del video implica.

CONFIGURACIONES DE VENTANA

Existen varias clases de ventana que son familiares al usuario de la PC, estas por lo general, entran alguna de las 2 configuraciones de ventanas más populares: ventanas apiladas o ventanas estratificadas. Las primeras pueden ser vistas como un subconjunto de las segundas. Las ventanas estratificadas ofrecen más características que las apiladas, pero estas resultan más eficientes, debido a la mayor rapidez con que ejecutan sus funciones (al desplegar o destruir una ventana), en relación con su contraparte (ventanas estratificadas).

Pero antes de hablar por separado de cada una de estas clasificaciones, anotemos que un programa puede estar ligado a un sólo tipo de configuración.

■ VENTANAS APILADAS.

Este tipo de configuración implica que cualquier operación que se lleve a cabo sobre una ventana (escribir texto, cambiar su color, destruirla, ocultarla, etc) debe ser realizada cuando la ventana este visible completamente (*full view*). Esto significa que dicha ventana no debera permanecer oculta o cubierta, ni siquiera en parte, por otra ventana.

Cuando una ventana apilada es establecida, el *software* que la soporta establecerá un "*buffer*" para almacenar la información de la RAM de video correspondiente a el área de pantalla sobre la que la ventana será desplegada. Despues de haber salvado dicha información en el *buffer* (que de ahora en adelante llamaremos "*buffer*-de-salvado"), la RAM de video será reescrita pero ahora con la información que generará el despliegue de la ventana.

Cualquier operación que el usuario desee hacer sobre la ventana, será realizada directamente sobre la RAM de video. Cuando una ventana es destruida, el contenido de su *buffer*-de-salvado es vuelto a escribir a la RAM de video, restableciendo así la imagen de video a su condición anterior al establecimiento de dicha ventana.

La condición de realizar operaciones sobre una ventana sólo cuando ésta se encuentre visible en su totalidad, en realidad no constituye una gran limitante pues por lógica el usuario siempre efectua operaciones sobre la ventana mas reciente; por ejemplo, resultaría ilógico escribir sobre una ventana que no se está viendo. De todas maneras es importante aclarar este aspecto, independientemente de que lo usual sea direccionar únicamente la última ventana apilada establecida en el sistema.

Las consecuencias de proceder de manera opuesta pueden acarrear errores: Por ejemplo, si se establecen en el sistema las ventanas "A" y "B" (esta última cubriendo parte de la primera ventana), y se procede a escribir texto sobre la ventana A, el resultado será que parte del texto será vertido dentro de la parte de la ventana "B" que coincide con la "A". De igual manera, si se destruyera la ventana "A" antes de haber destruido la "B", parte del *buffer*-de-salvado de la ventana "A" se escribirá encima de la parte coincidente con la ventana "B". Recordemos que cualquier operacion de ventana, dentro de esta configuración, se efectuara sobre al RAM de video, sin importar la existencia de posibles ventanas traslapadas.

Podemos decir en resumen que los aspectos caracteristicos del ambiente de ventanas apiladas son: direccionar la ventana más recientemente abierta y en destruir las ventanas en el orden inverso de su creacion o establecimiento.

La mayoria de los paquetes comerciales que manejan ventanas soportan la configuración de ventanas apiladas, pues las caracteristicas de las aplicaciones que dichos paquetes incluyen, se ajustan para operar exitosamente en este tipo de ambiente; además de resultar conveniente por las ventajas de rapidez en la ejecución.

■ VENTANAS ESTRATIFICADAS.

Este tipo de configuración ofrece mayor flexibilidad, así como algunas caracteristicas adicionales. Entre las cuales sobresale la capacidad de permitir direccionar cualquier operacion a una ventana previamente establecida, sin importar su grado de visibilidad o proximidad con otras ventanas.

Otro aspecto característico, en este tipo de configuración, es la inclusión de nuevas operaciones. Además de abarcar el conjunto usual de operaciones de ventana, las ventanas estratificadas incluyen otras adicionales que permiten: hacer movimientos en un plano bidimensional sobre la pantalla, y adelantar o retrasar ventanas independientemente del orden en que hayan sido creadas. (La palabra "adelantar", para los programadores significa desplegar la ventana en cuestión encima de las existentes en ese momento, para que quede completamente visible al usuario. Y por "retardar" se entiende pasar la ventana en cuestión atrás de todas las establecidas en el sistema). Esta última característica es la que da la impresión de un ambiente tridimensional.

Cuando una ventana estratificada es establecida, al igual que en el otro tipo de configuración, le es asignado un "*buffer*" para almacenar el contenido de video pero, en este caso, la ventana no será desplegada inmediatamente. Para desplegar una ventana se deberá verificar una operación explícita de despliegue. El *buffer*-de-salvado es inicializado con los valores de video que la ventana debería contener si fuera visible; así, cualesquiera operaciones posteriores llevadas a cabo sobre la ventana, mientras continúe no visible, serán llevadas a cabo sobre el *buffer*-de-salvado.

Cuando la operación de despliegue es direccionada a una ventana estratificada, el *software* procederá a intercambiar el contenido de la RAM de video y el del *buffer*-de-salvado de la ventana. Una vez desplegada la ventana, y sólo si se encuentra totalmente visible, todas las operaciones que se direccionen a ella, serán ejecutadas sobre la RAM de video.

Cuando la ventana direccionada no se encuentra totalmente visible - pudiendo estar cubierta total o parcialmente por otra(s) ventana(s) -, el proceso de determinación del área que se debe intercambiar resulta más complejo. Para las partes de la ventana que son visibles, el cambio será realizado directamente en la RAM de video. Pero para las áreas cubiertas por otras

ventanas, el cambio debera realizarse en el *buffer*-de-salvado respectivo a cada una de las ventanas traslapadas sobre la ventana direccionada. El algoritmo para determinar la dirección de memoria (ya sea que corresponda a la RAM de video o a un *buffer*-de-salvado determinado) en donde se efectuara el intercambio, debera inspeccionar el conjunto de ventanas establecidas siguiendo un orden; a partir de la ventana direccionada debera proceder hacia adelante a través de todas las ventanas más recientes, a fin de determinar si el cambio está ocurriendo en una area cubierta por una ventana establecida con posterioridad. Si se encuentra que efectivamente existen ventanas encima de la nuestra, la información de video involucrada será escrita en el *buffer*-de-salvado respectivo a cada una de las ventanas.

El empleo de ventanas apiladas o estratificadas dependerá de los requerimientos del sistema.

ASPECTOS BASICOS PARA EL MANEJO DE UN AMBIENTE DE VENTANAS

CAPITULO 6

INTRODUCCION

El propósito de este capitulo es presentar al programador las herramientas basicas generales de soporte de ventanas en cualquiera de sus tipos o configuraciones. Hemos dividido esta seccion en dos partes generales; en la primera mostraremos el código en lenguaje C de las funciones de bajo nivel que se relacionan con la arquitectura del *hardware* de la PC; y en la segunda, presentaremos una libreria de funciones de ventana. Funciones genericas para soportar el despliegue de todo tipo de información en cualquier tipo de ambiente de usuario.

El conjunto de las funciones en este capitulo incluidas, constituyen el fundamento de las funciones de ventana de menu, editor, etc. que presentaremos más adelante.

Por ultimo, solo resta anadir que para ejemplificar el uso de las funciones de libreria, incluiremos algunos programas muestra. (Toda la información se encuentra ampliamente documentada).

LIBRERIA DE FUNCIONES DE BAJO NIVEL

Las funciones presentadas en este apartado constituyen las operaciones de proposito general de bajo nivel (especificas a la IBM PC y a su *hardware*) para manipular el teclado y el despliegue en pantalla. Estas funciones representan el soporte de las funciones de libreria de alto nivel que veremos más adelante.

LISTADO #1

```
/*    IBMPC.C
 *----------*/
/* Funciones de Bajo-Nivel para direccionar el BIOS y Hardware
 * de la PC */

BLOQUE (1)
#pragma inline
#include <dos.h>
static union REGS rg;

BLOQUE (2)
/* --- posiciona el cursor --- */
void cursor(int x, int y)
{
     rg.x.ax = 0x0200;
     rg.x.bx = 0;
     rg.x.dx = ((y << 8) & 0xff00) + x;
     int86(16, &rg, &rg);
}

BLOQUE (3)
/* --- regresa la posición corriente del cursor --- */
void curr_cursor(int *x, int *y)
{
     rg.x.ax = 0x0300;
     rg.x.bx = 0;
     int86(16, &rg, &rg);
     *x = rg.h.dl;
     *y = rg.h.dh;
}
```

BLOQUE (4)
```
/* --- coloca el tamaño del cursor --- */
void set_cursor_type(int t)
{
     rg.x.ax = 0x0100;
     rg.x.bx = 0;
     rg.x.cx = t;
     int86(16, &rg, &rg);
}

char attrib = 7;
```

BLOQUE (5)
```
/* --- limpia pantalla --- */
void clear_screen()
{
     cursor(0, 0);
     rg.h.al = ' ';
     rg.h.ah = 9;
     rg.x.bx = attrib;
     rg.x.cx = 2000;
     int86(16, &rg, &rg);
}
```

BLOQUE (6)
```
/* --- regresa el modo de video corriente --- */
int vmode()
{
     rg.h.ah = 15;
     int86(16, &rg, &rg);
     return rg.h.al;
}
```

BLOQUE (7)
```
/* --- prueba evento Scroll-Lock --- */
int scroll_lock()
{
     rg.x.ax = 0x0200;
     int86(0x16, &rg, &rg);
     return rg.h.al & 0x10;
}
```

BLOQUE (8)
```
void (*helpfunc)(void);
int helpkey = 0;
int helping = 0;
```

BLOQUE (9)
```
/* --- lee un caracter de teclado --- */
int get_char()
{
     int c;
```

```
    while (1) {
        rg.h.ah = 1;                          (a)
        int86(0x16, &rg, &rg);
        if (rg.x.flags & 0x40)   {            (b)
            int86(0x28, &rg, &rg);            (c)
            continue;
        }
        rg.h.ah = 0;                          (d)
        int86(0x16, &rg, &rg);
        if (rg.h.al == 0)                     (e)
            c = rg.h.ah | 128;
        else
            c = rg.h.al;
        if (c == helpkey && helpfunc) {       (f)
            if (!helping)  {
                helping = 1;
                (*helpfunc)();
                helping = 0;
                continue;
            }
        }
        break;                                (g)
    }
    return c;                                 (h)
}
```

BLOQUE (10)

```
/* --- escribe en la RAM de video un caracter y su atributo ---*/
void vpoke(unsigned vseg, unsigned adr, unsigned chr)
{
    if (vseg == 0xb000)             /* modo monocromático */
        poke(vseg, adr, chr);
    else {
        _DI = adr;              /* offset del caracter de video */
        _ES = vseg;             /* segmento de video */
        asm cld;
        _BX = chr;              /* atributo y caracter */
        _DX = 0x3da;            /* estado del puerto de video */

    /*  espera a que el retrazo de video comience   */
        do
            asm in  al,dx;
        while (_AL & 1);
    /*  espera a que el retrazo de video pare   */
        do
            asm in  al,dx;
        while (!(_AL & 1));
        _AL = _BL;
        asm stosb;              /* almacena caracter */
    /*  espera a que el restrazo de video inicie   */
        do
            asm in  al,dx;
        while (_AL & 1);
```

```
     /*   espera a que el retrazo de video pare   */
          do
               asm in  al,dx;
          while (!( AL & 1));
          _AL = _BH;
          asm stosb;              /* almacena atributo */
     }
}
```

BLOQUE (11)

```
/* --- lee un caracter y su atributo de la RAM de video ---*/
int vpeek(unsigned vseg, unsigned adr)
{
     if (vseg == 0xb000)              /* modo monocromático */
          return peek(vseg, adr);
     asm push ds;
     _DX = 0x3da;                  /* estado del puerto de video */
     _DS = vseg;                   /* dir. de segmento de video */
     _SI = adr;                    /* offset del caracter de video */
     asm cld;
     /*  espera a que el "retrazo" de video inicie  */
     do
          asm in  al,dx;
     while (_AL & 1);
     /*  espera a que el "retrazo" de video pare  */
     do
          asm in  al,dx;
     while (!(_AL & 1));
     asm lodsb;                    /* obtiene caracter */
     _BL = _AL;
     /*  espera a que el "retrazo" de video inicie  */
     do
          asm in  al,dx;
     while (_AL & 1);
     /*  espera a que el "retrazo" de video pare  */
     do
          asm in  al,dx;
     while (!(_AL & 1));
     asm lodsb;                    /* obtiene atributo */
     _BH = _AL;
     _AX = _BX;
     asm pop ds;
     return _AX;
}
```

DOCUMENTACION LISTADO #1

BLOQUE [1]

- La directiva **#pragma inline** le indica a Turbo C que el programa contiene código de ensamble en línea.

- La directiva **#include** instruye al compilador para que durante su proceso incluya otro archivo fuente, que deberá estar delimitado entre comillas o pico-paréntesis. En la mayoría de los casos, el archivo antes mencionado es "stdio.h" que incluye las funciones de la biblioteca estándar.

```
#include "stdio.h"
#include <stdio.h>
```

La directiva **#include** <nombre-archivo> es utilizada cuando se desea buscar el archivo referido, tan sólo en los directorios preestablecidos. El uso de los picoparéntesis en lugar de las comillas instruye al compilador para realizar la búsqueda del archivo en un directorio con información estándar.
Utilizando la directiva **#include** "nombre-archivo" la búsqueda será llevada a cabo primeramente en el directorio donde reside el archivo fuente y luego en una secuencia de directorios preestablecidos.
Nosotros, por ahora sólo haremos uso de las funciones del DOS; de ahí la declaración : **#include** <dos.h>. Las funciones de interfaz con el DOS requieren el encabezado **dos.h**. Este archivo define una estructura conocida con el nombre de "*union*", que corresponde a los registros del procesador 8088/86 y es usada por algunas de las funciones de interfaz con el sistema.

- Instrucción: **static union REGS rg;**
union REGS es una unión definida en el archivo "dos.h" y está formada por dos estructuras que permiten accesar los registros *de hardware*, como palabra o como *byte*.

```
/*--------REGISTROS--------*/
struct WORDREGS
     {
       unsigned int ax, bx, cx, dx, si, di, cflag;
     };
struct BYTEREGS
     {
       unsigned char al, ah, bl, bh, cl, ch, dl, dh;
     };
union REGS {
     struct WORDREGS x;
     struct BYTEREGS h;
          };
```

Nota: En el archivo **dos.h** tambien se incluye la estructura **SREGS**, utilizada por algunas funciones para colocar los registros de segmento.

- Las variables estaticas constituyen un clase particular de almacenamiento (independientemente de las otras ya conocidas: **extern**, **auto**, etc). Las variables estáticas pueden a su vez ser internas o externas. Las variables **static** internas son locales a una funcion, del mismo modo que las automaticas, pero a diferencia de éstas, no son creadas y removidas de memoria al activar la función, su existencia es permanente. Las variables **static** internas proporcionan un medio de almacenamiento permanente y privado en una funcion.
Una variable **static** externa (nuestro caso) puede ser accesada en el resto del archivo fuente en el que está declarada, pero no en otro. Normalmente las funciones son objetos externos: sus nombres son conocidos globalmente. Pero, al declarar una función **static**, su nombre será inaccesible fuera del archivo en que se declare. Las variables y funciones **static** externas son empleadas para ocultar "elementos" y el proceso o rutinas que las manipulan, de modo que otras rutinas y datos no puedan entrar en conflicto con ellas. Usamos el tipo **static** cuando queremos realizar funciones muy generalizadas, funciones de librería que serán usadas por otros programadores.

BLOQUE (2)

- ANALISIS DE LA FUNCION: **void cursor (int x, int y)**
Esta funcion posiciona el cursor de la PC en la localidad de la pantalla especificada por las coordenadas (x,y). Las coordenadas (0,0) corresponden a la esquina superior izquierda. El rango de la coordenada "x" va de 0 a 79, y el de la coordenada "y" de 0 a 24.
void es uno de los tipos de datos basicos de C (además de **char**, **int**, **float**, etc.). El tipo **void** tiene dos usos, el primero: para declarar explicitamente que la función no regresara valor (nuestro caso); el segundo: para crear apuntadores genericos (void * variable;) expresión que significa que utilizaremos un tipo explicito de molde (*cast*) para un apuntador del tipo deseado.

- Notas Previas:
  - Declaracion: **rg.x.ax 0x0200;**
    rg: nombre de la union definida previamente.
    x : indica que accesaremos un registro de palabra.
    ax: es el registro de palabra a ser accesado.

  - Declaracion: **int86(16, &rg, &rg);**
    Su forma general:
    int int86(int num-int, union REGS *registross-entrada,
    union REGS *registros-salida);

El tipo de la declaración (int) se puede omitir. Los prototipos para int86() están en el archivo dos.h.
La función int86() es utilizada para ejecutar una interrupción de *software* especificada por "num-int" (numero de la interrupción).
Primero, el contenido de la unión "registros-entrada" es copiado a los registros del procesador, posteriormente se ejecuta la interrupción solicitada. A su regreso, la unión "registros-salida" contendrá los valores de los registros que el CPU ha devuelto de la interrupción. Si la bandera de Carry está encendida, se debe entender que ocurrió un error. El valor del registro AX es devuelto.
La función int86() es a menudo usada para llamar rutinas del BIOS de la IBM PC.

- CUERPO DE LA FUNCION:

```
AX ← 0200H
BX ← 0H
DX ← ((y << 8) & 0xff00) + x;
```

Se realizan los corrimientos necesarios para dejar en DH la coordenada de renglón (y), y en DL coordenada de columna (x).

```
int86(16, &rg, &rg);
```

Ejecución de la Int 16 (decimal) o Int 10H (hexadecimal), función 02H (AH = 02H). Interrupción del BIOS que coloca la posición del cursor.

Los parámetros de esta interrupción son:

Entrada.-

AH = 02H
BH = número de página
DX = posición del cursor
DH ← renglón
DL ← columna

Salida.- (nada)

BLOQUE (3)

- ANALISIS DE LA FUNCION: **void curr_cursor(int *x, int *y)**

Esta función lee la posición corriente del cursor y escribe las coordenadas (x,y) a las direcciones indicadas por los apuntadores pasados a la función.

- CUERPO DE LA FUNCION:

```
AX ← 0300H
BX ← 0H
```

Ejecución de la Int 10H función 03H. Esta es una interrupción del BIOS que regresa las coordenadas del cursor y el tipo del mismo.
Los parámetros de esta interrupción son:
Entrada.-

AH = 03H
BH = número de página

Salida.-

BH = número de página de video
CX = modo del cursor
CH = linea de inicio del cursor
CL = linea de finalizacion del cursor
DX = posicion del cursor
DH = renglón
DL = columna

Al término de la función, la variable "x" contendrá la coordenada de columna y la variable "y" la coordenada de renglón.

BLOQUE (4)

- ANALISIS DE LA FUNCION: **void set cursor type(int t)**
Esta funcion coloca el tamaño del cursor corriente. Nuestro *software* de soporte de ventanas reconocerá dos tamaños o tipos de cursor. El programa de editor de texto, encontrado en capitulos posteriores, utiliza el tamaño del cursor para indicar la colocación del modo "Insert/Overwrite". El cursor en forma de bloque, indica que el modo de insercion (Insert) está presente. El modo de sobreescritura "overwrite" prevalecerá cuando el cursor adopte forma de "caracter de subrayado".
El tamaño del cursor es especificado por el entero "t". el *byte* alto de este entero indica el inicio de la linea de "exploración" del cursor, y el *byte* bajo, el final de la linea de "exploración". Cuando t = 0x0106 se genera un cursor que ocupa las lineas de "exploración" de la 1 a la 6 de la caja del caracter (creando el cursor en forma de bloque). Cuando t = 0x0607 generara un cursor que sólo ocupará las lineas de exploracion 6 y 7 de la caja del cursor (apareciendo al usuario como un "caracter de subrayado).

- CUERPO DE LA FUNCION:
AX = 0100H
BX = 0H
CX = t (entero con la informacion del tipo de cursor)
Ejecución de la *interrupcion* 10H función 01. Interrupción del BIOS utilizada para colocar la altura del cursor de video, tambien conocida como colocación del tipo del cursor. Los parametros de esta interrupcion son:
Entrada.-
AH = 01H
CH = (*bits* 0-4) inicio (parte alta) de la linea de exploración del cursor.
CL = (*bits* 0-4) fin (parte baja) de la linea de exploración del cursor.
Nota: Para apagar el cursor sólo se debe llamar a esta interrupción con: CX = 2000H.
Salida.- (nada)

BLOQUE (5)

- ANALISIS DE LA FUNCION: **void clear_screen()**
  Esta función limpia la pantalla y posiciona el cursor en al esquina superior izquierda. La memoria de video es llenada de caracteres ASCII "blancos", el atributo de video para estos caracteres es tomado de la variable tipo caracter declarada como: char **attrib** = 7; que corresponde al atributo "normal" de video. Con este valor se limpia la pantalla dejándola en fondo negro con caracteres en blanco.
  Si se desea un atributo diferente, se deberá cambiar el valor de dicha variable antes de llamar a la función clear screen.

- CUERPO DE LA FUNCION:
  Primero hace un llamado a la función **cursor** con el siguiente parámetro: cursor(0,0); que posiciona el cursor en la esquina superior izquierda.
  AL = ' '
  AH = 09H
  BX = attrib
  CX = 2000 (80 cols x 25 ren = 2000)
  Ejecucion de la interrupción 10H función 9H, que almacena en la posición del cursor un numero especifico de caracteres ASCII con un atributo definido.
  Los parámetros de esta interrupción son:
  Entrada.-
  AH = 09H
  AL = caracter ASCII
  BH = página de despliegue
  BL = byte de atributo del caracter contenido en AL
  CX = el numero de caracteres a escribir

  Salida.- (nada)

BLOQUE (6)

- ANALISIS DE LA FUNCION: **int vmode()**
  Esta función regresa el modo de video corriente y es utilizada primordialmente para determinar si el *software* deberá direccionar memoria de video monocromática o memoria en modo texto del CGA, EGA o VGA.
  Esta función regresa un valor de 7 si la PC está operando en modo monocromatico. Cualquier otro valor indica la operación en modo texto, del CGA, EGA o VGA.

- CUERPO DE LA FUNCION:
  AH = 15
  Ejecución de la interrupcion 10H funcion 0FH, que regresa el modo de despliegue de video, el ancho de la pantalla y la página activa de video.

Los parámetros de esta interrupción son:
Entrada.-
AH = 0FH
Salida.-
AH = número de columnas en la pantalla
AL = modo de despliegue de video (texto o gráficos, y tipo de adaptador)
BH = página de despliegue activa

Al término de la función se regresará el modo de video.

BLOQUE (7)

- ANALISIS DE LA FUNCION: int scroll_lock ()
Verifica si está activa la condición scroll-lock.

- CUERPO DE LA FUNCION:
AX = 0200H
Ejecución de la interrupción 16H función 02H, que regresa un *byte* de estado indicando la condición de las teclas de función, también conocidas como teclas "*Shift*" o extendidas.
Los parametros de esta interrupción son:
Entrada.-
AH = 02H
Salida.-
AL = Byte de Banderas de teclado de ROM BIOS.

Bit	Tecla que representa
0	Shift derecho
1	Shift izquierdo
2	Ctrl
3	Alt
4	Scroll Lock
5	Num Lock
6	Caps Lock
7	Insert

Al término de la función se prueba el *bit* 4 del registro AL quedando finalmente en éste, el estado de la condición Scroll Lock.

BLOQUE (8)

- Declaración: void (*helpfunc)(void);
helpfunc es un apuntador a una función de tipo void (que no devolverá valor). Esta función será documentada en la sección de ventanas de ayuda de contexto sensitivo. Las variables **helpkey** y **helping** son inicializadas con un valor nulo, indicando que no se han solicitado ventanas de ayuda.

BLOQUE (9)

- ANALISIS DE LA FUNCION: **int get_char()**

Esta función es clave para el buen funcionamiento de nuestros programas TSR; sirve a diversos propósitos críticos. Su primer fin es leer un caracter del teclado haciendo uso de los servicios de teclado del ROM BIOS de la PC; proporcionando una entrada de teclado de un sólo caracter (sin repeticiones); mientras se evitan llamadas a funciones del DOS. La función get_char involucra las siguientes tareas:

- **Mientras el sistema esté esperando un pulso de tecla del usuario, la función get_char genera interrupciones de *software* al vector 28H (interrupción DOSOR). Lo concerniente a esta interrupción y su importancia en una utilería residente en memoria, fue ya explicado en el capitulo II.**
- **Cuando se presiona una tecla de función, el ROM BIOS de la PC regresa lo que podría concebirse como 2 pulsos de tecla. El primero contiene un valor nulo indicando, precisamente, que el pulso detectado corresponde a un tecla de función. Y el segundo contiene un código ASCII de 7 *bits*, valor unico para cada tecla extendida. Si se ignorara el caracter nulo, las teclas de función podrían confundirse con las teclas de caracteres ASCII válidos (letras, numeros, etc).**
- **La función get_char traduce las dos secuencias de teclas del ROM BIOS en valores de 8 *bits*. Es decir los dos *bytes* regresados al detectar la tecla extendida, son reducidos a un solo *byte*, simplemente agregando un "1" al *bit* más significativo del *byte* que contiene el código ASCII de 7 *bits*. Los valores resultantes, no podrán confundirse con el correspondiente a las teclas no extendidas. Estos valores están definidos como símbolos globales en el archivo fuente "keys.h".**
- **Otra tarea llevada a cabo por la función get_char consiste en verificar la activación de la función de ayuda (help) designada. El valor de la tecla de función de ayuda está definido en una variable entera global llamada helpkey. Esta variable es inicializada con un valor nulo, pero el *software* que procesa las ventanas de ayuda la colocará al valor de la tecla de función correspondiente. Cuando se presione la tecla de función de ayuda, la función get_char hará un llamado a la función de ayuda solicitada, mandando el control a la dirección señalada por el apuntador a función global helpfunc (siempre y cuando éste no tenga un valor nulo).**

CUERPO DE LA FUNCION:

(a) Ejecución de la interrupción 16H función 01H. Verifica la existencia de un pulso de tecla disponible en el buffer de teclado; si es encontrado, regresa el código ASCII y el código de exploración del caracter sensado.

Los parametros de esta interrupción son:
Entrada.-

AH = 01H

Salida.-

Bandera de Cero = 0 (pulso de tecla disp.)
AH = código de exploración
AL = caracter ASCII
Bandera de Cero = 1 (pulso de tecla no disp.)

(b) La operación "AND" a nivel de *bit* efectuada sobre el registro de banderas y el valor 40H, es realizada con el fin de conservar el estado del *bit* 6 que corresponde a la Bandera de Cero.
Respecto a la condicion "if": si el resultado es "1", implica que no hay caracter disponible en el buffer de teclado, caso en el que se ejecuta la interrupción 28H (DOSOK).

(c) Ejecución de la interrupción 28H. Esta interrupción se encuentra constantemente verificando el estado de la consola de E/S del DOS, para permitir a los programas TSR determinar si el sistema se encuentra en un estado de "seguridad" para poder utilizar operaciones de archivo y otras funciones de la interrupción 21H superiores a la 0CH.
La proposición **continue** forza la ejecución de una nueva iteración de la instruccion de repetición (for, while, do) que la anteceda. En los casos "while" y "do"; la parte de comprobación de la condición se ejecuta inmediatamente; en el caso "for", el control es transferido primero a la parte de incremento del ciclo, y posteriormente a la parte de prueba de la condición.

(d) A esta parte se entra cuando la Bandera de Cero contiene un valor nulo, indicando la existencia de un caracter disponible en el buffer de teclado listo a ser leído.
Ejecución de la interrupción 16H función 0H; que lee un caracter del buffer de teclado
Los argumentos de esta interrupción son:
Entrada.-

AH = 00H

Salida.-

AH = código de exploracion de la tecla o código extendido si AL = 0
AL = código ASCII del caracter o cero si es un caracter extendido.

(e) Si AL = 0, Se coloca un "1" en el *bit* más significativo de AH. Esto para traducir las *2* secuencias de teclas del ROM BIOS en un solo *byte* (como ya fue explicado). El resultado es almacenado en la variable "c".
Si AL <> 0, el caracter ASCII viene contenido en el registro AL y es almacenado directamente en la variable "c".

(f) Si la tecla de ayuda no ha sido presionada y el apuntador a la función de ayuda es nulo, el ciclo se da por terminado (*g*).

Si la tecla de ayuda ha sido presionada y existe apuntador a una funcion de ayuda: se verifica primero que *no exista alguna función de ayuda activada en ese momento* (helping = 0). Si es asi, se coloca la bandera de ayuda en "1" indicando la próxima ocurrencia de una función de ayuda. A continuación se ejecuta la función de ayuda requerida y se apaga la bandera; para volver a verificar la existencia de caracteres en el buffer de teclado (a).

(g) La proposicion break: origina la terminación del while, do, for o switch mas cercano. El control será pasado al inciso (h)

(h) Proposicion return: Mediante esta instrucción, las funciones regresan el control al punto en el que fueron llamadas. La proposicion return puede tener 2 formas: "return;" o "return (expresión)", cuando la función en cuestión devolvera algún valor. La expresión devuleta por una función puede ser vista como una asignación al tipo de la función en la que aparece.
Cuando el control alcanza el fin de una función se produce un "return" implicito sin regreso de valores. En nuestro caso se regresara el valor almacenado en la variable "c".

BLOQUE (10)

- ANALISIS DE LA FUNCION:

void vpoke(unsigned vseg, unsigned adr, unsigned chr)

Esta función escribe un caracter y su atributo en la RAM de video. Para comprender la manera en la que esta función es usada debemos conocer la naturaleza de la memoria de video de la PC y el mecanismo del "retrazo" de video (conceptos explicados en el capitulo anterior). Esta función utiliza código de ensamble en linea. El codigo de ensamble en línea es requerido unicamente para los programas que serán corridos sobre sistemas que utilicen la tarjeta CGA de IBM o equivalente. Esto debido a que solo el adaptador CGA *presenta el problema de "nieve" en pantalla.*
Esta función no requiere mas documentacion que la presentada en el listado.
Parametros de entrada:

vseg = segmento de video (0xb800 para CGA, 0xb000 para MA)

adr = offset al caracter de video desde el inicio del segmento de video: 0 para el primer caracter, 2 para el segundo, y asi sucesivamente.

chr = en el *byte* más significativo contiene el atributo de video, y en el menos significativo el caracter ASCII.

BLOQUE (11)

- ANALISIS DE LA FUNCION:

  int vpeek(unsigned vseg, unsigned adr)

  Esta función regresa el caracter de video y su atributo localizados en vseg:adr (dirección del segmento de video y offset del caracter). La descripción de la función anterior se aplica también a ésta. Esta función no requiere más documentación que la presentada en el listado.

LISTADO #2

```
/*  keys.h
 *------------*/
#define HT            9
#define RUBOUT        8
#define BELL          7
#define ESC          27
#define SHIFT_HT    143
#define CTRL_T       20
#define CTRL_B        2
#define CTRL_D        4
#define ALT_D       160

#define F1          187
#define F2          188
#define F3          189
#define F4          190
#define F5          191
#define F6          192
#define F7          193
#define F8          194
#define F9          195
#define F10         196

#define HOME        199
#define UP          200
#define PGUP        201
#define BS          203
#define FWD         205
#define END         207
#define DN          208
#define PGDN        209
#define INS         210
#define DEL         211
#define CTRL_HOME   247
#define CTRL_BS     243
#define CTRL_FWD    244
#define CTRL_END    245
```

DOCUMENTACION LISTADO #2

- Este listado es un archivo de cabecera incluido en muchos de los programas que presentaremos a continuación. El archivo contiene definiciones globales de valores de teclas de función regresados por la funcion **get char**.

- Declaración **#define**
Permite definir nombres o constantes simbolicas, al comienzo de un programa. Posteriormente el compilador reemplazara todas las apariciones "no entrecomilladas" del nombre por su cadena correspondiente. En el listado #2, las cadenas están compuestas por numeros; sin embargo, se puede utilizar cualquier texto como substituto del nombre.

LIBRERIA DE FUNCIONES DE VENTANA

El *software* presentado en este apartado incluye funciones de libreria que soportan un amplio rango de operaciones de ventanas de video. Las funciones estan divididas en subsistemas orientados a aplicaciones especificas: menus, ayudas de contexto sensitivo, y edición de texto. La libreria de ventanas de proposito general podrá ser empleada por los programas de aplicacion asi como por los subsistemas mismos.

En este apartado presentaremos 2 listados (**twindow.h** y **twindow.c**) que incluyen los archivos fuente de ventana. Cada listado es seguido por la documentación de su código correspondiente.

LISTADO #3

El siguiente listado define las estructuras de ventana y contiene los prototipos de las funciones de ventana. Deberá ser incluido en cualquier programa fuente que utilice las funciones de ventana.

```
/*    twindow.h
 * -------------*/
/*    Nota: La declaracion :  #define FASTWINDOWS.
 *    Se debe quitar de los comentarios para que el compilador de
 *    Turbo C la incluya; cuando se utilicen ventanas apiladas.
 *
 *    #define FASTWINDOWS
 */

/* --- colores de ventana --- */
#undef WHITE
#undef YELLOW
#undef MAGENTA
#define RED     4
#define GREEN   2
#define BLUE    1
#define WHITE   (RED+GREEN+BLUE)
#define YELLOW  (RED+GREEN)
#define AQUA    (GREEN+BLUE)
#define MAGENTA (RED+BLUE)
```

```
#define BLACK 0
#define BRIGHT 8
#define DIM 0
#define BORDER 0
#define TITLE 1
#define ACCENT 2
#define NORMAL 3
#define ALL 4
#define TRUE 1
#define FALSE 0
#define ERROR -1
#define OK 0

BLOQUE (1)
/* --- estructuras de control de ventana --- */
typedef struct  wnd {
     int  wv;        /* var. para indicar ventana visible */
     int  hd;        /* var. para indicar ventana oculta */
     char * ws;      /* apuntador al Bloque-de-salvado */
     char *_tl;      /* apuntador al Titulo de ventana */
     int  wx;        /* coordenada "x" sup. izquierda */
     int  wy;        /* coordenada "y" sup. izquierda */
     int  ww;        /* ancho de la ventana */
     int  wh;        /* altura de la ventana */
     int  wsp;       /* apuntador de scroll de ventana */
     int  sp;        /* apuntador de selección */
     int  cr;        /* posición "x" del cursor */
     int btype;      /* tipo de contorno de ventana */
     int wcolor[4]; /* colores de ventana */
     int  pn;        /* color normal anterior */
     struct  wnd * nx; /* apuntador a la sig. ventana */
     struct  wnd * pv; /* apuntador a la ventana previa */
} WINDOW;

BLOQUE (2)
typedef struct w_menu {
     char *mname;
     char **mselcs;
     void (**func)(int, int);
} MENU;

BLOQUE (3)
#define SAV     (wnd-> ws)
#define WTITLE  (wnd-> tl)
#define COL     (wnd-> wx)
#define ROW     (wnd-> wy)
#define WIDTH   (wnd-> ww)
#define HEIGHT  (wnd-> wh)
#define SCROLL  (wnd-> wsp)
#define SELECT  (wnd-> sp)
#define WCURS   (wnd-> cr)
#define WBORDER      (wnd->wcolor[BORDER])
#define WTITLEC      (wnd->wcolor[TITLE])
#define WACCENT      (wnd->wcolor[ACCENT])
```

```
#define WNORMAL      (wnd->wcolor[NORMAL])
#define PNORMAL      (wnd->_pn)
#define BTYPE        (wnd->btype)
#define NEXT         (wnd->_nx)
#define PREV         (wnd->_pv)
#define WCOLOR       (wnd->wcolor)
#define VISIBLE      (wnd->_wv)
#define HIDDEN       (wnd->_hd)



BLOQUE (4)
#define NW      (wcs[wnd->btype].nw)
#define NE      (wcs[wnd->btype].ne)
#define SE      (wcs[wnd->btype].se)
#define SW      (wcs[wnd->btype].sw)
#define SIDE    (wcs[wnd->btype].side)
#define LINE    (wcs[wnd->btype].line)


/* ======== Prototipos de Funciones y Macros ======== */


BLOQUE (5)
/* --- funciones de proposito general y macros --- */
void clear_screen(void);
int vmode(void);
void cursor(int, int);
void curr_cursor(int *, int *);
int cursor_type(void);
void set_cursor_type(int);
int get_char(void);
int scroll_lock(void);
void vpoke(unsigned, unsigned, unsigned);
int vpeek(unsigned, unsigned);

BLOQUE (6)
/* --- funciones de ventana y macros --- */
WINDOW *establish_window(int, int, int, int);
void set_border(WINDOW *, int);
void set_colors(WINDOW *, int, int, int, int);
void set_intensity(WINDOW *, int);
void set_title(WINDOW *, char *);
void display_window(WINDOW *);
void delete_window(WINDOW *);
void clear_window(WINDOW *);
void hide_window(WINDOW *);
void wprintf(WINDOW *, char *, ...);
void wputchar(WINDOW *, int);
void close_all(void);
void wcursor(WINDOW *, int x, int y);
void error_message(char *);
void clear_message(void);
int get_selection(WINDOW *, int, char *);
```

BLOQUE (7)

```
#define reverse_video(wnd) wnd->wcolor[3]=wnd->wcolor[2]
#define normal_video(wnd) wnd->wcolor[3]=wnd->_pn
#define rmove_window(wnd,x,y) repos_wnd(wnd, x, y, 0)
#define move_window(wnd,x,y) repos_wnd(wnd, COL-x, ROW-y, 0)
#define forefront(wnd) repos_wnd(wnd, 0, 0, 1)
#define rear_window(wnd) repos_wnd(wnd, 0, 0, -1)
```

BLOQUE (8)

```
/* --- funciones internas para el proceso de ventanas --- */
void accent(WINDOW *);
void deaccent(WINDOW *);
void scroll(WINDOW *, int);
void repos_wnd(WINDOW *, int, int, int);
void acline(WINDOW *, int);
void remove_delist(WINDOW *);
#define accent(wnd) acline(wnd, WACCENT)
#define deaccent(wnd) acline(wnd, WNORMAL)
#define clr(bg,fg,in) ((fg)|(bg<<4)|(in))
#define vad(x,y) ((y)*lnb+(x)*2)
#ifdef FASTWINDOWS
#define cht(ch,at) (((ch)&255)|((at)<<8))
#define displ(w,x,y,c,a) vpoke(VSG,vad(x+COL,y+ROW),cht(c,a))
#define dget(w,x,y) vpeek(VSG,vad(x+COL,y+ROW))
#define verify_wnd(w) (*(w)=listtail)!=NULL
#else
void displ(WINDOW *wnd, int x, int y, int ch, int at);
#endif
/* ------ funcion de editor ------- */
void text_editor(WINDOW *, char *, unsigned);
/* ------ función de menu ------ */
void menu_select(char *name, MENU *mn);
/* ----- funciones de ayuda ------- */
void load_help(char *);
void set_help(char *, int, int);
```

DOCUMENTACION LISTADO #3

En general:

El proposito de la variable global FASTWINDOWS es seleccionar la configuracion de ventanas apiladas por lo que, cuando este tipo de configuracion sea requerido, simplemente se deberá excluir dicha variable de los comentarios. En principio, la libreria presentada será compilada para la configuración de ventanas estratificadas.

La estructura WINDOW contiene todos los elementos necesarios para describir una ventana en el sistema. A cada ventana le es asignada una estructura de este tipo.

La estructura MENU, como su nombre lo indica, será usada para incluir opciones de menu de ventanas. Existirá un arreglo de estructuras MENU con una entrada para cada menu *pop-down*. Los aspectos concernientes a este punto serán vistos en un capítulo posterior dedicado al estudio de las ventanas de menú.

La lista de declaraciones **/define** es usada para hacer más legible el código en **twindow.c**. Los nombres de mnemónicos son igualados a los miembros de la estructura WINDOW cuya dirección es mantenida por el apuntador **wnd**. Por convención, dicho nombre de apuntador será reconocido por todas las funciones en el archivo **twindow.c**

BLOQUE (1)

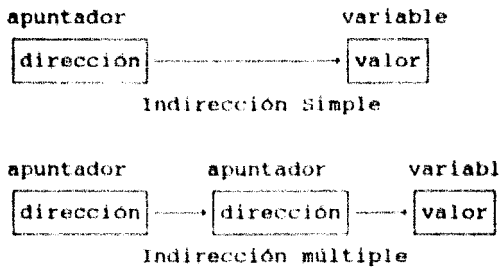
- Declaración: **typedef**
  Turbo C permite definir nuevos nombres de tipo de datos utilizando la palabra reservada **typedef**. Por ello no se debe entender que se permite crear una nueva clase de datos; mediante la declaración typedef se define un nuevo nombre para un tipo de datos existente. Esto contribuye a la portabilidad de programas que dependan del *hardware*. El segundo propósito de **typedef** es ayudar a documentar el código, permitiendo nombres descriptivos para los tipos de datos estándar.
  Su forma general es: **typedef** "tipo" "nombre";
  Para nuestro caso WINDOW será el nuevo nombre para referenciar una estructura del tipo "struct _wnd".

- Declaración: **struct**
  Una estructura en Turbo C es un conjunto de una o más variables, posiblemente de tipos diferentes, agrupadas bajo un mismo nombre para hacer más eficiente el manejo en su conjunto. Los elementos o variables citados en una estructura se denominan miembros. La palabra clave **struct** introduce la declaración de una estructura, que no es más que una lista de declaraciones encerradas entre llaves. Opcionalmente puede seguir un nombre a la palabra clave **struct**; se le llama nombre de la estructura y se puede emplear en declaraciones posteriores.

BLOQUE (2)

- Declaración: **char **mselcs;**
  Se trata de un apuntador a apuntador, o bien de un arreglo de apuntadores. Manejar el concepto de "arreglos de apuntadores" suele ser mas sencillo, debido a que tal como un arreglo simple es concebido, los índice mantienen un significado claro. El concepto de apuntadores a apuntadores puede ser confuso. Un apuntador a apuntador es una forma de "indirección" múltiple o una cadena de apuntadores. Ver figura 6.1.

FIG. 6.1 FORMAS DE INDIRECCIONAMIENTO



En el caso de un apuntador normal, el contenido del apuntador es la dirección de la variable con el valor buscado. En el caso de un apuntador a apuntador, el primer apuntador contiene la dirección de un segundo que apunta a la variable que contiene el valor deseado.

- Declaración: void (**func) ();
la variable func, es un apuntador a apuntador a una función que no regresa valor.

BLOQUE (3)

- Declaracion: #define SAV (wnd->_ws)
La variable SAV tendrá la dirección del primer elemento del arreglo "_ws" correspondiente al bloque-de-salvado de la ventana.

- Declaración: #define WBORDER (wnd->wcolor[BORDER])
El contenido de la variable WBORDER es un elemento que especifica el tipo de contorno de la ventana, elemento que pertenece al vector wcolor, y que se encuentra inicializado con "0".

BLOQUE (4)

- Declaraciones: #define NW (wcs[wnd->btype].nw)
#define NE (wcs[wnd->btype].ne)
etc.
la variable wcs representa un arreglo de estructuras definido en el listado #4 twindow.c. Este arreglo mantiene el conjunto de caracteres de borde de ventana. A continuación, se muestra una parte de su código de definición, con el objeto de simplificar la explicación en este punto.

```
        struct {
             int nw, ne, se, sw, side, line;
        } wcs[ ] = {
    wcs[0]    {218, 191, 217, 192, 179, 196 },
    wcs[1]    {                .              },
    wcs[2]                     .
    wcs[3]                     .
    wcs[4]    {                .              }
        };
```

En la estructura se definen ciertas variables enteras: nw, ne,... line. Posteriormente se declara un arreglo de 5 conjuntos de este tipo de estructura, cada una de las cuales define los códigos ASCII correspondientes a las partes laterales, fondo y tope del marco de la ventana. A esto se le llama arreglo de estructuras.
wnd→btype indicará el conjunto que estamos referenciando. btype puede tomar los valores 0, 1, 2, 3, 4 para los diferentes tipos de borde.

- Los operadores "." y "→" son usados para referenciar elementos individuales de estructuras y uniones.
Las estructuras y uniones son tipos de datos compuestos que pueden ser referenciados bajo un nombre único.
El operador "." es utilizado cuando la estructura o unión es global o cuando el código que se está referenciando está en la misma función donde se encuentra la declaración de la unión o la estructura.
El operador "→" hace referencia a un apuntador a una estructura o unión. En nuestro caso, **wnd** es el apuntador a una estructura WINDOW.

BLOQUE (5)

- En este bloque se declaran las funciones de propósito general (con sus respectivos tipos de parámetros) del listado fuente #1 ibmpc.c.

BLOQUE (6)

- En este bloque se declaran las funciones de ventana con sus respectivos tipos de parámetros del listado fuente #4 twindow.c

BLOQUE (7)

- Declaración:
#define reverse_video(wnd) wnd→wcolor[3] = wnd→wcolor[2]
La macro reverse_video(), será remplazada por la expresión wcolor[3] = wcolor[2].

Originalmente tenemos que wcolor[3] = NORMAL y wcolor[2] = ACCENT. Una vez invocada esta macro, todas las llamadas a las funciones wprintf y wputchar darán por resultado despliegues en colores ACCENT en lugar de usar el default NORMAL.

- Declaración:
#define normal_video(wnd) wnd->wcolor[3] = wnd->_pn
Una vez invocada esta macro, todas las llamadas a las funciones wprintf y wputchar darán por resultado "despliegues" en colores NORMAL. Esta función es llamada para anular el efecto de la función reverse video.

- Declaración:
#define rmove_window(wnd, x, y) repos_wnd(wnd, x, y, 0)
A partir de la macro **rmove_window** todas las subsiguientes macros definidas en este bloque, serán empleadas sólo en el caso de manejar ventanas estratificadas.
Cuando se llame a la macro rmove_window, se ejecutará la función repos_wnd con los parámetros indicados. Esta función desplaza una ventana en un plano, según el resultado de la suma entre los valores de los parámetros (x,y) y los de las coordenadas de la esquina superior izquierda de la ventana.

- Declaración:
#define move_window(wnd,x,y) repos_wnd(wnd, COL-x, ROW-y, 0)
Esta macro mueve una ventana de modo que su esquina superior izquierda es posicionada en las coordenadas especificadas por los parámetros (x,y).

- Declaración: **#define forefront(wnd) repos_wnd(wnd, 0, 0, 1)**
Esta macro traslada una ventana a la posición más reciente respecto a las demás ventanas establecidas en el sistema. Si la ventana se encuentra visible, es desplegada encima de todas las existentes en pantalla.

- Declaración:
#define rear_window(wnd) repos_wnd(wnd, 0, 0, -1)
Esta macro traslada una ventana a la posición menos reciente respecto a las demás ventanas establecidas en el sistema. Si la ventana se encuentra visible, será desplegada abajo de todas las existentes en pantalla.

BLOQUE (8)

- Declaración: **#define clr(bg, fg, in) ((fg)|(bg<<4)|(in))**
Define una macro donde: bg = color de fondo (o del segundo plano), fg = color del primer plano e in = intensidad.
Recordemos, del capítulo anterior, que la información concerniente a los colores del primer y segundo plano se encuentra en el *byte* de atributo de video. Por lo que, la información mantenida en las variables antes mencionadas debe ser combinada en sólo *byte*, que se ajuste a la

estructura del *byte* de atributo de video. Esta tarea es llevada a cabo en la segunda expresión de la instrucción **#define**. Se realiza una "OR" entre los valores de color para el primer y segundo plano e intensidad para que la combinación quede registrada en el *byte* de atributo. Para ello previamente se colocan los valores de color para cada plano y de intensidad, en las posiciones del *byte* de atributo correspondientes. De ahí que el color para el primer plano sea colocado directamente, y que se haya efectuado un corrimiento de 4 lugares a la izquierda para la colocación del color de fondo. En cuanto a la variable de intensidad, corresponderá el valor de "1" cuando se desee "acentuar" el brillo del primer plano.

- Declaración: **#ifdef FASTWINDOWS**

Las directivas **#ifdef** y **#ifndef**, constituyen un método de compilación condicional. Su significado: "si está definido" y "si no fue definido", respectivamente.
La forma general de **#ifdef** es:

```
#ifdef  nombre-de-macro
     secuencia de instrucciones
#else (opcional)
     secuencia de instrucciones
#endif
```

Se compilará la secuencia de instrucciones entre **#ifdef** y **#else** sólo si el nombre-de-macro hubo sido previamente definido en una declaración **#define**.
Para la directiva **#ifndef**, la forma general es la misma; pero a diferencia de la anterior, provocará la compilación de la secuencia de instrucciones correspondientes a su parte afirmativa cuando el nombre de la macro no haya sido definido previamente en una declaración **#define**.
Aplicando esto a nuestro caso: si la variable FASTWINDOWS se encuentra declarada, se realizará la compilación de las macros requeridas para la configuración de ventanas apiladas; en caso contrario se procederá a compilar el cuerpo del *#else*, que contempla el caso de ventanas estratificadas.

LISTADO #4

El siguiente listado contiene las funciones básicas para operaciones con ventana. Con estas funciones se podrá establecer una ventana, especificar su localización y tamaño, colocar colores, contornos y titulos; escribir texto en ella y trasladarla de una localidad a otra. Finalmente, también se podrá quitar una ventana de pantalla, de modo que la imagen previa de pantalla aparezca como estaba antes de que la ventana fuera establecida. Todas estas operaciones constituyen la base para el desarrollo de programas que manejen ventanas de video vistosas y atractivas al usuario.

```
/*    twindow.c
 *--------------*/

BLOQUE (1)
#include <stdio.h>
#include <ctype.h>
#include <stdarg.h>
#include <dos.h>
#include <alloc.h>
#include <stdlib.h>
#include <string.h>
#include "twindow.h"
#include "keys.h"
#define TABS 4
#define SCRNHT 25
#define SCRNWIDTH 80
#define ON  1
#define OFF 0
#define ERROR -1

/* --- prototipos locales ---- */
static void redraw(WINDOW *wnd);
static void wframe(WINDOW *wnd);
static void dtitle(WINDOW *wnd);
static int *waddr(WINDOW *wnd, int x, int y);
static void vsave(WINDOW *wnd);
static void vrstr(WINDOW *wnd);
static void add_list(WINDOW *wnd);
static void beg_list(WINDOW *wnd);
static void remove_list(WINDOW *wnd);
static void insert_list(WINDOW *w1, WINDOW *w2);
#ifndef FASTWINDOWS
static int verify_wnd(WINDOW **w1);
static int dget(WINDOW *wnd, int x, int y);
#endif
```

```
/* --- arreglo de caracteres de contorno --- */
struct {
      int nw, ne, se, sw, side, line;
} wcs[] = {
      {218,191,217,192,179,196},      /* linea sencilla */
      {201,187,188,200,186,205},      /* linea doble */
      {214,183,189,211,186,196},      /* lados doble, tope sen. */
      {213,184,190,212,179,205},      /* lados sen., tope doble */
      {194,194,217,192,179,196}       /* menù pop-down */
};

/* - Cabeza y cola de lista ligada de estructuras de ventana - */
WINDOW *listhead = NULL;
WINDOW *listtail = NULL;
int VSG;  /* direccion de segmento de video */
```

BLOQUE (2)

```
/* --- establecimiento de una ventana --- */
WINDOW *establish_window(x, y, h, w)
{
      WINDOW *wnd;
      VSG = (vmode() == 7 ? 0xb000 : 0xb800);
      if ((wnd = (WINDOW *) malloc(sizeof (WINDOW))) == NULL)
            return NULL;
      /*    ajuste de parametros fuera de limite    */
      WTITLE = "";
      HEIGHT = min(h, SCRNHT);
      WIDTH = min(w, SCRNWIDTH);
      COL = max(0, min(x, SCRNWIDTH-WIDTH));
      ROW = max(0, min(y, SCRNHT-HEIGHT));
      WCURS = 0;
      SCROLL = 0;
      SELECT = 1;
      BTYPE = 0;
      VISIBLE = HIDDEN = 0;
      PREV = NEXT = NULL;
      WBORDER=WNORMAL=PNORMAL=WTITLEC =
                         clr(BLACK, WHITE, BRIGHT);
      WACCENT = clr(WHITE, BLACK, DIM);
      if ((SAV = malloc(WIDTH * HEIGHT * 2)) == (char *) 0)
            return NULL;
      add_list(wnd);
#ifndef FASTWINDOWS
      clear_window(wnd);
      wframe(wnd);
#endif
      return wnd;
}
```

BLOQUE (3)

```
/* --- establece el contorno de ventana --- */
void set_border(WINDOW *wnd, int btype)
{
     if (verify_wnd(&wnd))     {
          BTYPE = btype;
          redraw(wnd);
     }
}
```

BLOQUE (4)

```
/* --- establece colores de ventana --- */
void set_colors(WINDOW *wnd,int area,int bg,int fg,int inten)
{
     if (vmode() == 7)    {
          if (bg != WHITE && bg != BLACK)
               return;
          if (fg != WHITE && fg != BLACK)
               return;
     }
     if (verify_wnd(&wnd))     {
          if (area == ALL)
               while (area)
                    WCOLOR [--area] = clr(bg, fg, inten);
          else
               WCOLOR [area] = clr(bg, fg, inten);
          redraw(wnd);
     }
}
```

BLOQUE (5)

```
/* ----- establece la intensidad de la ventana ----- */
void set_intensity(WINDOW *wnd, int inten)
{
     int area = ALL;

     if (verify_wnd(&wnd))     {
          while (area)    {
               WCOLOR [--area] &= ~BRIGHT;
               WCOLOR [area] |= inten;
          }
          redraw(wnd);
     }
}
```

BLOQUE (6)

```
/* ----- coloca el titulo de ventana ------ */
void set_title(WINDOW *wnd, char *title)
{
     if (verify_wnd(&wnd))     {
          WTITLE = title;
          redraw(wnd);
     }
}
```

BLOQUE (7)

```
/* -- redibuja una ventana cuando un atributo es modificado -- */
static void redraw(WINDOW *wnd)
{
#ifndef FASTWINDOWS
     int x, y, chat, atr;

     for (y = 1; y < HEIGHT-1; y++)
          for (x = 1; x < WIDTH-1; x++) {
               chat = dget(wnd, x, y);
               atr = (((chat >>8)&255) ==
                    PNORMAL ? WNORMAL : WACCENT);
               displ(wnd, x, y, chat&255, atr);
          }
     wframe(wnd);
#endif
     PNORMAL = WNORMAL;
}
```

BLOQUE (8)

```
/* --- despliega una ventana establecida --- */
void display_window(WINDOW *wnd)
{
     if (verify_wnd(&wnd) && !VISIBLE)  {
          VISIBLE = 1;
#ifdef FASTWINDOWS
          if (HIDDEN)     {
               HIDDEN = 0;
               vrstr(wnd);
          }
          else {
               vsave(wnd);
               clear_window(wnd);
               wframe(wnd);
          }
#else
          vswap(wnd);
#endif
     }
}
```

BLOQUE (9)

```
/* --- cierra todas las ventanas --- */
void close_all()
{
     WINDOW *sav, *wnd = listtail;

     while (wnd)     {
          sav = PREV;
          delete_window(wnd);
          wnd = sav;
     }
}
```

BLOQUE (10)

```
/* --- quita una ventana --- */
void delete_window(WINDOW *wnd)
{
     if (verify_wnd(&wnd))          {
          hide_window(wnd);
          free(SAV);
          remove_list(wnd);    /* quita ventana de la lista */
          free(wnd);
     }
}
```

BLOQUE (11)

```
/* --- oculta una ventana --- */
void hide_window(WINDOW *wnd)
{
     if (verify_wnd(&wnd) && VISIBLE)    {
#ifndef FASTWINDOWS
          vswap(wnd);
#else
          vrstr(wnd);
#endif
          HIDDEN = 1;
          VISIBLE = 0;
     }
}
```

BLOQUE (12)

```
#ifndef FASTWINDOWS
/* --- reposiciona la vetana en un plano de 3 ejes --- */
void repos_wnd(WINDOW *wnd, int x, int y, int z)
{
     WINDOW *twnd;
     int x1, y1, chat;
     if (!verify_wnd(&wnd))
          return;
     twnd = establish_window(x+COL, y+ROW, HEIGHT, WIDTH);
     twnd->tl = WTITLE;
     twnd->btype = BTYPE;
     twnd->wcolor[BORDER] = WBORDER;
     twnd->wcolor[TITLE] = WTITLEC;
     twnd->wcolor[ACCENT] = WACCENT;
     twnd->wcolor[NORMAL] = WNORMAL;
     twnd->pn = PNORMAL;
     twnd->wsp = SCROLL;
     twnd->cr = WCURS;
     if (z != 1)     {
          remove_list(twnd);
          if (z == 0)
               insert_list(twnd, wnd);
          else
               beg_list(twnd);
     }
```

```
     for (yl = 0; yl < twnd->_wh; yl++)
          for (xl = 0; xl < twnd->_ww; xl++) {
               chat = dget(wnd, xl, yl);
               displ(twnd, xl, yl, chat&255, (chat>>8)&255);
          }

     twnd->_wv = 1;
     vswap(twnd);
     hide_window(wnd);
     free(SAV);
     remove_list(wnd);
     *wnd = *twnd;
     insert_list(wnd, twnd);
     remove_list(twnd);
     free(twnd);
}
#endif
```

BLOQUE (13)

```
/* --- limpia el area de ventana --- */
void clear_window(WINDOW *wnd)
{
     register int xl, yl;

     if (verify_wnd(&wnd))
          for (yl = 1; yl < HEIGHT-1; yl++)
               for (xl = 1; xl < WIDTH-1; xl++)
                    displ(wnd,xl, yl, ' ', WNORMAL);
}
```

BLOQUE (14)

```
/* --- dibuja el contorno de la ventana --- */
static void wframe(WINDOW *wnd)
{
     register int xl, yl;

     if (!verify_wnd(&wnd))
          return;
     /*   despliega el titulo de la ventana    */
     displ(wnd,0, 0, NW, WBORDER);
     dtitle(wnd);
     displ(wnd,WIDTH-1, 0, NE, WBORDER);
     /* despliega los lados (partes laterales) de la ventana */
     for (yl = 1; yl < HEIGHT-1; yl++)  {
          displ(wnd,0, yl, SIDE, WBORDER);
          displ(wnd,WIDTH-1, yl, SIDE, WBORDER);
     }
     /*   despliega el fondo del marco de la ventana    */
     displ(wnd,0, yl, SW, WBORDER);
     for (xl = 1; xl < WIDTH-1; xl++)
          displ(wnd,xl, yl, LINE, WBORDER);
     displ(wnd,xl, yl, SE, WBORDER);
}
```

BLOQUE (15)

```
/* --- despliega el titulo de la ventana --- */
static void dtitle(WINDOW *wnd)
{
     int x1 = 1, i, ln;
     char *s = WTITLE;

     if (!verify_wnd(&wnd))
          return;
     if (s)    {
          ln = strlen(s);
          if (ln > WIDTH-2)
               i = 0;
          else
               i = ((WIDTH-2-ln) / 2);
          if (i > 0)
               while (i--)
                    displ(wnd, x1++, 0, LINE, WBORDER);
          while (*s && x1 < WIDTH-1)
               displ(wnd, x1++, 0, *s++, WTITLEC);
     }
     while (x1 < WIDTH-1)
          displ(wnd, x1++, 0, LINE, WBORDER);
}
```

