



UNIVERSIDAD POPULAR AUTÓNOMA DEL ESTADO DE PUEBLA

ESCUELA DE SISTEMAS COMPUTACIONALES

PROGRAMAS RESIDENTES EN MEMORIA

Trabajo de investigación presentado por:

José Luis Íñigo Ortega

para obtener el título de
Licenciado en Sistemas Computacionales.

Puebla Puebla, Abril de 1993



UPAEP – Secretaría General

Dirección General de Apoyos Académicos

Dirección del Centro de Recursos para el Aprendizaje y la Investigación.

Biblioteca Central - **Karol Wojtyla**

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 textos, imágenes, gráficas, 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 de donde la obtuvo mencionando el autor o autores involucrados en el documento.

Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

UPAEP
BIBLIOTECA CENTRAL
TESIS
USO ÚNICAMENTE EN SALA



A PAPÁ, MAMÁ Y ANGIE

GRACIAS ... POR TODO

48630

Agradezco al Departamento de Informática de la Universidad Popular Autónoma Del Estado De Puebla todas las facilidades proporcionadas para la elaboración de este trabajo, especialmente, al Ing. Luis Alberto Gómez G. asesor de esta tesis.

También agradezco al Ing. José Ma. Bedolla C., director de la Escuela de Sistemas Computacionales, la ayuda brindada durante la revisión final de este trabajo.

PREFACIO

El presente trabajo es una investigación acerca de los programas denominados 'Programas Residentes en Memoria', de su historia, de sus fundamentos tecnológicos y de sus características funcionales.

Con respecto a la vigencia del tema se puede decir que, ya desde 1986 se encuentran comentarios en las principales revistas de computación de los Estados Unidos que dicen que los programas residentes en memoria son un verdadero problema, no tanto porque sean difíciles de desarrollar sino porque generan demasiados conflictos en la computadora. Se decía también que el sistema operativo DOS (ya sea MS-DOS o PC-DOS) no fue concebido para realizar multitareas, que estaba por desaparecer para dejar paso al nuevo sistema operativo OS/2 y que al desaparecer ese, también desaparecerían los programas residentes.

Pero ahora en 1993, siete años después de esas predicciones, la realidad es muy diferente. DOS sigue siendo el sistema operativo más usado en todo el mundo y lo seguirá siendo todavía por bastantes años. Los programas residentes tampoco se han ido, sino que por el contrario ha aumentado su desarrollo y uso en todos los ámbitos de las computadoras personales, desde sencillas utilerías para manejo de teclado como el programa DOSKEY, que la misma Microsoft incluyó en la versión 5.0 de su MS-DOS, hasta complejos administradores de redes de área local.

Por lo tanto se puede decir que una investigación acerca de los programas residentes en memoria sí está justificada ya que es un tema muy vigente y muy actual dentro de la infinidad de áreas de estudio que tocan a las computadoras personales.

Algunas notas aclaratorias antes de comenzar:

El tema principal de este trabajo de investigación, son los programas TSR (Terminate and Stay Resident). Aún así, se ha dedicado un capítulo para el estudio de los Device Drivers y de los Virus, los cuales también pueden ser considerados como programas residentes en memoria.

Los temas tratados aquí son válidos exclusivamente para computadoras personales (PC o compatibles) utilizando el sistema operativo MS-DOS o PC-DOS versiones 2.0 o posteriores. Los programas de demostración fueron desarrollados utilizando únicamente dos lenguajes: pascal y ensamblador. Específicamente se utilizaron las siguientes versiones: Turbo Pascal versión 6.0 y Turbo Assembler versión 2.01 junto con Turbo Link versión 3.01.

Estos dos lenguajes fueron seleccionados porque muestran las dos caras de la moneda al desarrollar un programa residente. Por un lado tenemos un lenguaje muy amigable como lo es Turbo Pascal, el cual no permite optimizar al máximo el programa que se desarrolla. Por otro lado tenemos el lenguaje ensamblador, que nos proporciona un control absoluto del programa y de la computadora, pero el precio de este control es una mayor dificultad al programar.

Este trabajo asume que el lector posee conocimientos acerca de diversas áreas que corresponden a las computadoras personales, tales como: tipos de procesadores de la familia 80x86 de Intel y sus estructuras internas, las diferentes formas de direccionar la memoria de la computadora, los tipos de interacciones que se pueden dar a diferentes niveles en la computadora (acceso al hardware por medio de puertos, llamadas al BIOS y al sistema operativo), lenguajes de programación pascal y ensamblador que son los dos lenguajes que se utilizaron en la realización de los programas de demostración.

Por último, algunas de las palabras clave que aparecen a lo largo del texto, no han sido traducidas porque no existe una traducción satisfactoria para ellas y para no inventar palabras que pudieran distorsionar el significado o confundir al lector, se ha preferido utilizar las palabras originales en inglés. Algunas otras palabras son usadas en inglés y en español indistintamente, tal es el caso de las palabras 'programa residente' que muy a menudo serán sustituidas por la abreviatura TSR.

Esto