BLOQUE (16)

```
/* --- función "printf" orientado a ventanas --- */
void wprintf(WINDOW *wnd, char *ln, ...)
{
     char dlin[100], *dl = dlin;

     if (verify_wnd(&wnd))          {
          va_list ap;
          va_start(ap, ln);
          vsprintf(dlin, ln, ap);
          va_end(ap);
          while (*dl)
               wputchar(wnd, *dl++);
     }
}
```

BLOQUE (17)

```
/* --- escribe un caracter en la ventana --- */
void wputchar(WINDOW *wnd, int c)
{
     if (!verify_wnd(&wnd))
          return;
     switch (c)     {
          case '\n':
               if (SCROLL == HEIGHT-3)
                    scroll(wnd, UP);
               else
                    SCROLL++;
               WCURS = 0;
```

```
                break;
            case '\t':
                do displ(wnd,(WCURS++)+3,SCROLL+1,' ',WNORMAL);
                    while ((WCURS%TABS) && (WCURS+1) < WIDTH-1);
                break;
            default:
                if ((WCURS+1) < WIDTH-1) {
                    displ(wnd, WCURS+1, SCROLL+1, c, WNORMAL);
                    WCURS++;
                }
                break;
        }
}
```

BLOQUE (18)

```
/* --- coloca el cursor de ventana --- */
void wcursor(WINDOW *wnd, int x, int y)
{
    if (verify_wnd(&wnd) && x < WIDTH-1 && y < HEIGHT-1)   {
        WCURS = x;
        SCROLL = y;
        cursor(COL+x+1, ROW+y+1);
    }
}
```

BLOQUE (19)

```
/* --- permite al usuario hacer una selección de ventana --- */
int get_selection(WINDOW *wnd, int s, char *keys)
{
    int c = 0, ky;
    if (!verify_wnd(&wnd))
        return 0;
    SELECT = s;
    while (c != ESC && c != '\r' && c != BS && c != FWD)   {
        accent(wnd);
        c = get_char();
        deaccent(wnd);
        switch (c)          {
            case UP:  if (SELECT > 1)
                          SELECT--;
                      else
                          SELECT = SCROLL+1;
                      break;
            case DN:  if (SELECT < SCROLL+1)
                          SELECT++;
                      else
                          SELECT = 1;
                      break;
            case '\r':
            case ESC:
            case FWD:
            case BS:  break;
            default:  if (keys) {
                          ky = 0;
```

```
                    while (*(keys + ky));
                        if(*(keys+ky)==toupper(c) ||
                            *(keys+ky)==tolower(c))
                            return ky + 1;
                        ky++;
                    }
                }
                break;
            }
        }
        return    c == '\r' ? SELECT : c == ESC ? 0 : c;
}
```

BLOQUE (20)

```
union REGS rg;

/* --- hace un scroll de ventana hacia arriba o abajo --- */
void scroll(WINDOW *wnd, int dir)
{
    int row = HEIGHT-1, col, chat;

    if (!verify wnd(&wnd))
        return;
    if (NEXT == NULL && HEIGHT > 3 && VISIBLE)   {
        rg.h.ah = dir == UP ? 6 : 7;
        rg.h.al = 1;
        rg.h.bh = WNORMAL;
        rg.h.cl = COL + 1;
        rg.h.ch = ROW + 1;
        rg.h.dl = COL + WIDTH - 2;
        rg.h.dh = ROW + HEIGHT - 2;
        int86(16, &rg, &rg);
        return;
    }
    if (dir == UP) {
        for (row = 2; row < HEIGHT-1; row++)
            for (col = 1; col < WIDTH-1; col++)     {
                chat = dget(wnd, col, row);
                displ(wnd,col,row-1,chat&255,(chat>>8)&255);
            }
        for (col = 1; col < WIDTH-1; col++)
            displ(wnd, col, row-1, ' ', WNORMAL);
    }
    else {
        for (row = HEIGHT-2; row > 1; --row)
            for (col = 1; col < WIDTH-1; col++)     {
                chat = dget(wnd, col, row-1);
                displ(wnd,col,row,chat&255,(chat>>8)&255);
            }
        for (col = 1; col < WIDTH-1; col++)
            displ(wnd, col, row, ' ', WNORMAL);
    }
}
```

BLOQUE (21)

```
#ifndef FASTWINDOWS
/* --- calcula la direccion de un caracter de despliegue en
 *      ventana ---*/
static int *waddr(WINDOW *wnd, int x, int y)
{
     WINDOW *nxt = NEXT;
     int *vp;

     if (!VISIBLE)
          return (int *) (SAV+y*(WIDTH*2)+x*2);
     x += COL;
     y += ROW;
     while (nxt)    {
          if (nxt->_wv)
               if (x >= nxt->_wx && x <= nxt->_wx + nxt->_ww-1)
                    if (y >= nxt->_wy &&
                              y <= nxt->_wy + nxt->_wh-1)   {
                         x -= nxt->_wx;
                         y -= nxt->_wy;
                         vp = (int *)
                              ((nxt->_ws) +y*(nxt->_ww*2)+x*2);
                         return vp;
                    }
          nxt = nxt->_nx;
     }
     return NULL;
}
```

BLOQUE (22)

```
/* --- despliega un caracter en una ventana --- */
void displ(WINDOW *wnd, int x, int y, int ch, int at)
{
     int *vp;
     int vch = (ch&255)|(at<<8);

     if ((vp = waddr(wnd, x, y)) != NULL)
          *vp = vch;
     else
          vpoke(VSG,vad(x+COL,y+ROW),vch);
}
```

BLOQUE (23)

```
/* --- obtiene un caracter desplegado en ventana --- */
static int dget(WINDOW *wnd, int x, int y)
{
     int *vp;

     if ((vp = waddr(wnd, x, y)) != NULL)
          return *vp;
     return vpeek(VSG,vad(x+COL,y+ROW));
}
```

```
/* ======== Funciones de Video de Bajo Nivel ======== */
BLOQUE (24)
/* - intercambia la imagen de video con el buffer-de-salvado -*/
static void vswap(WINDOW *wnd)
{
     int x, y, chat;
     int *bf = (int *) SAV;
     for (y = 0; y < HEIGHT; y++)
          for (x = 0; x < WIDTH; x++)    {
               chat = *bf;
               *bf++ = dget(wnd, x, y);
               displ(wnd, x, y, chat&255, (chat>>8)&255);
          }
}
# else
BLOQUE (25)
/* --- salva la memoria de video en el buffer-de-salvado --- */
static void vsave(WINDOW *wnd)
{
     int x, y;
     int *bf = (int *) SAV;

     for (y = 0; y < HEIGHT; y++)
          for (x = 0; x < WIDTH; x++)
               *bf++ = vpeek(VSG, vad(x+COL, y+ROW));
}

BLOQUE (26)
/* --- restablece la memoria de video tomando el contenido del
 *     buffer-de-salvado --- */
static void vrstr(WINDOW *wnd)
{
     int x, y;
     int *bf = (int *) SAV;

     for (y = 0; y < HEIGHT; y++)
          for (x = 0; x < WIDTH; x++)
               vpoke(VSG,vad(x+COL,y+ROW), *bf++);
}
#endif

BLOQUE (27)
/*- intesifica o desintensifica la linea apuntada por SELECT -*/
void acline(WINDOW *wnd, int set)
{
     int x, ch;

     if (!verify_wnd(&wnd))
          return;
     for (x = 1; x < WIDTH - 1; x++)    {
          ch = dget(wnd, x, SELECT) & 255;
          displ(wnd, x, SELECT, ch, set);
     }
}
```

```
/* ======== Funciones de Listas Ligadas ======== */
BLOQUE (28)
/* --- agrega una ventana al final de la lista --- */
static void add_list(WINDOW *wnd)
{
     if (listtail) {
          PREV = listtail;
          listtail->_nx = wnd;
     }
     listtail = wnd;
     if (!listhead)
          listhead = wnd;
}
BLOQUE (29)
/* --- agrega una ventana al inicio de la lista --- */
static void beg_list(WINDOW *wnd)
{
     if (listhead) {
          NEXT = listhead;
          listhead->_pv = wnd;
     }
     listhead = wnd;
     if (!listtail)
          listtail = wnd;
}

BLOQUE (30)
/* --- quita una ventana de la lista --- */
static void remove_list(WINDOW *wnd)
{
     if (NEXT)
          NEXT->_pv = PREV;
     if (PREV)
          PREV->_nx = NEXT;
     if (listhead == wnd)
          listhead = NEXT;
     if (listtail == wnd)
          listtail = PREV;
     NEXT = PREV = NULL;
}

BLOQUE (31)
/* --- inserta w1 después de w2 --- */
static void insert_list(WINDOW *w1, WINDOW *w2)
{
     w1->_pv = w2;
     w1->_nx = w2->_nx;
     w2->_nx = w1;
     if (w1->_nx == NULL)
          listtail = w1;
     else
          w1->_nx->_pv = w1;
}
```

```
BLOQUE (32)
#ifndef FASTWINDOWS
/* --- verifica la existencia de una ventana en la lista --- */
static int verify_wnd(WINDOW **wl)
{
     WINDOW *wnd;

     wnd = listhead;
     if (*wl == NULL)
          *wl = listtail;
     else {
          while (wnd != NULL) {
               if (*wl == wnd)
                    break;
               wnd = NEXT;
          }
     }
     return wnd != NULL;
}
#endif

BLOQUE (33)
WINDOW *ewnd = NULL;

/* --- mensajes de error --- */
void error_message(char *s)
{
     ewnd = establish_window(50, 22, 3, max(10, strlen(s)+2));
     set_colors(ewnd, ALL, RED, YELLOW, BRIGHT);
     set_title(ewnd, " ERROR! ");
     display_window(ewnd);
     wprintf(ewnd, s);
     putchar(BELL);
}

BLOQUE (34)
void clear_message()
{
     if (ewnd)
          delete_window(ewnd);
     ewnd = NULL;
}
```

■ **DOCUMENTACION LISTADO # 4**

En general:

Las declaraciones de datos externas en twindow.c incluyen: los prototipos para las funciones que son locales al archivo fuente, un arreglo de estructuras que define los 5 tipos de contorno que las ventanas pueden tener y los apuntadores a la cabeza y cola de la lista ligada de estructuras WINDOW. El

contorno de una ventana es controlado por un miembro en la estructura WINDOW, dicho miembro es un offset (de entero) dentro de la tabla de tipos de contorno. La entrada a la que dicho offset apunta contiene 6 valores, cada uno de los cuales representa uno de los lados o esquinas de la ventana. El primer valor representa la esquina superior izquierda o "noroeste". Los nombres de variables (nw, ne, se, sw) indicaran la esquina que esta representada. El entero side representará los lados verticales del contorno. El entero line será empleado para hacer referencia a las lineas horizontales del tope y fondo del contorno.

Los dos apuntadores a la estructura WINDOW: listhead y listtail mantendrán la dirección de cabeza y cola de la lista ligada, respectivamente. En el mismo orden en que las ventanas sean establecidas, serán agregadas a la lista. Inicialmente los apuntadores listhead y listtail serán nulos. Al ser establecida la primer ventana, se asignará la memoria requerida para una estructura WINDOW y su dirección será copiada a los apuntadores antes mencionados. En la lista ligada la cabeza apuntará a la primer ventana establecida en el sistema. Cuando una segunda ventana es establecida, su dirección es copiada en el apuntador de cola de la lista. La dirección de la segunda ventana será pasada al apuntador " nx" (apuntador a la siguiente ventana) de la primer ventana; y la dirección de la primera será escrita en el apuntador " pv" (apuntador a la ventana previa) de la segunda ventana. La lista ligada es una estructura de datos bidireccional llamada lista doblemente ligada.

BLOQUE (1)

- Muchas de las funciones de libreria de Turbo C, son en realidad definiciones de macros contenidas en los encabezados de los archivos.
  Los archivo de cabecera estandar incluidos en twindow.c (mediante la directiva #include) son:

Archivos de Cabecera	Descripcion
stdio.h	Usado para flujo de E/S
ctype.h	Funciones para manejo de caracteres
stdarg.h	Listas de arg. de longitud variable
dos.h	Funciones de interfaz con el DOS
alloc.h	Funciones para asignación dinámica
stdlib.h	Para declaraciones de miscelánea
string.h	Funciones de cadenas de caracteres

Además de los archivos de cabecera estándar, twindow.c incluye nuestros archivos de cabecera: twindow.h y keys.h.

BLOQUE (2)

- **ANALISIS DE LA FUNCION:WINDOW *establish_window(x, y, h, w)**
  Esta función establece una ventana pero no la despliega. Los parámetros "x, y" corresponden a las coordenadas de la esquina superior izquierda de la ventana. Los parámetros "h, w" corresponden a la altura y ancho de la ventana en posiciones de caracter respectivamente.
  La función no permitirá que los parametros proporcionados excedan los límites de la pantalla.
  La ventana se establece con atributos *default*. Su contorno consta de líneas sencillas, color blanco brillante sobre fondo negro, sin título. Estos atributos podrán ser modificados por llamadas subsecuentes a otras funciones descritas posteriormente.
  La ventana mas recientemente establecida aparecerá en pantalla al frente de las anteriores.
  Esta función regresa un apuntador a la estructura WINDOW, la cual está definida en el archivo de encabezado twindow.h. Este apuntador será usado para llamadas subsecuentes a funciones de ventana, para identificar la ventana que se quiere direccionar.

- CUERPO DE LA FUNCION:
  El primer paso llevado a cabo por la función establish_window es inicializar la variable **VSG** a la dirección del segmento de la RAM de video. Posteriormente, se asigna memoria para una estructura WINDOW y se inicializa esta estructura con las características *default* de la ventana. Tambien se asigna memoria para el buffer-de-salvado de la ventana, cuya dirección es escrita en el miembro correspondiente de la estructura WINDOW. Con la estructura así inicializada, la función llama a **add list** para agregar la estructura a la lista ligada de ventanas. El area de texto de la ventana es limpiada, y el contorno de la ventana es dibujado (siempre y cuando se estén procesando ventanas estratificadas). Todas estas operaciones son llevadas a cabo sobre el buffer-de-salvado, ya que la ventana no ha sido desplegada todavía.

  Declaración:
  if ((wnd = (WINDOW *) malloc(sizeof (WINDOW))) == NULL)
  - sizeof regresa el tamaño en *bytes* que ocupará la estructura de ventana WINDOW, dicho valor le entra como parámetro a la función **malloc**, que reserva ese espacio en memoria regresando: un valor NULL si no encontró espacio suficiente, o la dirección en memoria de la estructura, en caso contrario. Esta dirección, al ser afectada por el "cast", se convierte en un apuntador a una estructura de tipo WINDOW, que es asignado a la variable **wnd**.
  - Nota: Las funciones de asignacion dinámica de memoria del lenguaje C son: malloc() y free(). Éstas trabajan sobre el *heap*. La función malloc() asigna memoria y

regresa un apuntador tipo void al inicio de ésta, o un valor NULL si no existe suficiente espacio para cubrir la petición. La función free() regresa la memoria previamente asignada al *heap* para su posible reutilización. Los prototipos para estas funciones se encuentran en el archivo de cabecera stdlib.h.

Declaración: `HEIGHT = min(h, SCRNHT);`
La función **min()** regresa el menor valor de sus argumentos (h o SCRNHT). La variable SCRNHT (definida con un valor de 25) representa la altura máxima permitida para una ventana. Por tanto, esta instrucción implica la validación del límite de renglones máximos permitidos para la creación de una ventana. El valor regresado por min() será almacenado en la variable HEIGHT.
Declaración: `COL = max(0, min(x, SCRNWIDTH - WIDTH));`

- SCRNWIDTH está definido con un valor de 80 (columnas) que representa el valor máximo de columnas en pantalla.
- WIDTH representa el ancho de ventana (número de columnas) ya validado.
- La función max() regresará el mayor valor de sus parametros, que quedará almacenado en la variable COL.

Declaración:

```
if ((SAV = malloc(WIDTH * HEIGHT * 2)) != (char *) 0)
```

Las variables WIDTH y HEIGHT ya vienen validadas. Se realiza una multiplicación de la altura de la ventana por su ancho, y el resultado por 2. Este último factor es involucrado debido a que cada columna cuenta con 2 *bytes* (atributo y caracter). El resultado final nos da el área total a reservar en memoria para esta ventana. La función de asignación de memoria es llevada a cabo por malloc(), que regresará un apuntador tipo void al primer *byte* de la región de memoria reservada. Este apuntador es copiado a la variable SAV (miembro de la estructura WINDOW que apunta al buffer-de-salvado de la ventana). Posteriormente se verifica si SAV es un apuntador nulo. Pero, como esta verificación no se puede hacer directamente debido a que SAV es un apuntador a caracteres; a dicho apuntador le es aplicado un "cast" al valor "0"; convirtiendolo en un apuntador nulo a caracteres.

Declaración: `add_list(wnd);`
Esta función es invocada siempre que la verificación anterior haya arrojado un apuntador no nulo. Y su propósito es agregar una estructura de ventana al final de la lista ligada (mas adelante se analizará esta función).

Declaraciones: `clear_window(wnd);`
`wframe(wnd);`
Estas funciones serán invocadas en caso de no estar definida la macro FASTWINDOWS; es decir, son funciones privativas para ventanas estratificadas. La primera función limpia el área de ventana, y la segunda, dibuja el marco de la misma.

Estas operaciones son llevadas a cabo sobre el buffer-de-salvado.

BLOQUE (3)

- ANALISIS DE LA FUNCION:
  void set border(WINDOW *wnd, int btype)

Esta funcion establece el tipo de marco o contorno de ventana. El parametro entero btype puede contener alguno de los valores siguientes:

0 - lineas sencillas para todos los lados.
1 - lineas dobles para todos los lados.
2 - fondo y tope de lineas sencillas; partes laterales de lineas dobles.
3 - fondo y tope de lineas dobles; partes laterales de lineas sencillas.
4 - menu de ventana especial *pop-down* con contorno de lineas sencillas.

- CUERPO DE LA FUNCION:

Esta función modifica las caracteristicas de una ventana previamente establecida. Primero, se invoca la función verify_wnd() para asegurar que el programa que la invoca, está pasando la dirección de una ventana ya establecida. Luego, modifica el atributo de contorno. Finalmente invoca a la funcion redraw() de manera que el cambio pueda ser observado en pantalla.
En particular; la función verify_wnd() tiene como parámetro de entrada un apuntador a estructura de ventana, y como salida un valor"0" si la ventana no existe en la lista, o un valor "1" en caso contrario.
En caso de haber verificado la existencia de la ventana en la lista, se almacena en la variable BTYPE, el tipo de marco seleccionado.
Finalmente se invoca a la funcion redraw para redesplegar la ventana con el nuevo atributo.

BLOQUE (4)

- ANALISIS DE LA FUNCION:
  void set colors(WINDOW *wnd, int area, int bg, int fg, int inten)

Esta función establece los colores de una ventana. El parámetro area puede ser alguno de los siguientes: ALL, BORDER, TITLE, ACCENT, NORMAL. Este parámetro indica el area (o areas) afectada(s) de la ventana. La variable ALL afecta todas las áreas, BORDER coloca los colores para el contorno de la ventana, ACCENT representa el area usada para las barras de menu, empleada tambien para intensificar texto (afectado por NORMAL), finalmente la variable NORMAL representa el atributo normal de video (blanco intensificado sobre fondo negro).

Los enteros bg y fg especifican los colores para el area de fondo y primer plano respectivamente. Los colores pueden ser: RED, GREEN, BLUE, WHITE, YELLOW, AQUA, MAGENTA y BLACK. El entero inten especifica la intensidad de los caracteres del primer plano y puede ser: BRIGHT o DIM.

- CUERPO DE LA FUNCION:
Esta función modifica los colores de una ventana previamente establecida.
Primeramente se verifica en que modo de video se está trabajando, si monocromatico o color. En el primer caso se hacen las validaciones necesarias para que los colores de fondo y primer plano correspondan solamente a la combinacion de blanco y negro. En el caso de monitor a color las validaciones no son necesarias.
Posteriormente, se verifica que la ventana direccionada exista en la lista liqada por medio de verify_wnd().
En caso afirmativo, se procede a salvar en el arreglo de colores de ventana (WCOLOR []) los parámetros proporcionados.
Finalmente se invoca a la funcion redraw para redesplegar la ventana con el nuevo atributo.

BLOQUE (5)

ANALISIS DE LA FUNCION:
void set intensity(WINDOW *wnd, int inten)
Esta función establece la intensidad del primer plano para todas las áreas de ventana. El valor de inten puede ser: BRIGHT o DIM.

CUERPO DE LA FUNCION:
Esta función modifica la intensidad de las areas de una ventana previamente establecida.
Primero se invoca a verify_wnd() por la misma razón expuesta en casos anteriores. Posteriormente se procede a modificar la condición de intensidad (independientemente de su valor anterior) en todas las áreas de la ventana, segun el valor indicado en la variable inten.
Finalmente se invoca a la funcion redraw para redesplegar la ventana con el nuevo atributo.

BLOQUE (6)

- ANALISIS DE LA FUNCION:
void set title(WINDOW *wnd, char *title)
Esta función almacena la cadena correspondiente al titulo de la ventana, en el arreglo de caracteres WTITLE especificado en la estructura WINDOW.

BLOQUE (7)

- ANALISIS DE LA FUNCION: **static redraw(WINDOW *wnd)**
  Esta función "redespliega" una ventana cuando alguno de sus atributos haya sido modificado, y siempre que la configuración empleada corresponda a ventanas estratificadas.

- CUERPO DE LA FUNCION:
  La declaracion static: provoca que la función sólo pueda ser utilizada en el archivo twindow.c, permaneciendo inaccesible a otros archivos.
  Si se trata de ventanas estratificadas, se procede de la siguiente manera:
  - Se utilizan dos instrucciones "for", una para controlar el despliegue de renglones y otra para el despliegue de columnas del area de texto de la ventana.
  - La funcion dget() regresa un caracter y su atributo, almacenados en la variable entera chat.
  - La variable atr es llenada con el valor de chat desplazado en 8 posiciones a la derecha para colocar el *byte* de atributo del caracter en la parte baja del entero y anular la parte alta del entero correspondiente al valor ASCII.
  - Para el despliegue de los caracteres con sus nuevos atributos se emplea la funcion displ().
  - Finalmente se hace una llamado a la función wframe() para desplegar el contorno de la ventana.

BLOQUE (8)

- ANALISIS DE LA FUNCION: **void display window(WINDOW *wnd)**
  Esta función despliega una ventana previamente establecida. Para evitar confusion visual, esta funcion es llamada despues de haber establecido todos los atributos y, de ser posible, despues de que la ventana haya sido inicializada con texto.

- CUERPO DE LA FUNCION:
  Esta función opera de manera diferente para ventanas apiladas y estratificadas. En cualquier caso, la función no reaccionara si la ventana está visible en pantalla. Pero en caso contrario y tratandose de ventanas estratificadas; la función display window intercambiara la memoria de video con el buffer-de-salvado de la ventana, mediante una llamada a la funcion vswap().
  Para el caso de ventanas apiladas, primeramente se realiza una prueba para determinar si la ventana direccionada se encuentra oculta. En caso afirmativo, el buffer-de-salvado de la ventana es escrito hacia la RAM de video mediante una llamada a la función vrstr(); el caso opuesto implica que la ventana nunca ha sido desplegada, entonces se hace un llamado a la funcion vsave() para salvar el contenido

corriente de la RAM de video en el buffer-de-salvado. Finalmente las funciones clear_window() y wframe() son invocadas para desplegar la ventana vacía.

BLOQUE (9)

- ANALISIS DE LA FUNCION: **void close_all()**
  Esta función elimina todas las ventanas establecidas.

- CUERPO DE LA FUNCION:
  Para llevar a cabo esta tarea, la función "navega" sobre la lista ligada de estructuras WINDOW, haciendo llamadas a la función delete window.

  Mientras el apuntador de ventana no sea nulo:
  Se guarda en la variable "sav" el apuntador de la ventana previa a la que se desea eliminar.
  Se elimina la ventana actual.
  Se actualiza el apuntador de ventana con la dirección del apuntador "sav".
  Finalmente todas las ventanas habrán sido eliminadas de la lista ligada, y el apuntador de ventana contendrá una dirección nula.

BLOQUE (10)

- ANALISIS DE LA FUNCION: **void delete_window(WINDOW *wnd)**
  Esta función elimina una ventana visible (desplegada por la función display window) encontrada en la lista ligada de estructuras WINDOW; restableciendo la imagen de pantalla a su estado anterior.

- CUERPO DE LA FUNCION:
  La secuencia general seguida por esta función es:
  Verificar la existencia de la ventana en la lista ligada; en caso afirmativo: ocultar la ventana direccionada, liberar la memoria ocupada por su buffer-de-salvado (apuntada por SAV), remover la estructura WINDOW de la lista ligada, y finalmente liberar la memoria ocupada por dicha estructura.

BLOQUE (11)

- ANALISIS DE LA FUNCION: **void hide_window(WINDOW *wnd)**
  Esta función oculta una ventana que ya haya sido desplegada, devolviendo la pantalla a su contenido previo. Nota: la ventana todavía existe y puede, ser direccionada para llevar a cabo operaciones sobre ella.

- CUERPO DE LA FUNCION:
  La secuencia general seguida por esta función es:
  Esta función se ejecuta si la ventana direccionada se encuentra visible.
  Para ventanas estratificadas: se hace un llamado a la función vswap() para intercambiar el buffer-de-salvado con la RAM de video.
  Para ventanas apiladas: se invoca a la función vrstr() para restablecer la RAM de video a partir del contenido del buffer-de-salvado.
  Finalmente, y para ambos casos, se prende la bandera de "ventana oculta" (HIDDEN = 1) y se apaga la bandera de "ventana visible" (VISIBLE = 0).

BLOQUE (12)

- ANALISIS DE LA FUNCION:
  void repos wnd(WINDOW *wnd, int x, int y, int z)
  Esta función opera para ventanas estratificadas. Es invocada por alguna de las macros siguientes: move_window, rmove_window, rear_window y forefront; definidas en el archivo twindow.h. La función repos_wnd cambia la posición de la ventana siguiendo los pasos enumerados a continuación: Primero, establece una ventana temporal, a continuación coloca dicha ventana dentro de la lista ligada y en una posición específica, de acuerdo con la macro ejecutada, posteriormente escribe el contenido original de la ventana en el buffer-de-salvado de la ventana temporal, y finalmente procede a desplegar la ventana temporal, ocultando la ventana original.

- CUERPO DE LA FUNCION:
  Se crea un apuntador a una estructura WINDOW (para el establecimiento de la ventana temporal).
  Si la venta existe en la lista ligada:
    Se establece una ventana temporal -twnd- a través de la función **establish_window**.
    Se hace una copia de la ventana a ser movida (ventana original -wnd-) en la ventana temporal (con las nuevas coordenadas (x,y) o (0,0,z)).
      Si la ventana será trasladada hacia atrás de todas las ventanas (0,0,-1):
        - Elimina la estructura de la ventana temporal de la lista ligada, encontrada al final de ésta, como resultado de la función establish_window.
        - Agrega la estructura de la ventana temporal al inicio de la lista.
      Si la ventana será movida en el plano "x,y":
        - Se elimina la estructura de la ventana temporal de la lista.

- Se inserta en la lista la ventana temporal en la posición siguiente a la original.

Se inicializa el buffer-de-salvado de la ventana temporal con el contenido (texto) de la ventana a ser movida.
Se coloca a "1" la bandera "VISIBLE" de la ventana temporal.
Se efectúa el intercambio de la RAM de video con el buffer-de-salvado de la ventana temporal. (apareciendo dicha ventana en pantalla en sus nuevas coordenadas).
Se elimina del sistema la ventana original:
Se oculta la ventana original.
Se libera la memoria ocupada por el buffer-de-salvado de la ventana original.
Se remueve la ventana original de la lista ligada.
El apuntador de la ventana original es actualizado con la dirección del apuntador de la ventana temporal. Esto, para poder seguir manejando el apuntador "wnd" reconocido por todas las funciones de este programa.
Se inserta el nuevo apuntador de la ventana original a continuación del apuntador de la ventana temporal en la lista ligada.
Se elimina el apuntador de la ventana temporal de la lista ligada.
Y finalmente se libera la memoria ocupada por la estructura de la ventana temporal.

BLOQUE (13)

- ANALISIS DE LA FUNCION: **void clear_window(WINDOW *wnd)**
Esta función realiza "Clear Screen", escribiendo blancos en el área de datos de la ventana, mediante la función displ().

BLOQUE (14)

- ANALISIS DE LA FUNCION: **static wframe(WINDOW *wnd)**
Esta función despliega el marco de la ventana, mediante la función displ().

- CUERPO DE LA FUNCION:
Si la ventana existe en la lista ligada:
  - Se Despliega el titulo de la ventana.
  Para ello primeramente se manda a escribir en pantalla el caracter de contorno correspondiente a la esquina superior izquierda de la ventana.
  El siguiente paso es desplegar el título (si existe).
  La función que lleva a cabo esta operación despliega junto con el título, los caracteres de contorno que complementan el tope de la ventana. Dicha función será vista en el siguiente bloque.

Posteriormente se manda a escribir en pantalla el caracter de contorno correspondiente a la esquina superior derecha de la ventana.

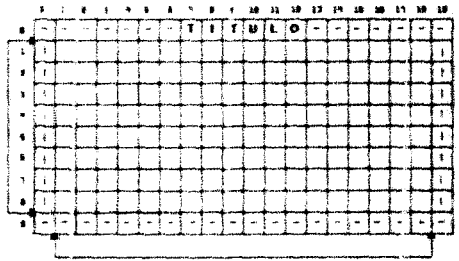
- Despliegue de los lados (partes laterales) de la ventana.
  La instrucción "for" lleva el control de los renglones y las funciones displ(), el de las columnas. Mediante la primera función displ() se desplegará la parte lateral izquierda de la ventana, y mediante la segunda, se desplegará la parte lateral derecha de la misma.
- Despliegue de la parte inferior del contorno de la ventana.
  Para ello primeramente se manda a escribir en pantalla el caracter de contorno correspondiente a la esquina inferior izquierda de la ventana.
  A continuación se colocan las líneas de contorno correspondientes al fondo de la ventana, excluyendo las correspondientes a las esquinas inferiores del marco.
  Posteriormente se manda a escribir en pantalla el caracter de contorno correspondiente a la esquina inferior derecha de la ventana. (La figura 6.2 ilustra las operaciones llevadas a cabo en este bloque).

BLOQUE (15)

- ANALISIS DE LA FUNCION: **static dtitle(WINDOW *wnd)**
  Esta función despliega el título y los caracteres de contorno que complementan el tope de la ventana; mediante la función displ().

- CUERPO DE LA FUNCION:
  Si la ventana existe en la lista:
  - Se verifica si lleva título. En caso afirmativo: Se almacena la longitud de la cadena -terminada en NULL- que forma el título *(longitud calculada* por la función strlen(s)). En caso contrario: se procede al despliegue de los caracteres de contorno del tope de la ventana.
  - Se verifica si el título excede los límites de la ventana. En caso afirmativo: Se limpia el contenido del contador de caracteres de contorno del tope de la ventana. En caso contrario: Dicho contador es llenado con el número de caracteres que delimitarán el título tanto a la derecha como a la izquierda (-- TITULO --).
  - Si la longitud del título no sobrepasa el ancho de la ventana: Se despliega el primer conjunto de caracteres de contorno que antecede al título, luego se procede al despliegue del mismo y finalmente al de los caracteres de contorno posteriores al título.

FIG. 6.2 ILUSTRACION DEL DESPLIEGUE DEL MARCO DE LA VENTANA



NOTA : EJEMPLO DE VENTANA CON ALTURA = 10 Y ANCHO = 20

BLOQUE (16)

- ANALISIS DE LA FUNCION:

void wprintf(WINDOW *wnd, char *in, ...)

Esta función es una versión de la función "printf" orientada a ventanas. Se hace uso de la función estándar **sprintf** para crear un string de una línea que será desplegada en la ventana. (Nota: Hay que asegurarse que la cadena resultante sea menor de 100 caracteres, o en su defecto modificar la extensión del arreglo **dlin[]**).

- CUERPO DE LA FUNCION:

La función wprintf() trabaja con un número variable de parámetros. Los puntos suspensivos en la lista de parametros de la función, indican la presencia de un numero variable de parámetros con diferentes tipos de datos.
La secuencia general seguida por esta función es:
Si la ventana existe en lista:

Se declara la variable "ap" del tipo va list, va list es un tipo de arreglo especial definido en **stdarg.h**, utilizado para declarar una lista de parametros.
Se invoca la función **va start** que establece el inicio y fin de la lista.
Se hace un llamado a la función **vsprintf**, que es una versión de la función estándar **sprintf** que aceptara un parámetro de tipo **va list**. Los parametros pasados a la función **wprintf** son procesados por vsprintf en el arreglo de caracteres **dlin[]**.
Posteriormente se ejecuta la función **va end**, instrucción complementaria de la función va start.
Finalmente, el arreglo dlin[] es desplegado en la ventana, caracter por caracter mediante la función **wputchar**.

Descripción de las funciones:
void va start(va list apuntador argumento, ultimo parámetro)
void va end(va list apuntador argumento)
Los prototipos de estas funciones estan definidos en el archivo stdarg.h. Ambas funciones trabajan conjuntamente para permitir que un numero variable de argumentos sea pasado a una función. El ejemplo más comun de funciones que aceptan un numero variable de parametros como argumento es "printf". La variable de tipo va list, que en ambos casos encontrada como argumento, está definida en el archivo stdarg.h. El procedimiento general para la creación de una función que pueda tomar un numero variable de argumentos es el siguiente (esta función en nuestro caso es wprintf): la función debera tener al menos un parámetro conocido (en nuestro caso existen 2) precediendo la lista de parámetros de longitud variable. Se identifica como "ultimo parámetro" al que se localice en el extremo derecho de la lista de parámetros conocidos (*in, en nuestro caso). Mediante una llamada a va start() debe ser inicializada la variable "apuntador argumento", antes de accesar cualquiera de los

parametros de longitud variable. Finalmente, una vez que todos los parametros han sido leídos y antes de regresar al proceso que hizo la peticion, se debe invocar a la funcion va_end() para asegurar el restablecimiento apropiado de la pila. Si la funcion va_end() no fuera llamada, cabria la posibilidad de que el programa parara el sistema.

Descripcion de la funcion:
int vsprintf(char *buf, char *format, va_list apunt_arg)
Los prototipos de esta función se encuentran en el archivo stdio.h. Tambien se deberá incluir el archivo stdarg.h. La función vsprintf es equivalente a sprintf, a excepción del formato de la lista de argumentos, que ahora consistirá de un apuntador a la lista de argumentos de longitud variable. Dicho apuntador debe ser de tipo va_list.

Descripción de la funcion:
int sprintf(char *buf, char *format, lista argumentos)
Esta función es identica a printf(), excepto que la salida generada es colocada dentro del arreglo apuntado por **buf**. El valor regresado por esta función corresponde al número de caracteres actualmente colocados dentro del arreglo. En cuanto al *string* apuntado por format, mencionemos que bajo su control se escriben los argumentos (especificados como tercer parametro de la funcion) dentro del arreglo apuntado por **buf**. El *string* apuntado por format, consta de 2 tipos de elementos: el primero, es utilizado para preparar los caracteres que serán impresos en pantalla; y el segundo, contiene los comandos de formato que definen la manera en la que los argumentos serán desplegados. El comando de formato es comenzado con un simbolo "%" y es seguido por el código del formato (ejemplo %d).

BLOQUE (17)

- ANALISIS DE LA FUNCION: **void wputchar(WINDOW *wnd, int c)**
  Esta funcion es una version orientada a ventanas de la función putchar. Escribe el caracter contenido en la variable "c" dentro de la ventana en la localización corriente del cursor de ventana. Despues de la escritura del caracter, la posicion del cursor es adelantada una localidad. Si el caracter corresponde a "nueva línea" (\n), el cursor es colocado en la siguiente linea, columna 0. Si el caracter es un tabulador (\t), el cursor es colocado en la siguiente posicion de tabulador en la ventana. Dicha posicion viene dada cada 4 posiciones de caracter en la ventana.
- CUERPO DE LA FUNCION:
  La localizacion del cursor en la ventana está en función de 2 miembros de la estructura WINDOW: WCURS (columna) y SCROLL (renglón). La función reacciona ante los caracteres de nueva linea y tabulador de la siguiente manera: Para el primer caso, si la variable SCROLL está en el fondo de la ventana,

el texto de la misma es desplazado una linea hacia arriba; de otro modo, la variable SCROLL es incrementada al siguiente renglon. En ambos casos, la variable WCURS es colocada en la columna cero. Para el segundo caso, cuando el caracter tabulador es pasado a la función wputchar, se despliegan 5 blancos a partir de la posición corriente del cursor contenida en la variable WCURS, la cual sera colocada al siguiente tope de tabulador en la ventana. Cuando se trate de otros caracteres, estos son desplegados en la ventana, y la variable WCURS es incrementada. Las cadenas de caracteres que excedan el ancho de la ventana serán truncadas.

BLOQUE (18)

- ANALISIS DE LA FUNCION:

void wcursor(WINDOW *wnd, int x, int y)

Cada ventana cuenta con una posición logica de cursor. Es decir, en el sistema de ventanas, cada ventana posee coordenadas propias, aunque relativas a las coordenadas reales de la pantalla. Al igual que las posiciones de cursor en la pantalla completa, la posicion lógica del cursor de ventana fluctua de la posicion 0,0 (esquina superior izquierda del area de texto de la ventana) a la posición correspondiente a la esquina inferior derecha del área de texto de la ventana. El objeto de la función que estamos analizando es precisamente "reposicionar" dicho cursor de ventana.
Las funciones **wputchar** y **wprintf** despliegan texto con relación a esta posición lógica del cursor.

- CUERPO DE LA FUNCION:
En primer termino, se actualizan los valores de las variables WCURS y SCROLL a las coordenadas especificadas por el programa que invoco a la función wcursor. Las coordenadas pasadas a esta funcion, ya vienen validadas (no exceden los limites de la ventana).
Antes de terminar, se hace una llamado la función **cursor()** para colocar el cursor en la localidad de pantalla relativa a la posición del cursor de ventana.

BLOQUE (19)

- ANALISIS DE LA FUNCION:

int get_selection(WINDOW *wnd, int s, char *keys)

Esta funcion permite utilizar una ventana como menu. Para utilizar esta función se debe establecer una ventana con antelacion y escribir algunas lineas de texto en ella (por medio de la funcion **wprintf**, por ejemplo). Optativamente, haciendo uso de la funcion **set colors**, también se puede establecer los valores de color ACCENT para la ventana (el *default* es letras negras sobre fondo blanco).

Una vez tomadas las acciones anteriores, se puede proceder a invocar la función get_selection(). Función que tiene como objeto manejar la ventana como un menu, en la que cada línea de texto representa una opción. Las opciones son "realzadas" por el color ACCENT. El entero s mantiene el número de la opción presente (según el movimiento del cursor sobre las líneas de la ventana) y es usado para colocar el cursor de barra de menu "realzado" sobre dicha opción (s=1, implica primera opción; s=2, segunda opción y así sucesivamente).
La barra de menu se puede mover hacia arriba o hacia abajo por medio de las teclas de dirección (*arrow keys*), la selección de menu se efectúa con la tecla *Enter*; y la tecla *Esc* es utilizada para salir del proceso.
El apuntador de teclas **keys** contiene la dirección de un arreglo de valores de tecla que puede ser usado para llevar a cabo selecciones en el menu. Algunos sistemas de menu permiten al usuario tener la opción de llevar a cabo selecciones haciendo uso de la barra de menú luminosa, o bien directamente introduciendo el pulso de tecla correspondiente a la opción deseada. El sistema aquí presentado será de este tipo con opción a funcionar sólo con barra de menu, para lo cual se deberá pasar al apuntador **keys** un valor nulo (NULL).
La función get selection devuelve un valor entero correspondiente al elemento seleccionado en el menu (el valor "1" indica primera opción); o bien el valor "0", en caso de haber sensado la tecla *Esc*. Simultáneamente y en caso de haberse presionado cualesquiera de las teclas "flecha derecha o izquierda", la función devolverá los valores de las variables **FWD** o **BS** (definidos en keys.h),

- CUERPO DE LA FUNCION:
La macro SELECT hace referencia a un miembro de la estructura WINDOW y es usada para seguir la localización del cursor de barra en la ventana. Las funciones **accent()** y **deaccent()** cambian el atributo de video de las líneas de selección para "encender" (ACCENT) o "apagar" (NORMAL) la barra de cursor. Conforme el usuario presione las teclas de "flecha arriba" o "flecha abajo", la función altera el valor de la variable SELECT. El programa que llama a la función get_selection puede opcionalmente pasar la dirección de un arreglo de caracteres que contiene una lista de "pulsos de tecla" para realizar las selecciones de menu directamente identificando los códigos de cada uno de ellos. Si el usuario presiona alguna de esas teclas, la selección correspondiente es llevada a cabo; tal y como sucedería si la barra del cursor estuviera colocada en la línea apropiada y la tecla *Enter* hubiera sido presionada.

Si la ventana existe en la lista:
- Se almacena en SELECT el numero de la opción seleccionada (1 es la primera, 2 la segunda, etc.).

Siempre que el caracter sea diferente de: *Esc*, *Enter*, flecha izquierda (←) y flecha derecha (→), se hace lo siguiente:
Se acentua (se realza) la linea indicada por SELECT.
Se lee un caracter desde el *buffer* de teclado y se almacena en la variable "c".
Se desacentua la linea indicada por SELECT.
El programa reacciona de forma distinta para cada uno de los siguientes casos:
Si el caracter almacenado en "c" es flecha hacia arriba (↑), entonces:
Se verifica que la barra del cursor (cursor de selección) no este posicionada en la primera opción del menú. Caso en el que: Sube la barra del cursor a la siguiente opción del menu. En caso contrario: Se coloca la barra del cursor en la última opción del menú.
Si el caracter almacenado en "c" es flecha para hacia (↓), entonces:
Se verifica que la barra del cursor no esté posicionada en la última opcion del menu. Caso en el que: se baja la barra del cursor a la siguiente opción del menú. En caso contrario: la barra del cursor es colocada en la primera opcion del menu.
Si el caracter almacenado en "c" es: *Esc*, *Enter*, flecha izquierda (←) o flecha derecha (→) entonces, por el momento no se lleva a cabo accion alguna.
Si el caracter almacenado en "c" es cualquier otro, entonces:
Se verifica que exista un apuntador al arreglo de teclas de seleccion. En caso afirmativo: se explora dicho arreglo para determinar si el caracter en "c" corresponde a alguno del arreglo, caso en el que la función get_selection devolverá el numero de opción de menú seleccionada. En caso contrario: si el apuntador al arreglo es nulo, la función get_selection devolverá lo siguiente: el valor de SELECT, si "c" = *Enter*; un "0" si "c" = *Esc*, o bien el contenido de "c", si esta variable contiene cualquier otro caracter.

BLOQUE (20)

- ANALISIS DE LA FUNCION: **void scroll(WINDOW *wnd, int dir)**
  Esta funcion hace el desplazamiento (*scroll*) en una linea hacia arriba o hacia abajo de la porción de texto de la ventana. Si la ventana en la que se realizará el *scroll* es la más recientemente establecida y está visible, entonces se utilizará la funcion de *scroll* de ROM BIOS debido a su mayor rapidez en relación con el *scroll* de *software*. Por el contrario, si la ventana no es la más recientemente establecida o si unicamente cuenta con sólo una línea de texto, el scroll se realizará mediante *software* utilizando las funciones **dget()** y **displ()** para leer y escribir caracteres de texto desde y hacia la ventana.

Nota: La razón por la que no se usara la función de *scroll* del ROM BIOS , cuando la ventana cuente unicamente con una línea de texto; radica en que tal situación genera un error en la IBM PC y en algunas AT compatibles; ocasionando un despliegue de video extrano. El error fue arreglado por IBM en el BIOS de la AT, pero las compañías fabricantes de *clonos* que imitaron a IBM acarreando el error, no pudieron solucionar el problema.

- CUERPO DE LA FUNCION:

Primero se inicializa la variable de renglon a la posición de la ultima linea del área de texto de la ventana.
Si la ventana existe en la lista:
Se verifica que la ventana direccionada sea la más reciente en el sistema, que tenga más de una linea de texto, y que esté visible en pantalla.

En caso afirmativo:

Se hace uso de la función de scroll del ROM BIOS, interrupción 10H funcion 06H ó 07H. En el primer caso (función 06H) se hace un scroll de ventana de la página activa hacia arriba, colocando una linea vacia (llena de Blancos) en el fondo de la ventana. En el segundo caso, se hace un scroll de ventana de la página activa hacia abajo, colocando una linea vacia en el tope de la ventana.
Los parámetros de entrada de la interrupción son:

```
ah = 06H;       si dir = UP
ah = 07H;       si dir = DN
al = 01H;       número de lineas a desplazar.
bh = WNORMAL;   atributo para la nueva linea vacia.
cl = COL+1;     columna y renglón de la esquina
ch = ROW+1;     superior izquierda del área de
                texto de la ventana.
dl = COL+WIDTH-2 ; columna y renglón de la esquina
dh = ROW+HEIGHT-2; inferior derecha del área de
                   texto de la ventana.
```

Una vez ejecutada la interrupción el control es devuelto al programa que invocó a la funcion.

En caso contrario:

Se lleva a cabo el scroll mediante nuestro *software*. Para ilustrarlo nos basaremos en una ventana de ejemplo de dimensiones especificas (ALTURA = 10 y ANCHO = 20)

Si el scroll sera realizado hacia arriba (dir= UP) se efectúa lo siguiente:

Los ciclos "for" delimitan el area de texto a ser desplazada. Todas las lineas de dicha area serán recorridas hacia arriba en una posición, empezando en la segunda linea y terminando en la última. Para ello, se hace uso de las funciones: dget() (para leer las lineas de la ventana, caracter por caracter, en su posición original) y displ() (para desplegar, caracter por caracter, las lineas leidas por la función anterior en su nueva

posición; la línea 2 será reescrita sobre la línea 1, la 3 sobre la 2, y así sucesivamente). Ver figura 6.3a.
Se utiliza un tercer ciclo para llenar con blancos la última línea del área de texto de la ventana. Ver figura 6.3b.

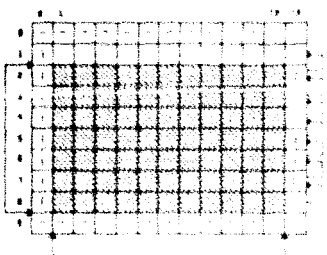
Si el scroll será realizado hacia abajo (dir= DN) se realiza lo siguiente:
Los ciclos "for" delimitan el área de texto a ser desplazada. Todas las líneas de dicha área serán recorridas hacia abajo en una posición, empezando por la penúltima línea y terminando en la primera. Para ello se hace uso de las funciones: dget() (para leer las líneas, caracter por caracter, en su posición original) y displ() (para desplegar las líneas leídas por la función anterior en su nueva posición; la línea 7 será reescrita sobre la línea 8, la 6 sobre la 7, y así sucesivamente. Ver fig. 6.4a.
Se utiliza un tercer ciclo para llenar con blancos la primer línea del área de texto de la ventana. Ver figura 6.4b.

BLOQUE (21)

- ANALISIS DE LA FUNCION:
  static int *waddr(WINDOW *wnd, int x, int y)

Esta función opera únicamente en un sistema de ventanas estratificadas. La función devuelve una dirección de tipo entero relativa a la posición (coordenadas "x,y") del caracter y su atributo dentro de la ventana direccionada. Si la ventana no es visible, la función regresa la dirección relativa a la posición del caracter dentro del buffer-de-salvado de dicha ventana. En caso de que la ventana se encuentre visible, se explora la lista ligada para determinar si existen ventanas posteriormente establecidas a la que se está direccionando, y así verificar si alguna de ellas está tapando la posición de caracter (x,y) pasada a la función waddr(). En caso afirmativo, la función devolverá la dirección correspondiente a la posición del caracter pero respecto al buffer-de-salvado de la ventana que esté cubriendo dicha posición de caracter. En el caso de que las coordenadas (x,y) de la ventana direccionada no estuvieran cubiertas por alguna ventana establecida con posterioridad, o que no existieran ventanas establecidas posteriormente a la direccionada; la función regresará un apuntador NULL indicando al programa que llamó a la función, que direccione la RAM de video.

FIG. 6.3 SCROLL HACIA ARRIBA

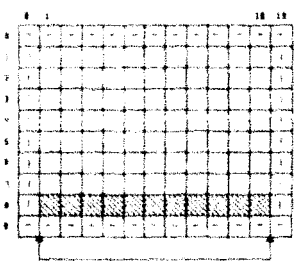


A) PRIMERA FASE :

EL AREA SOMBREADA SE TRASLADA UN RENGLON HACIA ARRIBA

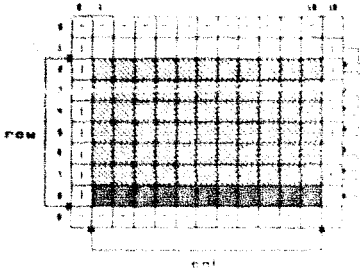
B) SEGUNDA FASE :

EL AREA SOMBREADA ES LLENADA CON BLANCOS



NOTA : EJEMPLO DE VENTANA CON ALTURA = 10 Y ANCHO = 20

FIG. 6.4 SCROLL HACIA ABAJO

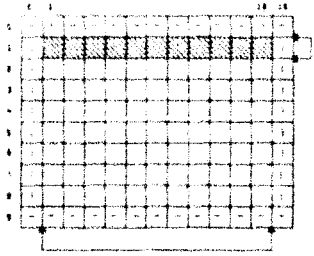


A) PRIMERA FASE

EL AREA SOMBREADA ES TRASLADADA UN RENGLON HACIA ABAJO

B) SEGUNDA FASE :

LA LINEA SOMBREADA ES LLENADA CON BLANCOS



NOTA : EJEMPLO DE VENTANA CON ALTURA = 10 Y ANCHO = 20

- CUERPO DE LA FUNCION:

Se declara un apuntador a una estructura de ventana (nxt) para almacenar la direccion de la ventana posterior a la que se encuentra direccionada en la lista ligada.
Si la ventana no esta visible:
La funcion regresa la direccion dentro del buffer de salvado, correspondiente a la posicion de caracter deseada de la ventana direccionada. Esta direccion es obtenida a partir de la direccion base del buffer-de-salvado, mantenida por el apuntador SAV y el offset a la posicion de caracter, calculado a partir de las coordenadas (x,y) y el ancho de la ventana.
Si la ventana se encuentra visible:
Las coordenadas (x,y) son colocadas a su posicion correspondiente dentro de la ventana direccionada.
Mientras exista una ventana posterior a la direccionada (nxt diferente de 0):

- Se verifica si la posicion de caracter deseada se encuentra visible (en RAM de video) o si permanece tapada por alguna de las ventanas posteriores. Para ello se efectuan 2 "IF'S". El primero verifica si la ventana siguiente (en la lista ligada) se encuentra visible, el segundo si la coordenada en "x" dada se encuentra dentro de los limites del ancho de dicha ventana, y el tercero si la coordenada en "y" no sobrepasa la altura de la ventana.
- En caso de que la posicion de caracter coincida con el área de despliegue de alguna ventana posterior:
  Las coordenadas (x,y) son ajustadas a la posición que les correspondería dentro de la ventana que las cubre.
  La funcion regresa la direccion correspondiente a las coordenadas (x,y) que le fueron pasadas, dentro del buffer de salvado de la ventana que cubre dicha posicion de caracter. Esta direccion es obtenida a partir de la direccion base del buffer-de-salvado, mantenida por el apuntador nxt->ws y el offset a la posición de caracter, calculado a partir de las coordenadas (x,y) y el ancho de la ventana siguiente.

En caso que las coordenadas (x,y) no coincidan con el área de despliegue de alguna ventana posterior, o no existan ventanas posteriores a la direccionada:
La funcion regresa un apuntador nulo, indicando que se debe direccionar la RAM de video directamente.

BLOQUE (22)

- ANALISIS DE LA FUNCION:
  void displ(WINDOW *wnd, int x, int y, int ch, int at)

Esta función es llamada para desplegar un caracter y su atributo de video en una ventana. Solo opera en un ambiente de ventanas estratificadas. Las coordenadas (x,y) indican la

posicion en la ventana donde se desea desplegar el caracter **ch** con atributo **at**.

- CUERPO DE LA FUNCION:
La función hace un llamado a **waddr** para determinar si el caracter debera ser escrito hacia un area de salvado o hacia la RAM de video.
Primero se almacena en la variable **vch** el atributo y caracter a escribir.
En seguida se verifica si la ventana direccionada está visible y si las coordenadas (x,y) se encuentran cubiertas por alguna ventana posterior, o si la ventana direccionada no esta visible. En caso afirmativo: en la dirección regresada por la función **waddr** se almacena el contenido de la variable **vch**. En caso contrario: se direcciona directamente la RAM de video y se utiliza la función **vpoke** para almacenar el atributo y caracter en la RAM de video. Los parametros de entrada de la función **vpoke** son: segmento de video, direccion (adr) y atributo-caracter. La variable **adr** representa un offset calculado por la macro **vad(x,y)** dadas las coordenadas (x,y). Su definicion es:

```
vad(x,y) = ((y) * 160 + (x) * 2).
```

BLOQUE (23)

- ANALISIS DE LA FUNCION:

```
static int dget(WINDOW *wnd, int x, int y)
```

Esta función es llamada para recuperar un caracter y su atributo de video desde una ventana. Sólo opera en un ambiente de ventanas estratificadas.

- CUERPO DE LA FUNCION:
La función hace un llamado a **waddr()** para determinar si el caracter debera ser leido de un area de salvado o de la RAM de video.
Primero se verifica si la ventana direccionada está visible y si las coordenadas (x,y) se encuentran cubiertas por alguna ventana posterior, o si la ventana direccionada no está visible. En caso afirmativo: la función devuelve el caracter almacenado en la direccion dada por **waddr()**. En caso contrario: se obtiene un caracter desde la RAM de video por medio de la funcion **vpeek()**. En esta función también hace uso de la macro **vad()** para calcular el offset del caracter.

BLOQUE (24)

- ANALISIS DE LA FUNCION: **static vswap(WINDOW *wnd)**
Esta funcion intercambia el contenido del buffer de salvado de una ventana estratificada con la RAM de video, o bien con los buffers-de-salvado de las ventanas posteriormente establecidas que cubran la ventana direccionada. Para

efectuar el intercambio se hace uso de las funciones **displ()** y **dget()**.

- CUERPO DE LA FUNCION:
Se declara un apuntador a enteros inicializado con el apuntador al buffer-de-salvado de la ventana direccionada. Mediante dos ciclos "FOR" se van variando las coordenadas correspondientes para: a) leer el contenido del buffer de salvado e irlo almacenarlo en la variable **chat** en cada iteracion; b) leer el contenido de la RAM de video mediante la funcion dget - cada caracter leido es almacenado en el buffer de salvado -; c) para desplegar el contenido de la variable **chat** en la RAM de video haciendo uso de la función **displ()**

BLOQUE (25)

- ANALISIS DE LA FUNCION: **static vsave(WINDOW *wnd)**
Esta función trabaja en un ambiente de ventanas apiladas. Su objeto es copiar el contenido de la RAM de video al buffer_de_salvado de la ventana direccionada.

- CUERPO DE LA FUNCION:
Se declara un apuntador a enteros inicializado con el apuntador al buffer_de_salvado de la ventana direccionada. Se procede a leer el contenido de la RAM de video (caracter por caracter) por medio de la funcion **vpeek()**, "al tiempo" de leer cada caracter, éste es almacenado en el buffer_de_salvado de la ventana direccionada).

BLOQUE (26)

- ANALISIS DE LA FUNCION: **static vrstr(WINDOW *wnd)**
Esta función trabaja en un ambiente de ventanas apiladas. Su objeto es copiar el contenido del buffer-de-salvado de la ventana direccionada en la RAM de video.

- CUERPO DE LA FUNCION:
Se declara un apuntador a enteros inicializado con el apuntador al buffer-de-salvado de la ventana direccionada. Mediante los ciclos anidados y la función **vpoke()** se escribe (caracter por caracter) el contenido del buffer-de-salvado de la ventana direccionada en la RAM de video.

BLOQUE (27)

- ANALISIS DE LA FUNCION: **void acline(WINDOW *wnd, int set)**
Esta funcion es llamada por las macros accent() y deaccent() para cambiar la configuración de color (ACCENT o NORMAL) de una línea de la ventana. La función acline() trabaja en sistemas de ventanas apiladas o estratificadas.

- CUERPO DE LA FUNCION:
  Si la ventana existe en la lista ligada:
  Por medio de la funcion dget() se lee cada uno de los caracteres de la linea seleccionada; en cada iteración del ciclo, el caracter presente es almacenado en la variable **ch** (sin incluir su atributo) e inmediatamente, haciendo uso de la función displ(), es vuelto a escribir con su nuevo atributo (WACCENT "acentuado" o WNORMAL "desacentuado") segun el parametro set. Asi al finalizar el ciclo la linea seleccionada habra cambiado de atributo.

NOTA: Las 4 bloques siguientes que veremos (del 2 8 al 31), corresponden a funciones de listas ligadas. Para todos ellos emplearemos la siguiente notación.
P(wnd).- para representar la direccion de la ventana previa a la direccionada dentro de la lista ligada.
S(wnd).- para representar la dirección de la ventana posterior a la direccionada dentro de la lista ligada.
Las letras "A, B, C,...." denotan direcciones de ventana.
"wnd1 ← wnd2"; para denotar que al apuntador wnd1 se le asignará la dirección almacenada en wnd2.

Aprovecharemos tambien este paréntesis para recordar que ***listhead*** es el apuntador a la primera estructura de ventana en la lista; y ***listtail***, el apuntador a la ultima estructura de ventana en la lista.

BLOQUE (28)

- ANALISIS DE LA FUNCION: **static add list(WINDOW *wnd)**
  Esta función agrega la estructura WINDOW direccionada por el apuntador wnd, al final de la lista ligada.

- CUERPO DE LA FUNCION:
  Se verifica si la ventana direccionada no sera la primera de la lista.
  - En caso afirmativo:
    Se procede segun el siguiente ejemplo. Se tiene una lista ligada con 3 estructuras de ventana (A, B y C) y se desea agregar una cuarta (D).

    Estado original de la lista: A ↔ B ↔ C → NULL
    ↑
    listtail

    Para esta función interesa actualizar las direcciones de los apuntadores PREV y/o NEXT de las ventanas afectadas (C y D) por la accion de agregar una nueva

ventana al final de la lista; así como la dirección de los apuntadores listtail y listhead.
Siguiendo el código:
• Actualización de apuntadores de ventana.

D
- P(D) ← listtail = C
- S(D) ← NULL (por estar al final de la lista)

C
- listtail→_nx = S(C) ← D

• Actualización de apuntadores de lista:
listtail ← D

Estado final de la lista: A ↔ B ↔ C ↔ D → NULL
↑
listtail

- En caso contrario (la ventana D será la primera en la lista):
• Actualización de apuntadores de lista:
listtail • listhead ← D

BLOQUE (29)

- ANALISIS DE LA FUNCION: static beg_list(WINDOW *wnd)
Esta función agrega la estructura WINDOW direccionada por el apuntador wnd, al inicio de la lista ligada.

- CUERPO DE LA FUNCION:
Se verifica si la lista no se encuentra vacía.
- En caso afirmativo:
Consideremos el siguiente ejemplo. Se tiene una lista ligada con 3 estructuras de ventana (A, B y C) y se desea agregar una cuarta (D) al inicio de la lista.

Estado original de la lista: A ↔ B ↔ C → NULL
↑
listhead

Las ventanas afectadas en este caso serán: A y D, así como los apuntadores listhead y listtail.
Siguiendo el código:
• Actualización de apuntadores de ventana.

D
- P(D) ← NULL (esta al inicio de la lista)
- S(D) ← listhead = A

A
- listhead→_pv = P(A) ← D

• Actualización de apuntadores de lista:
listhead ← D

Estado final de la lista: D ↔ A ↔ B ↔ C → NULL
↑
listhead

- En caso contrario (la ventana D será la unica en la lista):
• Actualización de apuntadores de lista:
listhead ← listtail ← D

BLOQUE (10)

- ANALISIS DE LA FUNCION: **static remove list(WINDOW *wnd)**
Esta función quita de la lista ligada la estructura WINDOW direccionada por el apuntador wnd.

- CUERPO DE LA FUNCION:
- Si la ventana a remover se encuentra intermedia en la lista:
Consideremos el siguiente ejemplo. Se tiene una lista ligada con las ventanas A, B, C y D; de la que se desea remover la ventana B.

Estado original de la lista: A ↔ B ↔ C ↔ D → NULL
↑ ↑
listhead listtail

Las ventanas afectadas en este caso seran: A, B y C.
Siguiendo el código:
• Actualización de apuntadores de ventana.

C { P(S(B)) ← P(B) :. P(C) ← A

A { S(P(B)) ← S(B) :. S(A) ← C

B { P(B) ← NULL
S(B) ← NULL

Estado final de la lista: A ↔ C ↔ D → NULL
↑ ↑
listhead listtail

- Si la ventana a remover se encuentra al inicio de la lista se procederá de la siguiente manera: Supóngase la lista siguiente de la cual se desea remover la ventana A:

A ↔ B ↔ C ↔ D → NULL
↑ ↑
listhead listtail

Las ventanas afectadas en este caso serán: A y B; así como el apuntador listhead.
Siguiendo cada paso del código:
• Actualizacion de apuntadores de ventana.

B { P(S(A)) ← P(A) :. P(B) ← NULL

• Actualizacion de los apuntadores de lista:
listhead ← S(A) = B.

A { P(A) ← NULL
S(A) ← NULL

Estado final de la lista: B ↔ C ↔ D → NULL
↑ ↑
listhead listtail

- Si la ventana a remover se encuentra al final de la lista se procederá de la siguiente manera: Se tiene la lista siguiente de la cual se desea remover la ventana D:

A ↔ B ↔ C ↔ D → NULL
↑ ↑
listhead listtail

Las ventanas afectadas en este caso serán: C y D; así como el apuntador listtail.
Siguiendo cada paso del código:
• Actualizacion de apuntadores de ventana.

C { S(P(D)) ← S(D) :. S(C) ← NULL

• Actualizacion de los apuntadores de lista:
listtail ← P(D) = C.

D { P(D) ← NULL
S(D) ← NULL

Estado final de la lista: A ↔ B ↔ C → NULL
↑ ↑
listhead listtail

BLOQUE (21)

- **ANALISIS DE LA FUNCION:**
  static insert_list(WINDOW *w1, WINDOW *w2)
  Esta función inserta una estructura WINDOW (w1) dentro de la lista ligada a continuación de otra estructura WINDOW especificada (w2).

- CUERPO DE LA FUNCION:
  - Si la ventana a insertar será colocada al final de la lista, se procede según el siguiente ejemplo:
    Se tiene una lista ligada con 3 estructuras de ventana (A, B, y C) en la que se desea insertar una cuarta (D) a continuación de la ventana C.

    Estado original de la lista: A ↔ B ↔ C → NULL
    ↑
    listtail

    Las ventanas afectadas en este caso serán: C y D; así como el apuntador listtail.
    Siguiendo cada paso del código:

    • Actualización de apuntadores de ventana.

    D
    - w1→_pv ← w2 :. P(D) ← C
    - w1→_nx ← w2→_nx :. S(D) ← S(C) = NULL

    C
    - w2→_nx ← w1 :. S(C) ← D

    • Actualización de apuntadores de lista:
      listtail ← D

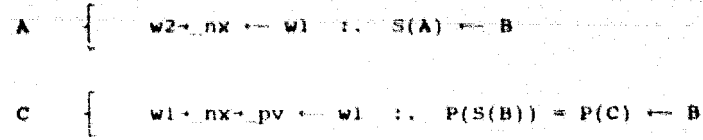
    Estado final de la lista: A ↔ B ↔ C ↔ D → NULL
    ↑
    listtail

  - Si la ventana a insertar será colocada en una parte intermedia de lista, se procede según el siguiente ejemplo:
    Considerese el estado de la lista siguiente, a la que se insertará la ventana B.

    Estado original de la lista: A ↔ C ↔ D → NULL

    Las ventanas afectadas en este caso serán: A, B y C.
    Siguiendo el código:
    • Actualización de apuntadores de ventana.

    B
    - w1→_pv ← w2 :. P(B) ← A
    - w1→_nx ← w2→_nx :. S(B) ← S(A) = C



BLOQUE (32)

- ANALISIS DE LA FUNCION: **static verify wnd(WINDOW **wl)**
Función exclusiva para ventanas estratificadas. Su objetivo es buscar en la lista ligada la dirección a una estructura WINDOW especificada. La función regresa un valor verdadero o falso reflejando la presencia o ausencia de la estructura WINDOW en la lista. Si el apuntador especificado a la estructura WINDOW contiene un valor nulo, la función regresa la dirección de la estructura WINDOW más reciente (la última) en la lista. El parametro **wl de esta función es un apuntador a apuntador a una estructura WINDOW. El formato seguido para invocar esta función es: verify wnd(&wnd)

- CUERPO DE LA FUNCION:
Primero se declara un apuntador auxiliar (wnd) a una estructura WINDOW; el cual es inicializado a la dirección de la primera estructura de ventana en la lista (listhead); esto, para tener una referencia a partir de la cual se explorará la lista. A continuación se verifica si el apuntador wl especificado tiene un valor nulo. En caso afirmativo: la función devolverá en wl la dirección de la ultima estructura de ventana en la lista (listtail), En caso contrario: la lista ligada será explorada actualizando la dirección del apuntador wnd y comparandola con el contenido de wl a fin de verificar la existencia de la estructura de ventana a la que este apunta.
La funcion regresara un valor "falso" si la dirección wl no fue encontrada, y un valor "verdadero" en caso contrario.

BLOQUE (33)

- ANALISIS DE LA FUNCION: **void error message(char *s)**
Esta función despliega el mensaje de error apuntado por "s", a la vez que hace sonar una alarma. El mensaje es desplegado en una ventana en el cuadrante inferior derecho de la pantalla. Este mensaje permanece sobre la pantalla aun despues de que la funcion haya regresado.

- CUERPO DE LA FUNCION:
  Se establece la ventana de error.
  Se colocan los colores de ventana (de preferencia llamativos, en nuestro caso: rojo y amarillo).
  Se coloca el titulo de la ventana.
  Se despliega la ventana
  Se escribe el mensaje de error dentro de la ventana mediante la funcion wprintf.
  Finalmente se hace sonar la alarma por medio de la función putchar(BELL) de Turbo C.

BLOQUE (34)

- ANALISIS DE LA FUNCION: **void clear_message()**
  Esta función quita el ultimo mensaje de error desplegado por la funcion error_message.

- CUERPO DE LA FUNCION:
  Se verifica la existencia de alguna ventana de error.
  En caso afirmativo: remueve dicha ventana tanto de la lista como de pantalla. En caso contrario, simplemente regresa al programa que la invocó.

PROGRAMAS DE EJEMPLO QUE HACEN USO DE LA LIBRERIA DE FUNCIONES DE VENTANA

Con el fin de mostrar al programador la forma de utilizar las funciones de libreria de ventana, en este apartado incluiremos algunos programas ejemplo. Cada uno de ellos consta de 3 módulos: un programa manejador, otro con el cuerpo de la función principal a la que está orientado el ejemplo (que ejecuta ciertas funciones de la libreria de ventanas), y un archivo "proyecto" (extensión prj), que Turbo C utiliza para construir el programa ejecutable.

Los programas ejemplo, en este apartado incluidos, aunados a los que se verán en los capitulos posteriores, serán integrados en un sólo módulo ejecutable que ilustrara el funcionamiento de todas las modalidades de ventanas. (Remitirse al capitulo IX).

■ **EJEMPLO No.1**
PROGRAMA QUE MUEVE UNA VENTANA EN DOS DIMENSIONES.

Para llevar a cabo el desplazamiento de ventanas, este programa hace uso de las funciones move_window y rmove_window. Funciones de uso exclusivo en un ambiente de ventanas estratificadas.

LISTADO #5

LISTADO 5.1
Programa Manejador.

```
/* ------ move.c ------ */
void testmove(void);

void main()
{
     testmove();
}
```

LISTADO 5.2
Programa que contiene el código de la función principal.

```
/* ------------ testmove.c ------------ */

#include "twindow.h"
#include "keys.h"

void testmove()
{
     WINDOW *wndA, *wndB, *wndC;
     int c;

     wndA = establish_window(5, 5, 9, 19);
     wndB = establish_window(10, 1, 9, 23);
     wndC = establish_window(11, 8, 9, 12);
     set_colors(wndA, ALL, RED, YELLOW, BRIGHT);
     set_colors(wndB, ALL, AQUA, YELLOW, BRIGHT);
     set_colors(wndC, ALL, WHITE, YELLOW, BRIGHT);
     display_window(wndA);
     display_window(wndB);
     display_window(wndC);
     wprintf(wndB, "\n    Hay algo mas    ");
     wprintf(wndB, "\n importante que la  ");
     wprintf(wndB, "\n     logica:    ");
     wprintf(wndB, "\n  La imaginacion.");
     wprintf(wndB, "\n\n Alfred Hitchcock.");
     do   {
          int x = 0, y = 0;
          c = get_char();
          switch (c)     {
               case FWD: x++;
                         break;
               case BS:  --x;
                         break;
               case UP:  --y;
                         break;
               case DN:  y++;
               default:  break;
          }
          if (x || y)
               rmove_window(wndB, x, y);
     } while (c != ESC);
     delete_window(wndA);
     get_char();
     delete_window(wndC);
     get_char();
     delete_window(wndB);
```

LISTADO 5.3
Programa que construye el archivo ejecutable.

```
/* ------ move.prj ------ */

move
testmove (twindow.h, keys.h)
twindow (twindow.h, keys.h)
ibmpc.obj
```

COMENTARIOS AL LISTADO #5

Para ilustrar el mecanismo de desplazamiento de ventanas, el programa testmove.c establece 3 ventanas, les asigna colores, y antes de desplegarlas, escribe un parrafo dentro de la segunda ventana (mostrando cómo se puede escribir texto en una ventana parcialmente cubierta por otra). Después de correr el programa, p éste esperará a que se introduzcan los pulsos de tecla siguientes: teclas de dirección (↑, ↓, →, ←) o la tecla "Esc".

Cada vez que se presione una tecla de dirección, la ventana intermedia (que contiene el texto) será movida en una posición de caracter de acuerdo a la dirección indicada. Al presionar la tecla "Esc" la primer ventana establecida será eliminada; y las otras, presionando (a continuación) cualesquiera otros 2 pulsos de tecla. Con lo que finalmente el programa terminará y regresará el control al sistema operativo.

Observaciones: El movimiento de ventanas aquí ejemplificado, ilustra el concepto de los buffers-de-salvado. A este respecto cabe aclarar que, debido al tiempo de procesamiento requerido para examinar cada buffer cuando un caracter es escrito en una ventana, la ejecución de las funciones involucradas, puede degradarse en tanto mayor sea el tamaño y número de ventanas en el sistema; por el incremento en tiempo de procesamiento que esto representa. Esta deficiencia será notoria si se trabaja en una máquina basada en el micro 8086.

■ **EJEMPLO No. 2**

PROGRAMA QUE TRASLADA UNA VENTANA ATRAS O AL FRENTE DE TODAS.

LISTADO #6

LISTADO 6.1
Programa manejador.

```
/* ------ prom.c ------ */
void promote(void);
void main()
{
     promote();
}
```

LISTADO 6.2
Programa que contiene el código de la función principal.

```
/* ------------ promote.c ------------ */
#include "twindow.h"
#include "keys.h"

void promote()
{
     WINDOW *wndA, *wndB, *wndC;
     int c;

     wndA = establish_window(5, 5, 9, 19);
     wndB = establish_window(10, 3, 9, 20);
     wndC = establish_window(13, 8, 9, 12);
     set_colors(wndA, ALL, RED, YELLOW, BRIGHT);
     set_colors(wndB, ALL, AQUA, YELLOW, BRIGHT);
     set_colors(wndC, ALL, WHITE, YELLOW, BRIGHT);
     display_window(wndA);
     display_window(wndB);
     display_window(wndC);
     wprintf(wndA, "\n\n Ventana A");
     wprintf(wndB, "\n\n Ventana B");
     wprintf(wndC, "\n\n Ventana C");
     do   {
          c = get_char();
          switch (c)     {
               case 'a': forefront(wndA);
                         break;
               case 'b': forefront(wndB);
                         break;
               case 'c': forefront(wndC);
                         break;
               case 'A': rear_window(wndA);
                         break;
```

```
                case 'B': rear_window(wndB);
                          break;
                case 'C': rear_window(wndC);
                          break;
                default:  break;
            }
        } while (c != ESC);
        delete_window(wndA);
        get_char();
        delete_window(wndC);
        get_char();
        delete_window(wndB);
}
```

LISTADO 6.3
Programa que construye el archivo ejecutable.

```
/* ------ prom.prj ------ */

prom
promote (twindow.h, keys.h)
twindow (twindow.h, keys.h)
ibmpc.obj
```

■ COMENTARIOS AL LISTADO #6.

El listado 6.2 "promote.c" utiliza el mismo patrón de 3 ventanas empleado en el programa de ejemplo anterior, con la diferencia de que, en este caso, se escribirá texto en cada una de las ventanas.

Después de correr el programa, este reaccionará de la siguiente manera: Al sensar los pulsos de tecla "a", "b" o "c", colocara la ventana asociada a esa letra, al frente de las demás. Si se presionan las teclas "A", "B" o "C" trasladará la ventana indicada atras de las demás. Si se presiona la tecla "Esc" eliminara la primer ventana establecida. Y cualesquiera otros dos pulsos subsiquientes provocaran la eliminación de las otras ventanas. Con lo que finalmente el programa terminará y regresara el control al sistema operativo.

La función foretront es utilizada para adelantar una ventana a la posición más corriente colocandola encima de todas las demás, y la función rear_window, para atrasar una ventana a la posición menos reciente, colocando debajo de las demás. Esta característica cobra importancia en un sistema de numerosas ventanas en el que el usuario requiere cambiar continuamente de una ventana a otra.

EJEMPLO No. 3

PROGRAMA QUE ASIGNA TITULOS Y CAMBIA LOS COLORES DE VENTANA.

LISTADO #7

LISTADO 7.1

Programa Manejador.

```
/* -------- color.c ------- */
void ccolor(void);
void main()
{
     ccolor();
}
```

LISTADO 7.2

Programa que contiene el codigo de la funcion principal.

```
/* ------------ ccolor.c ------------ */

#include "twindow.h"
#include "keys.h"

void ccolor()
{
     WINDOW *wndA, *wndB, *wndC;
     int c;

     wndA = establish window(8, 8, 9, 19);
     wndB = establish window(13, 6, 9, 20);
     wndC = establish window(16, 11, 9, 12);
     set colors(wndA, ALL, RED, YELLOW, BRIGHT);
     set colors(wndB, ALL, AQUA, YELLOW, BRIGHT);
     set colors(wndC, ALL, WHITE, YELLOW, BRIGHT);
     display window(wndA);
     display window(wndB);
     display window(wndC);
     do   {
          c = get char();
          switch (c)     {
               case 'r':
                    set title(wndB, " ROJO ");
                    set colors(wndB, ALL, RED, WHITE, BRIGHT);
                    break;
               case 'a':
                    set title(wndB, " AZUL ");
                    set colors(wndB, ALL, BLUE, WHITE, BRIGHT);
                    break;
               case 'v':
                    set title(wndB, " VERDE ");
```

```
                    set_colors(wndB, ALL, GREEN, WHITE, BRIGHT);
                    break;
               default:
                    break;
          }
     } while (c != ESC);
     delete_window(wndA);
     get_char();
     delete_window(wndC);
     get_char();
     delete_window(wndB);
}
```

LISTADO 7.3
Programa que construye el archivo ejecutable.

```
/* -------- color.prj ------- */

color
ccolor (twindow.h, keys.h)
twindow (twindow.h, keys.h)
ibmpc.obj
```

■ COMENTARIOS AL LISTADO #7.

El listado 7.2, se basa en el mismo patron de 3 ventanas de los ejemplos anteriores (misma posicion, mismo tamaño), pero estará orientado a demostrar la asignación de titulos y colores de ventana. Cada ventana será desplegada con un color determinado y sin titulo. La ventana intermedia sera el objeto de la demostracion; al presionar los pulsos de tecla "r", "a" o "v", aparecerá el titulo de dicha ventana ("ROJO", "AZUL" o "VERDE" segun la tecla pulsada), a la vez que cambiará de color al indicado por su nuevo titulo. Para salir del proceso se deberá presionar la tecla "Esc", con lo que la primer ventana establecida será eliminada, cualesquiera otros dos pulsos de tecla subsecuentes provocaran la eliminacion de las demas ventanas. Con lo que finalmente el programa terminará y regresará el control al sistema operativo.

■ EJEMPLO No. 4
PROGRAMA QUE REUNE LAS FUNCIONES ESENCIALES DE LA LIBRERIA DE VENTANAS.

El siguiente ejemplo combina las caracteristicas de ventanas mostradas en los ejemplos anteriores agregando tres caracteristicas mas: el uso de la funcion get_selection, para procesar un menu vertical; el uso de la funcion hide_window para ocultar la ventana de menu, y la funcion set_intensity, para cambiar la intensidad del primer palno de las ventanas.

LISTADO #8

LISTADO 8.1
Programa Manejador.

```
/* ---- tabla.c ------ */

#include "twindow.h"
void funtabla(void);
void main()
{
      load_help("tcprogs.hlp");
      funtabla();
}
```

LISTADO 8.2
Programa que contiene el código de la función principal.

```
/* ---------------- funtabla.c ----------------- */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "twindow.h"
#include "keys.h"

/* -- prototipos locales -- */
void get_tabla(int s);
int ht (char **tb);
int wd (char **tb);
char *titles [ ] = {                                          (a)
      " 1: FUNCIONES TRIGONOMETRICAS ",
      " 2: VALORES DE ANGULOS ",
      " 3: RELACIONES TRIGONOMETRICAS ",
      " 4: NUMEROS COMPLEJOS ",
      " 5: CONSTANTES NOTABLES ",0
};
WINDOW *pno [ ] = {0, 0, 0, 0, 0};                            (b)
static int x [ ] = {20, 15, 29, 10, 17};
static int y [ ] = {5, 10, 13, 18, 6};
```

```
static int wcl [] [2] = {      (BLUE, WHITE),
                               (MAGENTA, WHITE),
                               (RED, WHITE),
                               (GREEN, WHITE),
                               (AQUA, WHITE)  };
char *tabla1 [] = {                                              (c)
     "senA + senB = 2 sen ½(A + B) cos ½(A + B)",
     "senA - senB = 2 cos ½(A + B) sen ½(A + B)",
     "cosA + cosB = 2 cos ½(A + B) cos ½(A - B)",
     "cosA - cosB = 2 sen ½(A + B) sen ½(B - A)",
     "",
     "senA senB = ½[cos(A - B) - cos(A + B)]",
     "cosA cosB = ½[cos(A - B) + cos(A + B)]",
     "senA cosB = ½[sen(A - B) + sen(A + B)]",
     0
};
char *tabla2 [] = {
     "Angulo en Grados        Angulo en Radianes",
     "       0°                        0",
     "      15°                       π/12",
     "      30°                       π/6",
     "      45°                       π/4",
     "      60°                       π/3",
     "      90°                       π/2",
     "     120°                      2π/3",
     "     135°                      3π/4",
     "     180°                       π",
     "     210°                      7π/6",
     "     270°                      3π/2",
     "     360°                      2π",
     0
};
char *tabla3 [] = {
     " tanA = senA / cosA ",
     " cotA = 1 / tanA = cosA / senA ",
     " secA = 1 / cosA ",
     " cscA = 1 / senA ",
     " sen²A + cos²A = 1 ",
     " csc²A - cot²A = 1 ",
     0
};
char *tabla4 [] = {
     "(a+bi) + (c+di) = (a+c) + (b+d)i",
     "(a+bi) - (c+di) = (a-c) + (b-d)i",
     "(a+bi) (c+di) = (ac-bd) + (ad+bc)i",
     "(a+bi)/(c+di) = [(ac+bd)/(c²+d²)] +",
     "                [(bc-ad)/(c²+d²)]i",
     "",
     "Si x = r cosθ, y = r senθ; entonces:",
     "x + iy = r(cosθ + isenθ)",
     "donde r = √(x²+y²)",
     "[r1(cosθ1+isenθ1)] [r2(cosθ2+isenθ2)] =",
     "r1r2[cos(θ1+θ2) + isen(θ1+θ2)]",
     "[r1(cosθ1+isenθ1)] / [r2(cosθ2+isenθ2)] =",
```

```
        "(r1/r2) [cos(θ1-θ2) + isen(θ1-θ2)]",
        0
};
char *tabla5 [] = {
        " π = 3.14159 26535 89793 23...",
        " e = 2.71828 18284 59045 23...",
        "√2 = 1.41421 35623 73095 04...",
        "√3 = 1.73205 08075 68877 29...",
        "√5 = 2.23606 79774 99789 69...",
        "",
        "ln 10 = 2.30258 50929 94...",
        "ln  2 = 0.69314 71805 59...",
        "ln  3 = 1.09861 22886 68...",
        "",
        "1 radian = 180°/π = 57.29577...",
        "1° = π/ 180 rad = 0.017453...",
        "",
        "√e = 1.64872 12707 00128 14...",
        "√π = 1.77245 38509 05516 02...",
        0
};                                                          (d)
char **tabla [] = {tabla1,tabla2,tabla3,tabla4,tabla5,0};

void funtabla()                                             (e)
{
        int s = 0, i, c;
        WINDOW *mn;
        char **cp;

        cursor(0, 25);
        mn = establish_window(0, 0, 7, 35);
        set_title(mn, " Selección de Tabla ");
        set_colors(mn, ALL, BLUE, GREEN, BRIGHT);
        set_colors(mn, ACCENT, GREEN, WHITE, BRIGHT);
        display_window(mn);
        cp = titles;
        while (*cp)
                wprintf(mn, "\n%s", *cp++);
        while (1) {
                set_help("tablamen", 30, 10);
                s = get_selection(mn, s+1, "12345");
                if (s == 0)
                        break;
                if (s == FWD || s == BS) {
                        s = 0;
                        continue;
                }
                hide_window(mn);
                get_tabla(--s);
                c = 0;
                set_help("tablas  ", 5, 15);
                while (c != ESC)    {
                        c = get_char();
                        switch (c)      {
```

```
                    case FWD: rmove_window(pno[s], 1, 0);
                              break;
                    case BS:  rmove_window(pno[s], -1, 0);
                              break;
                    case UP:  rmove_window(pno[s], 0, -1);
                              break;
                    case DN:  rmove_window(pno[s], 0, 1);
                              break;
                    case DEL: delete_window(pno[s]);
                              pno[s] = NULL;
                              break;
                    case '+': forefront(pno[s]);
                              break;
                    case '-': rear_window(pno[s]);
                    default:  break;
               }
               if (c > '0' && c < '6')
                    get_tabla(s = c - '1');
          }
          forefront(mn);
          display_window(mn);
     }
     close_all();
     for (i = 0; i < 5; i++)
          pno[i] = NULL;
}
/* -- activa una tabla por numero de opción -- */
static void get_tabla(int s)
{
     char **cp;
     static int lastp = -1;
     if (lastp != -1)
          set_intensity(pno[lastp], DIM);
     lastp = s;
     if (pno[s])
          set_intensity(pno[s], BRIGHT);
     else {
          pno[s] = establish_window
               (x[s], y[s], ht(tabla[s]), wd(tabla[s]));
          set_title(pno[s], titles[s]);
          set_colors(pno[s],AIL,wcl[s][0],wcl[s][1],BRIGHT);
          set_border(pno[s], 1);
          display_window(pno[s]);
          cp = tabla[s];
          while (*cp)
               wprintf(pno[s], "\n %s", *cp++);
     }
}

/* -- calcula la altura de la ventana a desplegar -- */      (f)
static int ht(char **tb)
{
     int h = 0;
     while (*(tb + h++)) ;
```

```
    return h * 3;
}
/* -- calcula el ancho de la ventana a desplegar -- */
static int wd(char **tb)
{
    int w = 0;
    while (*tb)          {
        w = max(w, strlen(*tb));
        tb++;
    }
    return w + 4;
}
```

LISTADO 8.3
Programa que construye el archivo ejecutable.

```
/* ---- tabla.prj ------ */

tabla
funtabla (twindow.h, keys.h)
thelp (twindow.h, keys.h)
twindow (twindow.h, keys.h)
ibmpc.obj
```

■ COMENTARIOS AL LISTADO No 8.

El programa se basará en un patrón de 5 ventanas (tablas de funciones matematicas) sobre las que se podrá efectuar las operaciones a continuación enumeradas. Primeramente el programa desplegara un menu de ventana que listara los titulos de las tablas; permitiendo seleccionar alguna de ellas posicionando la barra de cursor en la opcion deseada y presionando la tecla Enter, o bien, simplemente tecleando el numero de opcion asociado a la tabla. La tabla seleccionada sera desplegada pudiendo también serlo las demas, tecleando su numero de opcion asociado. La ultima tabla seleccionada será resaltada con intensidad brillante, indicando que es la ventana corriente sobre la que se podrán efectuar las operaciones. Para colocar la ventana corriente enfrente de las demas se deberá presionar la tecla "+"; para enviarla al fondo, la tecla "-"; para moverla en dos dimensiones, las teclas de direccion, y para destruirla, la tecla "Del". Si se desea recuperar alguna tabla previamente eliminada, simplemente se deberá teclear su numero asociado. Para regresar al menu, se deberá presionar la tecla "Esc", misma que al ser nuevamente presionada provocara la terminación del proceso.

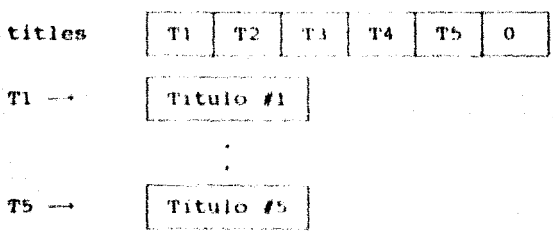
En este programa la tecla F1 es utilizada como una tecla de función de ayuda de contexto sensitivo, que provocará el despliegue de una ventana con un mensaje de ayuda relativo a lo que, en ese momento, esté realizando el usuario. En el capitulo VII se mostrará, detalladamente, como incluir esta caracteristica.

■ DOCUMENTACION DEL PROGRAMA DE EJEMPLO #4.

- En el programa manejador se incluye una llamada a la función de ayuda load help(), con lo cual, queda cargado el archivo con las ventanas de ayuda requeridas en el programa funtabla.c.

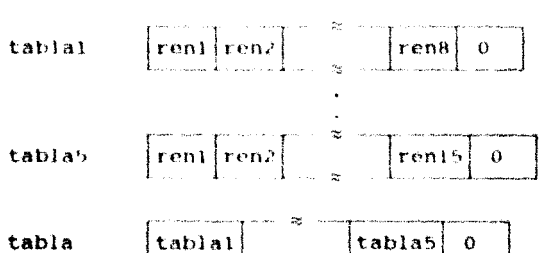
- Observaciones respecto al codigo del programa funtabla.c.

a) El arreglo **titles**, contiene las direcciones de los titulos de las 5 tablas.



b)
- El arreglo **pno** es inicializado con 5 elementos nulos; y contendra las direcciones de las estructuras de ventana para cada tabla.
- Los arreglos **x[]**, **y[]**, son inicializados con las coordenadas de la esquina superior izquierda de cada ventana.
- El arreglo **wcl** es llenado con los colores del fondo y primer plano de cada ventana.

c) y d) Los arreglos tabla1, tabla2, ..., tabla5 son llenados con el texto de su tabla asociada.



e) En la llamada a la función principal funtabla():
- Se establece la ventana de menú con todos sus atributos.
- Se despliega la ventana de menú.

- Mediante la funcion **wprintf** se escriben las opciones de la ventana de menu.
- Se establece una ventana de ayuda para ser activada durante el despliegue del menu.
- La variable "s" contendra el numero de tabla seleccionada, o cero si se tecleo "Esc".
- Si fue seleccionada alguna tabla: la ventana de menu es ocultada y la funcion get_tabla es invocada para desplegar la tabla seleccionada.
  Posteriormente, se establece una ventana de ayuda para ser activada durante el despliegue de las tablas.
  La tabla seleccionada podra ser movida, adelantada, atrasada o destruida de acuerdo a los pulsos de tecla proporcionados.
  En esta fase también podrá desplegarse cualquier otra tabla, simplemente tecleando su numero asociado.
  Para regresar al menu se debera teclear "Esc".
- Si desde el menu se vuelve a teclear "Esc", el proceso terminara.

(f)
- La funcion **ht()** calcula la altura de la ventana que formara la tabla, en base al numero de lineas del texto previamente introducido. Al total de lineas se le suma 4, a fin de incluir las lineas del contorno superior e inferior de la ventana y titulo de la misma.
- La funcion **wd()** calcula el ancho de la ventana que formara la tabla, en base al tamano de la línea de mayor longitud del texto previamente introducido. Al tamano de esa linea se le suma 4, con el fin de tomar en cuenta las lineas de contorno y márgenes izquierdo y derecho de la ventana.

VENTANAS DE AYUDA DE CONTEXTO SENSITIVO

CAPITULO 7

INTRODUCCION

El primer problema al que se enfrenta el usuario cuando tiene contacto con un nuevo sistema, radica en su incapacidad de poder conducirse con agilidad por todas las opciones de éste; esto ocurre debido a la no familiaridad con el programa y su lenguaje de usuario. La experiencia ha demostrado que, no importa el esfuerzo que se dedique al desarrollo de un lenguaje de usuario que se explique por si mismo, los usuarios siempre tendrán preguntas ¿Qué tecla deberá ser presionada? ¿Que proceso viene a continuacion? ¿Qué hace determinado elemento del menú? etc. Para resolver este problema y dar a conocer lo que el sistema puede y no puede hacer, así como otros detalles particulares; tradicionalmente, los proveedores incluyen un manual de usuario y/o un programa de consulta identificado como guía automatizada o *tutorial*. La desventaja de estas alternativas radica en que el usuario debe desviar la atención del sistema para leer el manual o correr el *tutorial*. Se podría pensar que el mantener un despliegue constante del manual de usuario podría solucionar el problema pero no es así. Esta opción queda descartada debido a que sólo respondería a las espectativas de un usuario inexperto; pero en tanto éste creciera en el conocimiento del sistema, la información de ayuda no le sería necesaria, por lo que su despliegue en pantalla además de esteril le resultaría molesto.

La mejor opcion adoptada ya, por la inmensa mayoria de los disenadores de *software*, se basa en los sistemas interactivos. Estos, poseen la caracteristica de poner a disposicion del usuario la informacion de ayuda requerida en cada fase del sistema proporcionando un ambiente amigable basado en la natural inquietud, por parte del usuario, de tener respuesta a sus dudas en cualquier instante.

Este capitulo esta dedicado a mostrar el mecanismo basico para la construcción de un ambiente interactivo basado en la técnica de ventanas de ayuda *popup*, activadas por una tecla de función, que informaran al usuario de todas las alternativas por las que éste puede optar respecto a la fase del sistema en que se encuentre. Este tipo de ventanas, reciben el nombre de ventanas de ayuda de contexto sensitivo.

La tecla de funcion <F1> es reconocida por la industria de *software* como la tecla de función estandar de ayuda (Existen algunos paquetes de *software* que utilizan otras teclas).

El calificativo "contexto sensitivo" de una ventana, es debido a que el mensaje de ayuda desplegado es relativo al contexto corriente del programa. Estas ventanas tambien son identificadas como ventanas sensibles al contexto.

Es tarea del diseñador del sistema, decidir cuánta y qué tipo de ayuda se proporcionará al usuario. Algunos sistemas ofrecen diversos niveles de ayuda dependiendo de la experiencia del usuario. Esta técnica ha sido empleada por años por los diseñadores del procesador de palabra *WordStar*. Los niveles de ayuda fluctuan desde un simple mensaje "Presione Esc para regresar a...." hasta un completo manual de usuario en línea. Esta alternativa ha sido adoptada por algunos desarrolladores de *software* que prefieren distribuir el manual de usuario de esta forma, en lugar del gasto que representa el imprimirlo. Este procedimiento ha originado diversas situaciones en favor y en contra: primero, los usuarios han llegado a asociar manuales extravagantes con la calidad del *software*; y por otro lado, se ha contribuido a la "piratería" de *software* (los libros costosos tienden a desalentar a las personas que ha esto se dedican). Sin embargo, desde la perspectiva del usuario, resulta comodo trabajar en sistemas que requieran un uso mínimo del manual de usuario; es decir, el usuario se está acostumbrando a recibir lo que se conoce como "ayuda en línea".

El objetivo de este capítulo es mostrar el procedimiento y herramientas básicas para la creación de un ambiente de ventanas de ayuda de contexto sensitivo. La información aquí presentada es dividida en dos partes, tal como en el capítulo anterior, la primera consta de las funciones de ayuda de librería básicas, y la segunda de algunos ejemplos de aplicación.

Es tarea del diseñador del sistema, decidir cuánta y qué tipo de ayuda se proporcionará al usuario. Algunos sistemas ofrecen diversos niveles de ayuda dependiendo de la experiencia del usuario. Esta tecnica ha sido empleada por años por los disenadores del procesador de palabra *WordStar*. Los niveles de ayuda fluctúan desde un simple mensaje "Presione Esc para regresar a...." hasta un completo manual de usuario en linea. Esta alternativa ha sido adoptada por algunos desarrolladores de *software* que prefieren distribuir el manual de usuario de esta forma, en lugar del gasto que representa el imprimirlo. Este procedimiento ha originado diversas situaciones en favor y en contra: primero, los usuarios han llegado a asociar manuales extravagantes con la calidad del *software*; y por otro lado, se ha contribuido a la "piratería" de *software* (los libros costosos tienden a desalentar a las personas que ha esto se dedican). Sin embargo, desde la perspectiva del usuario, resulta comodo trabajar en sistemas que requieran un uso mínimo del manual de usuario: es decir, el usuario se está acostumbrando a recibir lo que se conoce como "ayuda en linea".

El objetivo de este capitulo es mostrar el procedimiento y herramientas básicas para la creacion de un ambiente de ventanas de ayuda de contexto sensitivo. La informacion aquí presentada es dividida en dos partes, tal como en el capitulo anterior, la primera consta de las funciones de ayuda de libreria básicas, y la segunda de algunos ejemplos de aplicación.

ACLARACIONES PREVIAS A LA PROGRAMACION DE VENTANAS DE AYUDA

Para el soporte del concepto de ventanas de contexto sensitivo se diseñará un conjunto de "funciones de ayuda" que estarán basadas en las funciones de ventana presentadas en el capítulo VI y en un archivo de texto independiente (archivo de ayuda).

Las funciones de ayuda serán invocadas por el programa de aplicacion, que deberá indicarles qué archivo de ayuda utilizar y qué ventana de ayuda es la corriente en el sistema. El archivo contiene el texto de todas las ventanas de ayuda. Las funciones de ayuda desplegaran la ventana corriente en cuanto la tecla de función de ayuda sea presionada.

Un programa que utilice el "*software* de ayuda", presentado en este capitulo, debera proporcionar los siguientes elementos para la interfaz con las funciones de ayuda:

a) Realizar una llamada a función para especificar el nombre del archivo de texto de ayuda.
b) Especificar un valor para la tecla de función de ayuda.
c) Mediante otras llamadas a función, establecer la ventana de ayuda corriente y sus caracteristicas.
d) Usar la funcion de entrada de teclado de bajo nivel **get char** (descrita en el capitulo VI -archivo ibmpc.c-) para leer los pulsos de tecla. Esto, debido a que dicha función fue diseñada expresamente para validar el caso el pulso de tecla correspondiente a la tecla de función de ayuda. Cuando la funcion get char detecte la tecla <F1> pasara el control del sistema al proceso de ventanas de ayuda.

Respecto a este ultimo inciso, cabe senalar que existen otros métodos para monitorear el teclado con el fin de detectar la tecla de función de ayuda. Algunos de estos metodos involucran el tener que ligar el programa al vector de interrupcion de teclado del BIOS. Esta acción debe siempre evitarse por las razones siquientes: En primer lugar, toda entrada de teclado requerida para trabajar en un ambiente interactivo puede ser manejada perfectamente por la función get char. En segundo lugar, las implicaciones de ligar un programa a un vector de interrupción pueden ser graves si este es terminado anormalmente (estos aspectos fueron estudiados en el capitulo III). Y en tercer lugar, cabe aclarar que como el *software* desarrollado en esta tesis puede ser utilizado en programas residentes en memoria y éstos son ligados al vector de interrupción de teclado para sus propios fines, no es conveniente que la función encargada de detectar la tecla de función de ayuda sea también ligada a dicho vector de interrupción.

El uso de la función get char podria aparentar la pérdida de ciertas facilidades estándares de C; que se manifestaria en la no utilizacion de las siguientes funciones: **scanf()** o **getchar()** de la libreria estandar, la funcion de entrada de consola **getch()** proporcionada por Turbo C; asi como en la restriccion del uso del dispositivo lógico **stdin** exclusivamente para entrada de teclado, excluyendo la posibilidad de redirigirlo hacia otro dispositivo de flujo de entrada (archivos, etc.)

La perdida de las funciones **scanf()** y **getchar()** de hecho no afecta al programador "en linea", debido a su uso infrecuente en un ambiente interactivo. En este tipo de ambiente, prácticamente dichas funciones son poco recomendables pues pueden originar problemas tales como provocar el aborto del programa corriente al detectar la combinacion Ctrl/C. En si, dichas funciones no trabajan bien con teclas de función o de cursor. Turbo C las incluye para mantener la compatibilidad con UNIX. Sus orígenes se remontan a los sistemas de computadora con teletipos a manera de terminales.

Respecto al dispositivo lógico **stdin**, cabe aclarar que está dirigido a programas que pueden tomar su entrada de: teclado, la salida generada por otros programas o de un archivo. Y este tipo de programas tampoco suelen ser corridos en un ambiente interactivo.

■ EL ARCHIVO DE TEXTO DE LAS VENTANAS DE AYUDA.

El texto de cada una de las ventanas de ayuda se encuentra descrito en un archivo de texto o archivo ASCII, que identificaremos como archivo de ayuda. La información respectiva a cada ventana se encuentra separada por mnemónicos (de 8 caracteres) delimitados por picoparéntesis, mismos que serán utilizados por el *software* de ayuda para identificar y localizar el texto de la ventana direccionada. El numero y la longitud de las líneas de texto entre cada mnemónico identificador definirán la altura y ancho de la ventana en cuestión.

Todos los programas de ejemplo utilizados en este capítulo y en los posteriores, utilizarán el *software* de ayuda presentado en los listados 9 y 10. El listado 9, mostrado a continuación, contiene el archivo de ayuda general (**tcprogs.hlp**) para proporcionar ayuda de contexto sensitivo en línea.

LISTADO #9

```
/*   tcprogs.hlp
 *--------------*/

<citas    >
 Presionar 1, 2, ó 3
 para desplegar cita.
<menu     >
 Usar ! ! . .    para mover la barra de cursor.
 Y presionar "Enter" para seleccionar.
```

```
<tablamen>
 Usar ↑ ↓ para:
      mover la barra de cursor.
      "Enter" para:
      seleccionar la opción indicada
      por el cursor.
      No. de opción (1-5) para:
      seleccionar cualquier otra tabla.
<tablas  >
 Usar:
 ↑ ↓  → ←   para mover la tabla.
 Número (1-5) para seleccionar
              la tabla deseada.
 " + " para colocar la tabla al
      frente.
 " - " para colocar la tabla al
      fondo.
<notepad >
-----Movimiento de cursor-----    -----Movimiento de página-----
↑↓→←    = Mueve el Cursor         Ctrl-Home = Inicio de Archivo
Ctrl-T  = Tope de Ventana         Ctrl-End  = Fin de Archivo
Ctrl-B  = Fondo de Ventana        PgUp      = Página Anterior
Ctrl →  = Palabra siguiente       PgDn      = Página Siguiente
Ctrl ←  = Palabra anterior
Home    = Inicio de Linea         ---------Comando de Editor--------
End     = Fin de Linea            Scroll Lock = No Autoformateo

-------Comandos de Bloque-------  -------Comandos de Edición-------
F4  = Fomatea Párrafo             F2 or Esc = Salvar y Salir
F5  = Marca Inicio de Bloque      F3        = Borra Archivo
F6  = Marca Fin de Bloque         Ins       = Insert/Overwrite
F7  = Mueve Bloque                Del       = Borra caracter
F8  = Copia Bloque                Backspace = Borra
F9  = Borra Bloque                Ctrl-D      Borra Palabra
F10 = Desmarca Bloque             Alt-D     = Borra Linea
<end>
```

FUNCIONES DE VENTANA DE AYUDA DE CONTEXTO SENSITIVO

Para instalar ventanas de ayuda de contexto sensitivo en un programa, éste deberá invocar a las funciones **load_help()** y **set help()** cuyo código fuente se encuentra en el archivo "thelp.c" mostrado a continuación.

LISTADO #10

```
/*   thelp.c
     ---------- */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "twindow.h"
#include "keys.h"

#define MAXHELPS 25
#define HBG WHITE
#define HFG BLACK
#define HINT DIM

#define TRUE 1
#define FALSE 0

BLOQUE (1)
static struct helps {
     char hname [9];
     int h, w;
     long hptr;
} hps [MAXHELPS+1];

static int hp = 0;
static int ch = 0;
static int hx, hy;
FILE *helpfp = NULL;
long ftell();
char *fgets();
void help();
char helpname[64];
void getline(char *lineh);
```

BLOQUE (2)

```
/* ----- archivo de definicion para cargar la ayuda---- */
void load_help(char *hn)
{
     extern void (*helpfunc)();
     extern int helpkey;
     char lineh [80];
     if (strcmp(helpname, hn) == 0)
          return;
     helpfunc = help;
     helpkey  = 1;
     hp = 0;
     strcpy(helpname, hn);
     if ((helpfp = fopen(helpname, "r")) == NULL)
          return;
     getline(lineh);
     while (1) {
          if (hp    MAXHELPS)
               break;
          if (strncmp(lineh, "<end>", 5) == 0)
               break;
          if (*lineh != '<')
               continue;
          hps[hp].h = 3;
          hps[hp].w = 18;
          strncpy(hps[hp].hname, lineh+1, 8);
          hps[hp].hptr = ftell(helpfp);
          getline(lineh);
          while (*lineh != '<')     {
               hps[hp].h++;
               hps[hp].w = max(hps[hp].w, strlen(lineh)+2);
               getline(lineh);
          }
          hp++;
     }
}
```

BLOQUE (3)

```
/* ----- obtiene una linea de texto del archivo de ayuda ---- */
static void getline(char *lineh)
{
     if (fgets(lineh, 80, helpfp) == NULL)
          strcpy(lineh, "<end>");
}
```

BLOQUE (4)

```
/* ---- establece la pantalla de ayuda corriente ---- */
void set_help(char *s, int x, int y)
{
     for (ch = 0; ch < hp; ch++)
          if (strncmp(s, hps[ch].hname, 8) == 0)
               break;
     hx = x;
     hy = y;
}
```

BLOQUE [9]

```
/* ---- despliega la ventana de ayuda corriente ---- */
void help()
{
     char ln [80];
     int i, xx, yy;
     WINDOW *wnd;
     extern int helpkey;
     if (hp && ch != hp) {
          curr_cursor(&xx, &yy);
          cursor(0, 25);
          wnd = establish_window(hx, hy, hps[ch].h, hps[ch].w);
          set_colors(wnd, ALL, HBG, HFG, HINT);
          display_window(wnd);
          fseek(helpfp, hps[ch].hptr, 0);
          for (i = 0; i < hps[ch].h-3; i++)  {
               getline(ln);
               wprintf(wnd, ln);
          }
          wprintf(wnd, " <F1> para regresar");
          while (get_char() != helpkey)
               putchar(BELL);
          delete_window(wnd);
          cursor(xx, yy);
     }
}
```

DOCUMENTACION LISTADO #10

En general:

Las primeras 4 declaraciones *#define* del archivo "*theip.c*", establecen los parámetros globales que serán usados en el sistema de ayuda. El parámetro MAXHELPS esta colocado al número máximo de ventanas de ayuda que el programa puede soportar a la vez. Las variables HBG, HFG y HINT corresponden a los colores de fondo, primer plano e intensidad de las ventanas de ayuda.

Para especificar un valor de tecla de función de ayuda diferente a <F1> (el *default*), se debe cambiar el valor de la variable entera global "**helpkey**" declarada en el archivo **ibmpc.c**. Para ello se debe "incluir" en el programa, el archivo fuente "keys.h" y posteriormente emplear alguno de los valores de tecla definidos en dicho archivo. Por ejemplo para cambiar <F1> por <F2> se procedería de la siguiente manera:

```
#include "keys.h"
extern int helpkey;
helpkey = F2;
```

Ahora bien, la función de ayuda estándar "help" también puede ser cambiada. El archivo ibmpc.c incluye un apuntador a la función de ayuda (helpfunc), normalmente inicializado a un valor nulo para poder ser modificado a la dirección de la función de ayuda deseada (en nuestro caso la función "help"). Para utilizar otra función de ayuda diferente, se debe proceder de la siguiente manera:

```
extern void (*helpfunc) ();
void nueva_func();
helpfunc = nueva_func;
```

Existen tres formas para deshabilitar la "ayuda" del sistema: colocando a cero la tecla de función de ayuda, inicializando el apuntador a la función de ayuda a "NULL", y llamando a la función set_help enviando como parámetro un apuntador a caracteres de longitud "cero". Para rehabilitar la ayuda se debe invertir la secuencia anterior.

BLOQUE (1)

Declaración: **static struct helps[]**
Esta estructura define una ventana de ayuda, en base a: el nombre del mnemónico identificador de ventana (arreglo hname[9]), altura y ancho de la misma, y un *offset* a caracter en el archivo de texto tcprogs.hlp.
El arreglo hps (arreglo de estructuras tipo helps), contiene una entrada para cada ventana en el archivo de texto y es construido por la función load_help.

BLOQUE (2)

ANALISIS DE LA FUNCION: **void load_help(char *hn)**
Esta función es llamada para cargar el archivo de texto que contiene la descripción de las ventanas de ayuda, o cambiarlo por un archivo de ayuda diferente. La función abre el archivo de ayuda y lo explora identificando los mensajes de ayuda asociados a cada ventana, para ir construyendo una tabla con los nombres de los identificadores de ventana, su tamaño y posición dentro del archivo de ayuda pasado a esta función por el apuntador hn.

CUERPO DE LA FUNCION:
helpfunc : apuntador a la función de ayuda.
helpkey : código de la tecla de función de ayuda.
lineh80[ ]: arreglo auxiliar para hacer comparaciones con las líneas del archivo de ayuda.

Se verifica si el archivo sobre el que se desea trabajar es el mismo que se ha estado empleando. En caso afirmativo, la función regresa al programa que la invocó puesto que todo ya ha sido inicializado. En caso contrario, El apuntador

helpfunc es inicializado a la dirección de la función de ayuda *default* (help), la variable helpkey es inicializada a "F1"; y el contador de número de ventanas de ayuda (hp) es inicializado a cero. Posteriormente se hace una copia (función strcpy) del nombre del archivo de ayuda introducido (hn) en el arreglo de caracteres helpname.
Posteriormente se verifica si el archivo de ayuda no pudo ser abierto por medio de la función fopen, para la lectura; en caso afirmativo: la función regresa al programa que la invocó. En caso contrario: el apuntador de archivo helpfp contendrá la dirección del archivo de ayuda. Posteriormente se utiliza la función getline para obtener una línea de texto del archivo de ayuda, a partir de donde apunte el indicador de línea en helpfp, y la almacena en el arreglo lineh y dicho indicador en helpfp es incrementado a la siguiente línea de texto en el archivo.
Posteriormente, se entra a un ciclo que finalizará hasta terminar de examinar todos los mnemónicos identificadores de ventana en el archivo de ayuda.

- Se verifica si el contador del numero de ventanas ya alcanzo el máximo permitido (MAXHELPS). En caso afirmativo: termina el ciclo. En caso contrario: se verifica si la línea de texto leída corresponde al mnemónico identificador de finalización del archivo de ayuda (<end>). En caso afirmativo: termina el ciclo. En caso contrario se verifica que la línea de texto leída no corresponda al mnemónico identificador de ventana. En caso afirmativo: vuelve al inicio del ciclo. En caso contrario: los miembros de la estructura helps siguientes, son inicializados para la entrada correspondiente en el arreglo de ventanas de ayuda: "h" y "w" son inicializados a los valores *default* de altura y ancho, en el miembro hname se introduce el mnemónico identificador de ventana (8 caracteres), al miembro hptr se le introduce el *offset* de caracter correspondiente al mnemónico encontrado. A continuación se obtiene una nueva línea del archivo de texto.
  Posteriormente, se calcula la altura de la ventana de ayuda que esta en función del numero de líneas de texto, y el ancho de la ventana que esta en función de la longitud de la línea más grande en el texto de la ventana. Finalmente los miembros de altura y ancho de la estructura helps contendrán los siguientes valores, para la entrada correspondiente en el arreglo de ventanas de ayuda: "h" la suma del *default* más el numero de líneas calculado; y "w" la línea más larga del texto o el "*default*" dado anteriormente en caso de que la línea de texto sea menor a este.
  Finalmente el contador de ventanas de ayuda es incrementado, regresando el control al inicio del ciclo.

- Declaracion: **int strcmp(char *str1, char *str2)**
  Esta función compara lexicográficamente dos cadenas terminadas en nulo, y regresa un entero basado en el resultado, como se muestra a continuación:

Valor	Significado
< 0	str1 < str2
= 0	str1 = str2
> 0	str1 > str2

- Declaracion: **char *strcpy(char *str1, char *str2)**
  Esta función es utilizada para copiar el contenido de str2 a str1; str2 debe ser un apuntador a una cadena terminada en nulo. Esta función regresa un apuntador a str1.

- Declaracion: **FILE *fopen(char *fname, char *mode)**
  Esta función abre un archivo cuyo nombre es apuntado por fname, y regresa la trayectoria asociada a este. El tipo de operaciones que serán permitidas sobre el archivo son definidas por el valor de mode ("r" para lectura, "w" para escritura, etc.) SI esta funcion tuvo éxito en la apertura del archivo especificado se regresa un apuntador de tipo FILE, en caso contrario, regresará un apuntador NULL.

- Declaracion: **long ftell(FILE *stream)**
  Esta funcion regresa el valor corriente del indicador de posicion de archivo para la trayectoria (stream) especificada. Este valor es el numero de bytes del indicador respecto al inicio del archivo.

BLOQUE (3)

ANALISIS DE LA FUNCION: **static void getline(char *lineh)**
Esta función obtiene una linea de texto del archivo de ayuda.

CUERPO DE LA FUNCION:
Por medio de la función fgets se lee una linea de texto del archivo helprp, la almacena en el arreglo lineh, y devuelve la direccion del arreglo lineh, siempre que no se haya alcanzado el fin del archivo de ayuda; en caso contrario, la función regresa un apuntador nulo y getline devolvera en el arreglo lineh el mnemonico identificador de fin de archivo (<end>).

- Declaracion:
  char *fgets(char *str, int num, FILE *stream)
  Esta funcion lee hasta num-1 caracteres desde la trayectoria indicada por "stream" y los coloca dentro del arreglo de caracteres apuntado por "str".
  Los caracteres son leidos hasta que un caracter nueva línea aparezca, o hasta que el límite especificado en

num sea alcanzado, o hasta que sea alcanzado el fin de archivo. Despues de que los caracteres han sido leidos, se coloca un caracter NULL en el arreglo inmediatamente despues del ultimo caracter leido. Si la función tuvo éxito en la lectura regresara la dirección del arreglo str, en caso de haber alcanzado el fin de archivo, se regresara un valor NULL.

BLOQUE (4)

ANALISIS DE LA FUNCION:
void set help(char *s, int x, int y)
Esta función valida la existencia de una ventana de ayuda proporcionada, mediante la comparación del arreglo de 8 caracteres "s" con los mnemonicos identificadores de ventana en el archivo de ayuda. El mnemónico se encuentra entre picoparéntesis en el archivo de ayuda, pero el arreglo no los incluye. Los enteros "x", "y" especifican las coordenadas de la esquina superior izquierda de la ventana de ayuda, lo cual permite a los usuario usar la misma ventana para diferentes contextos y en diferentes posiciones en la pantalla.

CUERPO DE LA FUNCION:
La funcion busca en el arreglo de estructuras de ventanas de ayuda, un nombre que iguale al nombre de la ventana pasado por el llamador de la funcion. La variable entera "ch" es utilizada como indice en el arreglo de ventanas de ayuda, hasta que la igualacion sea encontrada, con lo que esta variable contendrá el numero de la ventana de avuda que se desea desplegar. Posteriormente, las variables "hx", "hy" son colocadas a las coordenadas de la esquina superior izquierda de la ventana de ayuda pasadas por el llamador de la función.

BLOQUE (5)

ANALISIS DE LA FUNCION: **void help()**
Esta función es llamada por la función get char (encontrada en el archivo ibmpc.c) cuando la tecla de función de ayuda designada es presionada.

CUERPO DE LA FUNCION:
La funcion help salva la localización corriente del cursor para luego posicionarlo en una localidad fuera de la pantalla. Una ventana de ayuda es establecida de acuerdo a las coordenadas y dimensiones almacenadas en las variables "hx", "hy" y en los miembros "h" y "w" de la estructura helps para la entrada "ch" en el arreglo de ventanas de ayuda. El indicador de posición de caracter del archivo de ayuda es actualizado al valor contenido en el miembro hptr

de la estructura helps correspondiente a la entrada "ch" en el arreglo de estructuras.
De acuerdo a esa posicion de caracter, cada linea de texto es leida y escrita en la ventana de ayuda. La ultima linea escrita instruye al usuario para que presione la tecla de ayuda una segunda vez para limpiar la ventana de ayuda y regresar. Entonces el programa espera que la tecla de ayuda sea presionada, al tiempo que destruye la ventana y restablece el cursor a la posicion que mantenia antes de que la funcion de ayuda fuera invocada.

- Declaracion:
  int fseek(FILE *stream, long offset, int origin)
  Esta funcion coloca el indicador de posición de caracter del archivo asociado a la trayectoria indicada en stream, de acuerdo a los valores de *offset* y origin. Su proposito principal es dar soporte a operaciones de E/S aleatorias. El *offset* es el numero de bytes desde origin para establecer la nueva posición. La variable origin puede ser alguno de estos valores: "0", para indicar el inicio del archivo; "1", para indicar el origen a partir de la posición corriente; y "2", para indicar el fin del archivo.
  El *offset* debe ser de tipo long int, en la mayoria de los casos, para soportar archivos mayores de 64Kb.
  Esta función puede ser utilizada para mover el indicador de posición a donde se desee en el archivo, aún más allá del final, sin embargo es un error colocar el indicador de posicion antes del comienzo del archivo.

PROGRAMA DE EJEMPLO QUE HACE USO DE LAS FUNCIONES DE VENTANA DE AYUDA DE CONTEXTO SENSITIVO

Con el fin de mostrar al programador la forma de utilizar las funciones de ayuda de contexto sensitivo, en este apartado incluimos el siguiente programa de ejemplo. El cual consta de 3 módulos: un programa manejador, otro con el cuerpo de la función principal, y un archivo "proyecto" (extension prj), que Turbo C utiliza para construir el programa ejecutable.

■ **EJEMPLO No. 1**
PROGRAMA QUE DESPLIEGA FRASES CELEBRES PROPORCIONANDO UNA VENTANA DE AYUDA.

LISTADO #11

LISTADO 11.1
Programa Manejador.

```
/* ------ sayings.c ------ */

#include "twindow.h"
void maxims(void);

void main()
{
     load_help("tcprogs.hlp");
     maxims();
}
```

LISTADO 11.2
Programa que contiene el codigo de la funcion principal.

```
/* ------- maxims.c ------- */

#include "twindow.h"
#include "keys.h"
```

```
void maxims()
{
     int c;
     WINDOW *wnd;

     set_help("citas   ", 50, 10);
     wnd = establish_window(5, 10, 3, 50);
     set_title(wnd, "Presione F1 para ayuda");
     set_colors(wnd, ALL, RED, WHITE, DIM);
     display_window(wnd);
     while ((c = get_char()) != ESC)    {
          switch (c)          {
               case '1':
                    wprintf(wnd, "\nDime con quien andas \
y te diré quien eres ");
                    break;
               case '2':
                    wprintf(wnd, "\nMas sabe el diablo por \
viejo que por diablo");
                    break;
               case '3':
                    wprintf(wnd, "\nUn peso ahorrado \
es un peso ganado");
                    break;
               default:
                    break;
          }
     }
     delete_window(wnd);
}
```

LISTADO 11.3
Programa que construye el archivo ejecutable.

```
/* ------ sayings.prj ------ */
sayings
maxims (twindow.h, keys.h)
thelp (twindow.h, keys.h)
twindow (twindow.h, keys.h)
ibmpc.obj
```

■ COMENTARIOS AL LISTADO #11

El programa del listado 11.1 carga el archivo de ayuda tcprogs.hlp mediante una llamada a la función load_help, y luego invoca a la función maxims, cuyo código se encuentra en el listado 11.2. Esta secuencia fue seleccionada así, para que maxims.c pueda ser integrada en los programas de capítulos posteriores, que proporcionan ejemplos de procesamiento de ventanas de menu de barra deslizante y de utilerías residentes en memoria.

El programa del listado 11.2 maxims.c, muestra como utilizar las funciones set help y get char en los programas de aplicación. Unicamente será utilizada una ventana de ayuda. La función maxims.c crea una ventana con todas sus caracteristicas y espera a que un pulso de tecla sea introducido. Si este corresponde a uno de los siguientes: "1", "2", o "3"; se desplegará en la ventana, el "dicho popular" asociado a ese valor. Si se presiona "Esc", el programa terminara. Si se presiona <F1>, la ventana de ayuda será desplegada.

VENTANA DE EDITOR DE TEXTO

CAPITULO 8

INTRODUCCION

Muchos de los paquetes TSR comerciales incluyen editores de texto presentados como procesadores de palabra, *notepads* o simplemente como editores de programas de aplicación. El editor de texto constituye una de las aplicaciones más comunes entre los programas residentes (en cuanto a utilerías de escritorio se refiere). El objeto de esta tesis es, en primera instancia, presentar una metodología general para la construcción de programas residentes en memoria; sin embargo, con el objeto de dar un enfoque didáctico también se incluyen algunos ejemplos ilustrativos de algunos de los programas TSR más comunes. Razón por la cual consideramos conveniente dedicar el presente capítulo al tratamiento exclusivo del programa editor de texto. Nuestro objetivo: presentar al programador inexperto las herramientas generales para la construcción de un programa editor de texto que incluya un conjunto de funciones completo.

Particularidades:

Presentaremos el editor en una ventana de tamaño definido, creada tal como se ha venido manejando en los capítulos anteriores. La introducción/modificación del texto de la ventana será en formato libre. El *notepad* de *Sidekick* es un ejemplo de las aplicaciones que emplean dicho formato. De hecho, nuestro programa implementará un *notepad* muy similar al de *Sidekick*. Nuestro editor de texto podrá también ser usado en programas de entrada de datos que manejen línea múltiple. Por ejemplo en aplicaciones que involucren patrones de entrada de datos que requieran un campo o texto descriptivo (observaciones, comentarios, descripciones, nomenclaturas, etc.).

El primer paso en la programación de nuestro programa será establecer una ventana y crear un *buffer*, en el cual se introducirán físicamente los datos de texto. El *buffer* de texto es un arreglo con espacio suficiente para almacenar los bytes correspondientes al área de texto de la ventana.

Si se establece una ventana de 42 caracteres por línea y un *buffer* de 4000 caracteres, se habrá creado un área de texto de 40 caracteres de largo (pues el borde de la ventana ocupa dos posiciones de caracter) por 100 caracteres de ancho.

El texto introducido por el usuario, se irá colectando en el *buffer* de texto, sin incluir los caracteres nueva línea o tabulador. Si al llamar la función de edición por primera vez, el *buffer* ya contenía texto, la información existente será desplegada en el formato de línea de longitud fija.

COMANDOS DEL EDITOR DE TEXTO

La figura 8.1 muestra una tabla con el conjunto de comandos de edición que maneja nuestro programa editor de texto. Dicha tabla estará contenida en la ventana de ayuda del programa. La activación de la ventana de ayuda se efectuará al presionar la tecla de función 'F1'. A continuación veremos brevemente algunos aspectos de cada uno de los comandos del editor.

FIG.8.1 COMANDOS DE EDICION

Movimiento de Cursor		Movimiento de Página	
↑↓→←	= Mueve el Cursor	Ctrl-Home	= Inicio de Archivo
Ctrl-T	= Tope de Ventana	Ctrl-End	= Fin de Archivo
Ctrl-B	= Fondo de Ventana	PgUp	= Página Anterior
Ctrl →	= Palabra siguiente	PgDn	= Página Siguiente
Ctrl ←	= Palabra anterior		
Home	= Inicio de Linea	Comando de Editor	
End	= Fin de Linea	Scroll Lock	= No Autoformateo

Comandos de Bloque		Comandos de Edición	
F4	= Fomatea Parrafo	F7 or Esc	= Salvar y Salir
F5	= Marca Inicio de Bloque	F3	= Borra Archivo
F6	= Marca Fin de Bloque	Ins	= Insert/Overwrite
F7	= Mueve Bloque	Del	= Borra caracter
F8	= Copia Bloque	Backspace	= Borra
F9	= Borra Bloque	Ctrl-D	= Borra Palabra
F10	= Desmarca Bloque	Alt-D	= Borra Linea

Movimiento del Cursor.

<↑,↓,→,←>

Para mover el cursor a lo largo de toda la pantalla, se utilizan las teclas de flecha. Por cada pulso de tecla de flecha hacia arriba o hacia abajo, se efectuará un scroll de pantalla en un linea segun la dirección indicada. Si los movimientos son permanentes,

se realizara el scroll hasta alcanzar en inicio o fin del *buffer* (según sea el caso).

<Ctrl/T> <Ctrl/B> Mueven el cursor al tope y fondo de la pantalla respectivamente.

<Ctrl/ →> Mueve el cursor al inicio de la siguiente palabra en el *buffer* de texto.

<Ctrl/←> Mueve el cursor al inicio de la palabra previa en el *buffer* de texto.

<Home> Posiciona el cursor al inicio de la linea corriente.

<End> Posiciona el cursor al final de la linea corriente.

<Tab> Avanza el cursor hacia la siguiente marca de tabulador.

<Shift/Tab> Retrocede el cursor hacia la marca de tabulador previa.

Movimiento de Páginas.

<PgUp> y **<PgDn>** Muestran la página anterior y posterior respectivamente.

<Ctrl/Home> Muestra la primer página relativa al *buffer* posicionando el cursor al inicio de la primer linea.

<Ctrl/End> Muestra la ultima pagina del *buffer* y posiciona el cursor en la primer columna de la linea que sigue a la última linea en el *buffer*.

Comandos de Bloques.

Son comandos que operan sobre bloques de texto previamente delimitados (definidos sobre límites de linea).

<F4> Forma un parrafo a partir de las lineas de texto marcadas como un bloque.

<F5> Marca la primer línea en un bloque (relativa a la posición del cursor). Los bloques son desplegados en colores ACCENT (iluminados).

<F6> Marca la ultima línea en un bloque.

<F7> Mueve un bloque a la línea donde el cursor se encuentra posicionado. El movimiento es no-destructivo; es abierto un espacio para dar cabida a dicho bloque.

<F8> Copia un bloque en la línea corriente. La diferencia entre este comando y el anterior (MOVE) radica en que, en este caso el bloque original permanece en su lugar.

<F9> Borra un bloque recorriendo hacia arriba el texto posterior al bloque eliminado (deja el texto como si el bloque nunca hubiera existido).

<F10> Desmarca un bloque.

Comandos de Edición.

Durante la edición, el programa realizará automáticamente lo que se cono ce como reformateo de párrafo. El editor define un párrafo como un conjunto o grupo de líneas de texto encontradas arriba de una línea en blanco. El reformateo de párrafos puede ser suprimido colocando el teclado en modo Scroll Lock.

<F3> Borra todo el texto del *buffer* después de haber hecho una confirmación.

<Ins> Conmuta los modos Insert/Overwirte cambiando también la forma del cursor para indicar el modo.

<Del> Borra el caracter en el que el cursor se encuentre posicionado recorriendo el texto posterior una posición a la izquierda.

<Backspace> Borra el caracter encontrado a la izquierda del cursor y recorre tanto el cursor como el texto posterior, una posición a la izquierda.

<Ctrl/D> Borra la palabra indicada por el cursor.

<Alt/D> Borra la línea indicada por el cursor.

<F2> o <Esc> Para terminar la edición de texto y regresar al programa principal.

FUNCION DE EDICION DE TEXTO

La edición de texto cuenta con una función que deberá ser invocada por un programa principal. Para utilizar dicha función, previamente deberá ser establecida una ventana en la cual la entrada de texto será procesada.

LISTADO #12

```
/*   editor.c
/* ---------- */
#include <stdio.h>
#include <ctype.h>
#include <mem.h>
#include <conio.h>
#include <alloc.h>
#include "twindow.h"
#include "keys.h"

BLOQUE (1)
#define TRUE 1
#define FALSE 0
#define TAB 4
#define NEXTTAB (TAB-(x%TAB))
#define LASTTAB (((wwd-1)/TAB)*TAB)
#define PREVTAB (((x-1)%TAB)+1)
#define curr(x,y) (bfptr+(y)*wwd+(x))
#define lineno(y) ((int)(bfptr-topptr)/wwd+(y))
extern int VSG;
int last_x, last_y;
static int wnt;
static int wwd;
static int wsz;
static char *topptr;
static char *bfptr;
static char *lstptr;
static int lines;
static char *endptr;
static int blkbeg;
static int blkend;
static int inserting;
static WINDOW *wnd;
static int do_display_text = 1;
```

```
/* --- declaración de funciones locales --- */
void erase_buffer(int *x, int *y);
int lastword(int x, int y);
void last_char(int *x, int *y);
void test_para(int x, int y);
int trailing_spaces(int y);
int first_wordlen(int y);
void paraform(int x, int y);
int blankline(int line);
void delete_word(int x, int y);
void delete_line(int y);
void delete_block(void);
void copy_block(int y);
void move_block(int y);
void mvblock(int y, int moving);
void findlast(void);
void find_end(int *x, int *y);
void carrtn(int *x, int *y, int insert);
void backspace(int *x, int *y);
void fore_word(int *x, int *y, char *bf);
int spaceup(int *x, int *y, char **bf);
void back_word(int *x, int *y, char *bf);
int spacedn(int *x, int *y, char **bf);
void forward(int *x, int *y);
int downward(int *y);
void upward(int *y);
void display_text(void);
void disp_line(int y);
void insert_line(void);
```

BLOQUE (2)

```
/* ---Procesa la entrada de texto y comandos de edición ---- */
(2.1)
void text_editor(WINDOW *wnd1, char *bf, unsigned bsize)
{
     char *b, *buff;
     int depart = FALSE, i, c;
     int x, y, svx, svlw, tx, tabctr = 0;

     wnd = wnd1;
     wht = HEIGHT-2;
     wwd = WIDTH-2;
     wsz = wwd * wht;
     topptr = bfptr = bf;
     lines = bsize / wwd;
     endptr = bf + wwd * lines;
     blkbeg = 0;
     blkend = 0;
     inserting = FALSE;
     x = 0;
     y = 0;
     display_text();
     findlast();
```

(2.2)

```
    /*   lee texto y comandos del teclado   */
    while (TRUE)    {
         last_x = COL + 1 + x;
         last_y = ROW + 1 + y;
         cursor(last_x, last_y);
         buff = curr(x, y);
         if (tabctr)     {
              --tabctr;
              c = ' ';
         }
         else {
              c = get_char();
              clear_message();
         }
         switch (c)      {
              case '\r':      carrtn(&x, &y, inserting);
                              break;
              case DN:  downward(&y);
                              break;
              case PGUP:      y = 0;
                              for (i = 0; i < wht; i++)
                                   upward(&y);
                              break;
              case PGDN:      y = HEIGHT - 2;
                              for (i = 0; i < wht; i++)
                                   downward(&y);
                              y = 0;
                              break;
              case '\t':      if (x + NEXTTAB < wwd)    {
                                   if (inserting)
                                        tabctr = NEXTTAB;
                                   else
                                        x += NEXTTAB;
                              }
                              else
                                   carrtn(&x, &y, inserting);
                              break;
              case SHIFT_HT:
                              if (x < TAB)    {
                                   upward(&y);
                                   x = LASTTAB;
                              }
                              else
                                   x -= PREVTAB;
                              break;
              case CTRL_FWD:
                              fore_word(&x, &y, buff);
                              break;
              case CTRL_BS:
                              back_word(&x, &y, buff);
                              break;
              case CTRL_B:
                              y = wht - 1;
```

```
                    break;
case CTRL_T:
                    y = 0;
                    break;
case CTRL_HOME:
                    x = y = 0;
                    bfptr = topptr;
                    display_text();
                    break;
case HOME:          x = 0;
                    break;
case CTRL_END:
                    find_end(&x, &y);
                    display_text();
                    break;
case END: last_char(&x, &y);
                    break;
case UP:  upward(&y);
                    break;
case F2:
case ESC: depart = TRUE;
                    break;
case '\b':
case BS:  if (curr(x, y) == topptr)
                         break;
                    backspace(&x, &y);
                    if (x == wwd - 1)
                         last_char(&x, &y);
                    if (c == BS)
                         break;
                    buff = curr(x, y);
case DEL: movmem(buff+1, buff, wwd-1-x);
                    *(buff+wwd-1-x) = ' ';
                    disp_line(y);
                    test_para(y+1, y);
                    break;
case ALT_D:         delete_line(y);
                    break;
case CTRL_D:delete_word(x, y);
                    test_para(x, y);
                    break;
case INS: inserting = TRUE;
                    insert_line();
                    break;
case F3:  erase_buffer(&x, &y);
                    break;
case F4:  paraform(0, y);
                    break;
case F5:  blkbeg = lineno(y) + 1;
                    if (blkbeg > blkend)
                         blkend = lines;
                    display_text();
                    break;
case F6:  blkend = lineno(y) + 1;
```

```
                    if (blkend < blkbeg)
                        blkbeg = 1;
                    display_text();
                    break;
          case F7:  move_block(y);
                    break;
          case F8:  copy_block(y);
                    break;
          case F9:  delete_block();
                    break;
          case F10: blkbeg  blkend  0;
                    display_text();
                    break;
          case FWD: forward(&x, &y);
                    break;
{2.1}     default:  if (!isprint(c))
                        break;
                    if (curr(x, y) >= endptr-1 ||
                    (lineno(y)+1 >= lines && inserting
                    && *curr(wwd-2, y) != ' '))
                    {
                    error_message(" Fin de Buffer ");
                        break;
                    }
                    if (inserting) {
                    buff = curr(x, y);
                    movmem(buff, buff + 1, wwd-1-x);
                    }
                    buff = curr(x, y);
                    if (buff < endptr)  {
                        if (buff >= lstptr)
                            lstptr = buff + 1;
                        *buff = c;
                        disp_line(y);
                    }
                    buff  curr(wwd-1, y);
                    if (endptr && *buff != ' ')  {
                    for (b = buff+1; b < endptr; b++)
                        if (*b==' ' && *(b + 1)==' ')
                            break;
                    movmem(buff+1, buff+2, b-buff-1);
                    *(buff+1) = ' ';
                    svx = x;
                    svlw = lastword(x, y);
                    x = wwd-1;
                    if (*(buff-1) != ' ')
                        back_word(&x, &y, buff);
                    tx = x;
                    carrtn(&x, &y, TRUE);
                    if (svlw)
                        x = svx-tx;
                    else {
                        y  svx;
                        --y;
```

```
                              )
                         }
                         forward(&x, &y);
                         break;
               }
               if (depart)
                    break;
          }
          inserting = FALSE;
          insert_line();
}

BLOQUE (3)
/* --- limpia el contenido del buffer --- */
static void erase_buffer(int *x, int *y)
{
     int c = 0;
     WINDOW *sur;

     sur = establish_window(18, 11, 4, 35);
     set_colors(sur, ALL, RED, YELLOW, BRIGHT);
     display_window(sur);
     wprintf(sur, " Borrar el texto de la ventana\n Está seguro ?
          (s/n)");
     while (c != 's' && c != 'n')  {
          c = get_char();
          c = tolower(c);
          if (c == 's')  {
               lstptr = bfptr = topptr;
               *x = *y = 0;
               setmem(bfptr, lines * wwd, ' ');
               blkbeg = blkend = 0;
               display_text();
          }
     }
     delete_window(sur);
}

BLOQUE (4)
/* - verifica si determinada palabra es la última en la línea -*/
static int lastword(int x, int y)
{
     char *bf = curr(x, y);

     while (x++ < wwd-1)
          if (*bf++ == ' ')
               return 0;
     return 1;
}

BLOQUE (5)
/* - posiciona el cursor en el último caracter desplegable  */
/*   de la línea - */
static void last_char(int *x, int *y)
```

```
{
     char *bf;

     *x = wwd-1;
     bf = curr(0, *y);
     while (*x && *(bf + *x) == ' ')
          --(*x);
     if (*x && *x < wwd - 1)
          (*x)++;
}

BLOQUE (6)
/* --- verifica si un parrafo debe ser "reformateado" --- */
static void test_para(int x, int y)
{
     int ts, fw;

     if (!scroll_lock() && y < lines)   {
          ts = trailing_spaces(y);
          fw = first_wordlen(y+1);
          if (fw && ts > fw)
               paraform(x, y);
     }
}

BLOQUE (7)
/* ---- cuenta el numero de blancos sobrantes de una línea --- */
static int trailing_spaces(int y)
{
     int x = wwd-1, ct = 0;
     char *bf = curr(0, y);

     while (x >= 0) {
          if (*(bf + x) != ' ')
               break;
          --x;
          ct++;
     }
     return ct;
}

BLOQUE (8)
/* --- devuelve la long. de la primer palabra de una línea --- */
static int first_wordlen(int y)
{
     int ct = 0, x = 0;
     char *bf = curr(0, y);

     while (x < wwd-1 && *(bf+x) == ' ')
          x++;
     while (x+ct < wwd-1 && *(bf+x+ct) != ' ')
          ct++;
     return ct;
}
```

BLOQUE (9)

```
/* --- formatea un parrafo --- */
static void paraform(int x, int y)
{
     char *cp1, *cp2, *cpend, *svcp;
     int x1;

     if (blankline(lineno(y)+1))
          return;
     if (!blkbeg)   {
          blkbeg = blkend = lineno(y)+1;
          blkend++;
          while (blkend < lines)   {
               if (blankline(blkend))
                    break;
               blkend++;
          }
          --blkend;
     }
     if (lineno(y) != blkbeg-1)
          x = 0;
     x1 = x;
     cp1 = cp2 = topptr + (blkbeg - 1) * wwd + x;
     cpend = topptr + blkend * wwd;
     while (cp2 < cpend) {
          while (*cp2 == ' ' && cp2 < cpend)
               cp2++;
          if (cp2 == cpend)
               break;
          /* en una palabra */
          while (*cp2 != ' ' && cp2 < cpend) {
               if (x1 >= wwd - 1)  {
                    /* palabra truncada */
                    svcp = cp1 + (wwd - x1);
                    while (*--cp1 != ' ')    {
                         *cp1 = ' ';
                         --cp2;
                    }
                    x1 = 0;
                    blkbeg++;
                    cp1 = svcp;
               }
               *cp1++ = *cp2++;
               x1++;
          }
          if (cp2 < cpend)    {
               *cp1++ = ' ';
               x1++;
          }
     }
     while (cp1 < cpend)
          *cp1++ = ' ';
     blkbeg++;
     if (blkbeg <= blkend)
```

```
            delete_block();
        blkbeg = blkend = 0;
        display_text();
        findlast();
}

BLOQUE (10)
/* --- verifica si toda la linea contiene blancos --- */
static int blankline(int line)
{
        char *cp;
        int x;

        cp = topptr + (line-1) * wwd;
        for (x = 0; x < wwd; x++)
                if (*(cp + x) != ' ')
                        break;
        return (x == wwd);
}

BLOQUE (11)
/* --- borra palabra --- */
static void delete_word(int x, int y)
{
        int wct = 0;
        char *cp1, *cp2;

        cp1 = cp2 = curr(x, y);
        if (*cp2 == ' ')
                while (*cp2 == ' ' && x + wct < wwd)    {
                        wct++;
                        cp2++;
                }
        else {
                while (*cp2 != ' ' && x + wct < wwd)    {
                        wct++;
                        cp2++;
                }
                while (*cp2 == ' ' && x + wct < wwd)    {
                        wct++;
                        cp2++;
                }
        }
        movmem(cp2, cp1, wwd - x - wct);
        setmem(cp1 + wwd - x - wct, wct, ' ');
        display_text();
        findlast();
}
BLOQUE (12)
/* --- borra una linea --- */
static void delete_line(int y)
{
        char *cp1, *cp2;
        int len;
```

```
     cp1 = bfptr + y * wwd;
     cp2 = cp1 + wwd;
     if (cp1 < lstptr)   {
          len = endptr - cp2;
          movmem(cp2, cp1, len);
          lstptr -= wwd;
          setmem(endptr - wwd, wwd, ' ');
          display_text();
     }
}
```

BLOQUE (13)

```
/* --- borra un bloque --- */
static void delete_block()
{
     char *cp1, *cp2;
     int len;

     if (!blkbeg || !blkend)  {
          putchar(BELL);
          return;
     }
     cp1 = topptr + blkend * wwd;
     cp2 = topptr + (blkbeg - 1) * wwd;
     len = endptr - cp1;
     movmem(cp1, cp2, len);
     setmem(cp2 + len, endptr - (cp2 + len), ' ');
     blkbeg = blkend = 0;
     lstptr -= (cp1 - cp2);
     display_text();
}
```

BLOQUE (14)

```
/* --- mueve y copia bloques de texto --- */
static void mvblock(int y, int moving)
{
     char *cp1, *cp2, *hd;
     int len;
     if (!blkbeg || !blkend)  {
          putchar(BELL);
          return;
     }
     if (lineno(y) >= blkbeg-1 && lineno(y) <= blkend-1)    {
          error_message("No puede mover o copiar un bloque
                         dentro de si mismo");
          return;
     }
     len = (blkend - blkbeg + 1) * wwd;
     if ((hd = malloc(len)) == 0)
          return;
     cp1 = topptr + (blkbeg-1) * wwd;
     movmem(cp1, hd, len);
     cp2 = topptr + lineno(y) * wwd;
     if (moving)    {
```

```
          if (lineno(y) > blkbeg-1)
               cp2 -= len;
          do_display_text = 0;
          delete_block();
          do_display_text = 1;
     }
     if (cp2+len <= endptr)    {
          movmem(cp2, cp2 + len, endptr - cp2 - len);
          movmem(hd, cp2, len);
     }
     free(hd);
     blkbeg = blkend = 0;
     display_text();
}
```

BLOQUE (15)

```
/* --- copia el bloque marcado a partir de la linea */
/*     indicada por el cursor --- */
static void copy_block(int y)
{
     mvblock(y, FALSE);
     findlast();
}
```

BLOQUE (16)

```
/* --- mueve un bloque --- */
static void move_block(int y)
{
     mvblock(y, TRUE);
}
```

BLOQUE (17)

```
/* -- encuentra el ultimo caracter desplegable en el buffer --*/
static void findlast()
{
     register char *lp = endptr - 1;
     register char *tp = topptr;
     while (lp > tp && (*lp == ' ' || *lp == '\0'))    {
          if (*lp == '\0')
               *lp = ' ';
          --lp;
     }
     if (*lp != ' ')
          lp++;
     lstptr = lp;-
}
```

BLOQUE (18)

```
/* --- posiciona el curosr al final del texto del buffer --- */
static void find_end(int *x, int *y)
{
     int ct;
     bfptr = lstptr;
     ct = (lstptr - topptr) % wsz;
     bfptr -= ct;
```

```
    if (bfptr + wsz > endptr)
        bfptr = endptr - wsz;
    *y = (ct / wwd);
    *x = 0;
    downward(y);
}

BLOQUE (19)
/* --- procesa un "carriage return" --- */
static void carrtn(int *x, int *y, int insert)
{
    int insct;
    char *cp, *nl;
    int ctl = 2;
    cp = curr(*x, *y);
    nl = cp + ((cp - topptr) % wwd);
    if (lineno(*y) + 2 < lines)
        if (insert && nl < endptr)    {
            insct = wwd - *x;
            while (ctl--)  {
                if (endptr > cp + insct) {
                    movmem(cp, cp+insct, endptr-insct-cp);
                    setmem(cp, insct, ' ');
                }
                else if (ctl == 1)
                    setmem(cp, endptr - cp, ' ');
                cp += insct * 2;
                insct = *x;
            }
        }
    *x = 0;
    downward(y);
    if (insert)    {
        test para(*x, *y);
        display text();
    }
    if (lineno(*y) + 2 < lines)
        if (insert)
            if ((lstptr + wwd) <= endptr)
                if (lstptr > curr(*x, *y))
                    lstptr += wwd;
}

BLOQUE (20)
/* --- traslada el cursor una posición a la izquierda --- */
static void backspace(int *x, int *y)
{
    if (*x == 0)   {
        *x = wwd - 1;
        upward(y);
    }
    else
        --(*x);
}
```

BLOQUE (21)
```
/* -- traslada el cursor al inicio de la palabra siguiente -- */
static void fore_word(int *x, int *y, char *bf)
{
     while (*bf != ' ')  {
          if (spaceup(x, y, &bf) == 0)
               return;
          if (*x == 0)
               break;
     }
     while (*bf  = ' ')

if (spaceup(x, y, &bf) == 0)
               return;
}
```

BLOQUE (22)
```
static int spaceup(int *x, int *y, char **bf)
{
     if (*bf == lstptr)
          return 0;
     (*bf)++;
     forward(x, y);
     return 1;
}
```

BLOQUE (23)
```
/* --- traslada el cursor una posición a la derecha --- */
static void forward(int *x, int *y)
{
     int ww = wwd;
     (*x)++;
     if (*x == ww)  {
          downward(y);
          *x = 0;
     }
}
```

BLOQUE (24)
```
/* --- traslada el cursor al inicio de la palabra anterior --- */
static void back_word(int *x, int *y, char *bf)
{
     spacedn(x, y, &bf);
     while (*bf == ' ')
          if (spacedn(x, y, &bf) == 0)
               return;
     while (*bf != ' ')  {
          if (*x == 0)
               return;
          if (spacedn(x, y, &bf) == 0)
               return;
     }
     spaceup(x, y, &bf);
}
```

BLOQUE (25)

```
static int spacedn(int *x, int *y, char **bf)
{
     if (*bf == topptr)
          return 0;
     --(*bf);
     backspace(x, y);
     return 1;
}
```

BLOQUE (26)

```
/* --- traslada el cursor una linea abajo --- */
static int downward(int *y)
{
     if (*y < wht - 1)   {
          (*y)++;
          return 1;
     }
     else if ((bfptr + ws2) < endptr)   {
          bfptr += wwd;
          scroll(wnd, UP);
          disp_line(wht-1);
          return 1;
     }
     return 0;
}
```

BLOQUE (27)

```
/* --- traslada el cursor una linea arriba --- */
static void upward(int *y)
{
     if (*y)
          --(*y);
     else if ((topptr + wwd) <= bfptr)   {
          bfptr -= wwd;
          scroll(wnd, DN);
          disp_line(0);
     }
}
```

BLOQUE (28)

```
/* --- despliega todas las lineas de una ventana --- */
static void display_text()
{
     int y = 0;

     if (do_display_text)
          while (y < wht)
               disp_line(y++);
}
```

```
BLOQUE (29)
/* --- despliega una linea --- */
static void disp_line(int y)
{
     int x = 0, atr = WNORMAL;

     if (blkbeg || blkend)
          if (lineno(y) >= blkbeg-1)
               if (lineno(y) <= blkend-1)
                    atr = WACCENT;
     while (x < wwd)      {
          displ(wnd, x+1, y+1, *(bfptr+y * wwd+x), atr);
          x++;
     }
}

BLOQUE (30)
/* -- cambia la forma del cursor segun el modo de edición -- */
static void insert_line()
{
     set_cursor_type(inserting ? 0x0106 : 0x0607);
}
```

■ **DOCUMENTACION LISTADO #12**

BLOQUE (1)

En este bloque se agrupan diversas declaraciones **#define** que servirán para el control de las marcas de tabulador en la ventana del editor. La variable global **TAB** es colocada a un valor de "4", con lo que las marcas de tabulación quedan establecidas cada 4 posiciones de caracter.

#define NEXTTAB.-	Coloca el cursor en la siguiente marca de tabulacion.
#define LASTTAB.-	Coloca el cursor en la ultima marca de tabulacion de la linea corriente.
#define PREVTAB.-	Coloca el cursor en la marca de tabulacion previa.

Las siguientes macros son usadas para actualizar coordenadas y los apuntadores en el *buffer*:

#define curr(x,y).-	Transforma las coordenadas a su direccion asociada en el *buffer* de texto.
#define lineno(y).-	Convierte el argumento "y" (que representa el numero de *linea ebn* la ventana del editor) al numero de linea correspondiente en el *buffer* de texto.

BLOQUE (2)

DESCRIPCION DE LA FUNCION:
void text_editor(WINDOW *wnd1, char *bf, unsigned bsize)

Al ser invocada, la función despliega en la primer página del *buffer* el texto de la ventana, o blancos (si se encontraba vacía), permitiendo la entrada de texto desde el teclado, así como la introducción de comandos de edición. Cuando la función regresa al programa que la invocó, el *buffer* contendrá el texto que fue introducido y modificado por el usuario. Los parametros de entrada a la función son: el apuntador **wnd1** que especifica la dirección de la ventana de edición previamente establecida, el apuntador **bf** que direcciona el *buffer* de texto donde será almacenada la entrada de datos del editor, y el entero **bsize** que contiene el tamaño de *buffer* de texto. El número de líneas en el *buffer* de texto estara en función del tamaño del *buffer* y el ancho de la ventana.

(2.1)
En esta sección se inicializan las variables que proporcionan las características de las ventana de edición, así como los apuntadores en el *buffer* de texto.

x,y coordenadas de columna y renglon.
wht Representa la altura del área de texto de la ventana (sin incluir los caracteres del marco).
wwd Representa el ancho del área de texto de la ventana (sin incluir los caracteres del marco).
wsz Representa el tamaño del área de texto de la ventana.
lines Representa el número máximo de líneas que el *buffer* de texto puede contener.
topptr Es un apuntador a la primera posición de caracter en el *buffer* de texto.
bfptr Este apuntador va variando conforme el usuario haga paginaciones y *scrolls* en el *buffer* de texto, de manera que siempre quede apuntando a la posición de caracter localizada en la esquina superior izquierda de la ventana.
bf Apuntador auxiliar.
endptr Apunta a la ultima posición de caracter en el *buffer* de texto.
lstptr Apunta al ultimo caracter desplegable en el *buffer*.
blkbeg Marca el inicio de un bloque.
blkend Marca el final de un bloque.
depart Bandera que al tomar el valor "TRUE" indica la finalización de la entrada de texto en el editor.
tabctr Contador auxiliar en las funciones de tabulación.

Una vez inicializadas las variables antes descritas, la función **display_text** es invocada para desplegar el contenido del *buffer* asociado al editor. Inmediatamente después se

invoca la función **findlast** para que, en el caso de la existencia de texto previamente introducido, esta función localice el ultimo caracter introducido en el *buffer* y actualice el apuntador **lstptr**.

(2.2)
En esta sección se lleva a cabo la introducción de texto al *buffer*, así como la ejecución de comandos de edición.
Mediante un ciclo de control se lleva a cabo la introducción de texto al *buffer* o la ejecución de los comandos de edición. Para salir de este ciclo de control la variable **depart** deberá tener un valor de "TRUE", lo cual sólo sucede al introducir F2 o Esc que terminan la edición de texto y regresan al programa principal.
Dentro del ciclo de control, las coordenadas(x,y) de entrada son convertidas a sus respectivas coordenadas de ventana, y el apuntador *buff* es actualizado a la dirección en el *buffer* de texto correspondiente a las coordenadas(x,y).
A continuación se verifica si se ha dado algún comando de tabulación, en cuyo afirmativo éste es ejecutado y en caso contrario, se analizan los caracteres tecleados por el usuario para ver si corresponden a teclas de función o caracteres de control (la documentación de los comandos involucrados en este caso, queda implícita en la explicación dada al inicio de este capítulo) o forman parte del texto mismo.

(2.3)
Si el caracter tecleado no corresponde a un comando de edición, y es un ASCII desplegable, será copiado dentro del *buffer*. Primero se realiza una prueba para asegurarse que la introducción del caracter en el *buffer*: a) no rebasará la dirección final del mismo o b) que la acción de introducir dicho caracter, estando activado el modo "Insert", no provocará el desplazamiento del ultimo caracter del *buffer* más allá de la dirección final del mismo. Si esta prueba fue superada, antes de efectuar la copia del caracter en el *buffer*, se procede a verificar el estado *Insert/Overwrite*. Si el modo insert se encuentra activo, los caracteres posteriores son desplazados una posición a la derecha. En cualquiera de los casos, se procede al insertar el caracter en el *buffer*. Si el caracter introducido ocupa la ultima posición del texto, se actualiza el apuntador **lstptr** y la línea corriente es desplegada mostrando el cambio ocurrido.

Posteriormente se verifica si el nuevo caracter introducido da como resultado una palabra truncada. Las acciones que se toman al respecto se ilustran en el siguiente ejemplo:
(Condiciones: 10 columnas por línea)

a) Se verifica si la columna de prueba (ultima columna de una línea) -en nuestro caso columna #9- contiene un caracter diferente de blanco (caso palabra truncada). Coloca el apuntador "b" donde encuentre dos "blancos" seguidos o (si no los hubiere) al final del *buffer*.

```
Linea 0                    | Linea 1
0 1 2 3 4 5 6 7 8 9        | 0 1 2 3 4 5 6 7 8 9
i h o l a   b u e n        | d i a !
                  ↑        |       ↑
                buff       |       b
```

b) Se desplaza la linea siguiente a la palabra truncada en una posicion a la derecha y se coloca un blanco al inicio de dicha linea, con el fin de crear un espacio entre la palabra truncada y el texto de la línea siguiente.

```
Linea 0                    | Linea 1
0 1 2 3 4 5 6 7 8 9        | 0 1 2 3 4 5 6 7 8 9
i h o l a   b u e n        | d d i a !
i h o l a   b u e n        |   d i a !
```

c) La función **back word** coloca la coordenada "x" al inicio de la palabra truncada (con el fin de servir a la función carrtn -siguiente inciso- para pasar la palabra truncada al inicio de la linea siguiente).

```
Linea 0                    | Linea 1
0 1 2 3 4 5 6 7 8 9        | 0 1 2 3 4 5 6 7 8 9
i h o l a   b u e n        |   d i a !
            ↑   ↑          |
            x   buff       |
```

d) Se ejecuta la funcion **carrtn**, cuyo efecto será el siguiente:

```
Linea 0                    | Linea 1
0 1 2 3 4 5 6 7 8 9        | 0 1 2 3 4 5 6 7 8 9
i h o l a                  | b u e n   d i a !
```

e) Finalmente se actualiza el valor de la coordenada "x" a la posición que le corresponde según el contexto de la edición. Si el truncamiento ocurrió por "inserción" en alguna parte de la linea, la coordenada "x" se colocará a continuación de la ultima letra insertada. Y si el truncamiento ocurrio al estar editando, la coordenada "x" sera posicionada en la ultima letra de la palabra truncada.

BLOQUE (3)

DESCRIPCION DE LA FUNCION:
static void erase buffer(int *x, int *y)
Esta funcion limpia (llena de blancos) el *buffer* de texto a través de la función **setmem**. En primer lugar abre una ventana y pide confirmación al usuario antes de llevar a cabo su función. Todos los apuntadores y variables son reinicializados.

BLOQUE (4)

DESCRIPCION DE LA FUNCION: **static int lastword(int x, int y)**
Esta función verifica si la coordenada "x" se encuentra en la ultima palabra de la linea representada por la coordenada "y". La funcion devuelve el valor "0" cuando la coordenda "x" no se encuentra en la ultima palabra de la linea o cuando, aunque formara parte de dicha palabra, esta no es truncada. Y devuelve el valor "1" en caso contrario.

BLOQUE (5)

DESCRIPCION DE LA FUNCION: **void last char(int *x, int *y)**
Coloca la coordenada "x" a una posicion adicional a partir de donde se encuentra el ultimo caracter desplegable en la linea.

BLOQUE (6)

DESCRIPCION DE LA FUNCION: **void test para(int x, int y)**
Verifica si un párrafo deberá ser reformateado. La funcion se ejecuta si el modo scroll lock no se encuentra activo y la linea indicada por la coordenada "y" no sobrepase el numero maximo de lineas permisibles en el *buffer*. Posteriormente se invoca la funcion **trailing_spaces** para contar el numero de blancos sobrantes de la linea corriente (y). A continuacion se hace un llamado a la funcion **first_wordlen** que devuelve la longitud de la primer palabra de la linea siguiente (y+1), para ver si dicha palabra cabe en los espacios sobrantes de la linea anterior. En caso afirmativo, se invoca la funcion **paraform** que se encargara de reformatear todo el texto a partir de esa linea.

BLOQUE (7)

DESCRIPCION DE LA FUNCION: **static int trailing_spaces(int y)**
Esta función cuenta el numero de espacios sobrantes al final de una linea.

BLOQUE (8)

DESCRIPCION DE LA FUNCION: **static int firts_wordlen(int y)**
Devuelve la longitud de la primer palabra de una linea dada.

BLOQUE (9)

DESCRIPCION DE LA FUNCION:
static void paraform(int x, int y)
Esta funcion formatea un bloque marcado o bien todo el texto (hasta encontrar una linea en blanco o el fin del *buffer*) a partir de la linea indicada por la coordenada "y". Las acciones tomadas por esta función se ilustran en el siguiente ejemplo:

Condiciones: <F4> es presionada en la linea 0, cada linea consta de 10 columnas y se tienen 3 lineas con texto. Para hacer referencia a la variable **blkbeg** utilizaremos la letra "I", y para referenciar la variable **blkend** emplearemos la letra "F".

a) Si la linea corriente esta llena de blancos, la función simplemente regresa al proceso que la invocó.

b) Si no existe marcación de un bloque, la función inicializa sus propias marcas de bloque, colocando la variable **blkbeg** a la linea corriente e incrementando **blkend** hasta que su contenido sea el numero de linea anterior a una linea vacia, o bien al numero de linea anterior al fin del *buffer*. Los apuntadores auxiliares **cp1** y cp2 son inicializados a la dirección del primer caracter de la linea indicada por **blkbeg**. El apuntador auxiliar **cpend** es inicializado a la dirección del inicio de la linea posterior a la indicada por **blkend**.

```
Linea 0     Linea 1     Linea 2     Linea 3
0123456789  0123456789  0123456789  0123456789
yo    tu    el    nos   ustedes
↑                       ↑
cp1                     F
cp2                           ↑
1                          cpend
```

c) Las variables cp2 y cpend delimitan el espacio sobre el que se llevara a cabo la compactación de blancos. Cuando cp2 = cpend indicara que todo el bloque ya ha sido formateado. El apuntador cp1 siempre estará posicionado en la localidad a partir de la cual se debera insertar el texto apuntado por cp2 para eliminar los espacios innecesarios entre palabras. Las fases de este inciso son mostradas a continuación:

```
Linea 0     Linea 1     Linea 2     Linea 3
0123456789  0123456789  0123456789  0123456789
yo    tu    el    nos   ustedes
   ↑  ↑
  cp1 cp2

0123456789  0123456789  0123456789  0123456789
yo tu tu    el    nos   ustedes
      ↑     ↑
     cp1   cp2
0123456789  0123456789  0123456789  0123456789
yo tu el    el    nos   ustedes
         ↑  ↑     ↑
        cp1 svcp cp2
          x1
```

d) La variable "x1" siempre tiene el indice de cp1 y sirve para verificar si la palabra que se va a trasladar a esa posicion cabe en dicha linea o se truncaria (viola

la columna de prueba). En la fase anterior se ve que tanto cp1 como x1 se encuentran en la columna de prueba, por lo que el intento por trasladar la palabra "nos" daria como resultado un truncamiento. Por lo que el programa procederá de la siguiente manera:
Para evitar el posible truncamiento, se comenzará con el formateo de la linea siguiente (para indicar que la linea siguiente será la corriente, blkbeg es incrementado en 1). Para ello se actualizan los apuntadores afectados (x1 y cp1 deberán estar al inicio de la linea siguiente, efecto para el cual es utilizado el apuntador auxiliar svcp).

```
Linea 0      Linea 1      Linea 2      Linea 3
0123456789   0123456789   0123456789   0123456789
yo tu el     el   nos     ustedes
             ↑    ↑
          svcp    cp2
           cp1
            x1
             I

Linea 0      Linea 1      Linea 2      Linea 3
0123456789   0123456789   0123456789   0123456789
yo tu el     nos   nos    ustedes
                ↑         ↑
              cp1         cp2

0123456789   0123456789   0123456789   0123456789
yo tu el     nos ustede   ustedes
                    ↑         ↑
                  cp1         cp2
                   x1
                    └── Nota: Antes de insertar el
                              caracter "e" en la col.
                              de prueba verifica que
                              no se de truncamiento.
```

e) Caso especial de truncamiento.
Como hubo truncamiento el programa debera trasladar el apuntador cp1 a la columna 4 de la linea 1 (durante este proceso el programa ira introduciendo blancos) y cp2 debera ser colocado al inicio de la palabra que habia sido truncada. A continuacion se lleva a cabo el mismo proceso de actualizacion seguido en el inciso d).

```
Linea 0      Linea 1      Linea 2      Linea 3
0123456789   0123456789   0123456789   0123456789
yo tu el     nos ustede   ustedes
                    ↑    ↑    ↑       ↑
                  cp1  svcp  cp2      cpend
                   x1
```

```
0123456789    0123456789    0123456789    0123456789
yo tu el      nos           ustedes
                  ↑             ↑
                cp1           cp2
                              svcp

0123456789    0123456789    0123456789    0123456789
yo tu el      nos           ustedes
                                ↑
                               cp2
                          cp1=svcp
                                ↑
```

f) El texto formateado quedará como se muestra a continuación:

```
Linea 0       Linea 1       Linea 2       Linea 3
0123456789    0123456789    0123456789    0123456789
yo tu el      nos           ustedes
                                ↑             ↑
                                I           cp1   cp2=cpend
                                              x1

0123456789    0123456789    0123456789    0123456789
yo tu el      nos           ustedes
                                ↑             ↑
                                F           cp1
                                            cp2
                                            cpend
                                            I
```

Como al compactar el texto pudieron haber quedado líneas en blanco, se hace un llamado a la función **delete_block** para que las elimine. Finalmente las marcas de inicio y fin de bloque (**blkbeg** y **blkend**) son colocadas a cero, y las funciones **display_text** y **findlast** son invocadas para desplegar el texto ya reformateado y actualizar el apuntador **lstptr**.

BLOQUE (10)

DESCRIPCION DE LA FUNCION:
static int blankline(int line)
Esta función regresa 1 si la línea indicada está llena de blancos, y posiciona el apuntador **cp** al inicio de la línea corriente.

BLOQUE (11)

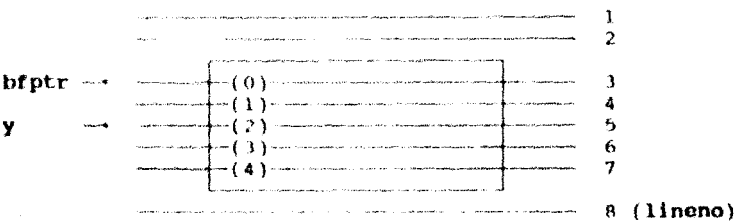
DESCRIPCION DE LA FUNCION:
static void delete word(int x, int y)
Esta función borra la palabra indicada por el cursor. Si la coordenada "x" esta posicionada en un "blanco", la función

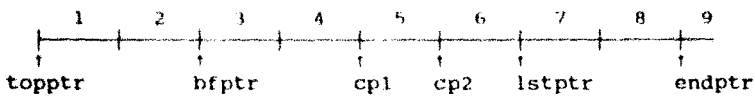
destruye todos los blancos hasta el inicio de la palabra siguiente; si este no es el caso, la función destruye la palabra desde la coordenada "x" hasta el inicio de la palabra siguiente.

BLOQUE (12)

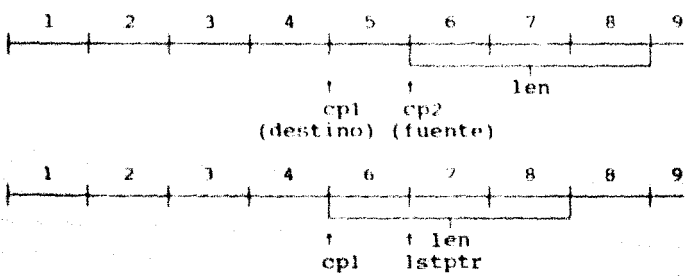
DESCRIPCION DE LA FUNCION:
static void delete line(int y)
Esta función borra una línea mediante un corrimiento del texto. Mediante las siguientes ilustraciones ejemplificaremos el procedimiento de esta función.

a) Suponer que se desea borrar la línea 2 de ventana que corresponde a la línea 5 en el *buffer*. El apuntador **lstptr** apuntara al inicio de la línea 7 del *buffer*.

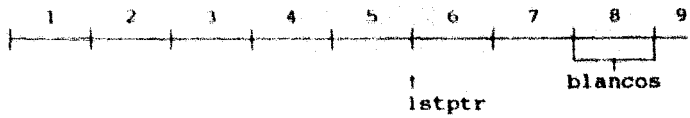




b) Se verifica si cp1 sobrepasa a lstptr, caso en el que se intenta borrar una línea llena de blancos, por lo que la función no se ejecutará. En caso contrario se procede como se ilustra a continuación:



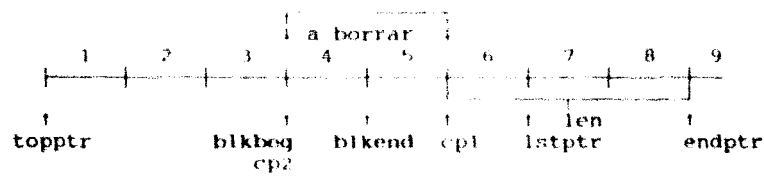
Finalmente se tiene:



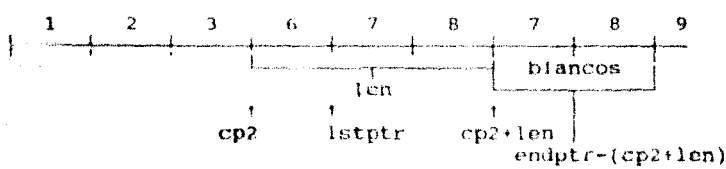
BLOQUE (13)

DESCRIPCION DE LA FUNCION:
static void delete_block()
Esta función borra el bloque delimitado por **blkbeg** y **blkend.**

a) Suponer que se desea borrar el bloque que engloba las líneas 4 y 5 respecto al *buffer*. El apuntador **lstptr** apuntará al inicio de la línea 7 del *buffer*.



b) Se verifica si existe marcación de bloque, en cuyo caso se procede según las figuras a continuación mostradas. En caso contrario sonará una alarma.



c) Finalmente se actualizan los números de línea en el *buffer* y las variables **blkbeg** y **blkend** se inicializan con cero.

BLOQUE (14)

DESCRIPCION DE LA FUNCION:
static void mvblock(int y, int moving)
Esta función es utilizada para mover y copiar un bloque de texto. La variable moving indicará el tipo de operación: TRUE para mover el bloque y FALSE para copiarlo.

a) Se verifica si no existe marcación de bloque, en cuyo caso sonara una alarma; en caso contrario se valida que el bloque no vaya a ser movido o copiado dentro de si mismo (si esto sucediera se desplegará un mensaje de error).

b) La variable "len" contendra el numero de lineas conformadas en un bloque. A continuación se comprueba si existe espacio suficiente en memoria para poder salvar, en un *buffer* temporal, el bloque marcado. Si es así se procede como se indica en el siguiente inciso, en caso contrario la función mvblock regresa al programa que la invocó.

c) Se copia el bloque marcado en el *buffer* temporal mediante la función **movmem**. Posteriormente se verifica el tipo de operacion a realizar (copia o movimiento). En el caso de una operación de movimiento de bloque se procede a borrarlo del *buffer* de texto mediante la función **delete_block**; posteriormente (y para ambos casos) se efectuan los pasos señalados en los siguientes incisos.

d) A partir de la linea indicada por el usuario, se abre un espacio en el *buffer* de texto para ahi insertar el contenido del *buffer* temporal (bloque marcado).

e) Finalmente, se devuelve al sistema la memoria ocupada por el *buffer* temporal; las variables blkbeg y blkend son inicializadas a cero y el texto modificado es desplegado.

BLOQUE (15)

DESCRIPCION DE LA FUNCION:
static void copy block(int y)
Esta función copia un bloque marcado a la linea indicada por el usuario. Hace uso de la función **mvblock** con **moving**=*FALSE* y de la función **findlast** para actualizar el apuntador **lstptr**.

BLOQUE (16)

DESCRIPCION DE LA FUNCION:
static void move_block(int y)
Esta función es llamada para mover un bloque marcado haciendo uso de la función mvblock.

BLOQUE (17)

DESCRIPCION DE LA FUNCION:
static void findlast()

Esta función encuentra el último caracter despleqable (diferente de blanco) en el *buffer* y actualiza el apuntador lstptr.

BLOQUE (18)
DESCRIPCION DE LA FUNCION:
static void find end(int *x, int *y)
Posiciona el cursor en la linea inmediata inferior a la última linea despleqable en el *buffer*.

BLOQUE (19)

DESCRIPCION DE LA FUNCION:
static void carrtn(int *x, int *y, int insert)
Ejecuta un "*Carriage Return Line-Feed*". Si el modo de insercion esta activo, la funcion divide la linea en cuestión mediante un corrimiento del contenido del *buffer* a la derecha, a partir de la posicion del cursor hasta el final de la linea corriente, y rellenando con blancos las posiciones generadas por el corrimiento. Hecho el corrimiento, la funcion formatea el texto posterior afectado. El apuntador lstptr es ajustado. El siguiente ejemplo ilustra el proceso de esta función.

a) Considerar: *buffer* con 2 lineas de texto con 10 columnas (**wwd**) cada una, modo de inserción activo y ejecución de la funcion estando el cursor en la columna 5 de la linea 0.

```
        Lineno 0      Lineno 1
        0123456789    0123456789
        Hola soy      Jim
        ↑     ↑       ↑
topptr      cp        nl
                      cp+insct
```

```
Aqui:
ctl  = 2
cp   = curr(5,0) = 5
nl   = cp + ((cp - topptr) % wwd)
     = 5  + ((5-0) % 10) = 10
```

b) A continuacion se verifica que al ejecutarse el carriage return no se sobrepase el final del *buffer*.

c) Superada la condición anterior, se determina el número de caracteres a bajar a la siguiente linea mediante: insct = wwd - *x = 10 - 5 = 5.

d) Mediante la variable ctl se lleva el control de los corrimientos en el *buffer*.

- Cuando **ctl** = 2
  Se efectua el corrimiento, haciendo uso de la función **movmem** y el relleno con blancos, con la función **setmem**. El apuntador **cp** es actualizado.
  cp = cp + (insct * 2) = 5 + (5 * 2) = 15
  (valor que corresponde a la columna 5, linea 1).

```
Lineno 0        Lineno 1
0123456789      0123456789
Hola            soy Jim
                    ↑
                   cp
```

- Cuando **ctl** = 1

```
Lineno 0        Lineno 1        Lineno 2
0123456789      0123456789      0123456789
Hola            soy             Jim
                                ↑     ↑
                                |  cp+insct*2
                             cp+insct
```

e) Se procede a colocar el cursor en su nueva posición (columna 0 de la linea 1) y a formatear el texto (en nuestro caso, a partir de la linea 1 en adelante).

f) El apuntador lstptr es actualizado y el texto es desplegado como sigue:

```
Lineno 0        Lineno 1
0123456789      0123456789
Hola            soy Jim
```

BLOQUE (20)

DESCRIPCION DE LA FUNCION:
static void backspace(int *x, int *y)
Coloca la coordenada de columna (x) una localidad a la izquierda; en caso de estar posicionados en la columna "0" de una linea, la funcion colocará la coordenada de renglón (y) en la linea anterior y la coordenada "x" en la ultima posición de columna en dicha linea.

BLOQUE (21)

DESCRIPCION DE LA FUNCION:
static void fore-word(int *x, int *y, char *bf)
Coloca la coordenada de columna y el apuntador del *buffer* al inicio de la palabra siguiente.

BLOQUE (22)

DESCRIPCION DE LA FUNCION:
static int spaceup(int *x, int *y, char **bf)
Coloca la coordenada de columna y el apuntador del *buffer* una localidad a la derecha, cambiando de linea si es necesario. Si se ha alcanzado el ultimo caracter desplegable en el *buffer*, la función devolverá "cero", en cualquier otro caso devolverá "1".

BLOQUE (23)

DESCRIPCION DE LA FUNCION:
static void forward(int *x, int *y)
Coloca la coordenada de columna una localidad a la derecha; en caso de estar posicionados en la última columna de una linea, la función colocará la coordenada de renglón en la linea siguiente, y la coordenada "x" en la primera posición de columna en dicha linea.

BLOQUE (24)

DESCRIPCION DE LA FUNCION:
static void back word(int *x, int *y, char *bf)
Coloca la coordenada de columna y el apuntador del *buffer* al inicio de la palabra anterior.

BLOQUE (25)

DESCRIPCION DE LA FUNCION:
static int spacedn(int *x, int *y, char **bf)
Coloca la coordenada de columna y el apuntador del *buffer* una localidad a la izquierda, cambiando de linea si es necesario. Si se ha alcanzado el inicio del *buffer*, la función devolverá "0", en cualquier otro caso devolverá "1".

BLOQUE (26)

DESCRIPCION DE LA FUNCION:
static int downward(int *y)
Coloca la coordenada de renglon en la linea siguiente. Regresa "1" si tuvo exito o "0" si alcanzó el fin del *buffer*. Esta funcion, a su vez, realiza el *scroll* hacia arriba en caso de ser necesario.

BLOQUE (27)

DESCRIPCION DE LA FUNCION:
static void upward(int *y)

Coloca la coordenada de renglon en la linea anterior. En caso de estar posicionados en la primer linea del *buffer*, la función regresará al programa que la invocó. Esta función, a su vez, realiza el *scroll* hacia abajo en caso de ser necesario.

BLOQUE (28)

DESCRIPCION DE LA FUNCION:
static void display text()
Despliega el contenido de una ventana, linea por linea, siempre que la bandera **do_display_text** esté encendida.

BLOQUE (29)

DESCRIPCION DE LA FUNCION:
static void disp_line(int y)
Despliega una linea de texto, empleando la función **displ** vista en el capitulo VI (libreria de funciones de ventana). Si la linea en cuestion forma parte de un bloque marcado, la funcion desplegara la linea en atributo acentuado.

BLOQUE (30)

DESCRIPCION DE LA FUNCION: **static void insert line()**
Coloca la forma del cursor " ▮ " para modo *insert*, y "_" para modo *overwrite*.

Coloca la coordenada de renglon en la linea anterior. En caso de estar posicionados en la primer linea del *buffer*, la función regresara al programa que la invoco. Esta función, a su vez, realiza el *scroll* hacia abajo en caso de ser necesario.

BLOQUE (28)

DESCRIPCION DE LA FUNCION:
static void display text()
Despliega el contenido de una ventana, linea por linea, siempre que la bandera **do display_text** esté encendida.

BLOQUE (29)

DESCRIPCION DE LA FUNCION:
static void disp line(int y)
Despliega una linea de texto, empleando la función **displ** vista en el capitulo VI (libreria de funciones de ventana). Si la linea en cuestion forma parte de un bloque marcado, la función desplegara la linea en atributo acentuado.

BLOQUE (30)

DESCRIPCION DE LA FUNCION: **static void insert line()**
Coloca la forma del cursor " ▮ " para modo *insert*, y "_" para modo *overwrite*.

PROGRAMA DE EJEMPLO QUE HACE USO DE LA FUNCION DE EDICION DE TEXTO

Basandonos en la funcion text editor realizaremos un *notepad* en linea. El cual consta de 3 modulos (tal como los programas de ejemplo anteriores) : un programa manejador, otro con el cuerpo de la funcion principal a la que está orientado el ejemplo, y un archivo "proyecto" (extension prj), que Turbo C utiliza para construir el programa ejecutable.

- **EJEMPLO No.1**
  PROGRAMA EDITOR DE TEXTO (*NOTEPAD*).

LISTADO #13

LISTADO 13.1
Programa Manejador.

```
/*       note.c
     ------------ */
#include "twindow.h"

void notepad(void);
char notefile [] = "note.pad";

void main()
{
     load help("tcprogs.hlp");
     notepad();
}
```

LISTADO 13.2
Programa que contiene el codigo de la funcion principal.

```
/* ------- notepad.c ------ */

#include <stdio.h>
#include <mem.h>
#include "twindow.h"
```

```
#define LWID 60
#define WHT 10
#define PADHT 20

char bf [PADHT] [LWID];
extern char notefile[];

void notepad()
{
     WINDOW *wnd;
     FILE *fp, *fopen();
     int i, ctr = 0;

     set_help("notepad ", 0, 0);
     setmem(bf, sizeof bf, ' ');
     if ((fp = fopen(notefile, "rt")) != NULL)     {
          while (fread(bf [ctr], LWID, 1, fp))
               ctr++;
          fclose(fp);
     }
     wnd = establish window
          ((80-(LWID+2))/2, (25-(WHT+2))/2, WHT+2, LWID+2);
     set_border(wnd, 3);
     set_title(wnd, "  Note Pad  ");
     set_colors(wnd, ALL, BLUE, AQUA, BRIGHT);
     set colors(wnd, ACCENT, WHITE, BLACK, DIM);
     display window(wnd);
     text_editor(wnd, bf[0], (unsigned) LWID * PADHT);
     delete_window(wnd);
     ctr = PADHT;
     while (--ctr)  {
          for (i = 0; i < LWID; i++)
               if (bf [ctr] [i] != ' ')
                    break;
          if (i < LWID)
               break;
     }
     fp = fopen(notefile, "w");
     for (i = 0; i < ctr+1; i++)
          fwrite(bf[i], LWID, 1, fp);
     fclose(fp);
}
```

LISTADO 13.3
Programa que construye el archivo ejecutable.

```
/* ------ note.prj ------ */
note
notepad (twindow.h)
editor (twindow.h, keys.h)
thelp (twindow.h, keys.h)
twindow (twindow.h, keys.h)
ibmpc.obj
```

COMENTARIOS AL LISTADO #13

El programa del listado 13.1 carga el archivo de ayuda tcprogs.hlp mediante una llamada a la función load_help; posteriormente invoca a la función **notepad**, cuyo código se encuentra en el listado 13.2. En este listado, el programa **notepad.c** opera en un archivo declarado en un arreglo externo definido como **note.pad**. Si el archivo ya había sido creado, **notepad.c** lo lee y almacena su contenido en un *buffer*. Este programa establece una ventana, le asigna un borde, título y colores para luego desplegarla; a continuación llama a la función text editor para procesar la introducción de texto. Cuando esta función regresa, **notepad.c** destruye la ventana y localiza la línea en el *buffer* en la que el último texto significativo haya sido escrito. Finalmente el texto introducido por el usuario, será salvado en el archivo **note.pad** al momento de salir del programa **notepad.c** presionando ‹Esc› o ‹F2›.

VENTANAS DE MENU

CAPITULO 9

INTRODUCCION

En los capitulos anteriores se han venido manejando los programas como módulos independientes; pero debido a que un programa interactivo en linea comunmente ofrece diversos procesos al usuario, en este capitulo nos daremos a la tarea de agrupar dichos programas independientes en uno solo que presentará las posibles opciones en forma de menu.

Existen muchas técnicas para presentar al usuario las opciones de un "sistema". A la mayoría de estas técnicas se les conoce como **menús** y al programa que despliega ventanas de menus y que permite seleccionar una operacion, se le denomina programa "ejecutor" del sistema o interfaz de usuario.

Anteriormente, se vio un ejemplo del uso de un tipo de menu en el programa de ejemplo **tabla.exe** del capitulo VI. Dicho menú proporciona una lista de opciones en una ventana. El usuario puede seleccionar presionando la tecla correspondiente a la opción deseada, o desplazando hasta ella el cursor de barra y presionando <Enter>. Esta tecnica de menu es muy común, eficaz y fácil de comprender.

Otro formato muy conocido para la elaboracion de menus, a menudo visto en computadoras que hacen uso de una interfaz de usuario con gráticos y *mouse*, es el "menu de barra deslizante" (técnica que emplearemos en este capitulo). Este formato consta de un menú horizontal en la parte superior de la pantalla que a su vez cuenta con menús verticales *pop-down* para cada una de las opciones. El usuario puede mover el cursor de barra deslizante de una opcion a otra sobre el menú horizontal con la ayuda de las teclas de direccion de cursor (← y →). El menu vertical correspondiente sera desplegado debajo de la opción seleccionada en el menú horizontal. Para seleccionar opciones del menu vertical, el usuario tiene a su disposicion otro cursor de barra que puede ser movido con las teclas de dirección de cursor (↓ y ↑). La ventaja de esta técnica de menú es que ocupa el mínimo espacio en pantalla dejando los menús de la aplicación, en su mayor parte, a la vista del usuario. Los menus verticales son ventanas desplegables, de modo que cubren temporalmente un área de la pantalla sin destruir la información encontrada debajo de éstas. Esta tecnica de presentación de menú debe ser ya familiar al lector, pues hoy día la mayoria de las aplicaciones hacen uso de ella.

EL PROGRAMA " EJECUTOR " DEL MENU DE VENTANA.

El programa "ejecutor" utiliza una función manejadora de menú que emplea la técnica de menú de barra deslizante. Los menús son construidos en base a ventanas y las opciones tanto para el menú de barra deslizante como para cada uno de los menus verticales son tomadas de una serie de tablas proporcionadas por el programa que invoca a la funcion manejadora.

La función manejadora de menu lleva a cabo la interfaz con el usuario, controlando el despliegue de los menús y la selección de opciones de usuario. También se encarga de la ejecución de las funciones de aplicación (opciones) en el programa. En resumen, el proceso es manejado completamente por un conjunto de tablas de menú que deben ser codificadas en el programa.

La tabla de menú primaria consiste en un arreglo de estructuras **MENU**. Dicha estructura está definida en el listado #3 **twindow.h**, del capitulo VI. Este arreglo tiene una entrada para cada una de las opciones que integran el menú horizontal de barra deslizante. Cada una de esas entradas contiene el nombre de la opción del menu horizontal a ser desplegada, así como algunos apuntadores para describir el contenido del menú vertical asociado a dicha opcion. A su vez los apuntadores incluyen: un apuntador a un arreglo de nombres de las opciones del menú vertical y otro a un arreglo de apuntadores a funciones de C (funciones de aplicacion). Posteriormente dichos "nombres" son desplegados en el menu vertical; las funciones de aplicacion son proporcionadas por el programa principal (en nuestro caso el programa ejemplo de menú) y son ejecutadas cuando el usuario hace la selección correspondiente en el menú.

La jerarquía de menus tiene dos niveles. El primer nivel está representado por las opciones del menú horizontal y se encuentra limitado a seis opciones (numero maximo de opciones permisibles en una linea). A su vez, cada opcion de este nivel genera un menú vertical desplegable que conforma el segundo nivel de la jerarquía. El menu vertical esta limitado a 21 opciones. Cada una de las opciones de este nivel ejecuta una función de aplicacion proporcionada por el programa que invocó a la función manejadora de menú.

Para tener niveles adicionales en la organización jerárquica de menú, se debe anadir un conjunto de tablas de menú y realizar llamadas recursivas a la función manejadora de menú. Debido a que esta función es reentrante una opcion, seleccionada en el menú vertical del segundo nivel, puede generar un nuevo menú horizontal.

El programa de aplicacion que mostraremos proporciona los arreglos de menu que describen la organizacion jerárquica antes mencionada.

FUNCION MANEJADORA DE MENU

Para utilizar las ventanas de menu, se cuenta con la función **menu_select** que deberá ser invocada por un programa principal. Para utilizar dicha función, previamente se deberá proporcionar lo siguiente: el arreglo de estructuras **MENU**, los arreglos a los que el arreglo de estructuras **MENU** apunta y las funciones de aplicacion que serán ejecutadas cuando las opciones del menú vertical sean seleccionadas por el usuario.

LISTADO #14

```
/*    tmenu.c
/* ---------- */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "keys.h"
#include "twindow.h"

extern int VSG;

WINDOW *open_menu(char *mnm, MENU *mn, int hsel);
int gethmenu(MENU *mn, WINDOW *hmenu, int hsel);
int getvmn(MENU *mn, WINDOW *hmenu, int *hsel, int vsel);
int haccent(MENU *mn, WINDOW *hmenu, int hsel, int vsel);
void dimension(char *sl[], int *ht, int *wd);
void light(MENU *mn, WINDOW *hmenu, int hsel, int d);

BLOQUE (1)
/* --- Despliega y procesa un menu --- */
void menu_select(char *name, MENU *mn)
{
     WINDOW *open_menu();
     WINDOW *hmenu;
     int sx, sy;
     int hsel = 1, vsel;

     curr_cursor(&sx, &sy);
     cursor(0, 26);
     hmenu = open_menu(name, mn, hsel);
```

```
    while ((hsel = gethmenu(mn, hmenu, hsel)) != 0)   {
         vsel = 1;
         while ((vsel = getvmn(mn, hmenu, &hsel, vsel)) != 0)  {
              delete_window(hmenu);
              set_help("", 0, 0);
              (*(mn+hsel-1)->func [vsel-1])(hsel, vsel);
              hmenu = open_menu(name, mn, hsel);
         }
    }
    delete_window(hmenu);
    cursor(sx, sy);
}

BLOQUE (2)
/* --- Abre un menu horizontal --- */
static WINDOW *open_menu(char *mnm, MENU *mn, int hsel)
{
    int i = 0;
    WINDOW *hmenu;

    set_help("menu    ", 30, 10);
    hmenu = establish_window(0, 0, 3, 80);
    set_title(hmenu, mnm);
    set_colors(hmenu, ALL, BLUE, AQUA, BRIGHT);
    set_colors(hmenu, ACCENT, WHITE, BLACK, DIM);
    display_window(hmenu);
    while ((mn+i)->mname)
         wprintf(hmenu, " %-10.10s ", (mn+i++)->mname);
    light(mn, hmenu, hsel, 1);
    cursor(0, 26);
    return hmenu;
}

BLOQUE (3)
/* --- Obtiene la opcion horizontal seleccionada --- */
static int gethmenu(MENU *mn, WINDOW *hmenu, int hsel)
{
    int sel;

    light(mn, hmenu, hsel, 1);
    while (TRUE)   {
         switch (sel = get_char())     {
              case FWD:
              case BS:  hsel = haccent(mn, hmenu, hsel, sel);
                        break;
              case ESC:  return 0;
              case '\r': return hsel;
              default:   putchar(BELL);
                        break;
         }
    }
}
```

BLOQUE (4)

```
/* --- Despliega la ventana vertical con sus opciones --- */
static int getvmn(MENU *mn,WINDOW *hmenu,int *hsel,int vsel)
{
     WINDOW *vmenu;
     int ht = 10, wd = 20;
     char **mp;

     while (1) {
          dimension((mn+*hsel-1)->mselcs, &ht, &wd);
          vmenu = establish_window(2+(*hsel-1)*12, 2, ht, wd);
          set_colors(vmenu, ALL, BLUE, AQUA, BRIGHT);
          set_colors(vmenu, ACCENT, WHITE, BLACK, DIM);
          set_border(vmenu, 4);
          display_window(vmenu);
          mp = (mn+*hsel-1)->mselcs;
          while(*mp)
               wprintf(vmenu, "\n%s", *mp++);
          vsel = get_selection(vmenu, vsel, "");
          delete_window(vmenu);
          if (vsel == FWD || vsel == BS)     {
               *hsel = haccent(mn, hmenu, *hsel, vsel);
               vsel = 1;
          }
          else
               return vsel;
     }
}
```

BLOQUE (5)

```
/* ---- Indica la opcion del menu horizontal que debe */
/*      ser intensificada ---- */
static int haccent(MENU *mn,WINDOW *hmenu,int hsel,int sel)
{
     switch (sel)   {
          case FWD:
               light(mn, hmenu, hsel, 0);
               if ((mn+hsel)->mname)
                    hsel++;
               else
                    hsel = 1;
               light(mn, hmenu, hsel, 1);
               break;
          case BS:
               light(mn, hmenu, hsel, 0);
               if (hsel == 1)
                    while ((mn+hsel)->mname)
                         hsel++;
               else
                    --hsel;
               light(mn, hmenu, hsel, 1);
               break;
          default:
```

```
                break;
        }
        return hsel;
}
```

BLOQUE (6)

```
/* --- Calcula la altura y ancho de una ventana de menú --- */
static void dimension(char *sl[], int *ht, int *wd)
{
        unsigned strlen(char *);

        *ht = *wd = 0;
        while (sl [*ht])
                *wd = max(*wd, strlen(sl [*ht]));
                (*ht)++;
        }
        *ht += 2;
        *wd += 2;
}
```

BLOQUE (7)

```
/* --- Intensifica una opción del menú horizontal --- */
static void light(MENU *mn, WINDOW *hmenu, int hsel, int d)
{
        if (d)
                reverse_video(hmenu);
        wcursor(hmenu, (hsel-1)*12+2, 0);
        wprintf(hmenu, (mn+hsel-1)->mname);
        normal video(hmenu);
        cursor(0, 26);
}
```

DOCUMENTACION LISTADO #14

BLOQUE (1)

DESCRIPCION DE LA FUNCION:
void menu_select(char *name, MENU *mn)

Al ser invocada, la función activa el proceso de menú de barra deslizante horizontal y espera a que el usuario seleccione alguna opcion. Sus parámetros de entrada son: el apuntador **name**, que contiene el titulo del menú de barra deslizante; y el apuntador **mn**, que contiene la dirección del arreglo de estructuras **MENU**.
Una vez que la funcion ha desplegado el menu horizontal, el usuario puede mover el cursor de barra deslizante haciendo uso de las teclas de direccion derecha e izquierda (→,←). Si el usuario presiona la tecla <Esc>, el proceso regresará al programa que invocó a la función **menu_select**. Ahora bien, si

el usuario presionó la tecla <Enter>, será desplegado el menú vertical asociado a la opción horizontal seleccionada. Mientras esté visible un menú vertical, el usuario puede moverse de un menú vertical a otro haciendo uso de las teclas de dirección izquierda o derecha. Cabe aclarar que al desplazarse de esta manera, sólo permanecerá en pantalla el menú vertical en el que se encuentre localizado el cursor. Para seleccionar alguna opción del menú vertical, el usuario deberá posicionarse en la opción deseada haciendo uso de las teclas de dirección y luego presionar <Enter>; en ese instante, tanto el menú vertical como el horizontal serán ocultados y ejecutará el programa asociado a la opción vertical seleccionada. Cada uno de estos programas son proporcionados por el *software* que invoco a la función **menu_select**. Al terminar la ejecución del programa los menús son desplegados nuevamente en pantalla y el proceso continúa. Presionando la tecla <Esc>, el menú vertical corriente es borrado y el control es devuelto a la rutina que procesa el menú horizontal, descrita en el párrafo anterior.

La descripción a nivel código de esta función es la siguiente:

La función **menu_select** es llamada para procesar un menú que se encuentra definido en un arreglo de estructuras **MENU**. Al iniciar la función, salva la localización corriente del cursor y coloca el cursor en una posición no visible en la pantalla.

Posteriormente, la función **open_menu** es invocada para crear y desplegar el menú horizontal en la parte superior de la pantalla. Un ciclo "*while*", se encarga de procesar las selecciones realizadas en el menú horizontal hasta que el usuario introduzca la tecla <Esc>. En cada iteración del ciclo se invoca a la función **gethmenu**, que regresará un valor cero cuando el usuario haya presionado la tecla <Esc> y un valor diferente de cero (que especifica el número de opción seleccionada) cuando el usuario haya realizado una selección en el *menú horizontal*. Un segundo ciclo "*while*", se encarga de procesar el menú vertical asociado con la opción seleccionada en el menú horizontal. En cada iteración del ciclo, se invoca a la función **getvmn** que regresará un valor cero cuando el usuario haya presionado la tecla <Esc>, o bien el valor de la opción seleccionada (cuando éste haya sido el caso). En ese momento, las ventanas de menú vertical y horizontal son borradas, la ventana de ayuda es inicializada con la dirección de la correspondiente al programa de aplicación seleccionado y dicho programa es ejecutado.

Una vez que el programa de aplicación finaliza, la función **menu_select** vuelve a desplegar las ventanas de menú horizontal y vertical, tal como se encontraban antes de la ejecución del programa de aplicación.

BLOQUE (2)

DESCRIPCION DE LA FUNCION:
static WINDOW *open_menu(char *mnm, MENU *mn, int hsel)

Esta función crea una ventana horizontal en la parte superior de la pantalla, coloca la ventana de ayuda para menu y despliega las opciones de menú horizontales por medio de la función **wprintf**, tomando el contenido del miembro **mname** de la estructura **MENU**. La primer opcion aparecerá intensificada, lo cual será llevado a cabo por medio de la función **light**.
Los parámetros de entrada para esta función son los siguientes:

mnm: Es un apuntador al titulo de la ventana de menú horizontal.
mn: Es un apuntador a la estructura MENU. El miembro que nos interesa de dicha estructura es ***mname*** que será usado para desplegar las opciones del menú horizontal.
hsel: Indica la opción horizontal a ser resaltada.

Finalmente esta función regresa, un apuntador a la ventana de menú horizontal a la funcion que la invoco.

BLOQUE (3)

DESCRIPCION DE LA FUNCION:
static int gethmenu(MENU *mn, WINDOW *hmenu, int hsel)

Esta funcion explora el teclado para ver si alguna opción del menu horizontal ha sido seleccionada. Mientras el usuario presione las teclas de direccion ← o →, la función **haccent** es invocada para mover la barra de cursor, intensificando la opcion indicada al presionar la tecla de dirección y actualizando el numero asociado a dicha opción.
Las opciones para salir de esta función son:

Esc: En ese momento la función devuelve un valor de cero indicando que no se desea seleccionar opción.
Enter: En ese momento la función devuelve el valor asociado a la opcion horizontal seleccionada (**hsel**).

BLOQUE (4)

DESCRIPCION DE LA FUNCION:
static int getvmn(MENU *mn,WINDOW *hmenu,int *hsel,int vsel)

Esta función despliega y procesa los menús verticales para cada una de las opciones del menu horizontal. El menú vertical es establecido por medio de una ventana y sus opciones son desplegadas tomando el contenido del miembro **mselcs** de la estructura **MENU**. La función **get_selection**

(vista en el capitulo VI) es usada para obtener del usuario la selección de alguna opcion vertical. Existen varias opciones para salir de esta función: con <Esc> devuelve un valor de cero indicando que el usuario no desea seleccionar sobre el menu vertical, con <Enter> se selecciona una opción vertical y se devuelve en **vsel** el valor asociado a dicha opción, o bien con las teclas de dirección ← o → con lo que **get selection** devolverá el código del caracter tecleado. En el momento en que la seleccion de la opción vertical sea realizada, la ventana vertical será borrada. El valor regresado en **vsel** desde **get selection** es pasado a la función que invocó a **getvmn**.
Ahora bien si **get selection** devuelve el código de las teclas ← o → , indica que el usuario desea desplazarse directamente de un menu vertical a otro, por lo que se hará uso de la función **haccent**, para avanzar o retroceder el cursor del menu de barra horizontal, luego el cursor de menú vertical será colocado en la primera opción de la ventana corriente y se continuara con el procesamiento del nuevo menu vertical seleccionado, tal como se explico anteriormente.

BLOQUE (5)

DESCRIPCION DE LA FUNCION:
static int haccent(MENU *mn,WINDOW *hmenu,int hsel,int sel)

Esta función indica qué opción del menu horizontal debe ser intensificada (por haber sido seleccionada). Siempre que se invoque esta función, el parámetro **sel** tendrá el código de las teclas de dirección derecha o izquierda (→ o ←).
Lo primero que hace esta función es avanzar o retroceder el cursor de menu horizontal de acuerdo a lo indicado en la variable **sel**. En cada avance o retroceso del cursor horizontal será intensificada la opción corriente y la opcion anterior será "opacada" por medio de la función **light()**. La variable **hsel** tambien será actualizada, esta indica en que opción del menú horizontal se encuentra el cursor actualmente. Finalmente, el valor de **hsel** actualizado es devuelto a la función que invoco a **haccent**.

BLOQUE (6)

DESCRIPCION DE LA FUNCION:
static void dimension(char *sl[], int *ht, int *wd)

Esta función es usada para calcular la altura y ancho de una ventana vertical. La altura dependerá del numero de opciones de la ventana vertical en cuestion; y el ancho, del número de caracteres de la opcion de mayor longitud de dicha ventana. Finalmente, a los valores de base y altura de ventana obtenidos se les agrega un valor de 2 (a cada uno); esto, para considerar los caracteres que serán ocupados al formar el marco de la ventana.

BLOQUE (7)

DESCRIPCION DE LA FUNCION:
static void light(MENU *mn, WINDOW *hmenu, int hsel, int d)

Esta función es usada para intensificar (modo de video inverso) la opcion corriente del menu horizontal y para opacar (modo de video normal) la opción anterior.

Al invocar esta función se le debe indicar por medio del parámetro **d** la acción que se desea ejecutar: para intensificar una opción (**d=1**), para opacar una opción (**d=0**). Por medio del parámetro **hsel** se le indica a la función la opción sobre la que debe modificar el atributo de video.

PROGRAMA DE EJEMPLO QUE HACE USO DE LA FUNCION MANEJADORA DE MENU

Basandonos en la función **menu_select**, en esta parte, realizamos un programa de ejemplo que ilustra el procesamiento de menú. Este programa consta de 3 módulos (tal como los programas de ejemplo anteriores) : un programa manejador, otro con el cuerpo de la función principal a la que está orientado el ejemplo y un archivo "proyecto" (extensión prj) que Turbo C utiliza para construir el programa ejecutable. El programa construye y ejecuta un menú que integra en un sólo módulo ejecutable todos los programas de ejemplo vistos en capítulos anteriores (del capítulo VI al VIII).

■ **EJEMPLO No.1**
PROGRAMA DE PROCESAMIENTO DE MENU.

LISTADO #15

LISTADO 15.1
Programa Manejador.

```
/* ---------- menu.c ---------- */

#include "twindow.h"
void exec(void);

char notefile [] = "note.pad";

void main()
{
     load_help("tcprogs.hlp");
     exec();
}
```

LISTADO 15.2
Programa que contiene el código de la función principal.

```
/* --- exec.c --- */

#include <stdio.h>
#include "twindow.h"
/* --- funciones locales --- */
void testmove(void);
void promote(void);
void ccolor(void);
void notepad(void);
void funtabla(void);
void maxims(void);
/* --- tablas con opciones de menu vertical --- */
char *dselcs[] = {
      " move ",
      " promote ",
      " colors ",
      NULL
};
char *pselcs[] = {
      " notepad ",
      " tabla ",
      " sayings ",
      NULL
};
static void (*dfuncs[])()={testmove,promote,ccolor};
static void (*pfuncs[])()={notepad,funtabla,maxims};
static MENU tmn[] = {
      {" demos ",    dselcs, dfuncs},
      {" programs ", pselcs, pfuncs},
      {NULL,NULL,NULL}
};

void exec()
{
      menu_select(" Menu Tesis ", tmn);
}
```

LISTADO 15.3
Programa que construye el archivo ejecutable.

```
/* ------ menu.prj ------ */
menu.c
exec.c (twindow.h, keys.h)
testmove (twindow.h, keys.h)
promote (twindow.h, keys.h)
ccolor (twindow.h, keys.h)
notepad (twindow.h)
maxims (twindow.h, keys.h)
funtabla (twindow.h, keys.h)
editor (twindow.h, keys.h)
```

thelp (twindow.h, keys.h)
tmenu (twindow.h)
twindow (twindow.h, keys.h)
ibmpc.obj

COMENTARIOS AL LISTADO #15

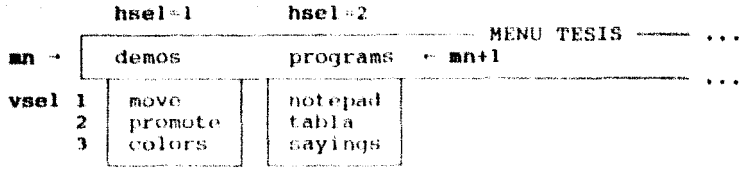
El programa del listado 15.1 carga el archivo de ayuda toprogs.hlp mediante una llamada a la función **load_help**; posteriormente invoca a la función **exec()**, cuyo código se encuentra en el listado 15.2. El programa **exec.c** contiene 2 arreglos de apuntadores a caracteres (**dselcs** y **pselcs**). El primero incluye todas las opciones de ventana vertical asignadas a la primer opción del menú horizontal, estas opciones corresponden a los nombres de los programas de demostración (creación, uso y movimiento) de ventanas vistos en el capítulo VI. En el segundo arreglo se tienen todas las opciones de ventana vertical asignadas a la segunda opción de la ventana horizontal, dichas opciones corresponden a los nombres de los programas de ejemplo mostrados al final de cada uno de los capítulos anteriores (del capítulo VI al VIII).

El programa **exec.c** también incluye 2 arreglos de apuntadores a funciones (**dfuncs** y **pfuncs**). El primero contiene las direcciones de los programas de demostración que serán ejecutados al ser seleccionada alguna opción del primer menú vertical (**dselcs**), y el segundo contiene las direcciones de los programas de ejemplo que serán ejecutados al seleccionar alguna opción de la segunda ventana vertical (**pselcs**).

También en el código de **exec.c** se tiene definido un arreglo de estructuras **MENU** denominado **tmn**. En este caso el arreglo únicamente tendrá 2 estructuras de menú definidas, debido a que sólo tenemos 2 opciones para la ventana de menú horizontal. Cabe mencionar que existe espacio suficiente para tener un máximo de seis opciones para el menú horizontal. Regresando a nuestro ejemplo, existen dos entradas en **tmn**, una para cada opción del menú horizontal. Cabe recordar que la estructura **MENU** definida en el listado #5 **twindow.h** (capítulo VI) tiene tres miembros: un apuntador a la cadena de caracteres que definen el nombre de la opción horizontal, un apuntador a un arreglo de nombres que corresponden a las opciones del menú vertical (que ya fueron definidos en **dselcs** y **pselcs**), y un apuntador a un arreglo de apuntadores a funciones (que puede ser **dfuncs** o **pfuncs** según sea el caso).

Finalmente, una vez que se ha realizado todo lo descrito anteriormente, también se debe dar el título de la ventana de menú horizontal para así poder pasar todos los parámetros a la función **menu_select()** y pueda ésta ser invocada para llevar a cabo el procesamiento de los menus.

Todo lo visto aquí puede quedar esquematizado de la siguiente manera:



Nota: Para salir, se debe presionar la tecla <Esc> 2 veces. Cuando el programa detecte el primer <Esc> terminará y quitará de pantalla el menú vertical, con el segundo <Esc> procederá de la misma manera con el menú horizontal al tiempo que saldrá también del programa.

Con este capítulo damos por terminada la exposición del tema "ambiente de ventanas" propio de los programas TSR. En esta parte hemos completado la biblioteca de funciones de ventana que se ha venido integrando desde el capítulo VI, proporcionando al programador las herramientas necesarias para el desarrollo de un propio ambiente interactivo. En el mercado existen algunos paquetes de bibliotecas de ventanas, pero como ya hemos mencionado con anterioridad, nuestra intención es la de involucrar a fondo al programador en los aspectos que consideramos de vital importancia para su capacitación en el área de Programas Residentes en Memoria.

PROGRAMA MANEJADOR DE TSRs

CAPITULO 10

INTRODUCCION

En este capitulo se presentan 2 programas que concretan todo el proceso (que hemos venido estudiando desde el capitulo III) para la implementación de programas residentes. En esta sección llevaremos a la practica la teoria presentada a lo largo de este trabajo. El primer programa constituye un ejemplo de TSRs pasivos y el segundo, de TSRs activos. El objeto de incluir ambos tipos de TSRs es el de mostrar al lector el contraste entre ellos; sin embargo la metodología propuesta en este libro (capitulo IV) se ve reflejada en el segundo programa. La función del programa TSR pasivo es mantener un despliegue constante en pantalla de la hora y fecha corrientes. El segundo programa es un manejador de TSRs de proposito general.

Siguiendo algunas pautas para la inicialización del ambiente, los "programas ejemplo" en Turbo C vistos en capitulos anteriores seran ligados al programa manejador. El conjunto de todas estas aplicaciones formaran un paquete de utilerías residente.

Anotaciones respecto al programa de utilería residente:

- Interrumpirá al sistema o a cualquier proceso corriente al detectar la *hotkey*.
- Abrirá, cerrara, leerá y escribirá archivos en disco.

- Utilizará funciones del ROM-BIOS para la entrada de teclado, y realizará el acceso directo a la memoria de video para manejar los despliegues en pantalla.
- Deberá ser compilado bajo el modelo de memoria *tiny* o *small* de Turbo C.
- Terminará por sí mismo, sin valerse del sistema operativo.

En el capítulo IX fueron integrados todos los "programas ejemplo" de los capítulos anteriores en un programa ejecutable controlado por un menú. En este capítulo se utilizará dicho programa para convertirlo en un paquete de utilería TSR.

PROGRAMA TSR PASIVO.
RELOJ EN LINEA

El listado 16.1 "reloj.c" constituye una utilería TSR simple que proporciona un despliegue constante de fecha y hora en la esquina superior derecha de la pantalla. Este programa, una vez residente, no hace llamadas al DOS; de modo que no necesita ser protegido para el problema de reentrancia del DOS.

LISTADO #16

LISTADO 16.1
Programa que contiene el código de la función principal.

```
/*   reloj.c
/* ---------- */

#include <stdio.h>
#include <dos.h>

BLOQUE (1)
unsigned vmode(void);
void vpoke(unsigned, unsigned, unsigned);
static void interrupt (*oldtimer)(void);
static void interrupt newtimer(void);
extern void interrupt (*ZeroDivVector)(void);
#define sizeprogram 375
unsigned intsp, intss;
unsigned myss, stack;
static union REGS rg;
struct date dat;
struct time tim;
unsigned vseg;
int running = 0;
char bf[20];
unsigned v;
char tmsk [] = " %2d-%02d-%02d %02d:%02d:%02d ";
int ticker = 0;
static struct SREGS seg;
```

BLOQUE (2)

```
void main()
{
     segread(&seg);
     /*  salva la pila del programa residente  */
     myss = _SS;
     /*  liga del vector de interrupción de reloj  */
     oldtimer = getvect(0x1c);
     setvect(0x1c, newtimer);
     stack = (sizeprogram - (seg.ds - seg.cs)) * 16 - 300;
     vseg = vmode() == 7 ? 0xb000 : 0xb800;
     /*  obtención de la fecha y hora corrientes  */
     gettime(&tim);
     getdate(&dat);
     /*  restablece el vector de int. de división entre "0"  */
     setvect(0, ZeroDivVector);
     /* conversión en TSR */
     rg.x.ax = 0x3100;
     rg.x.dx = sizeprogram;
     intdos(&rg, &rg);
}
```

BLOQUE (3)

```
/*  ISR de reloj  */
static void interrupt newtimer()
{
     (*oldtimer)();
     if (running == 0)   {
          running = 1;
          disable();
          intsp = _SP;
          intss = _SS;
          _SP = stack;
          _SS = myss;
          enable();
          if (ticker == 0)    {
               ticker = (((tim.ti_sec % 5) == 0) ? 19 : 18);
               tim.ti_sec++;
               if (tim.ti_sec == 60)    {
                    tim.ti_sec = 0;
                    tim.ti_min++;
                    if (tim.ti_min == 60)    {
                         tim.ti_min = 0;
                         tim.ti_hour++;
                         if (tim.ti_hour == 24)
                              tim.ti_hour = 0;
                    }
               }

               /*  formato para el despliegue de fecha/hora  */
               sprintf(bf,tmsk,dat.da_day,dat.da_mon,dat.da_year
                    % 100,tim.ti_hour, tim.ti_min, tim.ti_sec);
          }
          --ticker;
          /*  despliegue de fecha y hora  */
```

```
        for (v = 0; v < 19; v++)
            vpoke(vseg, (60 + v) * 2, 0x7000 + bf[v]);
        disable();
        _SP = intsp;
        _SS = intss;
        enable();
        running = 0;
    }
}
```

LISTADO 16.2

Programa que construye el archivo ejecutable

```
/* ------ reloj.prj ------ */
reloj
ibmpc.obj
```

■ **DOCUMENTACION LISTADO #16**

BLOQUE 1

El prototipo de la función:
void interrupt(*getvect (int intrr)) (void)
Se encuentra en la biblioteca **"dos.h"**. La función **getvect()** regresa la dirección de la ISR asociada con la interrupción especificada en intr. Este valor es regresado como un apuntador far.
La variable **newtimer** es declarada como una función de interrupción de Turbo C; esto significa que cuando la función sea llamada, los registros serán salvados en la pila y el segmento de datos contendrá la dirección del segmento de datos del programa al cual la ISR está ligada. La declaración de interrupción tambien garantiza que cuando la función regrese, los registros serán sacados de la pila. Al ejecutarse la instrucción de máquina de regreso de interrupción IRET, los registros: (PC) contador del programa, (CS) segmento de codigo y (FR) registro de banderas, serán sacados de acuerdo al numero de instrucciones "*push*" ocurridas al inicio de la interrupcion. La declaración: **#define sizeprogram 375** indica el tamaño del programa (375 párrafos de 16 *bytes* cada uno, equivalentes a 6 Kb).
Los miembros de las estructuras **dat** y **tim** son:

```
struct date {
     int  da_year;
     char da_day;
     char da_month;
}
struct time {
     unsigned char ti_min;     (minutos)
     unsigned char ti_hour;    (horas)
     unsigned char ti_hund;    (centésima de seg.)
     unsigned char ti_sec;     (segundos)
}
```

BLOQUE 2

En este bloque se llevan a cabo los pasos preliminares al establecimiento de la residencia en memoria.

a) Funcion: **segread(&seg)**
(**seg** declarada como estructura de tipo **SREGS**).
Esta funcion copia los valores corrientes de los registros de segmento dentro de la estructura **SREGS** encontrada en la biblioteca **dos.h**.

b) Se salva en la variable **myss** el segmento de pila del programa residente. Esto le permitira al programa restablecer su propia pila cuando este corriendo como TSR.

c) Funciones:
oldtimer =**getvect(0x1c)**;
setvect(0x1c, newtimer);
La función **getvect** regresa la dirección corriente de la ISR de reloj (1CH), misma que es almacenada en la variable **oldtimer**.
La función **setvect** coloca la nueva dirección de la ISR de reloj (**newtimer**) dentro de la tabla de vectores de interrupción en la localidad especificada por el número de interrupcion (1CH). (Ligado de interrupción).

d) Se hace el calculo del area de pila, con lo que el apuntador de pila del TSR es establecido de la siguiente manera.
stack = (375-(data segment - code segment)) * 16 - 300.
Y en la variable **vseg** se almacena el modo de video a ser empleado.

e) Se procede a obtener la fecha y hora corrientes para ser almacenadas en las estructuras **date** y **time** respectivamente.

f) Se restablece el Vector de Interrupción de División entre Cero a su dirección original (mediante la función **setvect**) antes de hacer el programa residente. (Ver capitulo IV cuadro G.5).

g) Finalmente se procede a establecer la residencia del programa en memoria.

```
-    rq.x.ax = 0x3100;          ( ah = 31h, al = 0 )
     rg.x.dx = sizeprogram;     ( dx = # párrafos a
                                     reservar)
     intdos(&rq, &rg);
```

forma general:
intdos(union REGS *in regs, union REGS *out regs)
Esta función es usada para accesar la llamada al sistema operativo especificada por el contenido de la unión apuntada por **in regs**. Esta función ejecuta una instrucción **INT 21H** y coloca el resultado de la operacion en la union apuntada por **out regs**. La unión **REGS** (encontrada en la biblioteca **dos.h**) define los registros de la familia de procesadores 8086/88.

BLOQUE 3

Este bloque incluye el codigo de la nueva ISR de reloj (**newtimer**) que sera ejecutado en cada *tick* del reloj, es decir 18.2 veces por cada segundo. Los pasos que esta involucra son mostrados a continuación:

a) Se ejecuta la ISR de reloj apuntada por **oldtimer**, con el objeto de seguir la cadena hasta llegar a la ISR original de reloj, y asi poder satisfacer todas las demandas de reloj del sistema. (Encadenamiento).

b) Se coloca una bandera (**running**) para evitar que en un mismo *tick* ocurra una nueva ejecucion de la ISR.

c) Se realiza la commutación de contexto propia de TSRs pasivos:
   - La macro **disable()** inhabilita todas las interrupciones excepto las no mascarables.
   - Se salva en la variable **intsp** el apuntador de pila del programa interrumpido.
   - Se salva en la variable **intss** el segmento de pila del programa interrumpido.
   - Se inicializa la variable **_SP** con la dirección del apuntador de pila del TSR (**stack**).
   - Se inicializa la variable **_SS** con la dirección del segmento de pila del TSR (**myss**).
   - La macro **enable()** habilita todas las interrupciones.
     Esto se realiza con el fin de que el TSR pueda utilizar su propia pila al ser ejecutado.

d) Esta parte se ejecutara cada segundo (18.2 "*ticks*" de reloj) para actualizar la hora y desplegarla en pantalla.
   En esta rutina (para evitar trabajar con cantidades fraccionarias) por cada segundo se asumirán 18 *ticks*, a excepción de los segundos multiplos de 5 que serán considerados de 19 *ticks*, con el objeto de corregir el error acarreado.
   Posteriormente se incrementa el contador de segundos cada 18 o 19 *ticks*, de manera que si se han llegado a acumular 60 segundos, entonces el contador de segundos es inicializado a cero y el contador de minutos es incrementado. Despues de haber alcanzado 60 minutos se inicializa a cero el contador de minutos y se incrementa el de horas. Después de alcanzar 24 horas, el contador de horas sera inicializado a cero, y el proceso continuará.
   Finalmente, se construye el despliegue de la fecha y hora, mediante la funcion **sprintf**.

e) Se lleva el control de "*ticks*" de manera que cuando el contador de ticks (**ticker**) llegue a cero se considere un nuevo segundo; momento en el que se realiza todo el proceso de actualizacion de la hora (indicado en el inciso anterior).

f) En cada tick de reloj se desplegará la fecha y hora corrientes, al tiempo que se restablece el segmento y el apuntador de pila del programa interrumpido de la manera siguiente:

- Se inhabilitan todas las interrupciones excepto las no mascarables.
- Se almacena en la variable SP la dirección del apuntador de pila del programa interrumpido (intsp).
- Se almacena en la variable SS la dirección del segmento de pila del programa interrumpido (intss).
- Se habilitan todas las interrupciones.

Esto se hace para que el sistema operativo utilice ahora la pila del programa interrumpido y no la del TSR, ya que el control será devuelto a dicho programa. La bandera running es apagada, indicando que la ISR de reloj esta lista para ser ejecutada nuevamente.

PROGRAMA TSR ACTIVO. MANEJADOR DE TSRs

En esta sección se incluyen dos programas (**popup.c** y **resident.c**) que le permitirán al TSR hacer uso de funciones del DOS cuando sea activado. Estos dos programas deberán ser ligados a la aplicación que se desea hacer residente y deberán ser modificados levemente para que se adapten a las necesidades de la misma. El primer programa incluye una función llamada **main** dentro de la cual se deberán colocar los parametros propios de la aplicación (así como su código). El segundo programa es propiamente el manejador de TSRs de proposito general, el cual controla: las interrupciones, el ligado a los vectores de interrupción, las posibles "disputas" entre el DOS y el BIOS, las llamadas al TSR, la validación de "programa ya residente", la suspensión y reanudación del TSR y finalmente la remoción del TSR de memoria.

Existe un tercer programa (popup.prj) que es propiamente la aplicación. Éste deberá cumplir con las siguientes normas para poder operar correctamente en el ambiente de TSRs:

- Debera ser construido con el modelo de memoria *tiny* o *small*.
- No deberá hacer uso de las funciones 0 a la 12 del DOS.
- Si el programa de aplicación cambia el directorio de disco que esta trabajando corrientemente, debera restablecer el directorio original antes de regresar el control al programa interrumpido.
- No debera utilizar operaciones en punto flotante.
- Al finalizar su ejecución no deberá terminar por medio del sistema operativo, es decir no deberá salir al DOS.

A continuación daremos explicación al porque deben evitarse operaciones en punto flotante: el paquete de Turbo C que maneja dichas operaciones utiliza varios vectores de interrupción, los cuales son ligados durante la ejecución de dicho paquete desde el código de arranque de Turbo C. Estos vectores no son devueltos hasta después de que el programa en C termine y regrese al código de arranque. El área donde se encuentran salvados dichos vectores, no se encuentra dentro del código de arranque; por lo que no podrán ser accesibles al programa TSR. Se concluye que si un programa TSR termina y se remueve por si mismo, dichos vectores no serán restablecidos nunca.

Finalmente, el programa de ejemplo del capitulo 1X, exec.c junto con todos los programas de ejemplo de ventanas, serán integrados para formar la utileria residente "manejadora de TSRs". Con ello damos por terminada la parte de "utilerías residentes de escritorio" de esta tesis.

LISTADO #17

LISTADO 17.1

```
/*   popup.c
/* --------- */

#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <dir.h>
#include "twindow.h"

static union REGS rg;

unsigned sizeprogram = (unsigned) (4e000L/16);

/*   definición de hotkey (Alt .)   */
unsigned scancode = 52;          /* código del punto */
unsigned keymask = 8;            /* se define Alt */

char signature [] = "POPUP";

char notefile[64];  /* Para almacenar la trayectoria y archivo a
                       ser usados con el editor */
```

```
/* --- Funciones locales --- */
int resident(char *, void interrupt (*)());
void resinit(void);
void terminate(void);
void restart(void);
void wait(void);
void resident_psp(void);
void interrupted_psp(void);
void exec(void);
void closefiles(void);
void popup(void);

void main(argc, argv)

/*    argc y argv: posibles parámetros que acompañan al programa
      popup.c al ser ejecutado.

      char *argv[]; /* argv[0] = dirección del primer parámetro */
(
      void interrupt ifunc();
      int ivec;

/*    Se verifica si el programa va a ser cargado en memoria por
      primera vez, caso en el que la función resident() devuelve
      un valor de cero. Si el programa ya estaba residente, la
      función devolverá el # del vector de int. de comunicaciones
      */
      if ((ivec = resident(signature, ifunc)) != 0)      {
            /*    Por lo tanto TSR ya está residente    */
            /*    Ahora se verifica si existe algún parámetro para
                  ejecutar la accion deseada    */
            if (argc > 1)  {
                  /*    Existencia de un parametro en la Línea de
                        Comandos:    */
                  rg.x.ax = 0;
                  /*    Parametro fue "quit"    */
                  if (strcmp(argv[1], "quit") == 0)
                        rg.x.ax = 1;
                  /*    Parámetro fue "restart"    */
                  else if (strcmp(argv[1], "restart") == 0)
                        rg.x.ax = 2;
                  /*    Parametro fue "wait"    */
                  else if (strcmp(argv[1], "wait") == 0)
                        rg.x.ax = 3;
                  if (rg.x.ax)   {     /*    Si AX > 0    */
                  /*    Llamada al vec. de int. de comunicaciones y
                        éste a su vez invoca las funciones de
                        resident.c para procesar los parámetros    */
                        int86(ivec, &rg, &rg);
                        return;
                  }
            }
            printf("\n\t Programa popup esta ya residente");
      }
```

```
    else {      /*    Esta sección solo se procesará cuando el
                      programa sea ejecutado por primera vez. Aquí
                      el programa es establecido residente en
                      memoria    */

        load_help("tcprogs.hlp");  /* carga archivo de ayuda */

        /*    Las siguientes instrucciones son necesarias para
              ser empleadas con el editor, que constituye uno de
              los programas de la utilería residente    */

    /*      Haciendo uso de la función getcwd() se copia la
            trayectoria del directorio corriente de hasta 64
            caracteres en el string notefile y regresa un apuntador
            a éste    */
            getcwd(notefile, 64);
            if (*(notefile+strlen(notefile)-1) != '\\')
                strcat(notefile, "\\");
            strcat(notefile, "note.pad");
    /*      Se imprime un mensaje en pantalla señalando que el
            programa de utilería ha sido cargado en memoria por vez
            primera    */
            printf("\nEl programa Popup ha sido cargado.");
            printf("\nPara activa al presionar Alt-punto. ");
            /*    T&SR    */
            resinit();      /*    Función que hace lo necesario para
                                  dejar el programa residente    */
    }
}

/* --- ISR de comunicaciones del TSR --- */
void interrupt ifunc(bp,di,si,ds,es,dx,cx,bx,ax)
{
    if (ax == 1)                /* Ejecuta parametro "salir" */
        terminate();
    else if (ax == 2)           /* Ejecuta parametro "reiniciar" */
        restart();
    else if (ax == 3)           /* Ejecuta parametro "esperar" */
        wait();
}
/* ------ Con la siguiente función se cierran todos los archivos
          utilizados por el TSR, cuando se desea finalizar su
          ejecución y regresar al programa interrumpido ------ */
void closefiles()
{
    /*    helpfp va a ser utilizado como apuntador al archivo de
          ayuda tcprogs.hlp    */
    extern FILE *helpfp;
    /*    Se conmuta el PID del programa interrumpido por el PID
          del TSR    */
    resident psp();
    /*    Si el archivo de ayuda fue abierto ...    */
    if (helpfp)
        fclose(helpfp); /* cierra el archivo de ayuda */
```

```
     /*     Se conmuta el PID del TSR por el PID del programa
           interrumpido */
     interrupted_pid();
}
     /* --- Funcion popup que contiene la utileria TSR --- */
void popup()
{
     int x, y;
     /*   salva posicion actual del cursor   */
     curr_cursor(&x, &y);
     exec();         /* se invoca la utileria TSR en C */
     /*   coloca cursor en su posicion original   */
     cursor(x, y);
}
```

LISTADO 17.2

```
/*   resident.c
/* ------------ */
#include <dos.h>
#include <stdio.h>
static union REGS rg;
static struct SREGS seg;
static unsigned mcbseg;
static unsigned dosseg;
static unsigned dosbusy;
static unsigned enddos;
char far *intdta;
static unsigned intsp;
static unsigned intss;
static char far *mydta;
static unsigned myss;
static unsigned stack;
static unsigned ctrl_break;
static unsigned mypsp;
static unsigned intpsp;
static unsigned pids[2];
static int pidctr = 0;
static int pp;
static void interrupt (*oldtimer)(void);
static void interrupt (*old28)(void);
static void interrupt (*oldkb)(void);
static void interrupt (*olddisk)(void);
static void interrupt (*oldcrit)(void);
extern void interrupt (*ZeroDivVector)(void);
static void interrupt newtimer(void);
static void interrupt new28(void);
static void interrupt newkb(void);
static void interrupt
newdisk(int,int,int,int,int,int,int,int,int,int,int,int);
static void interrupt
newcrit(int,int,int,int,int,int,int,int,int,int,int,int);
extern unsigned sizeprogram;
extern unsigned scancode;
```

```
extern unsigned keymask;
static int resoft = 0;
static int running = 0;
static int popflg = 0;
static int diskflag = 0;
static int kbval;
static int cflag;
static void dores(void);
static void pidaddr(void);
static void resterm(void);
void resident psp(void);
void interrupted psp(void);
void popup(void);

/* --- Establece y declara residencia --- */
void resinit()
{
/*    Haciendo uso de segread() se leen los registros de segmento
      que son copiados en los miembros de la estructura SREGS
      apuntada por seg.    */
      segread(&seg);
/*    Salvar segmento de pila del TSR (al momento de cargarlo)  */
      myss = seg.ss;
/*     Haciendo uso de la INT 21H función 34H, se obtiene la
      dirección de la bandera BUSY del DOS (esta dirección es
      salvada)    */
      rg.h.ah = 0x34;
      intdos(&rg, &rg);    /*     Regresa en es:bx, el segmento y
                                 offset donde el DOS almacena la
                                 bandera BUSY    */
      dosseg = _ES;
      dosbusy    rg.x.bx;
/*    Haciendo uso de getdta() se obtiene la dirección del dta del
      TSR, para luego salvar dicha direccion    */
      mydta = getdta();
/*    Haciendo uso de pidaddr() se obtiene(n) la(s) direccion(es)
      de PID en el DOS    */
      pidaddr();
/*    Haciendo uso de la funcion getvect() se obtienen (para luego
      salvarlas) las direcciones originales de los vectores de
      interrupcion especificados    */
      oldtimer = getvect(0x1c);
      old28 = getvect(0x28);
      oldkb = getvect(9);
      olddisk = getvect(0x13);
/*    Haciendo uso de la función setvect(), se coloca la dirección
      de la nueva ISR (especificada en su segundo argumento)
      dentro de la tabla de vectores de interrupción en la
      localidad especificada por el primer argumento de la
      función. Así los vectores de interrupción especificados son
      ligados al programa residente    */
      setvect(0x1c, newtimer);
      setvect(9, newkb);
      setvect(0x28, new28);
```

```
    setvect(0x13, newdisk);
/*  Se calcula la dirección del apuntador de pila del TSR   */
    stack = (sizeprogram - (seg.ds - seg.cs)) * 16 - 300;
/*  Se restablece el vector de interrup. de div. entre cero   */
    setvect(0, ZeroDivVector);
/*  Haciendo uso de la INT 21H  función 31H se convierte a la
    utilería en programa residente   */
    rq.x.ax = 0x3100;
    rq.x.dx = sizeprogram;
    intdos(&rq, &rq);
}


/* --- ISR de disco del BIOS --- */
static void interrupt
newdisk(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flgs)
/*  Todos los registros especificados en newdisk son salvados en
    pila al momento de ser ejecutada la ISR   */
{
    /*  se enciende la bandera para indicar al TSR que no puede
        interrumpir una operación de disco   */
    diskflag++;
    /*  se encadena newdisk con la ISR de disco anterior   */
    (*olddisk)();
    /*  los registros de maquina deben ser pasados al programa
        que invocó la ISR de disco tal y como regresaron de la
        llamada a olddisk   */
    ax = _AX;
    cx = _CX;
    dx = _DX;
    /*  se hace una llamada a newcrit() para poder regresar el
        registro de banderas (tal como fue regresado por
        olddisk) al programa que invocó a la ISR   */
    newcrit(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flgs);
    flgs = cflag;
    /*  se apaga bandera indicando que la ISR ha finalizado su
        ejecución   */
    --diskflag;
}


/* --- ISR de error crítico --- */
static void interrupt
newcrit(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flgs)
{
    /*  se le indica al DOS que ignore el error crítico   */
    ax = 0;
    /*  para regreso del reg. de banderas en newdisk   */
    cflag = flgs;
}
```

```
/* --- ISR de teclado --- */
static void interrupt newkb()
{

    /*  Mediante inportb(60H), se lee un byte desde el puerto
        del teclado; en dicho puerto se encuentra el código de
        exploración de la tecla pulsada. A continuación se
        procede a verificar si el byte leído corresponde al
        código de exploración de la hotkey  */
    if (inportb(0x60) == scancode) {
    /*  Si ya se ha tecleado el código de exploración de la
        hotkey, ahora se procede a ver si también se ha
        tecleado la parte complementaria del hotkey; para hacer
        esto, se lee el contenido de memoria en la dirección
        dada por el segmento 0, offset 0417 que corresponde a
        la máscara de teclas de estado: Alt, Ctrl, etc.  */
        kbval = peekb(0, 0x417);
    /*  Ahora se verifica que el TSR no se encuentre en un
        estado de suspensión (resoff=0) y que la parte de la
        hotkey que corresponde a la máscara de estado
        corresponda con la leída en el paso anterior, para
        poder proceder a la activación del TSR  */
        if (!resoff && ((kbval & keymask) == keymask) == 0) {
    /*  Se manda un "reset" por hardware al teclado. Primero se
        lee el Puerto B (61H) del 8255, luego se coloca y se
        envía la señal de acknowledge (ack) del teclado, se
        vuelve a enviar dicha señal que fue previamente
        "reseteada" y al hacerlo se restablece el contenido
        original del puerto B del 8255; finalmente se envía un
        comando de Fin de Interrupción (EOI) al registro de
        comandos del controlador de interrupciones 8259  */
            kbval = inportb(0x61);
            outportb(0x61, kbval | 0x80);
            outportb(0x61, kbval);
            outportb(0x20, 0x20);    /* EOI al 8259 */
    /*  Coloca bandera de hotkey (siempre que el TSR no esté
        desplegado, running 0) indicando que en cuanto sea
        posible el TSR deberá ser ejecutado  */
            if (!running)
                gspi[5] = 1;
            return;
        }
    }
    /*  En caso de que no se hubiera dado la hotkey, se
        encadena con la ISR de teclado para que se encargue de
        los pulsos de tecla sensados  */
    (*oldkb)();
}
```

```
/* --- ISR de reloj ---- */
static void interrupt newtimer()
{
     /*    Encadenamiento con la ISR de reloj previa, para poder
           atender primero todas las demandas de reloj del sistema
           */
     (*oldtimer)();

     /*    Si la bandera de hotkey esta prendida (popflq=1) y el
           DOS no está ocupado (bandera BUSY=0), y además no se
           está llevando acabo una operacion en disco, se puede
           proceder al despliegue del TSR; en caso de que alguna
           de estas condiciones no se cumpla el TSR no podrá ser
           activado   */
     if (popflq && peek(dosseg, dosbusy) == 0)
          if (diskflag == 0)  {
               outportb(0x20, 0x20);     /* EOI al 8259 */
     /*    La bandera de hotkey del TSR es apagada para que este
           pueda ser nuevamente activado con una nueva petición */
               popflq = 0;
               dores();  /* Ejecuta la utileria TSR */
          }
}

/* ------ ISR DOSOK que se activara cuando el TSR esté esperando
           que se le proporcione informacion de teclado o cuando
           el DOS este en espera de comandos ------ */
static void interrupt new28()
{
     /*    Encadenamiento con la vieja ISR DOSOK    */
     (*old28)();
     /*    Si la bandera hotkey se encuentra prendida (popflq=1) y
           el DOS ocupado (bandera BUSY=1), entonces se puede
           proceder al despliegue del TSR; en caso de que alguna
           de estas condiciones no se cumpla el TSR no podrá ser
           activado */
     if (popflq && peek(dosseg, dosbusy) != 0)  {

     /*    La bandera hotkey del TSR es apagada para que éste
           pueda ser nuevamente activado con una nueva petición */
           popflq = 0;
     /*    La utileria TSR es ejecutada, en caso de que la ISR de
           reloj no haya podido hacerlo    */
           dores();
     }
}

/* ------ Conmuta contexto de PSP del programa interrumpido al
           del TSR ------ */
void resident psp()
{
     /*    Se salva el PSP del programa interrumpido    */
     intpsp = peek(dosseg, *pids);
```

```
    /*   Se almacena en la dirección de memoria apuntada por el
         segmento dosseg y el offset pids[], el PSP del TSR
         (mypsp)   */
    for (pp = 0; pp < ptdctr; pp++)
         poke(dosseg, pids [pp], mypsp);
}

/* ------ Conmuta contexto del PSP del TSR al del programa
         interrumpido ------ */
void interrupted_psp()
{
    /*   Restablece el (los) PSP(s) del programa interrumpido en
         la(s) localidad(es) del DOS   */
    for (pp = 0; pp < ptdctr; pp++)
         poke(dosseg, pids [pp], intpsp);
}

/* ------ Ejecución del programa (utileria) residente. Aquí se
         lleva a cabo para tal fin la conmutación de contexto
         (pila, psp y dta) ------ */
static void dormir()
{
    /*   Se prende una bandera para evitar una segunda
         activacion del TSR en esté momento (evitar
         reentrancia)   */
    running = 1;

    /*    Lo primero que sucederá es que el TSR colocará su
         propio ambiente para poder trabajar correctamente. Esto
         involucra entre otras cosas, el indicarle al sistema
         operativo que el TSR utilizará su propia pila, su DTA,
         su PID o PSP. Para esto tendrá que salvar todas las
         condiciones actuales del ambiente (que involucra entre
         otras cosas información referente al ambiente del
         programa interrumpido) para poder restablecer dicho
         ambiente al final de la ejecución del TSR cuando éste
         tenga que regresar el control al programa interrumpido.
         Tambien se involucra el manejo del vector de int. de
         error critico y del Ctrl-Break, que como el TSR los
         modifica, deberá salvar también su estado previo   */

    /*   Se establece como corriente la pila del TSR, esto es
         conocido como conmutación de pilas del programa
         interrumpido a la del TSR   */
    disable();      /* se deshabilitan interrupciones */
    intsp = _SP;    /* salvar apuntador de pila de proq. int. */
    intss = _SS;    /* salvar seqmento de pila de proq. int. */
    _SP = stack;    /* se establece como corriente el apuntador
    _SS = myss;        de pila y seqmento de pila del TSR */
    enable();       /* se habilitan interrupciones */
    /*     Se obtiene y se salva la dirección del vector de
         interrupción 24H (Error critico)   */
    oldcrit = getvect(0x24);
```

```
/*    Colocar nuestra propia rutina de error crítico. Esto se
      debe hacer al momento del despliegue del TSR    */
setvect(0x24, new24);
/*      Mediante la INT 21H función 33H subfunción 00, se
      obtiene el estado del Ctrl-Break e inmediatamente se
      procede a salvarlo    */
rg.x.ax = 0x3300;
intdos(&rg, &rg);
ctrl_break = rg.h.dl;
/*      Mediante la INT 21H función 33H subfunción 01, se
      procede a deshabilitar el Ctrl-Break    */
rg.x.ax = 0x3301;
rg.h.dl = 0;
intdos(&rg, &rg);
/*      Se obtiene para luego salvarlo, el dta del programa
      interrumpido    */
intdta = getdta();
/*    Se coloca como corriente el DTA del TSR    */
setdta(mydta);
/*    Se coloca como corriente el PSP del TSR (conmutación o
      intercambio del PSP del programa interrumpido por el
      del TSR)    */
resident psp();

/*    Se ejecuta la utilería residente    */
popup();

/*      Cuando popup() termina su ejecución y regresa, son
      restablecidos a sus valores anteriores a la ejecución
      del TSR: el PID, el DTA, el vector de int. de error
      crítico, el Ctrl-break y el stack o pila del programa
      interrumpido. Y finalmente el TSR regresará el control
      al programa interrumpido. Todo esto viene mostrado paso
      a paso a continuación    */

/*    Se restablece como corriente el PSP del programa
      interrumpido.    */
interrupted_psp();
/*      Se restablece como corriente el DTA del programa
      interrumpido    */
setdta(intdta);
/*      Se restablece a su contenido previo el vector de
      interrupción de error crítico    */
setvect(0x24, oldcrit);
/*    Haciendo uso de la INT 21H  función 33H subfunción 01,
      se procede al restablecimiento del estado original del
      Ctrl-Break    */
rg.x.ax = 0x3301;
rg.h.dl = ctrl_break;
intdos(&rg, &rg);
/*      Se restablece como corriente la pila del programa
      interrumpido (conmutación de pilas)    */
disable();      /* se deshabilitan interrupciones */
```

```
    /*    Se restablece como corriente el apuntador de pila y el
          segmento de pila del programa interrumpido    */
    _SP = intsp;
    _SS = intss;
    enable();       /* se habilitan interrupciones */
    /*    Se apaga la bandera para indicar que se puede volver a
          activar el TSR    */
    running = 0;
}

static int avec = 0;

/* ------ La siguiente funcion verifica si el programa está ya
          residente; en caso de que no lo esté, liga al TSR una
          interrupcion disponible ------ */
unsigned resident(signature, ifunc)
char *signature;
void interrupt (*ifunc)();
{
    char *sq;
    unsigned dif;
    int vec;
    /*    Haciendo uso de segread() se leen los registros de
          segmento de maquina y se copian sus valores en los
          miembros de la estructura SREGS apuntada por seq.
          Posteriormente se procede a calcular y salvar el tamaño
          del área de código del TSR, que será utilizado más
          adelante para el calculo del segmento de datos del
          mismo.    */
    segread(&seq);
    dif = seq.ds-seq.cs;
    /*    Se procede a explorar los vectores de interrupción de
          usuario (60H a 67H) hasta encontrar un vector
          disponible (cuyo contenido sea NULL); una vez
          encontrado, se salva ese num. de vector, para que
          posteriormente sea ligado al TSR. Cabe aclarar que esto
          solo ocurrira la primera vez que el TSR es cargado en
          memoria    */
    for (vec = 0x60; vec < 0x68; vec++)     {
        if (getvect(vec)     NULL)      {
            if (!avec)
                avec = vec;
            continue;
        }
    /*    Aqui se procede a obtener el contenido del byte de
          memoria apuntado por la dirección regresada por la
          función peekb(peek(seq. de código de vect. de int.) +
          área de codigo), offset: direccion de signatura)), con
          lo que se obtiene la direccion correspondiente al
          segmento de datos del TSR y offset direcc. de la
          signatura (dicha dirección en nuestra utileria
          corresponderá con la localidad en la que está
          almacenada la signatura). Una vez leido el contenido de
          dicho byte, este es comparado con la signatura de la
```

```
        utileria TSR con el fin de comprobar si ésta ya habia
        sido cargada en memoria y de este modo evitar el tener
        cargadas varias copias de la misma utileria en memoria
        y el volver a ligar el vector de interrupción de
        comunicacion   */
        for (sq = signature; *sq; sq++)
              if (*sq!=peekb(peek(0,2+vec*4)+df,(unsigned)sq))
                    break;     /* si la signatura de la utileria
                                  no fue encontrada en esa dir.
                                  salir del for */
    /*   Si la signatura de nuestra utileria fue encontrada en
        esa direccion, nuestra utileria ya se encontraba
        residente y se regresa el num. del vector de int. de
        comunicaciones al programa que invoco a resident()   */
        if (!*sq)
              return vec;
    }
    /*   Si se encontró vector de usuario disponible, este es
        ligado al TSR y sera su vector de int. de
        comunicaciones   */
    if (avec)
        setvect(vec , [illegible]);
    return 0;        /*   resident() devuelve un valor de 0 para
                          indicar que la utilería puede ser
                          cargada a memoria (residente)   */
}
/* --- Localiza las direcciones de PID --- */
static void pidaddr()
{
    unsigned adr = 0;

    /*   Obtencion del PID corriente   */
    rq.h.ah = 0x51;
    intdos(&rq, &rq);    /* obtiene PSP del TSR */
    mypsp = rq.x.bx;
    /*   Se encuentra el final del segmento del DOS   */
    rq.h.ah = 0x52;
    intdos(&rq, &rq);
    enddos = _DS;
    enddos = peek(enddos, rq.x.bx-2);
    /*   Búsqueda de direcciones validas para almacenar PID(s)
        en el DOS   */
    while (pidctr < 2 &&
            ((dosseg<<4) + adr) < (enddos<<4)) {
    /*   Se exploran cada una de las direcciones de palabra en
        el DOS (comparando sus contenidos con el PSP del TSR)
        hasta encontrar las direcciones en donde se igualen
        contenidos o se llegue al final del DOS   */
        if (peek(dosseg, adr) == mypsp) {
    /*   Se escribe un PID artificial en la dirección
        encontrada, se comprueba si si pudo ser escrito para
        ver si es una direccion válida de PID   */
            rq.h.ah = 0x50;
            rq.x.bx = mypsp + 1;
```

```
                intdos(&rq, &rq);
                if (peek(dosseg, adr) == mypsp+1)
                    /*   Almacena en pids[], las direcciones
                         válidas para PSPs   */
                    pids[pidctr++] = adr;
                /*   El PID original es restablecido   */
                rq.h.ah = 0x50;
                rq.x.bx = mypsp;
                intdos(&rq, &rq);
            }
            adr++; /* Incrementa para buscar posible dir. de PID */
        }
}
/* --- Función de terminación --- */
static void resterm()
{
    void closefiles(void);

    closefiles();  /* cierra archivos abiertos por el TSR */
    /*   Restablece vectores de interrupción   */
    setvect(0x1c, oldtimer);
    setvect(9, oldkb);
    setvect(0x28, old28);
    setvect(0x13, olddisk);
    setvect(avec, (void interrupt (*)()) 0);
    /*   Se obtiene la dirección de segmento del primer MCB del
         DOS   */
    rq.h.ah = 0x52;
    intdos(&rq, &rq);
    mcbseg = ES;
    mcbseg = peek(mcbseg, rq.x.bx-2);
    /*   Se recorre toda la cadena MCB para la liberación de
         memoria   */
    segread(&seg);
    /*   Se verifica si el contenido del byte apuntado por el
         segmento mcbseg con offset 0 es 4DH, para ver si no se
         ha llegado al último MCB de la cadena   */
    while (peekb(mcbseg, 0) == 0x4d)  {
    /*   Se compara el PSP indicado en el MCB con el del TSR, y
         si son iguales se procede a la liberación de ese bloque
         de memoria   */
        if (peek(mcbseg, 1) == mypsp) {
            rq.h.ah = 0x49;
            seg.es = mcbseg+1;
            intdosx(&rq, &rq, &seg);
        }
    /*   Se calcula la dir. del siguiente MCB en la cadena   */
        mcbseg += peek(mcbseg, 3) + 1;
    }
}
/* --- Terminación del programa residente --- */
void terminate()
{
```

```
/*    Se verifica si el TSR todavia posee todos los vectores de
      interrupcion que tenia ligados, para de este modo proceder
      con la terminacion del mismo   */
      if (getvect(0x13) == (void interrupt (*)()) newdisk)
            if (getvect(9) == newkb)
                  if (getvect(0x28) == new28)
                        if (getvect(0x1c) == newtimer)      {
      /*    Como el TSR si posee todos los vectores que le fueron
            ligados, entonces si puede ser sacado de la memoria
            */
                              restorme();
                              return;
                        }
      /*    Si el TSR no posee alguno de los vectores que le fueron
            ligados, puede ser debido a que otro TSR que fue
            cargado despues del nuestro se ligo a si mismo a dicho
            vector, por lo que nuestra utileria no podra ser sacada
            de memoria y simplemente sera colocada en un estado de
            suspension   */
      resoff = 1;
}


/* --- Reactiva al programa residente --- */
void restart()
{
      resoff   0;     /* Apaga bandera de suspension de TSR */
}

/* --- Coloca al TSR en un estado de suspension ---*/
void wait()
{
      resoff = 1;     /* Enciende bandera de suspensión de TSR */
}
```

LISTADO 1".)

Programa de aplicacion.

```
/*    popup.prj
/* ---------- */

popup (twindow.h)
exec (twindow.h, keys.h)
testmove (twindow.h, keys.h)
promote (twindow.h, keys.h)
ccolor (twindow.h, keys.h)
notepad (twindow.h)
maxims (twindow.h, keys.h)
funtabla (twindow.h, keys.h)
editor (twindow.h, keys.h)
thelp (twindow.h, keys.h)
tmenu (twindow.h, keys.h)
twindow (twindow.h, keys.h)
```

resident
ibmpc.obj

COMENTARIOS AL LISTADO #17

Los tres programas que se utilizaron en la construcción del TSR son: **popup.c**, **resident.c** y el programa de utileria en Turbo C **popup.prj**. **Popup.c** (listado 17.1) y **resident.c** (listado 17.2) constituyen el programa manejador TSR. **Popup.c** es el programa que debe ser modificado conforme a los requerimientos del programa de aplicación, anadiendo el codigo necesario a la utileria; ademas en este se tienen diversas variables (como el *tamano* del programa de aplicacion **sizeprogram**, para la definición de *hotkey* **scancode** y **keymask**, etc) que deberán ser inicializadas con los valores que describen a la utileria o programa de apiicacion. Por otro lado, el programa **resident.c** siempre permanece fijo para cualquier TSR.

Observaciones: En el programa **popup.c** en la variable **sizeprogram** debera ser especificado el tamaño del programa en parrafos (un parrafo es de 16 *bytes*). Recordando que los programas en modelo *tiny* no pueden ser mayores de 64K (4096 parrafos) y los programas en modelo *small* no pueden ser mayores de 128K (8192 parrafos). El valor deseado de *hotkey* para activar la utileria debe ser asignado a la combinación de variables **scancode** y **keymask**. En nuestro caso la *hotkey* que activa el TSR es **Alt-"punto"**, con lo cual tenemos un **scancode** = 52 (codigo de exploracion de la tecla de punto) y **keymask** = 8 (codigo de la tecla Alt). En la variable **signature** debe proporcionarse la palabra clave o signatura de la utileria, que sera de utilidad para verificar si el TSR ya habia sido cargado en memoria y poder evitar asi el tener muchas copias del mismo TSR en memoria.

En nuestro caso, **popup.c** contiene la *funcion popup()* que es invocada cuando el programa manejador residente determina que la *hotkey* ha sido presionada y que se encuentra seguro para utilizar funciones del DOS. La funcion popup() ejecuta la funcion **exec()**, funcion que constituye el programa de utileria TSR (nuestro programa de ejemplo del capitulo IX).

El listado 17.3 incluye el programa que contiene el archivo "proyecto" que Turbo C utiliza para la construcción de **popup.exe**.

Para correr el programa de utileria residente construido por Turbo C (con el archivo de proyecto en **popup.prj**) se debe introducir el siguiente comando:

```
C>popup
```

Con este comando, el TSR quedara cargado en memoria y se desplegara el siguiente mensaje:

"El programa Popup ha sido cargado."
"Se Activa al presionar Alt-punto."

Si nuevamente fuera introducido el comando popup desde el prompt del sistema (C>), se desplegara el siguiente mensaje:

"El programa Popup esta ya residente."

Una vez que el TSR ya fue cargado en memoria, se puede ejecutar con parametros desde la linea de comandos del sistema, que le provocaran al TSR que se coloque en un estado de suspension, reanudacion o terminación.
Estos 3 comandos son:

```
C>popup wait
C>popup restart
C>popup quit
```

Finalmente cabe senalar que si se ha realizado un programa TSR, y se desea probarlo como tal, se tendran algunos inconvenientes. Primero, no se podrá utilizar el ambiente de Turbo C para realizar pruebas interactivas, ya que un TSR debe ser instalado desde la linea de comandos y se activa mediante una *hotkey*. Segundo, se debera apagar o "resetear" la computadora para sacar de memoria el TSR mientras la función de terminacion del mismo no esto trabajando o si otros TSR's fueron instalados despues del nuestro. Tercero, debido a que el TSR tiene ligados vectores de interrupcion, sus ciclos y abortos pueden tirar el sistema. Con todo esto podemos concluir que no es facil realizar pruebas a un TSR teniendolo residente en memoria. La solución consiste en probar el TSR como un programa transitorio. Como se pudo observar en el ejemplo de TSR de capitulo, la parte de utileria del programa fue desarrollada y probada aparte de las operaciones de TSR. Toda la interfaz al DOS, para establecer y operar al TSR en el ambiente de residencia en memoria es manejado por **popup.c** y **resident.c**: entonces lo mejor es probar el programa de utileria aparte y una vez que funcione correctamente ligarlo a los programas encargados de manejar la residencia en memoria, para después probarlo en el ambiente de TSR's.

La prueba final sera observar "como le va" a la utileria TSR en compañia de otros programas TSR. Esta prueba es a veces un juego de azar. Muchos de los programas TSR populares no pueden trabajar juntos, de modo que se deberán seleccionar aquellos con los que se desea ser compatible y efectuar pruebas que involucren el cargarlos todos en varias secuencias hasta obtener la secuencia correcta de carga, en la que todo trabaje bien.

SPOOLER DE IMPRESION

CAPITULO 11

INTRODUCCION

Se incluye este capitulo con el fin de mostrar otro tipo de programas residentes de importancia: los "spoolers de impresion".

En sus inicios las PC's satisfacian por completo las necesidades del usuario, pero conforme el tiempo pasaba, cada vez se hacia más necesario contar con ciertas capacidades "multitasking". Debido a ello, IBM y Microsoft incluyeron en la versión 1.1 del DOS una función de terminación y permanencia en memoria (TSR) y ademas una utileria sencilla para imprimir sin interrumpir el trabajo de la computadora. Ésta, era la primer utileria residente en memoria RAM (denominada MODE) que consistía de un mecanismo simple para redirigir la salida de la impresora. En 1983, con la versión 2.0 del DOS, aparece el programa residente PRINT.COM con la capacidad de realizar la impresion de archivos como tarea secundaria. Los programas MODE y PRINT del DOS sirvieron como prototipos para dar comienzo al desarrollo del software residente en RAM no direccionable por teclado.

En 1982 los discos RAM y los spoolers de impresión fueron las aplicaciones de este género más ampliamente difundidas. Hoy día, entre el 60 y 70% de las aplicaciones residentes en RAM, trabajan como funciones secundarias (de background) no direccionables, activadas con un comando especial. Dentro de esta categoría podemos mencionar: manejadores de impresora, software de disco-cache, software de tipos de letra, etc. Este tipo de programas son cargados en RAM pero no son usados de manera dinámica y, por lo general, no presentan conflictos de convivencia con otros TSR's.

FUNCIONAMIENTO BASICO DE UN SPOOLER DE IMPRESION

Un spooler de impresión intercepta básicamente dos interrupciones: la Int 17h (usada por la mayoría de los programas para imprimir caracteres) y la Int 1Ch (interrupción de reloj "timer-tick"; invocada por la rutina de interrupción 8h del BIOS).

Normalmente un programa de aplicación envia caracteres a la impresora haciendo uso de la Int 17h, está a su vez invoca una rutina del ROM BIOS que accesa el hardware para enviar el caracter hacia la impresora. (Ver fig. 11.1).

Cuando un programa de spooler de impresión se instala por sí mismo, el vector de interrupción 17h quedará apuntando a la rutina de impresión del spooler (la dir. original de la ISR de esta interrupción será salvada). La rutina del spooler está diseñada para que, al recibir una peticion de impresión, los caracteres sean almacenados en un *buffer* o spool en memoria. Paralelamente, por cada tick de reloj el spooler enviará los caracteres, almacenados en el *buffer*, hacia la impresora invocando a la rutina 17h original (Ver fig.11.2)

Cabe aclarar es probable la aparición de problemas al hacer interactuar un spooler de impresión con determinados programas que se apoderan por completo de la Int 1CH. Por ejemplo, programas compilados con alguno de los antiguos compiladores de BASIC. Al correr un programa de este tipo la impresora puede suspender su actividad, pudiendo reanudarla solo hasta que el usuario se salga del programa en cuestión.

FIG 11.1 ESQUEMA SIN SPOOLER

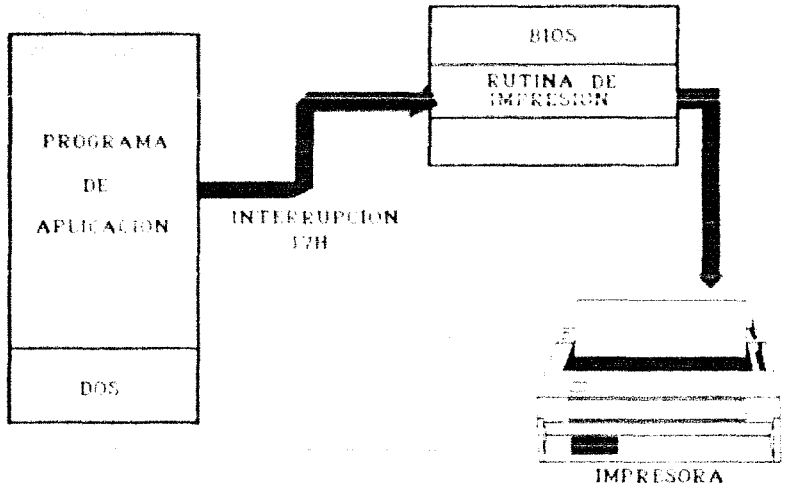
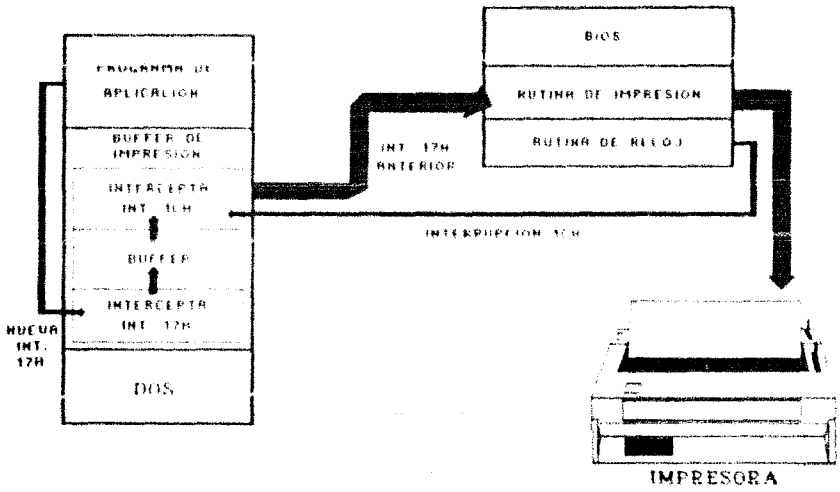


FIG 11.2 ESQUEMA CON SPOOLER



PROGRAMA RESIDENTE
SPOOLER DE IMPRESION

En esta seccion presentamos un spooler de impresión (PCSPOOL). Este programa sigue las pautas establecidas en el capitulo IV para el desarrollo de utilerias residentes. El programa imprime un archivo colocado previamente en una cola, que puede residir en disco o en memoria.

PCSPOOL incorpora muchas de las caracteristicas comunmente encontradas tanto en spoolers sencillos como en aquellos dirigidos a redes, al tener la capacidad de controlar simultáneamente hasta 3 impresoras y permitir realizar las siguientes operaciones sobre los archivos "encolados": salto, cancelación, supresion, separación por *form-feeds* y pausa (con mensajes prestablecidos).

Descripción general del funcionamiento de PCSPOOL:

Es una utileria residente en memoria (TSR). Cuatro de sus funciones básicas son controladas desde la linea de comandos; las otras pueden ser activadas desde una ventana *popup* mediante la hotkey default (**Ctrl-Alt-P**). Después de su instalación, PCSPOOL ocupa aproximadamente 7Kb de RAM.

La sintaxis para instalar PCSPOOL desde el prompt del DOS es:

pcspool /I [/1] [/2] [/3] [/Cnn|/D[d:\path]]

PCSPOOL permite detener momentaneamente la impresora (estado de pausa) y desplegar un mensaje. Sintaxis:

pcspool /P [/1] [/2] [/3] [comentario]

Comando para enviar un *form-feed* a la impresora:
pcspool /F [/1|/2|/3]

Para desinstalar la utileria:
pcspool /U

Nota: Los comandos **/I** (instalar), **/P** (pausa), **/F** (form-feed) y **/U** (desinstalar), deben ejecutarse desde el prompt del DOS por separado (no pueden ser combinados).

El conmutador (*switch*) /I instala pcspool, inicializa la(s) impresora(s) que se desee controlar por el programa, y determina si la cola de impresion sera almacenada en disco o en memoria convencional. Por ejemplo, para indicar que se quieren controlar 2 impresoras (LPT1 y LPT2) se debe introducir el siguiente comando:

pcspool /I /1 /2

Existe la opcion de cargar pcspool sin indicarle que impresora se desea controlar, para luego indicarle estos parámetros a través de la ventana *popup*.

Una limitante de pcspool radica en su incapacidad para controlar impresoras conectadas a puertos serie, aun cuando estas hayan sido inicializadas usando el comando "MODE LPTx := COMx:" del DOS. Esto, debido a que la rutina utilizada por pcspool para enviar datos a la impresora está disenada para controlar el puerto paralelo de impresora.

Usando el conmutador /Cnn durante la instalación, se le indica a *pcspool* que reserve "nnK" de memoria convencional para almacenar la cola (donde "nn" pueden variar de 1 hasta 64). Si no se especifica ningun valor, se reservan 16K de memoria por default. Entonces, para iniciar al programa pcspool controlando LPT1 y con una cola de 24K, se debió introducir:

pcspool /I /1 /C24

El conmutador alternativo /D[d:\path] permite colocar la cola de impresion en disco. Hay que notar que no hay espacios entre el /D, el primer caracter del drive opcional y la trayectoria.

Pcspool cuenta con soporte EMS limitado, por lo que se tiene la opcion de poderle introducir "la letra del drive de disco RAM EMS" para especificar el drive con el comando /D. Si no se especifica ningun drive o trayectoria, el archivo de cola sera colocado en el directorio raiz del drive C:.

Independientemente del numero de impresoras controladas por pcspool, se creara unicamente un archivo de cola denominado PCSPOOL.QUE. Su tamano esta limitado unicamente por la cantidad de espacio libre en ese drive. Asi, por ejemplo para indicar a pcspool que controle 2 impresoras (LPT1 y LPT2) y coloque el archivo de cola en el drive E: en el directorio \TMP\WORK, se deberá introducir:

```
pcspool /1 /1 /2 /DE:\TMP\WORK
```

Si se especifica un drive o trayectoria incorrecta, pcspool no será capaz de inicializar la cola. En ese caso, se desplegará un mensaje de error y el spooler no sera instalado. Cabe aclarar que si ni el conmutador /C, ni el conmutador /D son especificados, el default sera una cola de 16K en memoria convencional.

El conmutador /P envia un comando de pausa a la impresora indicada por /1, /2 o /3. Si no se especifica ningun número, se asume LPT1. Cuando el comando pausa es emitido, se escribe un registro a la cola de impresora. Este registro, es colocado inmediatamente despues del ultimo byte de datos que haya sido enviado a la cola en ese momento. El registro de pausa contiene la petición de pausa y un comentario u observacion si éste fue proporcionado al introducir el comando /P.

Cuando pcspool encuentra un registro de pausa en la cola, la impresora asociada es detenida y el comentario es salvado para ser desplegado cuando la ventana *popup* sea llamada por el usuario. Así, si por ejemplo tuviéramos los siguientes comandos:

```
pcspool /1 /1 /C24
copy ARCHIVO1.TXT LPT1
copy ARCHIVO2.TXT LPT1
pcspool /P /1 ARCHIVO3.TXT a punto de imprimirse.
copy ARCHIVO3.TXT LPT1
```

Al ser ejecutada esta secuencia de comandos se tendrá lo siguiente: la primer línea instala pcspool e indica que se controlará LPT1; la impresora comenzará a imprimir tan pronto como el copiado del ARCHIVO1.TXT inicie, después de imprimir el ARCHIVO3.TXT, pcspool parará a la impresora. En este momento, si la ventana *popup* es activada con Ctrl-Alt-P, el mensaje "Pausa. ARCHIVO3.TXT a punto de imprimirse" será desplegado en la línea correspondiente a LPT1. Y LPT1 no imprimirá los datos del ARCHIVO3.TXT hasta obtener un comando GO desde la ventana. Por esta razón el comando pausa es muy útil para la colocación de mensajes de peticion de acciones entre las salidas de impresión.

El parametro de mando /P de pcspool no debe ser confundido con el comando (P)ausa emitido vía la ventana *popup*. Aunque los dos son comandos de pausa, tienen efectos diferentes.

El conmutador /F provoca el envio de un *form-feed* a la impresora indicada por /1, /2 ó /3. Si no se especificó impresora, la alimentación de forma es enviada a LPT1. Así, por ejemplo si se desea enviar una alimentación de forma a LPT2, se debe introducir:

```
pcspool /F /2
```

El conmutador /U desinstala pcspool (de ser posible). Así, para sacar el programa de memoria se deberá introducir:

```
pcspool /U
```

El comando /U fallara si alguna de las interrupciones ligadas al spooler coincide con las empleadas por otro TSR ejecutado con posterioridad al nuestro. Antes de remover de memoria nuestro spooler, deben ser removidos los programas que puedan ocasionar este problema. El modo usual de llevar a cabo esta acción es desinstalando los TSR's en el orden inverso a su instalación.

Por otro lado, si se intenta desinstalar pespool mientras existan datos en cola, el programa desplegara el siguiente mensaje:

"EN ESPERA DE TERMINAR IMPRESION. ESC CANCELA DESINSTALACION.
PULSE CUALQUIER OTRA TECLA PARA DESINSTALACION INMEDIATA."

En caso de que ninguna tecla sea presionada, el spooler será desinstalado automaticamente una vez que la cola quede vacía.

■ LA VENTANA DE CONTROL *POPUP*.

La ventana *popup* que el programa despliega al sensar la hotkey (Ctrl-Alt-P), es la siguiente:

```
LPT      % Uso                  CPS      CI   CEC   TIEMPO   D: DES....
 1:   68% ▓▓▓▓▓▓▓▓---            311     54K   25K 00:01:22   F: Formf..
 2: Impresora no encontrada                                  J: Salta..
 3: Impresora no encontrada                                  CTRL+key..
```

% Uso : Monitorea el porcentaje de espacio de cola en uso. Aquí será desplegado un porcentaje numerico y una barra que indica el porcentaje de espacio de cola utilizado en ese momento, por la impresora.

CPS : Velocidad de impresión en caracteres por segundo.

CI : Número de caracteres actualmente impresos.

CEC : Número de caracteres en cola esperando ser impresos.

TIEMPO: Tiempo estimado para imprimir los caracteres que se encuentran actualmente en la cola, y se basa en el CPS calculado.

Los valores de CPS, CI y CEC son desplegadas de acuerdo al siguiente formato de salida:

Desplegado:	Indica:
1 hasta 9999	valor actual inferior a 10,000
10K hasta 999K	10,000 hasta 999,999
1M hasta 999M	1 millón hasta 999 millones
> 1G	Más de mil millones

Las líneas que despliegan el estado de hasta 3 posibles impresoras contienen información que nos indica las condiciones en las que se encuentran cada una de ellas.

Si las estadísticas con referencia a la impresora no están presentes en la ventana, aparecerá alguno de los siguientes mensajes:

- Impresora no lista: fuera de línea o apagada.
- Pausa, use GO para continuar o Pausa: *comentario*: impresora en estado de "pausa", introducir G para continuar imprimiendo.
- Flujo de cola: datos sacados de la cola de la impresora solicitada como resultado del comando SaltaJob o Cancela emitido por el usuario en la ventana *popup*.
- No usa spool actualmente: la impresora no es controlada actualmente por pcspool. Razones: impresora no incluida en el comando pcspool /I, o bien impresora deshabilitada con el comando (D)esactiva de la ventana *popup*.
- Impresora no encontrada: el BIOS no encontró puerto para la impresora especificada.

Mientras la ventana *popup* esté desplegada, se tienen disponibles diversas funciones de control: (D)Desactiva, (P)Pausa, (G)Go, (F)Formfeed, (R)Reset, (J)SaltaJob y (C)Cancela. Estas funciones pueden ser ejecutadas sobre LPT1 introduciendo la letra asociada a la función; sobre LPT2, presionando las teclas Ctrl y la letra asociada a la función; y sobre LPT3 con Alt-letra de la función.

El comando (D)Desactiva, provoca que la impresora indicada conmute de un modo spooleado a uno no spooleado. Todos los datos enviados a esta impresora serán impresos inmediatamente, sin ser almacenados en *buffer*. Todos los datos que estén en el spool destinados a esta impresora, serán retenidos y podrán ser impresos cuando ésta sea nuevamente spooleada; lo cual se lleva a cabo ejecutando el comando (G)Go de la ventana *popup*, para esa impresora.

El comando (P)Pausa (equivalente a introducir el comando "pcspool /P" desde el prompt del DOS) suspende la operación de la impresora. Para cualquiera de los dos comandos, la impresión es reanudada por medio del comando (G)Go de la ventana. La diferencia entre los comandos de pausa, es que el comando de ventana no inserta un registro de petición de pausa en la cola, tal como lo hace el comando pcspool /P.

Al momento de emitir la (P)Pausa desde la ventana, ya no serán enviados más datos a la impresora (aunque se puede continuar almacenando datos en la cola) hasta que se ejecute el comando (G)Go.

Cuando se desea estar seguro de que la siguiente sección de una salida a impresion comienza al inicio de una nueva pagina, se puede emitir el comando (F)orm feed para la impresora en actividad. Cabe aclarar que si este comando es ejecutado mientras el programa esta copiando datos a la cola, el *form-feed* resultará en un sitio arbitrario de la salida a impresión. En general, este comando no deberá ser usado mientras el programa este enviando datos a la cola.

Existen ocasiones en las que las impresoras necesitan ser reinicializadas despues de imprimir un conjunto de datos. Para esto se tiene el comando (R)eset, que inserta un *Reset* de impresora del BIOS en la cola de la impresora solicitada. Lo que provocará que la impresora se revierta a su modo de arranque. Cabe aclarar que si la impresora a ser reinicializada, tiene un *buffer* con datos, la inicialización puede hacer que esos datos en el *buffer* se pierdan. No existe garantía de que este comando trabaje con todas las impresoras. Hay que tener cuidado de no emitir este comando cuando se esten enviando datos a la cola de impresion o de la cola de impresión a la impresora, ya que en caso de hacerlo, la reinicialización ocurrira en un punto impredecible de la salida a impresión.

A veces, despues de que un archivo haya sido enviado a la cola, puede suceder que no se desee imprimirlo; en este caso, el comando (C)ancela deja fluir toda la información destinada a la impresora en cuestión. Una vez que la información ha sido sacada, los datos no pueden ser recuperados.

Por último, si el comando pospool /P es usado entre segmentos de salida a impresion, se puede saltar al siguiente segmento de datos en cola, usando el comando (J)SaltaJob. Al ser emitido este último, los datos de la impresora solicitada son sacados hasta encontrar el registro de pausa más proximo en la cola. Si no se encuentran registros de pausa posteriores, este comando tendrá el mismo efecto que el comando (C)ancela.

■ CLASIFICACION Y DEFINICION DEL PROGRAMA DE SPOOL.

Los spoolers simples con buffer en memoria, frecuentemente proporcionan un numero de opciones de control tales como: ir al inicio de la página, inicializar impresora, reimprimir la página anterior, etc. Ademas presentan una pantalla que muestra el estado o condición de las impresoras (bytes en cola, bytes impresos, etc.) y el control de la cola (cancelar cola, desinstalar, etc.).

Los programas de spool en redes tienen dos características principales: permiten a los usuarios compartir impresoras costosas y evitan que los datos enviados desde diferentes nodos individuales lleguen a ser mezclados en la impresora. Casi todas las redes utilizan spoolers con *buffers* en disco. Los datos a ser impresos son interceptados en los nodos de usuario y son enviados al "server", donde son colocados en disco hasta que esté listo para imprimirlos.

Comunmente se utilizan 3 metodos para indicarle a la red cuándo debe imprimir: 1) comando explícito emitido por el usuario, 2) al advertir la red que el usuario ha regresado al prompt del DOS o ha cerrado el archivo de impresión, y 3) al transcurrir una cantidad predeterminada de tiempo antes de que los datos de salida del nodo sean impresos.

Para evitar la mezcla de datos provenientes de diversos nodos de usuario, el sistema de spool de red debe mantener *buffers* separados para cada usuario en la red. Con esto se da la apariencia de que cada nodo de usuario cuenta con su propia impresora.

Cuando un nodo de usuario comienza el spool hacia la red, la mayoría de las redes abren un "print job" (trabajo de impresión). Éste es enviado, por la red, a la impresora apropiada, cuando encontrandose ésta disponible, un conjunto de datos haya completado el spool y el usuario haya informado a la red que la impresión está terminada. La red puede modificar los trabajos de impresión de diversos modos: cambiándoles su prioridad, colocandolos en un estado de retención para su posterior impresión, cancelando tareas individuales por completo o cambiando su destino final.

Cabe mencionar que la capacidad de redirigir un trabajo de impresión a otra impresora, puede algunas veces producir efectos secundarios inesperados. Por ejemplo, si una imagen de gráficos creada para una impresora laser fuera redirigida a una impresora incompatible de matriz de puntos, la salida impresa sería irreconocible y se podría sospechar erróneamente que el fallo fue debido a la impresora o al programa de spool.

Una vez distinguidos los tipos de spooler, se observa que pcspool engloba elementos de ambos. Cuando pcspool está activo, intercepta los datos a imprimir apoderandose de la interrupción 17H (rutina de interrupcion de impresion del BIOS). Si una aplicación envia datos a una impresora por algún otro medio, tal como E/S directa por hardware, los datos no serán interceptados por pcspool. Cuando la Int 17H es llamada por el programa de aplicación, pcspool determina si la impresora solicitada está spooleada, y si lo está redirige los datos a un *buffer* de registro de cola para esta impresora.

Los registros de la cola pueden ser colocados en alguna de las cuatro listas ligadas (LPT1, LPT2, LPT3 y la lista de registros libres). Cuando los datos están siendo escritos a la cola, se saca un registro de la lista de registros libres, los datos son trasladados a éste y es colocado en la lista asociada con la impresora solicitada. La colocación de un registro en una lista implica unicamente el movimiento de apuntadores (los registros de datos no son movidos).

A continuación, la interrupción de reloj invoca una rutina que verifica el estado de cada impresora controlada por pcspool y determina si los datos deberán ser enviados a ésta. Si es así, los registros son leidos de la lista en la cola de dicha impresora. Los datos del registro son pasados a un *buffer* en memoria y el registro es colocado nuevamente en la lista de registros libres.

Existe una condición de excepcion que cambia la secuencia normal del flujo de datos. Si una lista de registros de una impresora está vacia y su *buffer* en memoria tiene suficiente espacio para manejar los datos que están siendo "spooleados", los datos son trasladados directamente al *buffer* de memoria. Con esto se elimina el movimiento innecesario del registro a través de la cola durante las transferencias de información de bajo volumen.

■ COMO SE EFECTUA EL MANEJO DE COLAS.

La E/S asincrona a disco, es una de las funciones TSR más dificiles de codificar debido a que el DOS no es reentrante. Este problema se manifiesta cuando el spooler es configurado para mantener su cola en disco. Cuando una aplicación solicita una impresion via el DOS, este se vale de la rutina de interrupción 17H del BIOS, para tal efecto. El spooler intercepta los datos cuando la int 17h es invocada y los coloca en un *buffer*. Cuando el *buffer* esta lleno, debe ser escrito al disco de modo que pueda ser reutilizado para capturar mas datos que se deseen mandar a impresión. Para ello, el spooler debe hacer uso de funciones del DOS; pero el DOS no puede atenderlo debido a que aún no esta disponible.

Si el spooler pudiera, de alguna manera, almacenar los datos hasta que el DOS se desocupara, tendria que encargarse de 64K de datos (máxima extensión de una escritura del DOS) antes de que el DOS pudiera admitir una petición para escribir en la cola en disco. Ademas, a los usuarios no les interesará "donar" 64K de memoria para ser utilizados como *buffer* temporal.

Este problema es evitado en pespool examinando todas las peticiones de escritura emitidas via la función 40h de la INT 21H del DOS. Al realizarse una peticion de escritura, el spooler primero emite una llamada al IOCTL (función 44h) del DOS para determinar si el dispositivo al que se desea escribir es una impresora. Una de las muchas cosas que la función IOCTL del DOS puede hacer es regresar informacion con referencia al tipo de dispositivo representado por el manejador de archivo (*file handle*).

Si el manejador de archivo es un dispositivo de disco o una consola, la peticion de escritura es pasada a la función normal de la INT 21H del DOS. Si el manejador de archivo no es una consola, ni un dispositivo de disco, es muy probable que sea una impresora. Pespool no puede estar muy seguro todavia, debido a que posiblemente existan otros dispositivos en el sistema, tales

como el COM1. Por consiguiente, pcspool inmediatamente determina el tamaño máximo de los datos a escribir sin desbordamiento del *buffer* que pcspool tiene establecido; entonces hace uso de la función 40h del DOS para escribir el numero de bytes determinado. Si esto provoca que un registro de cola se sature, los datos restantes son escritos con una serie de llamadas al DOS diseñadas para llenar un solo registro de cola a la vez. Así Pcspool se asegura de que todas las operaciones de escritura a la impresora, por parte de del DOS, sean lo suficientemente pequeñas para poder ser almacenadas en su *buffer*. En otras palabras, divide grandes escritos en varios pequeños.

Conforme cada registro es llenado, es escrito en la cola hasta que todos los datos solicitados hayan sido escritos. Puesto que ningún escrito excederá el espacio disponible en un registro de cola en memoria, no se tiene que asignar más espacio en el *buffer* para estar seguro de que el DOS estará disponible. Y como pcspool tiene el control de la INT 21H del DOS cuando una petición de escritura es realizada, será capaz de superar el problema de "reentrancia" y escribir un registro de cola al archivo de cola en disco siempre que el spooler detecte que un registro de cola ha sido llenado.

Pcspool reserva un *buffer* temporal de 1K para cada impresora conectada a la computadora. Dicho *buffer* es utilizado durante el "viaje" de la cola a la impresora. Siempre que la cantidad de datos en dicho *buffer* se mantenga inferior a la cantidad de datos mantenidos en un registro de cola (y el DOS esté disponible), el registro de cola será leído y los datos serán trasladados al *buffer*. Debido a que los *buffers* de memoria son mayores al doble del área de datos de un registro de cola, los datos que van de la cola a la impresora son de hecho "buffereados" doblemente. En otras palabras, la impresora está siendo servida desde un *buffer* que consta efectivamente de dos registros de cola llenos. Esto hace menos probable que se le terminen los datos a la impresora a causa de una breve demora en la disponibilidad del DOS.

Cuando una cola en memoria es usada en lugar de disco, la disponibilidad del DOS no representa un problema. Debido a que pospool no necesita llamar al DOS para escribir en la cola, no necesita llevar a cabo ningun procesamiento especial para asegurar la disponibilidad del DOS durante el procesamiento de la INT 21H.

El tamano de la cola de pospool esta limitado por la memoria asignada a este durante la inicialización o por el espacio disponible en el drive logico indicado cuando el programa comenzo. En ambos casos, cuando no hay registros disponibles en la lista de registros libres, se colocan nuevos registros al final del espacio de cola hasta que todo el espacio disponible esté usado completamente; cuando esto ocurre, el spooler regresa un código de estado indicando que la impresora está ocupada.

■ CUESTIONES TECNICAS

El formato de la cola consta de una serie de registros de 512 bytes, cada uno de los cuales contiene un apuntador (en 2 bytes) al siguiente registro en la lista; un byte de control que indica si el registro es de datos, de pausa, o de reinicializacion; una palabra (2 bytes) con la longitud de los datos; y un área de datos de 507 bytes. El primer registro (registro 0) de la cola contiene un registro de encabezado, el cual a su vez contiene las estructuras para las listas de registros en la cola. La organización de los registros está esquematizada con detalle en la figura 11.3. La figura 11.4 muestra la relación entre registros para una sección de la cola y la organización de ésta. En donde el primer registro corresponde al encabezado que contiene los apuntadores base para las listas de registros presentes en la cola. La lista libre contiene los registros 2, 3, 4 y 7; la lista LPT1 contiene los registros 1 y 10; la lista LPT2 contiene los registros 5, 6, 8 y 9; la lista LPT3 no tiene registros. Se puede notar que el ultimo registro en cada lista contiene un cero para el apuntador al registro siguiente.

FIG. 11.3 FORMATO DEL REGISTRO DEL PCSPOOL

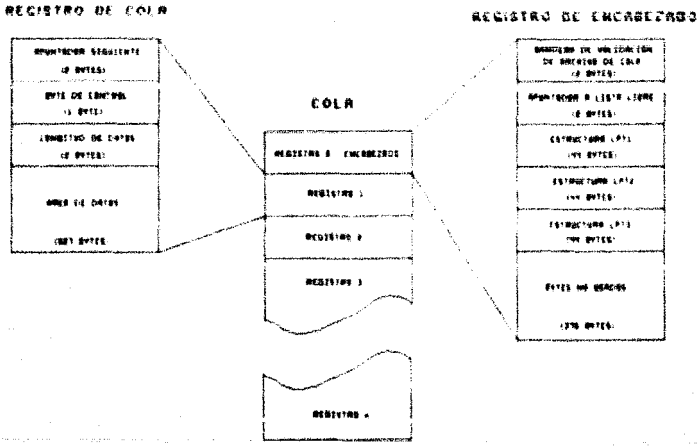
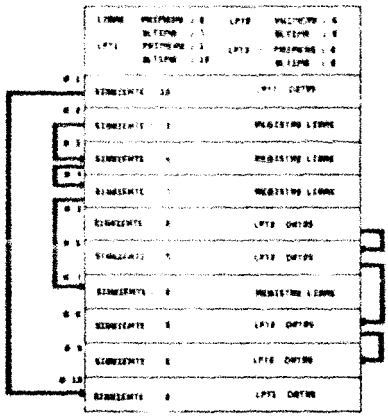


FIG. 11.4 ESQUEMA DE REGISTROS EN COLA



Cuando pcspool es inicializado, el registro de encabezado es leído en memoria. Si el spool fue cerrado apropiadamente antes de interrumpir al sistema y además fue utilizada cola en disco, el spooling puede continuar desde donde se había quedado antes de que el sistema fuera interrumpido.

El programa en si consta de tres secciones principales: Inicialización, rutinas encargadas del manejo de interrupciones y rutinas encargadas del manejo de la cola. La sección de inicialización del programa es muy extensa y lleva a cabo funciones tales como asignación de memoria, procesamiento de los comandos de inicialización (pausa, desinstalacion, etc.) antes de que pcspool se quede residente en memoria y regrese al DOS. El código de inicialización tambien establece las direcciones para la mayoria de los *buffers* del TSR y contiene el codigo para enviar mensajes de error cuando se introduzcan comandos erroneos.

Al concluirse la inicializacion, la mayor parte del código en la sección de inicializacion es sobrescrito con *buffers* de datos y una de las rutinas manejadoras de cola.

La mayor parte de la porcion TSR de pcspool esta constituida por las rutinas manejadoras de interrupciones. Todo el código para las interrupciones esta contenido en un segmento de código único y comparten campos de datos comunes, *buffers* de memoria, etc. Una vez que pcspool se hace residente, todas sus funciones son invocadas via interrupciones.

Las rutinas manejadoras de cola en memoria y en disco, están localizadas en segmentos separados al final del programa. Al determinarse el tipo de cola a emplear, el manejador correspondiente es trasladado a una porción de la memoria TSR. El otro manejador es sobrescrito o la memoria que ocupaba es liberada.

Este diseño conserva el resto del programa, relativamente independiente de un dispositivo. Es responsabilidad del manejador asegurarse de que los datos sean mandados hacia y obtenidos desde los registros de cola conforme sean solicitados. El manejador no debe "hacer suposiciones" acerca del contenido de los registros o de la cola. La única excepción aquí es el manejo del registro 0. En el caso de tener la cola en memoria, donde no hay necesidad de reanudar la impresión debido a que aquí no habrá interrupción, no hay motivo para mantener el registro cero en la cola, por lo que no será escrito, y el espacio será usado para el spool de datos adicionales. A continuación mostramos el listado #17 "PCSPOOL.ASM".

LISTADO NO. 17

```
title  "PCSPOOL  Spooler de Impresión"

svc_chars  equ   320/18           caracteres a/ft por servicio
svc_tests  equ   505              # a/ft de pruebas por caracter
lpt_dsps   equ   1024             tamaño del buffer de desvolso

;-----------------------------------------------
; definición de estructura para una impresora
;-----------------------------------------------

lpstruc    struc
lpprtnum   dw        ?            número de impresora
lpcomport  dw        0            dir. puerto de e/s de impr
lpfirstqr  dw        0            primer registro en cola
lplastqr   dw        0            último registro en cola
lpprtcnt   dd        0            caract. actualmente impresos
lpchrque   dd        0            caracteres en spool
lpdqbuf    dd        0            apuntador a buf. de desvolso
                                  ó "pseudo"
lplast     dd        0            última cuenta de carac. impresos
lpticks    dd        0            # de ticks para carac. impresos
lpdqin     dw        0            offset del sig. carac. a colocar
                                  actualizado por main
lpdqout    dw        0            offset del prox. carac. a imprimir
                                  actualizado por lpprt
lpcurbar   dw        0            apuntador a reg. de cola de spool
lpspools   db        80h          estado corriente de spool
lpstatus   db        0feh         estado actual de impresora
                                  ; 00 = ok
                                  ; 01 = no lista (hardware)
                                  ; 02 = pausa / mensaje
                                  ; 03 = pausa
                                  ; 04 = espera iniciar impresión
                                  ; fe = puerto no existente
                                  ; ff = [illegible]
lpcontrol  db        0            tipo de reg. de ctrl. actual
lpcps      dw        0            velocidad de impresora
lpchars    dw        svc_chars    carac. a enviar por servicio
lpwrite    db        0            1 = escritura necesaria
lpflush    db        0            1 = flujo de datos en cola
lpmodfy    db        0            1 = actualiza estado de cps
lpstruc    ends

;-----------------------------------------------
; definición de entrada al registro de cola
;-----------------------------------------------

qrlen          equ   512          long. de entrada a reg. de cola
                                  ; nota: modificar 512 requiere
                                  modificaciones en el código
                                  de la rutina [illegible]
qrdlen         equ   qrlen - 5    longitud de datos
qrstruc    struc                  estructura del reg. de cola
qrnextqr   dw        ?            ; siguiente reg. en cola
qrcontrol  db        ?            ; byte de control
                                  ; 00 datos
                                  ; 01 datos, sig. reg. es ctrl
                                  ; 02 dump datos reg. cola, pausa
                                  ; 04 inicializa impresora
qrbytes    dw        ?            ; bytes en registro
qrdata     db        qrdlen dup (?)   ; el resto son datos
qrstruc    ends
cgroup         group  mainline, comseg, dataq
mainline   segment para public 'code'        mainline
               assume cs:mainline, ds:mainline, es:mainline, ss:mainline
               org    100h
begin          jmp    start                  ; inicia proceso

;-----------------------------------------------
; Ésta es el área donde será cargada la cabecera de cola
;-----------------------------------------------

qheader    label  byte                       ; área de cabecera de cola
qhvalid    dw        0                       ; byte de signatura
qhfirstf   dw        0                       ; primer buffer libre
qhlpt1         lpstruc    <0>                estructura para lpt1
qhlpt2         lpstruc    <1>                estructura para lpt2
qhlpt3         lpstruc    <2>                ; estructura para lpt3
lplen          equ   qhlpt2 - qhlpt1         ; longitud de estructura lp
qhlen          equ   2 + (lplen*3)           ; longitud de cabecera de cola

;-----------------------------------------------
; control de vectores de interrupción
;-----------------------------------------------

intstkst   equ   12         ; debug          tamaño de pila para interrup
intblkcnt  equ   6                           # de rutinas de interrupción
intblklen  equ   13                          ; longitud de bloques
intblk1    equ   offset oldint13             primera a sustituir
oldint13   dd    0                           ; cs ip anterior
oldss13    dd    0                           ; ss sp cuando la int. es invocada
newss13    dd    0                           ; pila de trabajo durante int
               db    13h                     ; # de interrup. a apoderarse
               dw    offset int13            nuevo offset de interrupción
oldint09   dd    0                           cs ip anterior
oldss09    dd    0                           ss sp cuando la int. es invocada
newss09    dd    0                           pila de trabajo durante int
               db    09h                     ; # de interrup. a apoderarse
               dw    offset int09            nuevo offset de interrupción
oldint17   dd    0                           cs ip anterior
oldss17    dd    0                           ss sp al invocar interrup
newss17    dd    0                           ; pila de trabajo durante int
               db    17h                     # de interrup. a apoderarse
               dw    offset int17            ; nuevo offset de interrupción
oldint08   dd    0                           ; cs ip anterior
oldss08    dd    0                           ; ss sp al invocar interrup
```

```
newsat8  dd      0                  ;pila de trabajo durante int
                db      08h         ;# de interrup a apoderarse
         dw             offset int8 ;nuevo offset de interrupcion
oldint21 dd      0                  ;cs:ip anterior
oldss21  dd      0                  ;ss:sp al invocar interrup
newss21  dd      0                  ;pila de trabajo durante int
         db      21h                ;# de interrup a apoderarse
         dw      offset int21       ;nuevo offset de interrupcion
oldint28 dd      0                  ;cs:ip anterior
oldss28  dd      0                  ;ss:sp al invocar interrup
newss28  dd      0                  ;pila de trabajo durante int
         db      28h                ;# de interrup a apoderarse
         dw      offset int28       ;nuevo offset de interrupcion
svc_chars dw     svc_chars          ;caracteres por servicio
pcsflg   db      0                  ;banderas de spool
pcsflg1  equ     01h                ;  marca spool lpt1
pcsflg2  equ     02h                ;  marca spool lpt2
pcsflg3  equ     04h                ;  marca spool lpt3
pcsflgcm equ     08h                ;  memoria convencional
pcsflgdk equ     10h                ;  spool en disco
pcsflgem equ     20h                ;  EMS (futuro)
pcsflgst equ     18h                ;todos los bits tipo de spool
qfile    dd      0                  ;rutina de cola
qfnlen   dw      0                  ;longitud de rutina
qfnseg   equ     word ptr qfile+2   ;segmento de rtn de cola
dqrec    dw      0                  ;apt a reg de descolado
qfile    db      "\   "             ;direc directorio default
         db      63 dup (0)
nextqrnbr dw     1                  ;sig # de reg de cola
memqseg  dw      0                  ;segmento de memoria de spool
memqsize dw      16                 ;tamano K tamano contr regs
nextavail dw     qhandler           ;proximo byte disponible
startclr dw      0                  ;direcc de inicio limpiar mem
clrlen   dw      0                  ;extension a limpiar
dos_busy dw      2                  ;direcc de bandera "DOS" ocupado
i28flag  db      0                  ;usada por rtn de espera del BIOS
intsbusy db      0                  ;bandera "busy" de ints
i08flg   dw      0                  ;int 08 en uso
i17flg   db      0                  ;int 17 en uso (punto)
chkprflg db      0                  ;rtn chkprt en uso
qmuseflg db      0                  ;bandera de cola en uso
close_req db     0                  ;bandera de peticion de cierre
                                    ;0 = no solicitado
                                    ;1 = cierre solicitado
                                    ;2 = cierre terminado
i21hndl  dw      0                  ;manejador de archivos de usuario
i21hlen  dw  0                      ;longitud restante
i21hflg  db      0                  ;bandera de uso de int
tsr_active db    0                  ;bandera de tsr activo
tsr_req  db      0                  ;bandera de peticion de tsr
tsr_cnt  db      0                  ;contador de ticks del tsr
   db        "Hot key > "           ;apt visual para hotkey
shift_mask db           0ch         ;mascara de estado
                                    ;  shift derecho
                                    ;  shift izquierdo
                                    ;  tecla CTRL
                                    ;  tecla Alt
hot_key  db             19h         ;codigo de exploracion de hotkey

;----------------------------------------------------------------
;elementos de manejo de pantalla
;

lines    equ   5                    ;5 lineas
cols     equ   80                   ; de 80 columnas
monoattr equ   07h                  ;atributo monocromatico
colattr  equ   1fh                  ;atributo color
curattr  db             07h         ;atr a usar default monocrom.
                                    ;despliegue de mensajes de edo.
                                    ;(edo lp, longitud, mensaje)
dstat_msgs db  0,43,"ssss **********  sssK  ssssK  ssssK  hh:mm:ss"
         db    1,19,"Impresora no lista"
         db    2,5,"Pausa"
         db    3,27,"Pausa: use ^P para continuar"
         db    4,25,"Esperando intervencion"
         db    5,14,"Fin de cola"
         db    0ffh,24,"Impresora no encontrada"
         db    -1,25,"No usa spool actualmente"
dstat_nbr equ  7                    ;# de mensajes - 1

;----------------------------------------------------------------
; Esta rutina verifica el buffer para las impresoras y envia la infor-
; macion a imprimir mediante la rutina INT 17H original.
;----------------------------------------------------------------

toprt    proc                       ;envia a impresora
         push  ax                   ;salva registros
         push  bx
         push  cx
         push  dx
         push  si
         push  ds
         push  es
         push  cs                   ;salva nuestra dir. de segmento
         pop   ds                   ;ds = nuestro segmento
         mov   bx,offset qhlpt1     ;bx = bloque de control lpt1
toprt10: cmp   [bx].lpstatus,01h    ;¿disponible para imprimir?
         jbe   toprt15              ;si no, hacer algo
         cmp   [bx].lpstatus,4      ;¿necesita intervencion?
         je    toprt12              ;si, manejar c/BIOS
         jmp   toprt90              ;sino, siguiente impresora
toprt12: mov   ah,2                 ;ah = func de status de impr
         mov   dx,[bx].lpprtnbr     ;dx = # de impresora
         pushf                      ;salvar nuestras banderas
         call  oldint17             ;invoca rtn 17h anterior
         mov   [bx].lpstatus,0      ;coloca impresora "lista"
         mov   [bx].lpcurrec,0      ;sig registro de cola
         jmp   short toprt80        ;continuar c/codigo comun
```

```
topr15   mov   ax,[bx].lpdqin          ax = offset de entrada
         cmp   ax,[bx].lpdqout         ¿algo para imprimir?
         je    topr180                 si no, avanzar siguiente
         mov   ax,svc_chars            ax = cuenta por servicio
         mov   [bx].lpchars,ax         restablece cuenta
topr20   mov   cx,svc_tries            cx = # de veces a intentar
         mov   dx,[bx].lpstatport      dx = direcc. de puerto de edo.
topr22   in    al,dx                   pre carga (linea ocupada (?))
         in    al,dx                   al = edo. de puerto de impresora
         test  al,10h                  ¿impresora seleccionada?
         jnz   topr24                  si es, verifica si está lista
         mov   [bx].lpstatus,3         sino, problema de hardware
         jmp   short topr180           verifica sig. impresora
topr24   mov   [bx].lpstatus,2         a Inhibe edo. de cpu
         test  al,80h                  ¿impresora lista?
         jnz   topr25                  si es, imprime carácter
         loop  topr22                  sino, intentar después
         jmp   short topr180           intento realizado? sig. impr.
topr25   mov   dx,[bx].lpprtnbr        dx = # de impresora
         mov   [bx].lpstatus,0         indica impresora disponible
         push  bx                      salva apt. a estructura lp
         xor   ah,ah                   ah = imprime carácter
         les   si,[bx].lpdqbuf         es:si "puddle" de memoria
         mov   bx,[bx].lpdqout         bx = sig. carac. a imprimir
         mov   al,es:[bx+si]           al = carácter a imprimir
         inc   bx                      bx = sig. carácter
         cmp   bx,lpt_dqsz             ¿bx superior a máximo?
         jb    topr30                  si no, continuar
         xor   bx,bx                   sino, reinicializa apuntador
topr30   mov   cx,bx                   cx = nuevo offset
         pop   bx                      bx = bloque lpx
         mov   [bx].lpdqout,cx         salva nuevo offset del buffer
         pushf                         salva nuestras banderas
         call  oldint17                invoca rutina int 17h anterior
         add   word ptr [bx].lpprted,1 incrementa cuenta carac. impr.
         adc   word ptr [bx].lpprted+2,0
         cli                           deshabilita interrupciones
         sub   word ptr [bx].lpmquc,1  decrementa carac. en spool
         sbb   word ptr [bx].lpmquc+2,0
         sti                           rehabilita interrupciones
         cmp   [bx].lpdqin,cx          ¿caracteres a imprimir?
         je    topr180                 si no, sale
         dec   [bx].lpchars            ¿suficientes carac. enviados?
         jnz   topr20                  si no, [illegible]
topr180  add   bx,lpfrm                bx = sig. entrada a estruc. lp
         cmp   bx,offset qalpt5        ¿última impresora a servir?
         ja    topr190                 si sí, sale del ciclo
         jmp   topr10                  sino, atender otra impresora
topr190  pop   es                      restablece registros
         pop   ds
         pop   si
         pop   dx
         pop   cx
         pop   bx
         pop   ax
         ret                           retorna al llamador
topr     endp

;
; Esta rutina maneja el intento realizado por los usuarios de imprimir
; haciendo uso de la int. 17h del BIOS. La salida es dirigida por medio
; de una cola a la impresora si está lista, sino es almacenada en buffer
; para luego ser "spooleada".
;
; Llamadas Standard a la int 17
;
;
; Envía Carácter a Impresora
;         ah = 00h, código de función
;         al = carácter a imprimir
;         dx = número de impresora
;
; Regresa
;         ah = estado de la impresora
;
; Inicializa Impresora
;         ah = 01h, código de función
;         dx = número de impresora
;
; Regresa
;         ah = estado de la impresora
;
; Obtiene Estado de Impresora
;         ah = 02h, código de función
;         dx = número de impresora
; Regresa
;         ah = estado de impresora
;           01h, impresora ocupada
;           90h, impresora lista
; Llamadas Extendidas a la int 17
;
; [illegible] Bloque de Control
;         ah = 0C0h, código de función
;         dx = número de impresora
; Regresa
;         es:bx = bloque de control
; Construye Registro de Control (Flujo de Escritura Pendiente)
;         ah = 0C1h, código de función
;         dx = número de impresora
;         ds:si = string ASCIIZ para salvar desplegados
; Flujo de Escritura Pendiente
;         ah = 0C2h, código de función
;         dx = número de impresora
; Cancela Cola de Impresora (Flujo de toda la salida colocada en cola)
;         ah = 0C3h, código de función
;         dx = número de impresora
; Pregunta si el Spooler está Activo
;         ah = 0C4h, código de función
```

```
;   Regresa
;           di - 0b0bfh
;           si - segmento
; Saltar Archivo en Cola de Impresora (1 avanza hasta el registro pausa)
;           ah = 0c5h  codigo de funcion
;           dx - numero de impresora
; Verifica el Estado de la Cola de Impresion
;           ah - 0c6h  codigo de funcion
;           dx   numero de impresora
;   Regresa
;           ax   0   impresora no activada o en pausa
;                1 - impresora ocupada
; Cierra Cola
;           ah - 0c7h  codigo de funcion
;           dx - numero de impresora (0)
;

int17tbl  label  byte                      ; tabla de codigo funcion rutina
          db       [illegible]             ; imprime caracter
          dw       i17h40
          db       01h                     ; inicializa impresora
          dw       i17h45
          db       02h                     ; [illegible]
          dw       i17h50
          db       0c0h                    ; obtiene dir. de bloque de ctl
          dw       i17h55
          db       0c1h                    ; [illegible]
          dw       i17h60
          db       0c2h                    ; flujo de escritura pendiente
          dw       i17h65
          db       0c3h                    ; cancela cola de impresion
          dw       i17h70
          db       0c5h                    ; salta archivo en cola
          dw       i17h80
          db       0c6h                    ; verifica cola de impresora
          dw       i17h85
          db       0c7h                    ; coloca bandera de cierre de cola
          dw       i17h89
          db       0ffh                    ; marca el fin de lista
          dw       i17h90                  ; obtiene edo. de impr. default
i17hcnt   db       0                       ; contador de int 17
int17     proc                             ; spool para int. de impresora
          cmp    ah, 0c4h                  ; peticion de spooler activo?
          jne    i17h05                    ; no - continuar
          mov    di, 0b0bfh                ; instalado, en base de signatura
          mov    si, cs                    ; y en nuestro segmento
          iret                             ; regresa al llamador
i17h05    push   ds                        ; salva registros
          push   bx
          push   cs                        ; obtiene direccionamiento
          pop    ds
          mov    bx, offset qbtbl          ; bx - bloque de control lpt
i17h10    cmp    dx, [bx].lpprtno          ; impresora correcta?
          je     i17h15                    ; si - verificar si esta
                                           ; spooleado
          add    bx, lpbln                 ; bx - sig. bloque de impresora
          cmp    bx, offset qbtbl          ; ultima impresora?
          jne    i17h10                    ; no - intentar de nuevo
          jmp    short i17h19              ; sino - rtn int 17 anterior
i17h15    cmp    [bx].lpstatus, 0fch       ; esta spooleada?
          jb     i17h20                    ; si - continuar
i17h19    cmp    ah, 0c0h                  ; una de las nuestras?
          jb     i17h19b                   ; no - ve a int 17 anterior
          cmp    ah, 0c7h                  ; todavia una de las nuestras?
          ja     i17h19b                   ; no - ve a int 17 anterior
          cmp    ah, 0c0h                  ; obten bloque de ctrl de impr?
          je     i17h20                    ; si - continuar
          cmp    ah, 0c6h                  ; verificar impresora?
          jne    i17h19a                   ; no - regresa al llamador
          xor    ax, ax                    ; ax = 0 impresora desocupada
i17h19a   pop    bx                        ; restablece registros
          pop    ds
          iret                             ; luego regresa al llamador
i17h19b   pop    bx                        ; restablece registros
          pop    ds
          jmp    cs:oldint17               ; luego va a rtn int 17 anterior
                                           ; y regresa de esto al usuario
i17h20    mov    i17hcnt, 1                ; indica llamada a int 17
          push   ax                        ; salva el resto de los registros
          push   si
          mov    si, offset int17tbl       ; si - tabla funcion rutina
i17h25    cmp    ah, byte ptr [si]         ; apuntamos a esta funcion?
          je     i17h30                    ; si - salir de ciclo
          add    si, 3                     ; si - sig. entrada a lista
          cmp    byte ptr [si], 0ffh       ; fin de la lista?
          jne    i17h25                    ; no - entrar al ciclo
i17h30    jmp    word ptr 1[si]            ; ir a la rutina apropiada
                                           ; rutinas
;
i17h40           label  byte               ; imprime caracter
          cmp    [bx].lpwrite, 1           ; escritura necesaria?
          jne    i17h40i                   ; no - continuar
          call   writeq                    ; pudo enviar reg. de cola?
          jnc    i17h40i                   ; si - continuar
          jmp    i17h90                    ; sino - regresa c/busy
i17h40i   cli                              ; para info por 1 segundo
          add    word ptr [bx].lpnque, 1   ; incrementa total en cola
          adc    word ptr [bx].lpnque + 2, 0
          cli                              ; sin inicio de interrupciones
          push   bx                        ; salva aptr. a bloque de ctl
          mov    si, [bx].lpwrbuf          ; si - registro de cola
          inc    i17hflg                   ; proceso critico activo
          mov    bx, [si].qrbytes          ; bx - # de bytes en registro
          mov    [si + bx].qrdata, al      ; almacena carac. en reg. de cola
          inc    [si].qrbytes              ; incrementa contador de caracs.
          dec    i17hflg                   ; proceso critico terminado
          pop    bx                        ; restablece registro
          cmp    [si].qrbytes, qrdlen      ; reg. de cola lleno?
```

```
                 [illegible]                    [illegible]
          jmp    si7h[illegible]                sino solo regresa
si7h402   mov    [si].qrcontrol,0               hacer el tipo de reg. de datos
          call   writeqr                        p puede enviar registro
          jz     si7h401                        i no, hasta despues
          jmp    si7h[illegible]                sino, regresar al llamador
si7h403   mov    [bx].ipspool,[illegible]       indica impresora aun hara
          mov    [bx].[illegible]               y escritura requerida
          jmp    si7h90                         luego regresa al llamador

si7h43    label  byte                           [illegible] impresora
          mov    si,[bx].ipwork[illegible]      si: reg. de cola a trabajar
          mov    [si].qrcontrol                 marca reg. [illegible]
          call   writeqr                        envia reg. de cola [illegible]
          mov    [si].qrcontrol,4               es un registro de [illegible]
          call   writeqr                        envia reg. de cola encargada
          jmp    si7h[illegible]                luego regresa a [illegible]

si7h50    label  byte                           obtiene edo. de impresora
          mov    si,[bx].ipworkqr               si: reg. de cola a trabajar
          jmp    si7h90                         en rutina de salida

si7h55    label  byte                           obtiene bloque de ctrl. de impr
          mov    ax,ds                          ax: nuestro segmento de datos
          mov    es,ax                          es: nuestro seg. de datos
          pop    si                             restablece registros
          pop    ax
          add    sp,2                           salta restableciendo es:bx
          pop    ds
          iret                                  regresa
                                                es:bx: bloque de impresora

si7h60    label  byte                           construye reg. de control
          push   bp                             salva registro bp
          mov    bp,sp                          obtiene apt. base de pila
          push   di                             salva algunos otros registros
          push   si
          push   cx
          push   es
          push   ds
          mov    si,[bx].ipworkqr               si: reg. de cola a trabajar
          push   si                             [illegible]
          mov    [si].qrcontrol,[illegible]     indica reg. [illegible] control
          call   writeqr                        envia reg. de cola encargada
          mov    [si].qrcontrol,2               tipo reg. [illegible]
          lea    di,[si].qrdata+2               di: área de string
          push   ds                             establece
          pop    es                             es: nuestro segmento
          mov    si,[bp]+[illegible]            si: string de usuario
          mov    ds,[bp]+8                      ds:si: string usuario
          xor    cx,cx                          cx: contador de caracteres
si7h62    cmp    cx,qrdlen-1                    p se trasladó suficiente?
          je     si7h63                         i si, no trasladar más
          xor    al,al                          simula fin
          jmp    [illegible]                    [illegible]
si7h63    lodsb                                 al: carácter del usuario
si7h64    stosb                                 almacena en reg. de cola
          inc    cx                             cx: contador de carac. copiados
          or     al,al                          p último carácter?
          jnz    si7h62                         i no, [illegible]
          cmp    cx,33                          p mensaje necesita scroll?
          jbe    si7h66                         i no, salta carac. sobrante
          mov    byte ptr es:[di-1],[illegible] salva carac. de fin de mensaje
          inc    cx                             y calcular longitud
si7h66    inc    cx                             cx: # de caracs. + longitud
          pop    si                             si: registro de cola
          pop    ds                             restablece nuestro seg. de datos
          mov    [si].qrbytes,cx                salva contador de bytes
          sub    cx,2                           cx: longitud de string
          mov    word ptr [si].qrdata,cx        salva mensaje
          pop    es                             restablece el resto de los regs
          pop    cx
          pop    si
          pop    di
          pop    bp
          call   writeqr                        envia reg. de cola encargada
          jmp    short si7h90                   luego a rutina de salida

si7h65    label  byte                           flush de escritura pendiente
          mov    bx,offset qhip[illegible]      bx: bloque de control ips
si7h67    mov    si,[bx].ipworkqr               si: reg. de cola a trabajar
          cmp    [si].qrbytes,0                 p contiene algo?
          jes    si7h68                         i no, sig. impresora
          call   writeqr                        sino, envia reg. de cola
si7h68    add    bx,iplen                       bx: sig. bloque de impresora
          cmp    bx,offset qhip[illegible]      p última impresora?
          jne    si7h67                         i no, intenta de nuevo
          jmp    short si7h90                   en rutina de salida

si7h70    label  byte                           cancela cola de impresión
          mov    [bx].ipflush,[illegible]       indica borrado de cola
          jmp    short si7h82                   y va al código común

si7h80    label  byte                           salva archivo en cola de impr
          mov    [bx].ipflush,2                 indica en que cola se salvará
si7h82    mov    si,[bx].ipworkqr               si: reg. de cola a trabajar
          mov    ax,[si].qrbytes                ax: contenido de reg. de cola
          cli                                   detiene int. por [illegible] segundo
          sub    word ptr [bx].ipmaqu[illegible],ax   decrementa total en cola
          sbb    word ptr [bx].ipmaqu[illegible]+2,0
          sti                                   reanuda interrupciones
          mov    [si].qrbytes,si                limpia reg. de cola
          mov    [si].qrcontrol,0               y el byte de control
          mov    [bx].ipstatus,[illegible]      [illegible]

si7h90    pop    si                             regreso de llamadas normal
          pop    ax
          mov    ah,[bx].ipspools               regresa estado de spool
```

```
        pop     bx
        pop     ds
        ret                             ;regreso al llamador
                                        ;
si7h83  label   byte                    ;verifica edo. de cola de impr
        push    cx                      ;salva otros registros
        sub     ax,ax                   ;ax = 0 para impr. sin trabajo
                                        ;en curso
        cmp     [bx].ipstatus,?         ;¿en espera de pausa?
        jne     si7h86                  ;si, listo
        cmp     [bx].ipfirstqr,0        ;¿cola vacía?
        jne     si7h87                  ;no, regresa ahora
si7h86  mov     cx,[bx].ipdown          ;cx = reg. byte de entrada en
                                        ;marcha
        cmp     cx,[bx].ipdqout         ;¿paddle de impresora vacío?
        je      si7h88                  ;si
si7h87  inc     ax                      ;ax = 1 impresora ocupada
si7h88  pop     cx                      ;restablece registros
        pop     si
        add     sp,2
        pop     bx
        pop     ds
        ret                             ;regreso al llamador
                                        ;
si7h89  label   byte                    ;coloca bandera de cierre de
                                        ;cola de impresora
        mov     close_flg,1             ;enciende bandera de petición
                                        ;de cierre
        jmp     short si7h90            ;luego sale
int17   endp

;
; Esta rutina intentara copiar los datos en cola al área de memoria de
; impresora (paddle), pero si no está lista, los datos serán escritos
; en el spool
;
; Entra
;       bx  bloque de control de impresora
;
; Regresa
;       carry = cola incapaz, intentar después
;
;

writeqr proc
        push    si                      ;salva registros
        push    ds
        mov     si,[bx].ipwrkqr         ;si = registro de cola
        cmp     [si].qrcontrol,?        ;¿registro de control?
        je      writeqr30               ;si, escribirlo
        cmp     [bx].ipcontrol,0        ;¿vacío de impresora?
        jne     writeqr10               ;si, manda a cola
        call    movetoprt               ;traslada reg. de spool a impr
        jnc     writeqr10               ;todo bien, continuar
```

```
writeqr10 call  testgdos                ;¿DOS disponible?
        jc      writeqr25               ;no, intentar después
writeqr20 mov   dx,si                   ;dx = reg. de cola a trabajar
        mov     ax,[bx].ipprtnbr        ;ax = número de impresora
        mov     ah,1                    ;ah = escribe reg. de cola
        call    qh                      ;¿pudo escribir registro?
        jnc     writeqr30               ;si, continuar
writeqr25 mov   [bx].ipspoolis,0ch      ;indica impresora ocupada
        mov     [bx].ipwrite,1          ;indica petición de escritura
        pop     ds                      ;restablece registros
        pop     si
        stc                             ;coloca bandera de carry
                                        ;(problemas)
        ret                             ;regresa al llamador
writeqr30 mov   [si].qrcontrol,0        ;limpia byte de control
        mov     [si].qrbytes,0          ;contador de caracteres
        mov     [bx].ipwrite,0          ;bandera de petición de escr
        mov     [bx].ipspoolis,90h      ;indica impresora lista de nuevo
        pop     ds                      ;restablece registros
        pop     si
        clc                             ;limpia bandera de carry
        ret                             ;regresa al llamador "ok"
writeqr endp

;
; Esta rutina intentará trasladar el reg. de cola corriente al área de
; memoria de las rutinas de "desencola". Esto ayudará a reducir la E/S
; de spool cuando la impresora continúa trabajando con el programa de
; aplicación
;
; Entra
;       bx  bloque de control de impresora
;
; Regresa
;       carry   buffer de impresora no suficientemente vacío
;
;

movetoprt proc
        cmp     [bx].ipfirstqr,0        ;¿hay regs. en cola?
        jne     movetop10               ;si, trasladar a spool
        push    si                      ;salvar registros
        push    cx
        call    getprt                  ;obtener espacio de impresora
                                        ;disponible
        mov     si,[bx].ipwrkqr         ;si = reg. de cola a trabajar
        cmp     [si].qrbytes,cx         ;¿espacio suficiente en mem.?
        jbe     movetop20               ;si, almacena registro
        pop     cx                      ;restablece registro
        pop     si
movetop10 stc                           ;prender bandera de carry
        ret                             ;luego regresa con estado de
                                        ;no pudo trabajarlo
movetop20 call  moverecord              ;traslada registro al paddle
[illegible]
```

```
        pop     cx                      ; restablece registros
        pop     si
        clc                             ; limpia carry - indica "ok"
        ret                             ; luego regresa al llamador
movexqrt  endp

; ----------------------------------------------------------------
; Esta rutina regresará la disponibilidad de uso del DOS para E/S a
; archivos. Si el spooler no está haciendo uso de funciones del DOS
; para E/S de archivo (utiliza memoria convencional o EMS) entonces
; el regreso será siempre "ok"
;
; Regresa
;         carry = DOS no está disponible para E/S a archivo
; ----------------------------------------------------------------

uningdos  proc
        test    pcsflg, pcsflgdx        ; ¿se está usando cola en disco?
        jz            uningdos90        ; no - regresa ok
        cmp     usedmg, 2               ; ¿memoria de manera del DOS?
        jne     uningdos90              ; si - entonces está bien
        cmp     intbusy, 0              ; ¿una interrupción ocupada?
        jne     uningdos80              ; si - sale código espera
        push    es                      ; salva registros
        push    bx
        les     bx, dos_busy            ; es:bx = bandera BUSY del DOS
        cmp     byte ptr es:[bx], 0     ; ¿DOS ocupado?
        pop     bx                      ; restablece registros
        pop     es
        je            uningdos90        ; no - regresa al usuario
uningdos80 stc                          ; uno - indica DOS ocupado
        ret                             ; y regresa al llamador
uningdos90 clc                          ; limpia carry - es ok
        ret                             ; y regresa al llamador
uningdos  endp

; ----------------------------------------------------------------
; Esta rutina traslada un registro al área de memoria de la impresora
; y ajusta apuntadores
;
; Entra
;         bx - bloque de control de impresora
;         si - registro de cola a trasladar
;
; ----------------------------------------------------------------

moverecord proc
        push    ax                      ; salva registros
        push    cx
        push    di
        push    si
        push    es
        cld                             ; limpia bandera de dirección
        mov     di, [bx].ipdqin         ; di - offset de entrada
        mov     es, word ptr [bx].ipdqbuf+2     ; es = segmento de paddle
        mov     cx, lpt_dqsz            ; cx - fin de paddle
        sub     cx, di                  ; cx = máx. caracteres en las copias
        mov     al, [si].qrcontrol      ; al = tipo de registro
        mov     [bx].ipcontrol, al      ; salvalo
        cmp     al, 2                   ; ¿fin de reg. de datos?
        jne     moverec20               ; no - continúa
        xor     al, al                  ; al = nuevo tipo de registro
moverec20 mov   [bx].ipstatus, al       ; establece nuevo estado
        mov     ax, [si].qrbytes        ; ax = # de carac. en reg.
        mov     [si].qrbytes, 0         ; limpia contador
        mov     [si].qrcontrol, 0       ; y tipo de registro
        lea     si, [si].qrdata         ; si - areg. de datos en cola
        or      ax, ax                  ; ¿algo a trasladar?
        jz      moverec90               ; no - mejor salir
        cmp     ax, cx                  ; ¿espacio suficiente?
        jb      moverec30               ; no - necesita dos
        mov     cx, ax                  ; cx = cantidad a trasladar
        jmp     short moverec40         ; a código común
moverec30 push  cx                      ; salva 1er porción de long.
    rep movsb                           ; traslada 1a porción del texto
        pop     cx                      ; cx = 1er porción de long.
        sub     ax, cx                  ; ax = obtiene nueva longitud
        mov     cx, ax                  ; cx = establecida para rep movsb
        xor     di, di                  ; di - inicio de paddle
moverec40
    rep movsb                           ; traslada datos en cola a paddle
        cmp     [bx].ipstatus, 2        ; ¿mensaje de pausa?
        je      moverec90               ; si - no ajustar datos
        cmp     di, lpt_dqsz            ; ¿al fin del paddle?
        jb      moverec50               ; no - salvalo como está
        xor     di, di                  ; di - 1er byte del paddle
moverec50 mov   [bx].ipdqin, di         ; almacena nuevo apuntador final
moverec90 pop   es                      ; restablece registros
        pop     si
        pop     di
        pop     cx
        pop     ax
        ret                             ; luego regresa al llamador
moverecord endp

; ----------------------------------------------------------------
; Esta rutina intercambia operaciones de escritura que hagan uso de la
; función 40H para cualquier asunto con la impresora. La información a
; escribir es dividida en bloques de acuerdo al tamaño máximo especi-
; ficado por la escritura o bien conforme al espacio sobrante del
; buffer de spool disponible. Ahora bien si el manejador utiliza el
; spooler entonces el resto de la porción de escritura es enviado en
; tramos. Esto permitirá que las rutinas que escriben en disco tengan
; tiempo suficiente para vaciar sus buffers a disco
;
; ----------------------------------------------------------------

int21     proc                          ; manejador de interrup. 21
```

```
        cmp   cs:i21hflg,0        ; ¿int 21 en uso?
        je    i21h10              ; no, verifica y coloca
        jmp   cs:oldint21         ; luego ir al DOS
i21h10: inc   cs:i21hflg          ; indica que estamos aqui
        call  chkprt              ; verifica lectura / escritura
        cmp   ah,40h              ; ¿escribe a manejador de arch?
        je    i21h15              ; si, verifica y coloca
        jmp   short i21h18        ; sino, ir al DOS
i21h15: push  dx                  ; salva direccion de buffer
        push  ax                  ; y funcion
        mov   ax,4400h            ; ax = IOCTL: obtiene inf de disp
        pushf                     ; simula interrupcion
        call  cs:oldint21         ; invoca al DOS
        cmp   al,0c0h             ; ¿impresora acaso?
        pop   ax                  ; restablece ax
        pop   dx                  ; restablece dx
        je    i21h20              ; si, continuar
i21h18: dec   cs:i21hflg          ; apaga bandera de uso de int 21
        jmp   cs:oldint21         ; luego ir al DOS
i21h20: push  bx                  ; salva registros
        push  dx
        push  cx
        mov   cs:i17hcnt,0        ; limpia bandera de int 17 en uso
        mov   cs:i21hhfh,bx       ; salva manejador de archivo
        mov   cs:i21hlen,cx       ; longitud restante
i21h30: cli                       ; deshabilita ints
        call  chkprt              ; escribe sig. bufer completo
        call  getmax              ; obtiene max long. a escribir
        sti                       ; habilita ints
        or    bx,bx               ; ¿queda espacio para escribir?
        jz    i21h30              ; no, intenta de nuevo
        cli
        cmp   bx,cx               ; ¿espacio suficiente?
        jae   i21h40              ; si, continuar
        mov   cx,bx               ; cx = cantidad a escribir
i21h40: mov   ah,40h              ; ah = escribir al manejador
        mov   bx,cs:i21hhfh       ; bx = manejador original
        pushf                     ; simula interrupcion
        call  cs:oldint21         ; ¿error de escritura?
        jnc   i21h50              ; no, continuar
        pop   cx                  ; restablece registro
        jmp   short i21h70        ; y sale

i21h50: mov   cx,cs:i21hlen       ; cx = lo que resta
        or    ax,ax               ; ¿algo a imprimir?
        jz    i21h60              ; no, entonces sale
        sub   cx,ax               ; ¿falta algo?
        jz    i21h60              ; no, sale de ciclo
        mov   cs:i21hlen,cx       ; salva long. restante
        add   dx,ax               ; dx = sig. bloque fuente
        cmp   cs:i17hcnt,0        ; ¿int 17 usada?
        jne   i21h30              ; si, regresa a loop
        jmp   i21h40              ; sino, terminar
i21h60: pop   cx                  ; restablece registros
        mov   ax,cx               ; ax = cantidad de carac. escrit
        clc                       ; indica todo ok
i21h70: pop   dx                  ; restablece regs
        pop   bx
        dec   cs:i21hflg          ; apaga bandera de int 21 en uso
        iret                      ; y regresa al llamador
int21   endp

; Esta rutina verifica si existen peticiones de lecturas / escrituras a
; cola que necesitan ser ejecutadas; para esto invoca a las rutinas de
; manejo de cola para que se encarguen de estas operaciones
;

chkprt  proc
        cmp   cs:chkprtflg,0      ; ¿chkprt en uso?
        jne   chkprt01            ; si, sale
        cmp   cs:qmuxflg,0        ; ¿cola en uso?
        je    chkprt00            ; no, continuar
chkprt01: ret                     ; sino, regresa a llamador
chkprt00: inc  cs:chkprtflg       ; indica que estamos aqui, no
        push  ax                  ; salva regs
        push  bx
        push  cx
        push  dx
        push  si
        push  di
        push  cs                  ; obtiene direccionamiento
        pop   ds
        cmp   close_req,1         ; ¿peticion de cierre de cola?
        jne   chkprt05            ; no, continuar
        mov   ax,7                ; ax = funcion de cierre de spool
        call  dword ptr qrtn      ; invoca rutina de cola
        mov   close_req,2         ; indica cola cerrada
        jmp   chkprt00            ; luego regresa a llamador
chkprt05: mov  bx,offset qhlpe    ; bx = manej. de control list
chkprt10: cmp  [bx].lpflush,0     ; ¿se requiere flujo de cola?
        je    chkprt40            ; no, continuar
        mov   cl,[bx].lpflush     ; cl = modo de flujo
        mov   dx,dgrec            ; dx = reg. de descalce
        mov   si,dx               ; si = mismo registro
chkprt30: mov  ax,[bx].lpnum      ; ah = leer, al = # de impr
        call  qrtn                ; ¿obtiene sig. reg. de cola?
        jc    chkprt40            ; no, intentar despues
        cmp   [si].qrcontrol,1    ; ¿registro de datos?
        jne   chkprt35            ; no, continuar
        mov   ax,[si].qrbytes     ; ax = # de bytes en reg
        sub   word ptr [bx].lpnque,ax     ; actualiza total en spool
        sbb   word ptr [bx].lpnque+2,0
chkprt35: cmp  cl,1               ; ¿flujo de toda la cola?
        je    chkprt30            ; si, obtiene sig. reg. de cola
        cmp   [si].qrcontrol,2    ; ¿sig. reg. es pausa?
        jne   chkprt30            ; no, obtiene sig. reg. de cola
chkprt40: mov  [bx].lpflush,0     ; limpia bandera de flujo
```

```
        mov   [bx].lpcontrol, 0        ; último reg. de control
chkprt00 cmp  [bx].lpwrite, 1          ; ¿escritura requerida?
        jne   chkprt2                  ; no: verifica lectura
        call  writeqr                  ; envía reg. de cola
chkprt2 cmp   [bx].lpstatus, 1         ; ¿impresora imprimiendo?
        je    chkprt70                 ; no: siguiente
        call  getprt                   ; ax = máx. espacio en puddle
        cmp   [bx].lpcontrol, 0        ; ¿sig. reg. de cola en datos?
        je    chkprt45                 ; sí: obtenlo
        cmp   cx, lpt dqsz - 1         ; ¿buffer vacío?
        jne   chkprt70                 ; no: espera a que lo esté
        xor   ax, ax                   ; ax = 0
        mov   [bx].lpdqin, ax          ; limpia aptr. de entrada
        mov   [bx].lpdqout, ax         ; y aptr. de salida
chkprt45 cmp  cx, qrlen                ; ¿suficiente para lectura?
        jb    chkprt70                 ; no: sig. impresora
        mov   ax, [bx].lpprtnbr        ; ah = lee; al = # de impr.
        mov   dx, dqrec                ; dx = reg. de dem. [illegible]
        call  qb                       ; ¿obtuvo sig. reg. de cola?
        jc    chkprt70                 ; no: inténtalo después
        mov   si, dx                   ; si = registro
        jmp   short chkprt80           ; continúa código común
chkprt50 mov  si, [bx].lpwrkqr         ; si = reg. de trabajo
        cmp   i17flg, 0                ; ¿alguien realiza int 17h?
        jne   chkprt70                 ; sí: ejecutar después
chkprt80 call moverecord               ; transfiere reg. a puddle de impr.
chkprt70 add  bx, lplen                ; bx = sig. bloque de impresora
        cmp   bx, offset qblpt3        ; ¿última impresora?
        je    chkprt90                 ; sí: regresa a llamador
        jmp   chkprt10                 ; no: sig. impresora
chkprt90 pop  dx                       ; restablece registros
        pop   si
        pop   dx
        pop   cx
        pop   bx
        pop   ax
        dec   cs:chkprtflg             ; limpia bandera de rutina en uso
        ret                            ; y regresa al llamador
chkprt  endp
```

;
; Esta rutina devuelve el tamaño máximo de escritura que el sistema de
; spool puede soportar con ayuda del DMS en este momento. Este espacio
; disponible es actualmente un espacio de buffer muy pequeño
;
; Regresa
;   bx = máximo espacio disponible en el registro de cola a utilizar
;

```
getmax  proc
        push  ax
        push  cx
        push  si
        push  dx
        push  cs                       ; [illegible]
        pop   ds
        mov   ax, qrdlen               ; ax = valor máximo
        mov   bx, offset qblpt1        ; bx = bloque de ctrl lpt1
getmax10 mov  si, [bx].lpwrkqr         ; si = reg. de cola de la int. 17
        mov   cx, qrdlen               ; cx = tamaño máximo de reg.
        sub   cx, [si].qrbytes         ; cx = bytes libres
        cmp   ax, cx                   ; ¿esta impresora tiene menos?
        jb    getmax20                 ; no: sig. impresora
        mov   ax, cx                   ; ax = nuevo valor menor
getmax20 add  bx, lplen                ; bx = sig. bloque de impr.
        cmp   bx, offset qblpt3        ; ¿última impresora?
        jne   getmax10                 ; no: trata de nuevo
        mov   bx, ax                   ; bx = valor a regresar
        pop   dx                       ; restablece regs.
        pop   si
        pop   cx
        pop   ax
        ret                            ; regreso a llamador
getmax  endp
```

;
; Esta rutina devuelve el espacio disponible en el área de memoria de
; impresora (puddle)
;
; Entra
;   bx = bloque de control de impresora
;
; Regresa
;   cx = máximo espacio disponible en área de memoria de impresora
;

```
getprt  proc
        push  ax                       ; salva registros
        mov   ax, [bx].lpdqin          ; ax = sig. carac. de entrada
        mov   cx, [bx].lpdqout         ; cx = sig. carac. a imprimir
        cmp   ax, cx                   ; ¿buffer vacío?
        je    getprt10                 ; sí: usa máximo
        jb    getprt20                 ; casi lleno: ¿queda alguno?
        sub   ax, cx                   ; ax = lo que está utilizado
        mov   cx, lpt dqsz - 1         ; cx = máx. disponible
        sub   cx, ax                   ; cx = lo que no está utilizado
        pop   ax                       ; restablece registro
        ret                            ; y regreso a llamador
getprt10 mov  cx, lpt dqsz - 1         ; cx = tamaño del puddle
        pop   ax                       ; restablece registro
        ret                            ; y regreso a llamador
getprt20 sub  cx, ax                   ; cx = lo que resta
        dec   cx                       ; menos 1
        pop   ax                       ; restablece registro
        ret                            ; y regresa a llamador
getprt  endp
```

```
;
; Esta rutina verifica si las áreas de memoria de impresora necesitan
; ser rellenadas y verifica las banderas de "ejecución requerida" en
; la parte de apoyo
;

int28       proc
            sti                         habilita ints
            inc   cs:i28hflag           indica que se está usando
                                        rutina de int 28h
            call  chkprt                muestra de pendiente de impresión
            dec   cs:i28hflag           apaga bandera de uso de int 28h
            jmp   dword ptr cs:oldint28 se encadena a rut de int 28
                                        anterior
int28       endp

;
; Esta rutina está pendiente del empleo de la int 13 para la rutina DOS
; en uso
;

int13       proc
            sti                         habilita interrups
            inc   cs:intbusy            indica interrup en uso
            pushf                       simula llamada a interrup
            call  dword ptr cs:oldint13 invoca la rut int 13 anterior
            dec   cs:intbusy            apaga bandera de int en uso
            ret                         y regresa a llamador
int13       endp

;
; Intenta imprimir algunos caracteres al tick del reloj
;

int08       proc
            push  ds                    salva registros
            push  cs                    establece direccionamiento
            pop   ds
            pushf                       simula llamada a interrup
            call  oldint08              invoca servicios de rutina
                                        de reloj anterior
            inc   i08flg                bloquea otra llamada
                                        y checa si ya está
                                        ocupada
            cmp   i08flg,1              ¿int 08 en uso?
            jnz   i08h10                sí, salir rápido
            sti                         habilita interrupciones
            call  setprt                intenta imprimir
            call  tscpt                 calcular velocidad de cps
            call  suspdos               ¿DOS disponible, requerido?
            jz    i08h05                no, intentar después
            call  chkprt                se transfiere hacia la impr
i08h05:     mov   i08flg,0              apaga bandera para otra llamada
i08h10:     inc   tsr_cnt               incrementa contador de tsr
            cmp   tsr_req,0             ¿tsr demandado?
            jz    i08h30                no, intentar después
            cmp   tsr_active,0          ¿tsr activo?
            jnz   i08h30                sí, continuar
            cmp   chkprtflg,0           ¿impresora activa?
            jnz   i08h30                sí, continuar
            inc   tsr_active            bloquea otra llamada
            sti                         habilita ints
            call  suspdos               ¿DOS disponible/requerido?
            jz    i08h20                no, intentar después
            call  tsr                   ejecuta tsr
i08h20:     mov   tsr_active,0          apaga bandera tsr activo
i08h30:     pop   ds                    restablece registros
            iret                        y regresa al llamador
int08       endp

; Intercepta la hot key
;

int09       proc
            cmp   cs:tsr_active,0       ¿tsr en uso?
            jnz   i09h15                sí, salir
            push  ax                    salva regs usados
            in    al,60h                obtener código de exploración
            cmp   al,cs:hot_key         ¿nuestra hot key?
            je    i09h10                sí, continuar
            pop   ax                    restablece registro
            jmp   short i09h15          y salir
i09h10:     mov   ah,2                  ah = obtiene máscara de edo
            int   16h                   llama BIOS
            and   al,0fh                al = bits de estado
            cmp   al,cs:shift_mask      ¿iguala nuestro edo. la tecla?
            pop   ax                    restablece registro
            je    i09h20                sí, continuar
i09h15:     jmp   cs:oldint09           sino, trata tecla normal
i09h20:     pushf                       invoca int 9 anterior
            call  cs:oldint09           para procesar tecla
            sti                         habilita ints
            push  ax                    salva reg. de trabajo
i09h30:     mov   ah,1                  ah = obtiene estado de teclado
            int   16h                   ¿tecla disponible?
            jz    i09h40                no, salir de ciclo
            mov   ah,0                  ah = obtén tecla
            int   16h                   lectura de tecla
            jmp   i09h30                y repite hasta que no haya
                                        teclas
i09h40:     pop   ax                    restablece registro
            inc   cs:tsr_req            indica petición de tsr
            iret                        regresar al llamador
int09       endp
```

```
;------------------------------------------------------------------
; Esta rutina realiza el cálculo de los caracteres impresos por segundo
; y actualiza CPS al ser invocada por las rutinas de despliegue del tsr
;------------------------------------------------------------------

tscps       proc
            push  ax                        ; salva registros
            push  bx
            push  dx
            lea   bx, qblpt1                ; bx = 1er bloque de impresora
tscps10:    cmp   [bx].lptcntflg, 0         ; ¿ actualizar CPS?
            je    tscps30                   ; no, sig. impresora
tscps20:    mov   ax, tkcntfg               ; ax = # de ticks en este ciclo
            add   word ptr [bx].lpttcks, ax         ; acumula total de ticks
            adc   word ptr [bx].lpttcks + 2, 0       ; para impresión este tanto
            mov   [bx].lptcntflg, 0         ; apaga bandera de actualizar
                                            ; CPS
tscps30:    add   bx, lptlen                ; bx = sig. impresora
            cmp   bx, offset qblpt3         ; ¿ última impresora?
            jb    tscps10                   ; no, repite ciclo
            pop   dx                        ; restablece regs
            pop   bx
            pop   ax
            ret                             ; y regresa a llamador
tscps       endp

;------------------------------------------------------------------
; Esta rutina contiene la porción TSR de interfaz de usuario. Desplie-
; ga el estado corriente, controla impresoras y colas de impresión
;------------------------------------------------------------------

tsr         proc
            push  ax                        ; salva registros
            push  bx
            push  cx
            push  dx
            push  si
            push  di
            push  bp
            push  es
            mov   tsr_req, 0                ; limpia bandera de petición de
                                            ; TSR
            mov   ah, 0Fh                   ; ah = obtener edo. actual
            int   10h                       ; llama al BIOS
            push  bx                        ; salva página de video corriente
            mov   ah, 3                     ; ah = obtiene posición de cursor
            int   10h                       ; llama al BIOS
            push  dx                        ; salva posición de cursor
            push  cx                        ; y modo de cursor
            mov   ah, 1                     ; ah = coloca tipo de cursor
            mov   cx, 2020h                 ; = no cursor
            int   10h                       ; llama al BIOS
            call  swapdisp                  ; ¿ intercambio pags. de video?
            jnc   tsr20                     ; sí, continuar
            mov   ax, 0e07h                 ; ax = manda un beep
            int   10h                       ; llama al BIOS
            jmp   short tsr90               ; y regresa al llamador
tsr20:      mov   tsr_cnt, 0                ; limpia contador de ticks
            call  dstat                     ; despliega el edo. corriente
tsr30:      call  chkqrt                    ; verifica colas de impresión
            call  dcmds                     ; manejo de comandos
            jc    tsr80                     ; repite ciclo hasta que la
                                            ; salida sea seleccionada
            cmp   tsr_cnt, 36               ; ¿ actualizar pantalla?
            jb    tsr30                     ; no, sólo verifica colas
            jmp   short tsr20               ; sí, despliega inf. de edo.
tsr80:      call  swapdisp                  ; restablece pags. de video
tsr90:      pop   cx                        ; cx = modo original
            mov   ah, 1                     ; ah = coloca modo de cursor
            int   10h                       ; llama BIOS
            pop   dx                        ; dx = posición original
            pop   bx                        ; bh = # de página de video
            mov   ah, 2                     ; ah = coloca posición
            int   10h                       ; lo ejecuta
            pop   es                        ; restablece registros
            pop   bp
            pop   di
            pop   si
            pop   dx
            pop   cx
            pop   bx
            pop   ax
            ret                             ; y regresa al llamador
tsr         endp

;------------------------------------------------------------------
; Esta rutina despliega el estado corriente de cada impresora.
;------------------------------------------------------------------

dstat       proc
            push  cx                        ; salva registro
            mov   bx, offset qblpt1         ; bx = 1er impresora
dstat10:    mov   dl, [bx].lptstatus        ; dl = estado corriente
            xor   dh, dh                    ; dh = ningún mensaje
            cmp   dl, 2                     ; ¿ pausa / mensaje?
            jne   dstat15                   ; no, continuar
            mov   dh, word ptr [bx].lptqbuf + 2    ; dh = seg. de mensaje
dstat15:    lea   si, dstat_msgs            ; si = mensajes
            mov   cx, dstat_nbr             ; cx = msgs
dstat20:    cmp   dl, [si]                  ; ¿ es éste nuestro mensaje?
            jne   dstat30                   ; no, ve al sig.
            inc   si                        ; si = último mensaje
            jmp   short dstat40             ; continúa con código común
dstat30:    mov   al, [si]                  ; al = longitud de mensaje
            cbw                             ; ax = long. de mensaje
            add   ax, 2                     ; ax = lon. de bloque de msg
            add   si, ax                    ; si = sig. mensaje
            loop  dstat20                   ; ¿ más mensajes?
```

```
datas40  cmp   di, 0                     ; ¿se usa mensaje preparado?
         jne   datas50                   ; si - continuar
         call  dmbld                     ; construir mensaje
datas50  call  dmsvc                     ; traslada mensaje a desplegar
         cmp   [bx].lpstatus, 2          ; ¿pausa c/comentario?
         jne   datas60                   ; no - continuar
         les   si, [bx].lpdspbuf         ; es:si  buffer de despliegue
         mov   cx, es:[di]               ; cx = longitud de string
         cmp   cx, 35                    ; ¿comentario necesita scroll?
         jna   datas60                   ; no - continuar
         push  ds                        ; salva regs
         push  es                        ; salva seg. de buffer de despliegue
         pop   ds                        ; ds  buffer de despliegue
         mov   ah, ds:[37]               ; ah = 1er caracter
         mov   di, 2                     ; di  área de salida
         mov   si, 3                     ; si  área de entrada
         dec   cx                        ; cx = caracteres a mover
         cld                             ; de abajo a arriba
   rep   movsb                           ; corrimiento de string
         mov   [si-1], ah                ; salva caracter corrido
         pop   ds                        ; restablece ds
datas60  add   bx, lpien                 ; bx  sig. impresora
         cmp   bx, offset qkipt1         ; ¿terminado?
         jne   datas10                   ; no - repite ciclo
         pop   es                        ; restablece registro
         ret                             ; regreso a llamador
datas    endp

;
; Esta rutina construye el mensaje de estado de "impresora ok"
;
; Entra
;          bx  bloque de control de impresora
;          si  bloque de mensaje
;

dmbld    proc
         push  si                        ; salva registros
         push  di
         push  cx
         push  bx                        ; salva apuntador lpt
         mov   di, bx                    ; coloca a nuevo apuntador
         mov   ax, word ptr [di].lpticks         ; dx:ax = # de ticks
         mov   dx, word ptr [di].lpticks + 2
         mov   cx, 10                    ; cx = multiplicador
         call  mmul                      ; multiplica por 10
         mov   bx, 182                   ; cx:bx = 18.2ticks/seg
         xor   cx, cx
         call  ddiv                      ; convierte ticks a segs
         mov   cx, dx                    ; cx:bx = segundos
         mov   bx, ax
         mov   ax, word ptr [di].lpprted         ; dx:ax = caracs. impresos
         mov   dx, word ptr [di].lpprted + 2
         call  ddiv                      ; divide para obtener
                                         ; caracs./seg
         pop   bx                        ; restablece apuntador lpt
         mov   [bx].lpcps, ax            ; salva sel. en CPS
dmbld10  xor   dx, dx                    ; usa de doble palabra
         mov   di, si                    ; di  área de salida
         add   di, 22                    ; di  área de salida (lsb)
         call  dformat                   ; convierte a ascii
         mov   ax, word ptr [bx].lpprted         ; ax = parte de cuenta
         mov   dx, word ptr [bx].lpprted + 2     ; dx:ax = caracs. apuntados
         mov   di, si                    ; di  área de salida
         add   di, 28                    ; di  área de salida (lsb)
         call  dformat                   ; convierte a ascii
         mov   ax, word ptr [bx].lpmque          ; ax = parte de cuenta
         mov   dx, word ptr [bx].lpmque + 2      ; dx:ax = caracs. en cola
         mov   di, si                    ; di  área de salida
         add   di, 34                    ; di  área de salida (lsb)
         call  dformat                   ; convierte a ascii
         call  dgauge                    ; construye % [illegible]
         push  bx                        ; salva apuntador lp
         mov   ax, word ptr [bx].lpmque          ; ax = parte de cuenta
         mov   dx, word ptr [bx].lpmque + 2      ; [illegible]
         mov   bx, [bx].lpcps            ; cx:bx = CPS calculado
         xor   cx, cx
         call  ddiv                      ; segs. restantes para imgr
         pop   bx                        ; restablece registro
         mov   di, si                    ; di  registro de salida
         or    dx, dx                    ; ¿ 65535 segundos?
         jnz   dmbld30                   ; si - actualiza horas
         add   di, 33                    ; di  área de salida (lsb)
         mov   cx, 60                    ; cx = divisor
         div   cx                        ; ax = minutos, dx = segundos
         call  dtoa                      ; manda segundos
         xor   dx, dx                    ; dx = 0 para dividendo
         div   cx                        ; ax = horas, dx = minutos
         call  dtoa                      ; envia minutos
         mov   dx, ax                    ; dx  horas
         call  dtoa                      ; envia horas
         mov   byte ptr [si + 18]        ; para formato de tiempo
         mov   byte ptr [si + 41]
         jmp   short dmbld40             ; continua c/código común
dmbld30  push  bx                        ; salva apuntador lp
         mov   bx, 3600                  ; cx:bx = segundos por hora
         xor   cx, cx
         call  ddiv                      ; resto de horas por imprimir
         pop   bx                        ; restablece registro
         add   di, 38                    ; di  área de salida (lsb)
         call  dformat                   ; envia horas
         mov   word ptr [si + 40], [illegible]   ; para formato de tiempo
         mov   word ptr [si + 42], ...   ; usa " Hrs"
dmbld40  pop   cx                        ; restablece regs
         pop   di
         pop   si
         ret                             ; luego regresa a llamador
dmbld    endp
```

```
;
; Esta rutina imprime mensaje en pantalla
;
; Entra
;         bx    bloque de control de impresora
;         si    bloque de mensaje
;         es:di mensaje secundario o 0
;

dmove     proc
          push  bx                        ; salva registros
          push  di
          mov   ah, 3                     ; ah = checa pos de cursor
          mov   bh, byte ptr [bx].lppartido     ; bh = # de impresora
          inc   dh                        ; dh = # de renglon
          mov   dl, 3                     ; dl = # de columna
          mov   bh, 0                     ; bh = # de pag de salida
          int   10h                       ; llama BIOS
          xor   ch, ch                    ; ch = 0
          mov   cl, [si]                  ; cl = longitud
          push  cx                        ; salva longitud de mensaje
          inc   si                        ; si = mensaje
dmove10:  mov   ah, 0eh                   ; ah = escribe estilo tty
          lodsb                           ; al = caracter a escribir
          int   10h                       ; llama al BIOS
          loop  dmove10                   ; y repite ciclo hasta terminar
          or    di, di                    ; p mensaje secundario?
          jz    dmove20                   ; : no - continuar
          mov   ds, dx                    ; ds:0 = texto
          xor   si, si                    ; si = inicio de buffer
          xor   di, di                    ; di = 0 limpiar bandera
          lodsw                             ; ax = long de mensaje
          cmp   ax, 54                    ; p mayor que 54?
          jna   dmove15                   ; : no - deja tal como esta
          mov   ax, 54                    ; : maximo es 54
dmove15:  pop   cx                        ; restablece cx base
          add   cx, ax                    ; cx = longitud total
          push  cx                        ; salva longitud completa
          mov   cx, ax                    ; establece para ciclo
          jmp   dmove10                   ; regresa y envia mensaje
dmove20:  pop   ax                        ; ax = ultima cuenta de mensaje
          mov   cx, 63                    ; cx = long maxima
          sub   cx, ax                    ; p cx = long a borrar?
          jle   dmove40                   ; : no - no hacer nada
          mov   ax, 0e20h                 ; ax = escribe blancos
dmove30:  int   10h                       ; llama BIOS para cada caracter
          loop  dmove30                   ; y regresa a ciclo hasta
                                          ; finalizar
dmove40:  pop   di                        ; restablece registros
          pop   bx
          ret                             ; luego regresa a llamador
dmove     endp
```

```
; Esta rutina maneja los comandos de consola
;
; Regresa
;         carry    peticion de salir de TSR


dcmds     proc
          mov   ah, 1                     ; ah = lee estado de teclado
          int   16h                       ; p alguna tecla en buffer?
          jnz   dcmds10                   ; : si - continuar
          clc                             ; limpia carry
          ret                             ; y regresa al llamador
dcmds10:  xor   ah, ah                    ; ah = leer tecla de buffer
          int   16h                       ; obtiene tecla
          cmp   al, 1bh                   ; p Esc?
          jne   dcmds20                   ; : no - continuar
          stc                             ; coloca carry
          ret                             ; y regresa a llamador
dcmds20:  or    al, al                    ; p tecla Alt?
          jnz   dcmds22                   ; : no - continuar
          mov   dx, 2                     ; dx = lpt3
          jmp   short dcmds30             ; : a codigo comun
dcmds22:  cmp   al, ...                   ; p tecla Ctrl?
          jn    dcmds24                   ; : no - no alterar
          mov   dx, 1                     ; dx = lpt2
          jmp   short dcmds30             ; : a codigo comun
dcmds24:  xor   dx, dx                    ; dx = lpt1
dcmds30:  push  ax                        ; salva registro
          mov   al, lplen                 ; al = long de bloque de impr
          mul   dl                        ; ax = offset de impr solicitada
          add   ax, offset ablpt          ; ax = bloque de impr solicitado
          mov   bx, ax                    ; bx = bloque lpt
          pop   ax                        ; restablece registros
          cmp   [bx].lpstatus, 0ffh       ; p impresora no instalada?
          je    dcmds90                   ; : si - error
          cmp   ah, 20h                   ; p desactivar impresora?
          jne   dcmds40                   ; : no - intenta sig comando
          mov   [bx].lpstatus, 0ffh       ; coloca edo. a no instalada
          mov   [bx].lpcontrol, 0ffh
          clc                             ; limpia carry
          ret                             ; y regresa a llamador
dcmds40:  cmp   ah, 22h                   ; p formfeed a impresora?
          jne   dcmds45                   ; : no - sig comando
          mov   ax, 000ch                 ; ax = escribe
          jmp   short dcmds95             ; : salir
dcmds45:  cmp   ah, 24h                   ; p salta pob?
          jne   dcmds50                   ; : no - sig comando
          mov   ah, 0a5h                  ; ah = func salta pob
          int   17h                       ; llama BIOS
          jmp   short dcmds95             ; luego libera impresora
dcmds50:  cmp   ah, 19h                   ; p pausa a impresora?
          jne   dcmds55                   ; : no - sig comando
          cmp   [bx].lpstatus, ?          ; p pausa o continuacion?
```

```
         jz      dcmdn55              ; si, [illegible]
         dec     [bx].[illegible], 2  ; cambia a impresora en [illegible]
         mov     [bx].[illegible], 0
dcmdn55: clc                          ; limpiar carry
         ret                          ; y regresa a llamador
dcmdn53: cmp     ah, 13h              ; ¿reset a impresora?
         jne     dcmdn60              ; no, sig. comando
         mov     ah, 1                ; ah = inicializar impresora
         jmp     short dcmdn95        ; salir
dcmdn60: cmp     ah, 2ch              ; ¿cancela a impresora?
         jne     dcmdn65              ; no, sig. comando
         mov     ah, 0c5h             ; ah = func. de cancela impr.
         int     17h                  ; llama BIOS
         jmp     short dcmdn68        ; luego libera impresora
dcmdn65: cmp     ah, 23h              ; ¿"go" a impresora?
         jne     dcmdn90              ; no, sig. comando
dcmdn68: mov     [bx].[illegible], 0  ; hacer que [illegible] en cola
         mov     [bx].[illegible], 0  ; [illegible] impresora
         clc                          ; limpia bandera de carry
         ret                          ; y regresa al llamador
dcmdn90: mov     ax, 0e07h            ; ax = envía beep
         int     10h                  ; invoca BIOS
         stc                          ; limpia carry
         ret                          ; y regresa
dcmdn95: int     17h                  ; llama BIOS para func. de impr.
         clc                          ; limpiar carry
         ret                          ; y regresa
dcmds    endp

; Esta rutina formatea un entero de doble palabra en un campo de salida
; de 4 caracteres
;
; Entra
;        dx:ax   número a convertir
;        di      lsb del campo de salida
;
; Resultado
;        El campo de salida es formateado conforme a las sig. reglas
;
;          0 a 10000        nnnn
;          10001 a 999k     nnnK
;          1m a             nnnM

dformat  proc
         push    bx                   ; salva registro
         mov     cx, 4                ; cx = contador de caracts.
         or      dx, dx               ; ¿msh = 0?
         jnz     dform10              ; no, no es de este grado
         cmp     ax, 10000            ; ¿lsh 10000?
         jb      dform50              ; sí, no dividir
dform10: cmp     dx, 15               ; ¿procesa msh 15h?
         ja      dform30              ; no, debe estar en millones
         jl      dform20              ; sí, en los miles
         cmp     ax, 16960            ; ¿procesa lsh 10h?
         jnz     dform30              ; sí, establece divisor
dform20: mov     bx, 1000             ; es bx = divisor 1k
         xor     cx, cx
         mov     byte ptr [di], 'K'   ; coloca en 'K'
         jmp     short dform40        ; y continúa el código común
dform30: mov     bx, 16960            ; es bx = divisor 1m
         mov     cx, 15
         mov     byte ptr [di], 'M'   ; coloca en formato 'M'
dform40: call    ddiv                 ; división de 32 bits
         dec     di                   ; di: lsb del número
         mov     cx, 3                ; cx = cont. de cargas de ciclo
dform50: mov     bx, 10               ; constante para llamar
dform60: div     bx                   ; dx = dígito
         add     dx, '0'              ; dl = dígito ascii
         mov     [di], dl             ; almacenar en área de salida
         xor     dl, dl               ; dl = 0 para resto de divisiones
         dec     di                   ; di: sig. localidad
         or      ax, ax               ; ¿más dígitos significativos?
         jz      dform70              ; no, insertar blancos
         loop    dform60              ; sino, repetir ciclo hasta
                                      ; terminar
dform70: dec     cx                   ; ¿blancos requeridos?
         jl      dform90              ; no, todo concluido
         mov     dl, ' '              ; dl = blanco para suprimir
                                      ; ceros
dform80: mov     [di], dl             ; almacenar en área de salida
         dec     di                   ; di: sig. localidad
         loop    dform80              ; ciclo de blancos al inicio de
                                      ; string
dform90: pop     bx                   ; restablece registros
         ret                          ; regresa al llamador
dformat  endp

; Esta rutina convierte un número binario en dos caracteres ascii
; imprimibles
;
; Entra
;        dx      número a convertir
;        di      lsb de salida

dtoa     proc
         push    ax                   ; salva regs.
         mov     ax, dx               ; ax = número
         aam                          ; convierte a 2 núms. empacados
         or      ax, 3030h            ; ax = caracts. ascii
         xchg    ah, al               ; intercambia para almacenar
         sub     di, 3                ; di: sig. los menos signif.
         mov     word ptr [di + 2], ax ; salva caracts. ascii
         pop     ax                   ; restablece regs.
         ret                          ; y regresa a llamador
```

```
dtime       endp

; Esta rutina determina el porcentaje de uso del área de cola
;
; Entra
;           bx  bloque de control de impresion
;           si  linea de mensajes
; ----

dgauge      proc
            push  bx                        ; salva regs
            mov   ax,word ptr [bx].tpnque   ; obtiene caracs en cola
            mov   dx,word ptr [bx].tpnque+2 ; dx:ax = en spool
            mov   cx,100                    ; cx = multiplicador
            call  mmul                      ; dx:ax * 100
            push  ax                        ; salva producto
            push  dx
            mov   ax,nctlqrecs              ; ax = # de regs en cola
            xor   dx,dx                     ; dx:ax = # de regs
            mov   cx,qrlen                  ; cx = tamaño de reg de cola
            mul   cx                        ; dx:ax = espacio aprox. usado
                                            ;   en reg. de cola
            add   ax,lpt_dqsz               ; suma tamaño de pedido
            adc   dx,0                      ; dx:ax = bytes disponibles
            mov   cx,dx                     ; establece para division 32/32
            mov   bx,ax                     ; cx:bx = divisor
            pop   dx                        ; obtiene dividendo
            pop   ax
            call  ddiv                      ; divide para obtener % de uso
            cmp   ax,100                    ; ¿mayor que 100%?
            jna   dgauge10                  ; no: continuar
            mov   ax,100                    ; ax = maximo 100%
dgauge10:   push  ax                        ; salva reg
            mov   di,si                     ; di: string de salida
            add   di,4                      ; di: campo de salida
            call  dformat                   ; formatea el porcentaje
            pop   ax                        ; restablece reg
            push  cx                        ; salva reg
            push  ds                        ; establece dir. de segmento
            pop   es                        ;   a nuestro string
            mov   di,si                     ; di: string de salida
            add   di,7                      ; di: campo de salida
            xor   dx,dx                     ; dx = 0 para dividir
            mov   bx,10                     ; bl = divisor para % uso
            div   bx                        ; ax = cociente, dx = residuo
            push  ax                        ; salva # de bloques completos
            xor   cx,cx                     ; asume no grafica impresion
            or    ax,dx                     ; ¿caracteres requeridos?
            pop   ax                        ;   restablece cuenta
            jz    dgauge40                  ; no: borrarlo
            push  ax                        ;   salva de nuevo
            or    ax,ax                     ; ¿más bloques completos?
            jz    dgauge30                  ; no: intenta partes
            mov   cx,ax                     ; cx = contador de ciclo
            mov   al,219                    ; al = carac. de bloque completo
dgauge20:   cld                             ; asegura direcc. hacia adelante
  rep stosb                                 ; coloca bloques completos
dgauge30:   pop   cx                        ; restablece cuenta completamente
            or    dx,dx                     ; ¿bloques parciales?
            jz    dgauge40                  ; no: borra resto
            cmp   dx,5                      ; ¿parcial?
            jbe   dgauge35                  ; si: establece para uno
            mov   byte ptr [di],219         ; sino: otro completo
            jmp   short dgauge37            ; envia caracter
dgauge35:   mov   byte ptr [di],221         ; envia caracter parcial
dgauge37:   inc   di                        ; di: sig. posicion de salida
            inc   cx                        ; cx = long. corriente
dgauge40:   neg   cx                        ; cx = menos long. maxima
            add   cx,10                     ; cx = # de blancos
            mov   al,' '                    ; al = caracter de blanqueo
  repne stosb                               ; blanquea resto de linea
            pop   cx                        ; restablece regs
            pop   bx
            ret                             ; regresa al llamador
dgauge      endp

; Esta rutina hace una division de 32 bit sin signo
;
; Entra
;           dx:ax = dividendo
;           cx:bx = divisor
;
; Sale
;           dx:ax = cociente
; ----

ddiv        proc
            push  cx                        ; area de trabajo   [bp+8]
            push  bx                        ;   de pila         [bp+6]
            push  dx                        ;                   [bp+4]
            push  ax                        ;                   [bp+2]
            push  bp                        ; salva regs
            mov   bp,sp                     ; establece apuntador a argumentos
            push  si
            or    bx,cx                     ; ¿dividendo entre cero?
            jnz   ddiv2                     ; no: continuar
            xor   ax,ax                     ; sale: regresa cero
            xor   dx,dx
            jmp   short ddiv?               ; regresa a llamador
ddiv2:      mov   ax,cx                     ; ax = divisor msb
            or    ax,ax                     ; ¿32bit/16bit?
            jnz   ddiv3                     ; no: 32bit/32bit
                                            ; divide 32bit/16bit
            mov   cx,[bp+6]                 ; cx = divisor lsb
            mov   ax,[bp+4]                 ; ax = dividendo msb
            xor   dx,dx                     ; dx = 0
```

```
        div     cx              ; dx:ax = msh : lsh
        mov     bx, ax          ; bx = resultado temporal
        mov     ax, [bp + 2]    ; ax = dividendo lsb
        div     cx              ; dx:ax = resultado temp
        mov     dx, bx          ; dx = resultado msh temporal
        jmp     ddiv7           ; regresar
                                ; dividir 32bit / 32bit
ddiv3   mov     bx, ax          ; bx = divisor msh
        mov     cx, [bp + 6]    ; cx = divisor lsb
        mov     dx, [bp + 4]    ; dx = dividendo msh
        mov     ax, [bp + 2]    ; ax = dividendo lsb
ddiv4   shr     bx, 1           ; corrimientos de reducción
        rcr     cx, 1           ; de ambos operandos de 32bit
        shr     dx, 1
        rcr     ax, 1
        or              bx, bx  ; ¿ reducción terminada?
        jnz     ddiv4           ; no, regresa a ciclo
        div     cx              ; dx:ax = 1er resultado
        mov     si, ax          ; si = salva temporalmente
        mul     word ptr [bp + 8]
        xchg    cx, ax          ; intercambia valores temporales
        mov     ax, [bp + 6]    ; ax = divisor lsb
        mul     si
        add     dx, cx
        jb              ddiv5
        cmp     dx, [bp + 4]
        ja              ddiv5
        jb              ddiv6
        cmp     ax, [bp + 2]    ; ¿ resultado lsb original?
        jbe     ddiv6           ; si, salir
ddiv5   dec     si              ; dx
ddiv6   xor     dx, dx          ; dx = 0
        mov     ax, si          ; ax = resultado
ddiv7   pop     si              ; restablece registros
        pop     bp
        add     sp, 8           ; ajustar pila de área de trabajo
        ret                     ; y regresa al llamador
ddiv    endp

; Esta rutina hace una multiplicación de 32bit by 16bit
;
; Entra
;       dx:ax   multiplicando
;          cx   multiplicador
;
; Sale
;       dx:ax   producto

mmul    proc
        push    si              ; salva regs
        push    di
        mov     di, dx          ; salva valor superior
        mul     cx              ; multiplicar el más bajo
        push    ax              ; salva producto inferior
        mov     si, dx          ; salva producto superior
        mov     ax, di          ; ax = valor superior
        mul     cx              ; multiplica superior
        mov     dx, ax          ; dx = producto parcial
        add     dx, si          ; dx = producto supr completo
        pop     ax              ; dx:ax = producto 32 bit
        pop     di              ; restablece regs
        pop     si
        ret
mmul    endp

;
; qb   manejador de cola, rutina de interfaz de cola no dependiente del
;           del dispositivo
;
; Lectura de un registro de cola
;
; Esta función es usada para obtener el buffer de la cola de impresora.
; Las colas de impresora son colas fifo
;
; Al regreso, la bandera de carry indica una de las situaciones si-
; guientes
;           No existen más registros en la cola de impresora especificada
;           Se proporcionó un número de cola no válido
;
; Cuando un buffer expansible (disco, EMS) es usado y el área de regis-
; tros libres está vacía, uno o más registros nuevos serán colocados al
; final de la cola. La lectura de un registro automáticamente es colo-
; cada en el área de registros libres de la cola
;
; Entra ah = 0
;           al    cola a leer (0,1,2    lpt1,2,3)
;           ds:dx  buffer
;
; Sale  ax = # de registro de cola 1:256
;           carry = error
;           al    código de salida, 00 = Todo bien
;                                   01   número de cola inválido
;                                   02   no hay datos en la cola
;                                   03   no posible leer en este momento
;
;
; Escritura de un registro de cola
;
; Esta función es usada para añadir un registro de cola a la cola de
; la impresora. Esta función es necesaria para
;     conseguir un registro libre de la cola
;     (lo asigna si es necesario)
;           escribir el nuevo registro
;           actualizar los cadenas o ligas
```

```
; Al regreso la bandera carry indica cuando se produce algun error
; La función puede fallar por alguna de las siguientes causas
;         no hay más espacio en la cola
;         imposible escribir registro de cola (DOS en uso)
;         número de cola inválido
;
; Entra ah = 1
;         al = cola a escribir (0,1,2 = lpt1,2,3)
;         ds:dx = buffer
;
; Sale carry = error
;         al = código de salida, 00 = todo bien
;                                 01 = número de cola inválido
;                                 02 = espacio agotado
;                                 03 = imposible escribir en este
;                                      momento
;
; ------------------------------------------

qh      proc
        push  bx                    ; salva regs
        push  cx
        push  dx
        inc   qinusefg              ; indica qh en uso
        cmp   al,2                  ; ¿# de cola válido?
        jna   qh00                  ; sí, continuar
        mov   al,1                  ; al = error cola inválida
        jmp   short qh92            ; regresa c/error
qh00:   or    ah,ah                 ; ¿leer?
        jnz   qh01                  ; no, saltar llamada
        call  qhr                   ; sino, leer un registro
        jmp   short qh95            ; y regresar a llamador
qh01:   dec   ah                    ; ¿escribir?
        jnz   qh90                  ; no, mala función
        call  qhw                   ; sino, escribe registro
        jmp   short qh95            ; y regresa al llamador
qh90:   mov   al,0ffh               ; al = petición de func. errónea
qh92:   stc                         ; carry encendido
qh95:   dec   qinusefg              ; permite uso de rutina de nuevo
        pop   dx                    ; restablece regs
        pop   cx
        pop   bx
        ret                         ; regreso al llamador
qh      endp                        ; fin de manejador de cola ppal

; leer un reg. de cola
; ------------------

qhr     proc
        mov   bl,lpilen             ; bl = long. de estructura lp
        mul   bl                    ; ax = offset de buffer
                                    ;   corriente de impresora
        mov   bx,ax                 ; bx = cola solicitada
        add   bx,offset qhlpts      ; bx = cola solicitada
        cmp   [bx].lpfirstqr,0      ; ¿regs en cola?
        jnz   qhr05                 ; sí, obtén 1er registro
        mov   al,2                  ; al = no hay datos en cola
        jmp   short qhr90           ; regresa c/error
qhr05:  mov   cx,[bx].lpfirstqr     ; cx = sig. reg. de cola
        mov   ax,3                  ; ax = leer reg. de cola
        call  dword ptr qrtn        ; ¿lo logró?
        jnc   qhr10                 ; sí, actualiza colas
        mov   al,3                  ; al = no puede leer ahora
        jmp   short qhr90           ; regresar c/error
qhr10:  mov   si,dx                 ; si = reg. nuevamente leído
        mov   ax,[si].qrnextqr      ; ax = sig. # de reg. en cola lp
        mov   [bx].lpfirstqr,ax     ; quita reg. de cola lp
        or    ax,ax                 ; ¿fin de cola?
        jnz   qhr20                 ; no, añadir a área
                                    ;   de regs. disponibles
        mov   [bx].lplastqr,ax      ; sino, libera al último
                                    ; registro lpt
qhr20:  mov   ax,qhfirst            ; ax = 1er reg. libre
        mov   qhfirst,cx            ; 1er libre = reg. leído
        mov   [si].qrnextqr,ax      ; reg. de cola = nuevo sig.
        mov   ax,4                  ; ax = reescribe apuntador
                                    ;   como disponible
        call  dword ptr qrtn        ; escribe a la cola
        xor   al,al                 ; al = todo está bien
        clc                         ; no carry
        ret                         ; regreso al llamador
qhr90:  stc                         ; indica error
        ret                         ; regreso al llamador
qhr     endp

; ------------------
; escribir un reg. de cola
; ------------------

qhw     proc                        ; escribe un reg. de cola
        push  dx                    ; salva aptr. a buffer
        mov   bl,lpilen             ; bl = long. de estructura lp
        mul   bl                    ; ax = offset de buffer de
                                    ;   impresora corriente
        mov   bx,ax                 ; bx = cola solicitada
        add   bx,offset qhlpts      ; bx = cola solicitada
        mov   cx,qhfirst            ; ¿registros libres?
        jcxz  qhw05                 ; no, agrega uno nuevo
        mov   dx,offset qhfirst     ; dx = nuevo 1er disponible
        mov   ax,3                  ; ax = lee aptr. de 1er disp.
        call  dword ptr qrtn        ; ¿pudo leer apuntador?
        jnc   qhw20                 ; sí, continuar
        mov   al,3                  ; al = no puede ahora
        mov   qhfirst,cx            ; coloca reg. en área de
                                    ; regs. disponibles
        pop   dx                    ; restablece aptr. a buffer
        jmp   short qhw90           ; y regreso c/error
```

```
ghw15     mov   ax,3                 ; ax = obtiene nuevo registro
          call  dword ptr grtn       ; ¿hay reg. disponible?
          jnc   ghw20                ; si, continuar
          mov   al,3                 ; al = no puede escribir ahora
          pop   ds                   ; restablece sgmt. de buffer
          jmp   short ghw90          ; regresa c/error
ghw20     pop   si                   ; si = reg. ... # de reg
          mov   dx,si                ; dx = nuevo registro
          mov   [si].qrecnxt,0       ; coloca apt. a último reg
          mov   ax,2                 ; ax = escribe nuevo reg
          call  dword ptr grtn       ; ¿pudo hacerlo?
          jnc   ghw25                ; si, continuar
          mov   al,3                 ; al = no puede escribir ahora
          jmp   short ghw90          ; regresa c/error
ghw25     cmp   [bx].lpfirstqr,0     ; ¿cola vacía?
          je         ghw30           ; si, hacer cero y último
          xchg  cx,[bx].lplastqr     ; cx = último en cola
          mov   ax,4                 ; ax = escribe apt. únicamente
          lea   dx,[bx].lplastqr     ; dx = nuevo # de último reg
          call  dword ptr grtn       ; actualiza reg. previo y
                                     ; último
          jmp   short ghw80          ; regresa sin...
ghw30     mov   [bx].lpfirstqr,cx    ; coloca como 1er registro
          mov   [bx].lplastqr,cx     ; y último registro
ghw80     xor   al,al                ; al = no problema
          clc                        ; no carry
          ret                        ; regresa a llamador
ghw90     stc                        ; carry = problema
          ret                        ; regresa al llamador
ghw       endp

; ---------------------------------------------------------------
; Intercambia despliegue con buffer de memoria.
;         Regresa carry si modo de video incorrecto
; ---------------------------------------------------------------

swapdisp  proc
          push  bp                   ; salva regs
          mov   ah,0fh               ; obtiene modo de despliegue
          int   10h                  ; invoca BIOS
          cmp   al,2                 ; ¿modo blanco y negro?
          je    swpdsp10             ; si, manéjalo así
          cmp   al,3                 ; ¿modo color?
          je    swpdsp10             ; si, manéjalo así
          cmp   al,7                 ; ¿modo monocromático?
          je    swpdsp10             ; si, manéjalo así
          stc                        ; indica error
          jmp   short swpdsp90       ; regresa ahora
swpdsp10  mov   si,offset scrbuf     ; si = buffer de pantalla
          mov   di,lines             ; di = renglones a intercambiar
          xor   dh,dh                ; dh = inicia en línea cero
swpdsp15  mov   cx,cols              ; columnas a intercambiar
          xor   dl,dl                ; dl = inicia en pos. cero
swpdsp20  mov   ah,2                 ; ah = coloca pos. de cursor
          int   10h                  ; coloca cursor
          mov   ah,8                 ; ah = lee carac. & atributo
          int   10h                  ; invoca BIOS
          xchg  ax,[si]              ; intercambia con buffer
          inc   si                   ; si = siguiente
          inc   si                   ; carácter
          push  cx                   ; salva contador
          mov   bl,ah                ; bl = atributo a escribir
          mov   ah,9                 ; ah = escribe carácter & cursor
          mov   cx,1                 ; cx = # a escribir
          int   10h                  ; escribe carácter
          pop   cx                   ; restablece contador
          inc   dl                   ; dl = sig. columna
          loop  swpdsp20             ; repite ciclo hasta
                                     ; terminar la línea
          inc   dh                   ; dh = sig. línea
          dec   di                   ; ¿terminó todas las líneas?
          jnz   swpdsp15             ; no, sig. línea
          clc                        ; limpia bandera de error
swpdsp90  pop   bp                   ; restablece regs
          ret                        ; regreso al llamador
swapdisp  endp

; Rutina de inicio: proceso de arranque de spool
;
; Esta rutina realiza las siguientes funciones:
;           Invoca a CMD para ejecutar proceso inicial
;           Coloca a grtn como la rutina manejadora de cola
;           Coloca la dirección de grtn final
;           Calcula los párrafos a retener en memoria (incluyendo Memoria
;           Convencional si es necesario)
;           Coloca todos los vectores de interrupción
;           Se hace TSR
;

start     proc
          push  ds                   ; salvar ds
          call  cmd                  ; inicializa al sistema
  rep movsb                          ; coloca rtn manejadora
                                     ; de cola
          pop   ds                   ; restablece ds
          mov   qrtnseg,ds           ; actualiza nuevo manejador
          mov   dx,dataend           ; dx = bytes usados
          add   dx,15                ; sig. párrafo si se requiere
          mov   cl,4                 ; cl = contador de corrimiento
          shr   dx,cl                ; dx = contador de párrafos
          test  pcflg,pcflgm         ; ¿spool convencional?
          jz         start50         ; no, coloca área adicional
          mov   ax,es                ; ax = 1er párrafo
          add   ax,dx                ; ax = párrafo de cola en mem
          mov   memqseg,ax           ; salva segmento de área conv
          mov   ax,memqsize          ; ax = tamaño de mem. conv
          mov   bl,qrlen/16          ; bl = párrafos reg. de cola
          mul   bl                   ; ax = párrafos por cola
          add   dx,ax                ; dx = párrafos a retener
```

```
start0  push  ds                      salva la memoria a conservar
        mov   cx,chlen                cx = long. a limpiar
        mov   di,start              di = donde se limpiará
        mov   al,0                    al = valor de memoria inicial
        rep stosb                     limpia áreas asignadas
        mov   si,[illegible]          si = 1er bloque de interrup
        mov   cx,[illegible]          cx = # de ints. a manejar
start0  mov   ah,35h                  ah = obten vector de interrup
        mov   al,[si+2]               al = # de interrupción
        int   21h                     obtiene cola actual corriente
        mov   [si+2],es               salva segmento
        mov   [si],bx                 y offset de int. anterior
        mov   dx,[si+13]              dx = nueva interrupción
        mov   ah,25h                  ah = coloca vector de int
        int   21h                     establece nuevo vector
        add   si,[illegible]          si = siguiente entrada
        loop  start0                  coloca a sig. interrup
        dec   i21hflg                 indica que terminó de usar
                                      INT 21h
        pop   ds                      restablece memoria a conservar
        mov   ax,3100h                ax = TSR
                                      código de regreso = 0
        int   21h                     se hace residente
start   endp
        align 16                      alinea un párrafo
```

; Buffer de pantalla. (Pantalla de control de spool Pop up)

```
scrbuf  db                            [illegible] LPT    # Uso    CPS  CT  CTC
        db                            Tiempo  [illegible] Detectivo P Pausa  G [illegible]
        db                            [illegible]
        db                            [illegible] Formfeed R Reset  Esc
        db                            [illegible]
        db                            [illegible] Saltar [illegible] Cancela
        db                            [illegible]
        db                            [illegible]
        db
        db
scrbufcon equ  $ [illegible]
```

; Nota. Todo el código superior a esta parte será sobreescrito después
; de la instalación

```
scrbufend equ  scrbufcon [illegible] 400

qhandler  equ  scrbuf [illegible]
```

; El manejador de cola será trasladado a esta dirección después de la
; instalación

; Área del código de datos de instalación - disponible durante la
; instalación únicamente

```
hdrmsg    db    "PCSPOOL V1.0 (c) 1991. Spooler de Impresión ",13,10
          db    "TENIX II [illegible]" Dulce M. Mayo y Oscar A. Olivo"
          db    13,10
dollar    db    '$'
help      db    10
          db    "Uso: ",13,10,10
          db    "  PCSPOOL [illegible] [illegible]",13,10
          db    "  PCSPOOL [illegible] comentario  ]",13,10
          db    "  PCSPOOL [illegible]",13,10
          db    "  PCSPOOL [illegible]",13,10,"$"
invreq    db    10
          db    "Petición inválida: "
invreqd   db    " ",13,10,"$"
invopnd   db    10
          db    "Operando inválido: "
invopndd  db    " ",13,10,"$"
cantmq    db    10
          db    "Incapaz de manejar cola ",13
notup     db    10
          db    "Spooler no instalado ",13,10,"$"
upalrdy   db    10
          db    "Spooler ya instalado ",13,10,"$"
notspld   db    10
          db    "Impresora solicitada no spooleada ",13,10,"$"
notmsq    db    10
          db    "Esta versión no trabaja con multitasking "
          db    13,10,"$"
splalrdy  db    10
          db    "Tú sólo puedes especificar 1 tipo de spool ",13,10,"$"
invmem    db    10
          db    "El tamaño de memoria especificado no es válido "
          db    "Debe ser de [illegible]",13,10,"$"
baddrive  db    10
          db    "El drive especificado no es válido ",13,10,"$"
cantfree  db    10
          db    "Imposible desinstalar en éste momento ",13,10,"$"
freeok    db    10
          db    "Desinstalación Exitosa ",13,10,"$"
pausecl   db    10
          db    "Emisión de pausa exitosa ",13,10,"$"
waitmsg   db    10
          db    "En espera de terminar impresión. ESC cancela desinstalación "
          db    13,10
          db    "Pulse cualquier otra tecla para desinstalación inmediata "
          db    13,10,"$"
uninscan  db    10
          db    "Desinstalación cancelada. PCSPOOL todavía residente "
```

```
            db      13,10,'$'
ffmsh       db      10
            db      'Formfeed enviado a impresora ' 13,10,'$'
qfilnam     db      'PCSPOOL.QUE',0           ; nombre de archivo de cola
char C      db      'C'                       ; default mem. convencional

;------------------------------------------------------------
;
; Procedimiento de inicialización,      cs, ds, es  nuestro segmento
;
;                       Regresa  de si  rutina de cola
;                                  es:di  nueva dirección
;                                    cx  = long. a trasladar
;                                    di  = segmento de nueva dir
;
; 1. Explorar en busca de un comando
;        si hay error, envía mensaje de error & sale
;
; 2. Invoca al procesador de comando apropiado
;
; 3. Establece registros para el traslado de la rutina manejadora
;    de cola
;
;------------------------------------------------------------

cmd         proc                              ; encuentra/procesa comando
            push    es                        ; salva regs
            mov     dx, offset hdrmsg         ; ds:dx  mensaje de encabezado
            mov     ah, 9                     ; ah = imprime string ASCII$
            int     21h                       ; despliega encabezado
            mov     cx, 'DE'          ; cx = DE de DESQ
            mov     dx, 'SQ'          ; dx = SQ de DESQ
            mov     ax, 2b01h                 ; ax = coloca fecha inválida
            int     21h                       ; deja que el DOS lo ataque
            cmp     al, 0ffh                  ; ¿desqview activo?
            je      cmd0                      ; no, continúa
            mov     dx, offset nodesq         ; dx  no compatible con
                                              ;   desqview
            xor     al, al                    ; al   no ayuda necesaria
            call    die
cmd0        mov     ah, 0c4h                  ; ah   preguntar spooler activo
            int     21h                       ; ¿spooler corriendo?
            mov     bx, 81h                   ; bx  línea de comandos
            call    nxtop                     ; ¿hay operandos?
            jnc     cmd00                     ; sí, procésalo
            mov     dx, offset dullar         ; dx  mensaje nulo
            mov     al, 1                     ; al = dar ayuda
            call    die                       ; termina c/ayuda
cmd00       cmp     al, 'I'                   ; ¿petición de inicialización?
            jnz     cmd10                     ; no, verifica sig.
            call    init                      ; sino, invoca código de
                                              ;   inicialización
            jmp     short cmd90               ; regresa al llamador
cmd10       cmp     al, 'P'           ; ¿petición de pausa?
            jnz     cmd20                     ; no, verifica sig.
            jmp     pause                     ; sino, invoca rtn. de pausa
cmd20       cmp     al, 'U'                   ; ¿desinstalar?
            jnz     cmd30                     ; no, verifica sig.
            jmp     uninstall                 ; sino, invoca desinstalar
cmd30       cmp     al, 'F'                   ; ¿formfeed?
            jnz     cmd40                     ; no, error
            jmp     formfeed                  ; sino, envía un ff
cmd40       mov     invreqd, al               ; salva código de error
            mov     dx, offset invreq         ; ds:dx  mensaje de error
            mov     al, 1                     ; al = dar ayuda
            call    die                       ; envía mensaje y termina
cmd90       pop     es                        ; restablece regs
            les     di, qhandler              ; di  rtn. manejadora de cola
            mov     dx, di                    ; dx = offset también
            mov     cl, 4                     ; cl = cantidad de corrimiento
            shr     dx, cl                    ; dx = segmento offset
            mov     ax, cs                    ; ax = segmento del programa
            add     dx, ax                    ; dx = segmento de rtn.
                                              ;   manejadora de cola
            sub     si, si                    ; si = 0 offset
            mov     cx, qrtnlen               ; cx = tamaño de rutina
            cld                               ; dirección = hacia arriba
            mov     ds, qrtnseg               ; ds = seg. corriente de rtn.
            ret                               ; regreso al llamador
cmd         endp                              ; fin de rutina de exploración

;------------------------------------------------------------
; nxtop  encuentra el siguiente operando
; bx -> posición actual
; Regresa
;         bx    -> 1er carácter (posterior al slash) del siguiente operando
;         al    = 1er letra (en mayúscula) del operando
;         carry = fin de línea
;------------------------------------------------------------

nxtop       proc
            cmp     byte ptr [bx], 0dh        ; ¿fin de línea?
            je      nxtop80                   ; sí, no buscar más
            cmp     byte ptr [bx], '/'        ; ¿slash?
            je      nxtop50                   ; sí, regresa sig. carác.
            cmp     byte ptr [bx], ' '        ; ¿espacio en blanco?
            jna     nxtop20                   ; sí, salta blanco
; salta espacio no blanco
nxtop10     inc     bx                        ; bx  sig. carácter
            cmp     byte ptr [bx], 0dh        ; ¿fin de línea?
            je      nxtop80                   ; sí, no buscar más
            cmp     byte ptr [bx], '/'        ; ¿slash?
            je      nxtop50                   ; sí, regresa sig. carác.
            cmp     byte ptr [bx], ' '        ; ¿espacio blanco?
            ja      nxtop10                   ; no, salta
nxtop20     inc     bx                        ; bx  sig. carácter
            cmp     byte ptr [bx], 0dh        ; ¿fin de línea?
            je      nxtop80                   ; sí, no buscar más
```

```
        cmp   byte ptr [bx],'/'         ; ¿ slash?
        je          nstop30             ; si: regresa sig. carac.
        cmp   byte ptr [bx],' '         ; ¿ blanco?
        jne   nstop30                   ; si: obtén sig. carac.
        jmp   short nstop40             ; sino: regresa carácter
nstop30 inc   bx                        ; bx: byte posterior a slash
nstop40 mov   al,[bx]                   ; al = byte de operando
        cmp   al,'a'                    ; ¿ carac. >= minúscula de a?
        jna   nstop90                   ; no: regresar como está
        cmp   al,'z'                    ; ¿ carac. <= minúscula de z?
        jnb   nstop90                   ; no: regresar como está
        and   al,not 20h                ; convierte a mayúscula
        jmp   short nstop90             ; regresa a llamador, ok
nstop80 stc                             ; carry: fin de línea
        ret                             ; regreso al llamador
nstop90 clc                             ; no carry: al contiene carac.
        ret                             ; regreso a llamador
nstop       endp

;-----------------------------------------------------------------
; Procesa el comando de inicialización,
;                                       bx: sig. carácter a procesar
;                                       di: 0OBFh si spooler está activo
;-----------------------------------------------------------------

init        proc
        cmp   di,0OBFh                  ; ¿ spooler ya activo?
        jne   init05                    ; no: ok inicializa
        mov   dx,offset spalrdy         ; dx: mensaje de ya cargado
        xor   al,al                     ; al: no ayuda requerida
        call  dse                       ; despliega mensaje y termina
init05  push  es                        ; salva es
        mov   es,ds:[002ch]             ; es: ambiente
        mov   ah,49h                    ; ax: libéralo
        int   21h                       ; que el DOS lo haga
        push  bx                        ; salva registro
        mov   ah,34h                    ; ah: obtén dir. de bandera
                                        ;   busy del DOS
        int   21h                       ; invoca al DOS
        mov   word ptr dos_busy,bx      ; salva offset
        mov   word ptr dos_busy+2,es    ; y segmento de bandera BUSY
        pop   bx                        ; restablece regs.
        pop   es
init10  call  nstop                     ; ¿ más operandos?
        jc    init50                    ; no: procesa operandos
        cmp   al,'C'                    ; ¿ convencional?
        jne   init15                    ; no: siguiente
        call  getmem                    ; obtén tamaño de memoria
        jmp   init10                    ; obtén sig. operando
init15  cmp   al,'D'                    ; ¿ disco?
        jne   init20                    ; no: intentar con impr.
        call  getfile                   ; crea archivo de spool
        jmp   init10                    ; ir por sig. operando
init20  mov   cl,al                     ; prepara a probar impresora
        sub   cl,'1'                    ; colocar a 0,1 ó 2
        cmp   cl,2                      ; ¿ lpt1, 2 ó 3?
        ja    init30                    ; no: error
        mov   al,1                      ; al: bit
        shl   al,cl                     ; corrimiento a posición
                                        ;   apropiada
        or    psflg,al                  ; enciende bit para impr.
        jmp   init10                    ; obtén sig. operando
init30  mov   invopnd+d,al              ; salva operando inválido
        mov   dx,offset invopnd         ; dx: mensaje de error
        mov   al,1                      ; con ayuda
        call  dse                       ; indicar operador, terminar
init40  test  psflg,psflgx              ; ¿ algún tipo de spool indicado?
        jnz   init45                    ; si: establece mem. cte.
        or          psflg,psflg.m       ; sino: establece convencional
        lea   bx,char_t                 ; bx: 't' para spool conv.
        mov   ax,sfmqsize               ; ax: bytes para cola
        shl   ax,1                      ; ax: # de regs. en cola
        mov   mqsize,ax                 ; establece máx. cuenta de regs.
init45  call  mnq                       ; inicializa rutina de cola
init50  call  setmem                    ; inicializa memoria, apuntadores
                                        ;   a buffer, etc.
        call  kvsau                     ; inicializa el buffer de
                                        ;   pantalla
        ret                             ; regreso al llamador
init        endp

;-----------------------------------------------------------------
; Procesa el comando pausa  bx: sig. carácter a procesar
;                                       di: 0OBFh si spooler está activo
;-----------------------------------------------------------------

pause       proc
        cmp   di,0OBFh                  ; ¿ spooler activo?
        je          pause05             ; si: continuar
        mov   dx,offset notinstp        ; dx: mensaje de inactivo
        call  dse
pause05 xor   dx,dx                     ; asume impresora 0
        call  nstop                     ; encuentra sig. operador
        jc          pause10             ; si fin de línea
        cmp   byte ptr [bx],'/'         ; ¿ operador?
        jne   pause10                   ; no: debe ser comentario
        sub   al,'1'                    ; al = ajustar como # de impresora
        cmp   al,2                      ; ¿ número de impresora?
        ja          pause10             ; no: parte del comentario
        mov   dl,al                     ; dx = nuevo # de impresora
        call  nstop                     ; bx: sig. operador
pause10 mov   si,bx                     ; si: comentario si existe
        xor   cx,cx                     ; cx = contador de caracteres
pause15 cmp   byte ptr [bx],0dh         ; ¿ fin de línea?
        je    pause18                   ; si: hacerlo activo
        inc   bx                        ; bx: sig. carácter
        inc   cx                        ; cx = contador de caracs.
```

```
          jmp    pause15              ; sig caracter
pause10   mov    byte ptr [bx], 0     ; hace ASCIIZ el string
          mov    ah, 0c0h             ; ah = obten bloque de ctrl
          int    17h                  ; es:bx = bloque de control
          cmp    es:[bx].status, 0ffh ; ¿spooleado?
          jb     pause20              ; si, procesa pausa
          mov    dx, offset notsplst  ; dx = mensaje no spooleado
          mov    al, 3                ; al = da ayuda
          call   dic                  ; sale y termina
pause20   mov    ah, 0c1h             ; ah = construye reg. de ctrl
          int    17h                  ; solicita al spooler lo
                                      ;   haga
          mov    dx, offset pausesok  ; ds:dx = mensaje de terminación
          xor    al, al               ; al = no ayuda
          call   dic                  ; despliega mensaje y termina
pause     endp

;
; Procesa el comando formfeed, bx = sig. caracter a procesar
;

formfeed  proc
          xor    dx, dx               ; asume impresora 0
          call   nxtop                ; encuentra sig. operador
          jc     formfd10             ; si fin de linea, envía ff
          sub    al, '1'              ; al = ajustar como # de impr
          cmp    al, 2                ; ¿número de impresora?
          ja     formfd10             ; si no, ignóralo
          mov    dl, al               ; dx = número de impresora
formfd10  mov    ax, 000ch            ; ax = escribe un formfeed
          int    17h                  ; que el spooler lo haga
          cmp    ah, 1                ; ¿impresión ok?
          je     formfd10             ; si no, intenta de nuevo
          mov    dx, offset ffok      ; ds:dx = msg. de terminación
          xor    al, al               ; al = no ayuda
          call   dic                  ; despliega msg. y termina
formfeed  endp

;
; Procesa el comando de desinstalación
;                                     bx = sig. caracter a procesar
;                                     dx = 0b0fh si spooler está activo
;                                     es = segmento de spooler activo
;

uninstall proc
          cmp    dx, 0b0fh            ; ¿spooler activo?
          je     unins05              ; si, continuar
          mov    dx, offset notsetup  ; dx = mensaje de no instalado
          jmp    dic                  ;   termina
unins05   call   needwait             ; ¿requiere esperar?
          jnc    unins09              ; si no, continuar
          lea    dx, waitmsg          ; dx = mensaje de espera
          mov    ah, 9                ; ah = imprime mensaje ASCII$
          int    21h                  ; lo imprime
unins08   call   needwait             ; ¿requiere esperar aún?
          jnc    unins09              ; si no, continuar
          mov    ah, 0bh              ; ah = verifica edo. de teclado
          int    21h                  ; invoca servicio del DOS
          cmp    al, 0ffh             ; ¿caracter esperando?
          jne    unins08              ; si no, continúa esperando
          mov    ah, 08h              ; ah = obten caracter
          int    21h                  ; al = caracter
          cmp    al, 1bh              ; ¿Esc?
          jne    unins09              ; si no, continuar
          mov    dx, offset uncancel  ; dx = mensaje de cancelación
          jmp    short unins90        ;   y termina ahora
unins09   mov    ax, 0c?h             ; ah = cierre de cola solicitado
          int    17h                  ; invoca PC SPOOL residente
          mov    ds, si               ; ds = memoria del spooler
          mov    es, si               ; es = igualmente
          mov    di, intblk           ; bx = 1er bloque de interrup
          mov    cx, intblkcnt        ; cx = # de ints usadas
unins10   mov    ah, 35h              ; ah = obten interrupción
          mov    al, [di].??          ; al = # de interrupción
          int    21h                  ; es:bx = interrupción
          mov    ax, es               ; ax = segmento de interrup
          cmp    ax, si               ; ¿en nuestro segmento?
          jne    unins80              ; si no, no se puede
                                      ;   desinstalar
          add    di, intblklen        ; di = sig. bloque
          loop   unins10              ;   verificar sig. bloque
          mov    di, intblk           ; di = 1er bloque de interrup
          mov    cx, intblkcnt        ; cx = # de ints usadas
          cli                         ; deshabilita interrupciones
unins20   lds    dx, dword ptr es:[di+?]  ; ds:dx = rutina de int. anterior
          mov    ah, 25h              ; ah = coloca interrupción
          mov    al, es:[di+??]       ; al = interrupción a colocar
          int    21h                  ;   instala la interrupción
          add    di, intblklen        ; di = sig. bloque
          loop   unins20              ;   recoloca sig. interrup
          sti                         ;   habilita interrupciones
          mov    ah, 49h              ; ah = libera memoria del spooler
          int    21h                  ;   lo hace
          mov    dx, offset freeok    ; dx = liberación muy bien
          jmp    short unins90        ;   terminar
unins80   mov    dx, offset cantfree  ; dx = mensaje de no liberación
unins90   xor    al, al               ; al = no ayuda
          push   cs                   ; restablece al nuestro
          pop    ds                   ;   propio segmento
          call   dic                  ; terminar
uninstall endp

;
; Esta rutina verifica la necesidad de esperar el cierre del spooler(?)
; Regresa
;         carry = espera requerida
```

```
prtdwait  proc
          push  cx                          ; salva regs
          push  dx
          mov   cx, 3                       ; cx = contador de ciclo
          xor   dx, dx                      ; dx = lpt1
prtdwa10: mov   ax, 0c800h                  ; ah = verifica condicion de
                                            ;      espera
          int   17h                         ; invoca al manejador de impr
          or    al, al                      ; ¿requiere esperar?
          jz    prtdwa30                    ; no: verifica sig
prtdwa20: stc                               ; espera necesaria
          jmp   short prtdwa90              ; y sale c/carry colocado
prtdwa30: inc   dx                          ; dx = sig impresora
          loop  prtdwa10                    ; repite ciclo
          clc                               ; espera no necesaria
prtdwa90: pop   dx                          ; restablece regs
          pop   cx
          ret                               ; regresa al llamador
prtdwait  endp

;
; Obtiene el tamaño de la memoria de spool requerida.
;                                  bx "C" del operando
;
;                   regresa        queues es colocada al tamaño
;                                  pcxflgsm es colocada a uno
;

getmemsz  proc
          test  pcxflg, pcxflgsx            ; ¿algun tipo de spool indicado?
          jz    getmemsz10                  ; no: establece tamaño de mem
          mov   dx, offset splstrdy         ; dx = tipo de spool seleccionado
          mov   al, 1                       ; al = dar ayuda
          call  dic                         ; terminar
getmemsz10: or  pcxflg, pcxflgsm            ; indica con mem seleccionado
          mov   ax, memqsize                ; ax = tamaño de cola default
          cmp   byte ptr [bx+2], ' '        ; ¿espacio en blanco?
          jna   getmemsz80                  ; si: usa default
          cmp   byte ptr [bx+2], '/'        ; ¿slash?
          je    getmemsz80                  ; yes: usa default
          xor   ax, ax                      ; limpia ax para val calculado
          mov   cx, [bx+2]                  ; cx = carac de tamaño de spool
          sub   cx, '00'                    ; convierte a binario
          cmp   cl, 9                       ; ¿1er valor de caracter valido?
          ja    getmemsz70                  ; no: error
          mov   al, cl                      ; al = 1er valor
          cmp   ch, 9                       ; ¿segundo digito valido?
          ja    getmemsz80                  ; no: continuar
          mov   cl, 10                      ; cl = multiplicador
          mul   cl                          ; ax = ax * 10
          mov   cl, ch                      ; establece cx = 2do digito
          xor   ch, ch                      ; ch = 0
          add   ax, cx                      ; ax = tamaño total de memoria
          cmp   ax, 64                      ; ¿valor valido?
          ja    getmemsz70                  ; no: error
          cmp   ax, 4                       ; ¿limite inferior ok?
          jnb   getmemsz80                  ; no: error
getmemsz70: mov dx, offset invmsmsg         ; dx = tamaño de mem invalido
          mov   al, 1                       ; al = dar ayuda
          call  dic                         ; terminar
getmemsz80: shl ax, 1                       ; ax = # de regs en cola
          mov   memqsize, ax                ; coloca al maximo el contador
                                            ; de registros
getmemsz90: call  initq                     ; inicializa rutina de cola
          ret                               ; y regresa al llamador
getmemsz  endp

;
; Obtiene el nombre del archivo de spool,
; bx "D" del operando
;

getfile   proc
          test  pcxflg, pcxflgsx            ; ¿algún tipo de spool indicado?
          jz    getfile10                   ; no: establece archivo
          mov   dx, offset splstrdy         ; dx = tipo de spool seleccionado
          mov   al, 1                       ; al = dar ayuda
          call  dic                         ; terminar
getfile10: or   pcxflg, pcxflgsd            ; indica spool en disco escogido
          cmp   byte ptr [bx+2], ' '        ; ¿espacio en blanco?
          jna   getfile50                   ; si: usar default
          cmp   byte ptr [bx+2], '/'        ; ¿slash?
          je    getfile50                   ; si: usar default
          lea   si, [bx+2]                  ; si = sig carac del comando
          mov   di, offset gfile            ; di = area de trabajo de archivo
          cld                               ; traslada y explorar hacia
                                            ; arriba
getfile20: movsb                            ; traslada un byte al area de
                                            ; trabajo
          cmp   byte ptr [si], ' '          ; ¿espacio en blanco?
          jna   getfile40                   ; si: fin de parametros
          cmp   byte ptr [si], '/'          ; ¿conmutador?
          jne   getfile20                   ; si: fin de parametros
getfile40: cmp  byte ptr [di-1], ':'        ; ¿ultimo caracter dos puntos?
          je    getfile45                   ; si: no se desea trayectoria
          cmp   byte ptr [di-1], '\'        ; ¿ultimo caracter contraslash?
          je    getfile45                   ; si: trayectoria correcta
          mov   byte ptr [di], '\'          ; finaliza trayectoria
                                            ; c/contradiagonal
          inc   di                          ; di = sig caracter
getfile45: mov  byte ptr [di], 0            ; coloca nulo despues de
                                            ; la trayectoria
getfile50: mov  di, offset gfile            ; di = area de trabajo de arch
          mov   cx, 65                      ; cx = longitud a explorar
          xor   al, al                      ; para hacerlo nulo
          repne scasb
          dec   di                          ; di = 1er nulo
```

```
            mov     si, offset qfilenam         ; si = nombre del arch. de cola
getfile33:  movsb                               ; traslada un caracter
            cmp     byte ptr [si-1], 0          ; ¿terminado de trasladar?
            jnz     getfile33                   ; no: continua trasladando
getfile40:  call    msq                         ; inicializa rutina de cola
            ret                                 ; regresa al llamador
getfile     endp

;--------------------------------------------------------------
; actualiza todas las direcciones de memoria
;           qrtnlen = longitud de la rutina de cola
;
;           Regresa: Apuntadores de memoria
;                    actualizados
;                    Paddles de memoria de impresora "viva"
;                    Buffers de reg. de cola de impr. "viva"
;                    Pilas de interrupciones
;                    Registro único de desencolado
;
; Nota: El buffer de memoria convencional no puede ser establecido
;     aquí debido a que esta memoria debe primero ser liberada
;          antes de que el buffer sea asignado.
;          El apuntador a éste no es asignado en esta rutina. Éste es
;          asignado cuando la rutina es trasladada al área que resi-
;          dirá
;
;          Una "impresora viva" es aquella impresora cuya dirección
;     está anotada en el vector de impresora en la dirección
;     absoluta 40:8
;--------------------------------------------------------------

setmem      proc
            push    ax                          ; salva registros
            push    bx
            push    cx
            push    dx
            push    si
            push    di
            push    es
            mov     si, nxtavail                ; si = sig. byte disponible
            add     si, qrtnlen                 ; si = byte posterior al
                                                ;   manejador
            mov     startclr, si                ; salvar inicio de asignados
            mov     bx, offset qblpt1           ; bx = 1er impresora
            mov     cx, 3                       ; máximo de impresoras
            xor     ax, ax                      ; ax = 0
            mov     es, ax                      ; es = memoria baja
setmem10:   mov     di, [bx].lpprtnbr           ; di = # de impresora
            shl     di, 1                       ; di = di * 2
            mov     dx, word ptr es:[di] + 408h ; dx = dir. de base de impr.
            or      dx, dx                      ; ¿hay impresora?
            jz      setmem15                    ; no: salir
            cmp     dx, 10fh                    ; ¿impresora local?
            jb      setmem13                    ; no: sig. impresora
            inc     dx                          ; dx = puerto de estado
            mov     [bx].lpstatport, dx         ; salvalo
            mov     dx, si                      ; dx = sig. byte disponible
            push    cx                          ; salva cx
            mov     cl, 4                       ; cx = contador de corrimiento
            shr     dx, cl                      ; dx = dx / 16
            pop     cx                          ; restablece cx
            mov     ax, ds                      ; ax = segmento del programa
            add     ax, dx                      ; ax = segmento de paddle
            mov     word ptr [bx].lppagbuf + 2, ax ; salvalo
            add     si, lpt_dest                ; si = sig. byte
            mov     [bx].lpqrecp, si            ; salva dirección de reg. de cola
            push    cx                          ; salva reg.
            mov     cx, [bx].lpprtnbr           ; cx = # de impresora
            mov     al, 1                       ; al = un bit
            shl     al, cl                      ; al = máscara de bit
            pop     cx                          ; restablece cx
            test    psflg, al                   ; ¿impresora spooleada?
            jz      setmem11                    ; no: indica no spooleada
            mov     [bx].lpstatus, 0            ; establece edo. spooleada
            jmp     short setmem12              ; y continuar
setmem11:   mov     [bx].lpstatus, 0ffh         ; estado es no spooleada
setmem12:   add     si, qrlen                   ; si = sig. byte disponible
setmem13:   add     bx, lpsize                  ; bx = sig. estructura lp
            loop    setmem10                    ; establece sig. impresora
setmem15:   mov     bx, intblk1 + 8             ; bx = 1er seg. de pila nuevo
            mov     cx, intblkcnt               ; cx = # de segs. de pila nuevos
setmem20:   mov     [bx], si                    ; salva apuntador a pila
            mov     [bx] + 2, ds                ; y segmento de pila
            add     si, intstksz                ; si = sig. área de pila
            add     bx, intblklen               ; bx = sig. acumulador
            loop    setmem20                    ; construir sig. pila
            mov     dqrec, si                   ; si = reg. de desencolado
            add     si, qrlen                   ; si = sig. byte disponible
            mov     nxtavail, si                ; salva apuntador a nuevo
                                                ;   siguiente disponible
            sub     si, startclr                ; si = longitud a limpiar
            mov     clrlen, si                  ; salvalo
            pop     es                          ; restablece regs.
            pop     di
            pop     si
            pop     dx
            pop     cx
            pop     bx
            pop     ax
            ret
setmem      endp
rsetmts     proc
rsetmts     endp

;--------------------------------------------------------------
; Inicializa el manejador de cola.
;--------------------------------------------------------------
```

```
initq       proc
            call  findrtn              ; encuentra rutina de cola
            mov   word ptr qrtn, 3     ; establece offset para llamada
            xor   ax, ax               ; ax = 0 : instala spool
            call  dword ptr qrtn       ; invoca al manejador
            jnc   initq90              ; regresa si todo estuvo bien
            mov   dx, offset cantinitq ; dx: error de inicio de cola
            xor   al, al               ; al = no ayuda
            call  die                  ; termina c/error
initq90     ret
initq       endp

;
; Encuentra el manejador de cola
;           bx = 1er caracter de opción
;           Regresa: qrtnseg : segmento de rutina
;                    qrtnlen : longitud de rutina (en bytes)
;

findrtn     proc                       ; encuentra dir. absoluta
                                       ; de rutina
            mov   al, [bx]             ; al = 1er carac. de opción
            and   al, NOT 20h          ; lo hace mayúscula
            mov   si, offset mantien   ; si: inicio de 1er rutina
                                       ; de cola
findrtn10   cmp   [si+2], al           ; ¿es esta la rutina?
            je    findrtn50            ; sí: procesa dirección
            add   si, [si]             ; si: sig. rutina
            jmp   findrtn10            ; encuentra sig. rutina
findrtn50   mov   dx, [si]             ; dx = longitud de rutina
            mov   qrtnlen, dx          ; salvala
            mov   cl, 4                ; cl = cantidad de corrimiento
            shr   si, cl               ; divide entre 16
            mov   ax, cs               ; ax = segmento corriente
            add   ax, si               ; ax = segmento de manejador
            mov   qrtnseg, ax          ; salva segmento de rutina
            ret                        ; regresa al llamador
findrtn     endp

;
; inicializa relleno de pantalla
;

scrinit     proc                       ; inicializa buffer de pantalla
            push  ax                   ; salva registros
            push  bx
            push  cx
            push  dx
            push  si
            push  di
            mov   ah, 0fh              ; obtiene modo de despliegue
            int   10h                  ; BIOS
            cmp   al, 7                ; ¿modo Color?
            jne   scrinit10            ; No: déjalo en mono
            mov   curattr, colattr     ; usa atributo de color
scrinit10   mov   di, offset scrbufend ; di: fin de buffer
            mov   si, offset scrbufcon ; si: fin de constantes
            mov   al, curattr          ; al = atributo a usar
            mov   cx, 400              ; caracs. a mover
            std                        ; dirección hacia atrás
scrinit20   stosb                      ; salva atributo
            movsb                      ; traslada datos fijos
            loop  scrinit20            ; repite hasta que todo
                                       ; haya sido trasladado
            cld                        ; restablece dirección
            pop   di                   ; restablece regs.
            pop   si
            pop   dx
            pop   cx
            pop   bx
            pop   ax
            ret                        ; regreso al llamador
scrinit     endp

;
; termina c/mensaje de ayuda, dx: mensaje intermediario
;           AL debe ser diferente de cero si se desea ayuda
;

die         proc
            push  ax                   ; salva solicitud de ayuda
            mov   ah, 9                ; ah = imprime mensaje ASCII$
            int   21h                  ; imprime mensaje de error
            pop   ax                   ; restablece petición de ayuda
            or    al, al               ; ¿ayuda deseada?
            jz    die10                ; no: salte
            mov   dx, offset help      ; dx: mensaje de ayuda
            mov   ah, 9                ; ah = imprime mensaje ASCII$
            int   21h                  ; imprime la ayuda
die10       mov   ax, 4c01h            ; termina c/ayuda aviso
            int   21h                  ; regresa al DOS
die         endp
            align 16                   ; alinea párrafo
mainlen     label byte                 ; longitud de 'main'
mainline    ends

;
; Este es el módulo de interfaz de spool. Este mantiene la no depen-
; dencia de dispositivos para el ambiente del spool
; Regreso en General
;           bandera de carry = error (CF=1)
; Inicializa Spool
;           ax = 00h código de función
; Leer Registro de Cola
;           ax = 01h código de función
;           cx = número de registro
;           dx: buffer
; Escribe Registro de Cola
```

```
;       ah = 02h, código de función
;       al = número de registro
;       dx  buffer
; Leer Apuntador en Cola
;       ah = 03h, código de función
;       al = número de registro
;       dx  buffer


; Escribir Apuntador en Cola
;       ah  04h, código de función
;       al  número de registro
;       dx  buffer


; Asigna un Registro a la Cola
;       ah = 05h, código de función


; Regresa
;       al = número de registro


; Archivo de depósito
;       ah = 06h, código de función


; Cierra Spool
;       ah = 07h, código de función


;-----

dskq    segment para public 'code'      rutinas de spool en disco
        assume cs:dskq                  invocado via llamada remota
dskhdr  dw      diskend                 longitud de módulo
        db      13                      identificador de módulo
        jmp     short dsk               a rutina de disco
dskhandle dw    0                       manejador de disco
closeq  db      0                       cierre de cola requerido
dskfnt  label   byte                    tabla de funciones
        dw      dsk0                    inicializa Spool
        dw      dsk1                    Lee Registro de Cola
        dw      dsk2                    Escribe Registro de Cola
        dw      dsk3                    Lee Apuntador en Cola
        dw      dsk4                    Escribe Apuntador en Cola
        dw      dsk5                    Asigna un Registro a la Cola
        dw      dsk6                    Archivo de depósito
        dw      dsk7                    Cierra Spool
dsk     proc    far                     escribe de spool en disco
        push    ax                      salva registros
        push    bx
        push    dx
        add     ax,3                    ax = offset de rutina
        mov     si,ax                   si =
        mov     ah,62h                  obtener PSP del usuario
        int     21h                     llama al DOS
        push    bx                      salva PSP anterior
        mov     ah,50h                  ah = cambia a PSP
        mov     bx,dx                   bx  PSP de despacho
        int     21h                     llama al DOS
        push    cx                      salva último registro
        mov     bx,cs:diskhandle        bx = manejador de archivo
        jmp     word ptr cs:dskfnt[si]  a el punto de entrada apropiado

dsk1    label   byte                    inicializa Spool
        mov     ax,3d02h                ax = código de func. abrir
        lea     dx,qfile                dx  nombre de archivo
        int     21h                     ¿ se pudo abrir?
        jnc     dsk1a0                  s  leer encabezado
dsk1a0  mov     ah,3ch                  ah = código de func. crear
        lea     dx,qfile                dx  nombre de archivo
        mov     cx,0                    cx = atributo de archivo normal
        int     21h                     intenta crear spool
        jc      dsk1e0                  termina si hay error
        mov     cs:diskhandle,ax        salva manejador de disco
        mov     bx,ax                   bx = manejador de archivo
                                        de spool
        mov     ah,40h                  ah = func. escribe en archivo
        mov     cx,qblen                cx = longitud de encabezado de
                                        cola
        mov     dx,offset qheader       dx  lear buffer
        int     21h                     invoca al DOS para escribir
                                        registro
        jc      dsk1e0                  termina si hay error
        jmp     dsk1f0                  salta depósito al archivo
dsk1e0  jmp     dsk1e                   regresa con código
dsk1f0  mov     cs:diskhandle,ax        salva manejador de disco
        mov     bx,ax                   bx = manejador de archivo
                                        de spool
        xor     cx,cx                   cx = registro 0
        call    diskfcc                 posiciona archivo para
                                        encabezado de cola
        mov     ah,3fh                  ah = func. leer archivo
        mov     cx,2                    cx = cuenta registro de cola
        mov     dx,offset qheader       dx  buffer de lectura
        int     21h                     ¿ pudo leer encabezado?
        jc      dsk1e5                  s no  cierra y crea
        cmp     qheadid,0bcdefh         ¿ encabezado de cola válido?
        jne     dsk1e5                  s no  continua
dsk1e60 mov     ah,3fh                  ah = func. leer archivo
        mov     cx,qblen                cx = longitud de reg. de
                                        encabezado de cola
        mov     dx,offset qheader       dx  buffer de lectura
        int     21h                     ¿ pudo leer resto de
                                        encabezado?
        jnc     dsk1f0                  s no  continua
dsk1e5  mov     ah,3eh                  ah = cierra archivo
        int     21h                     cierra el archivo
        jmp     dsk1e0                  error  regresa error
```

```
dsk1[illegible]  mov   ax, 4202h         ; ax = función fin de archivo
         xor   cx, cx            ; cx = 0
         mov   dx, cx            ; cx:dx = 0 busca offset
         int   21h               ; llama al DOS
         mov   cx, qrlen         ; cx = longitud de reg. de cola
         div   cx                ; ax = # de regs. en cola
         mov   nextqrnbr, ax     ; salva # sig. para asignar
         jmp   short dsk9        ; y regresa al llamador

dsk2     label byte              ; Leer Registro de Cola
         call  dskrec            ; ¿busca el reg. correcto?
         jc          dsk9        ; si no, regresa error
dsk210   mov   ah, 3fh           ; ah = func. leer archivo
         jmp   short dsk3[illegible]  ; continúa código común

dsk3     label byte              ; Escribe Registro en Cola
         call  dskrec            ; ¿busca el reg. correcto?
         jc          dsk9        ; si no, regresa error
         mov   ah, 40h           ; ah = func. escribe archivo
dsk310   mov   cx, qrlen         ; cx = longitud de reg. de cola
         int   21h               ; llama al DOS
         jc          dsk9        ; si error, regresa rápido
         cmp   cs:closeq, 1      ; ¿necesita cerrar?
         je          dsk700      ; si, duplicado de archivo
         clc                     ; limpiar carry
         jmp   short dsk9        ; y regresa c/edo. apropiado

dsk4     label byte              ; Leer Apuntador en Cola
         call  dskrec            ; ¿busca el reg. correcto?
         jc    dsk9              ; si no, regresa error
         mov   ah, 3fh           ; ah = func. leer archivo
         jmp   short dsk520      ; continúa c/código común

dsk5     label byte              ; Escribe Apuntador en Cola
         call  dskrec            ; ¿busca el reg. correcto?
         jc    dsk9              ; si no, regresa error
         mov   ah, 40h           ; ah = func. de escritura en arch.
dsk520   mov   cx, 2             ; cx = longitud del aptr. de cola
         int   21h               ; llama al DOS
         jmp   short dsk9        ; regresa con estado

dsk6     label byte              ; Obtén sig. Registro en Cola
         mov   cx, nextqrnbr     ; cx = sig. # de registro
         inc   nextqrnbr         ; actualiza sig. # de reg.
         mov   cs:closeq, 1      ; establece cerrar sig. escritura
         clc                     ; limpia bandera de carry
         add   sp, 2             ; salta regresando cx
         jmp   short dsk910      ; regresa c/número de reg.

dsk7     label byte              ; Archivo de Depósito
dsk700   mov   cs:closeq, 0      ; limpia bandera de petición
                                 ; de cierre
         mov   ah, 45h           ; ah = duplica manejador de
                                 ; archivo
         int   21h               ; llama al DOS
         mov   bx, ax            ; bx = duplicado de manejador
                                 ; de archivo
         jmp   short dsk810      ; continúa c/código común

dsk8     label byte              ; Cierra Spool
         xor   cx, cx            ; cx = registro 0
         call  dskrec            ; posiciona archivo para
                                 ; reg. de encabezado de cola
         mov   qhvalid, 0b0bfh   ; hace registro válido
         mov   ah, 40h           ; ax = func. escribe a archivo
         mov   cx, qhlen         ; cx = long. de encabezado de
                                 ; cola
         mov   dx, offset qhdr   ; dx = buffer de lectura
         int   21h               ; ¿escritura de encabezado bien?
         jc    dsk9              ; si no, regresa c/edo. error
dsk810   mov   ah, 3eh           ; ah = func. cierra manejador
                                 ; de archivo
         int   21h               ; llama al DOS
         jmp   dsk9              ; regresa al llamador c/edo.

                                 ; Regreso al llamador c/edo.
dsk9     pop   cx                ; restablece registro
dsk910   pop   bx                ; bx = PSP del usuario
         pushf                   ; salva banderas (CF)
         mov   ah, 50h           ; ah = recoloca PSP del usuario
         int   21h               ; llama al DOS
         popf                    ; restablece banderas
         pop   dx                ; restablece registros
         pop   bx
         pop   si
         ret                     ; regresa c/bandera de carry
                                 ; como estado
dsk      endp

;
; Esta rutina toma el número de registro y posiciona el archivo para
; la siguiente lectura/escritura
;
; Entra
;        bx = manejador de archivo (file handle)
;        cx = número de registro
;
;  Regresa
;        bandera de carry = error (CF = 1)
;
;

dskrec   proc
         push  cx                ; salva registros
         push  dx
         mov   ax, qrlen         ; ax = long. de reg. de cola
         mul   cx                ; dx:ax = offset de registro
                                 ; en archivo
```

```
        mov   cx,dx             ; cx - palabra más significativa
        mov   dx,ax             ; cx:ax - offset en archivo
        mov   ax,4200h          ; ax - func. mueve apuntador
                                ;   de archivo para les func
        int   21h               ; llama al DOS
        pop   dx                ; restablece registros
        pop   cx
        ret
dsktras endp
        align 16                ; límite para sig. manejador
dsktend label byte
dsksg   ends

;
; Este es el manejador de spool de memoria convencional
;

convq   segment para public 'code'  ; spool en memoria convencional
        assume cs:convq             ; invocado via llamada remota
convtrm dw     convend              ; longitud de módulo
        db     'C'                  ; identificador de módulo
        jmp    short conv           ; a la rutina
convtbl label byte                  ; tabla de funciones
        dw     conv1                ; Inicializa Cola
        dw     conv2                ; Lee Registro de Cola
        dw     conv3                ; Escribe Registro de Cola
        dw     conv4                ; Lee Apuntador de Cola
        dw     conv5                ; Escribe Apuntador de Cola
        dw     conv6                ; Asigna Registro a la Cola
        dw     conv7
        dw     conv8
conv    proc   far                  ; interfaz de spool convencional
        push   si                   ; salva registros
        push   bx
        push   dx
        push   cx
        shl    ax,1                 ; ax - offset de rutina
        mov    si,ax                ; si - rutina a ejecutar
        jmp    word ptr cs:convtbl[si] ; al código apropiado

conv1   label  byte                 ; Inicializa Cola
        jmp    short conv9          ; y regresa a llamador

conv2   label  byte                 ; Lee Registro de Cola
        call   convrec              ; busca el "registro" correcto
        mov    cx,qrlen             ; cx - long de reg de cola
        call   convread             ; "lee" el registro
        jmp    short conv9          ; y regresa al llamador

conv3   label  byte                 ; Escribe Registro de Cola
        call   convrec              ; busca el "registro" correcto
        mov    cx,qrlen             ; cx - long de reg de cola
        call   convwrite            ; "escribe" el registro
        jmp    short conv9          ; y regresa al llamador

conv4   label  byte                 ; Lee Apuntador de Cola
        call   convrec              ; busca el apt correspondiente
        mov    cx,2                 ; cx - long del apt de cola
        call   convread             ; "lee" el apuntador
        jmp    short conv9          ; y regresa al llamador

conv5   label  byte                 ; Escribe Apuntador de Cola
        call   convrec              ; busca el apt correcto
        mov    cx,2                 ; cx - long de apt de cola
        call   convwrite            ; "escribe" el apuntador
        jmp    short conv9          ; y regresa al llamador

conv6   label  byte                 ; Asigna un Registro a la Cola
        mov    cx,nextqrecn         ; cx - sig. número de registro
        cmp    cx,maxqrec           ; ¿hay bastantes registros?
        jae    conv610              ; si no, regresa c/error
        inc    nextqrecn            ; actualiza # de reg siguiente
        clc                         ; limpia bandera de carry
        add    sp,2                 ; salta regresando cx
        jmp    short conv910        ; regresa c/# de registro
conv610 stc                         ; coloca bandera de carry-error
        pop    cx                   ; restablece registros
        pop    dx
        pop    bx
        pop    si
        ret                         ; regresa c/bandera de carry

conv7   label  byte
        jmp    conv9                ; regresa c/estado

conv8   label  byte
        jmp    conv9                ; regresa a llamador c/edo

                                    ; Regreso al llamador c/edo
conv9:  pop    cx                   ; restablece registros
conv910 pop    dx
        pop    bx
        pop    si
        clc                         ; limpia bandera de carry
        ret                         ; regresa c/bandera de carry
                                    ;   como estado
conv    endp

; Esta rutina toma el número de registro y posiciona la memoria para
; la siguiente lectura/escritura
;
; Entra
;       cx - número de registro
; Regresa
;       bx - offset del registro
;
```

```
convrec   proc
          push  cx                    ; salva registros
          push  dx
          mov   ax, qrlen             ; ax = long de reg de cola
          dec   cx                    ; cx = # de reg basado en 0
          mul   cx                    ; dx:ax = offset del reg en memoria
          mov   bx, ax                ; bx = dir registro
          pop   dx                    ; restablece registros
          pop   cx
          ret
convrec   endp

; ----------------------------------------------------
; Esta rutina "lee" un registro del spool (memoria convencional)
; Entra
;         bx = registro de spool
;         cx = longitud
;         ds:dx = registro del llamador
; ----------------------------------------------------

convread  proc
          push  cx                    ; salva registros
          push  si
          push  di
          push  es
          push  ds
          mov   si, bx                ; si = reg de spool
          mov   di, dx                ; di = buffer de usuario
          push  ds
          pop   es                    ; es:di = buffer de usuario
          mov   ds, memuseg           ; ds:si = reg de spool
          cld                         ; modo correcto
          rep movsb                   ; traslada registro
          pop   ds                    ; restablece regs
          pop   es
          pop   di
          pop   si
          pop   cx
          ret
convread  endp

; ----------------------------------------------------
; Esta rutina "escribe" un registro en el spool (memoria convencional)
; Entra
;         bx = registro de spool
;         cx = longitud
;         ds:dx = registro del llamador
; ----------------------------------------------------

convwrite proc
          push  cx                    ; salva regs
          push  si
          push  di
          push  es
          push  ds
          mov   si, dx                ; si = reg de usuario
          mov   di, bx                ; di = buffer del spool
          mov   es, memuseg           ; es:di = reg de spool
          cld                         ; modo de escritura
          rep movsb                   ; traslada registro
          pop   ds                    ; restablece regs
          pop   es
          pop   di
          pop   si
          pop   cx
          ret
convwrite endp
          align 16                    ; límite para sig. manejador
convend   label byte
codseg    ends
          end   begin
```

CONCLUSIONES

CONCLUSIONES

Después de haber concluido la realización de este trabajo, queremos simplemente hacer mención de algunos aspectos relativos a la utilidad, orientación y vigencia del tema que aquí hemos tratado.

Hoy día, existen diversas opiniones respecto a la eficiencia del sistema operativo DOS. Las opiniones son del todo variadas y contrastantes. Sin embargo en contra de las "predicciones" de quienes pensaban que el DOS desaparecería, este año salió al mercado la versión 5.0.

El que hoy se hable del DOS como un sistema operativo limitado, no implica que se esté generalizando a todos los ámbitos. Si bien se encuentra en desventaja respecto a los sistemas orientados a los micros 386 y 486; no podemos asegurar que deba ser forzosamente desplazado por éstos, o que su rango de aplicaciones sea muy restringido.

El hecho de que IBM y *Microsoft* hayan acordado mantener la "plataforma" del DOS para los 90's, pone en evidencia la vigencia de este sistema operativo, para el cual los programas TSR fueron diseñados.

Según algunos artículos de las revistas *PC Magazine* y *Computer Language* de este año, existe todavía un nutrido grupo de usuarios que ven cubiertas sus necesidades con las facilidades que el DOS ofrece y no encuentran una razón de peso para perder su inversión de años al cambiar radicalmente a otro sistema operativo. Se hace destacar que, dentro de las facilidades del DOS se distingue de manera importante el papel de los programas residentes en memoria.

Hemos aclarado hasta este momento ciertas cuestiones relativas a la actualidad del tema, ahora veamos algunos aspectos respecto a la utilidad del mismo.

En primera instancia, mencionemos que la información presentada puede servir de referencia tanto al estudiante como al técnico; sin embargo la mayor aportación de este trabajo radica en brindar al programador la instrucción necesaria para desarrollar utilerías residentes en memoria (teniendo en cuenta que esta información se ha mantenido reservada por las compañías de *software*) y así, abrir la posibilidad a los usuarios del DOS de establecer en su sistema, utilerías residentes que satisfagan sus necesidades particulares. Hoy día, existen en el mercado numerosas aplicaciones residentes de uso general; sin embargo, esto en nada ayuda cuando el usuario requiere un programa TSR específico para lograr la eficiencia deseada.

BIBLIOGRAFIA

BIBLIOGRAFIA

LIBROS

Dettmann, Terry R. DOS Programmer's Reference. Que corporation, Carmel, Indiana, 1988.

Hyman, Michael I. Advanced DOS: Memory Resident Utilities, Interrupts, and Disk Management with MS and PC DOS. MIS Press, Portland, Oreg., 1989.

Wadlow, Thomas A. Memory Resident Programming on the IBM PC. Addison-Wesley, Reading, Mass., 1987.

Stevens, Al. TURBO C: Memory-Resident Utilities, Screen I/O And Programming Techniques. MIS Press, Portland, Oreg., 1987.

Willen, David C and Jeffrey I. Krantz. 8088 Assembler Language Programming: The IBM PC. Howard W. Sams, Indianapolis, Ind., 1983

Schildt, Herbert. TURBO C: The Complete Reference. Borland Osborne/McGraw-Hill, Berkeley, California, 1988.

Duncan, Ray. Advanced MS DOS. Microsoft Press, Redmond, Washington, 1986.

Hall, Douglas V. Microprocessors And Interfacing: Programming and Hardware. McGraw-Hill, 1986.

REVISTAS

PERSONAL COMPUTING

SAYROLS México 1990, Ano 3 No.32.

PC WORLD

IDG Communications España. No.57 Agosto 1990.

PC MAGAZINE

PRODUCTIVITY. "Popping out your pop-ups". Agosto 1987. pp.419.

NEWS. " Memory-Resident Programs Jockey for RAM Positions". Marzo 25, 1986. pp.33.

PC TUTOR. Mayo 13, 1986. pp.314.

PC TUTOR. Octubre 15, 1985. pp.249.

SPECIAL REPORT "Taking Up Residence". Nov. 25, 1986. pp.163.

SPECIAL REPORT. "How RAM-Resident Programs Work (And Don't Work). Nov. 25, 1986. pp.176

Peter Norton. "A standard in the making". Mar 25, 1986. pp.79

Jim Seymour. "Looking for the Ultimate Pop-up". Mayo 13, 1986. pp.97.

UTILITIES. "Spooler Lets You Get Back to Work While You Print". Enero 15, 1991. pp.419.

BYTE

"TSRs Past and Future: MS-DOS and OS/2". Edición especial 1987. pp.49.

"Installing Memory-Resident Programs with C". Mar 1987. pp.129.

"Writing Assembly Language Interrupt Routines" Edición especial 1986. pp.249.

"Pull-Down Menus in C". Mayo 1987. pp.109.

"Writing Desk Accessories". Edición esepcial Dic. 1985. pp.105.

"Resident Headaches". Noviembre 1986. pp.361.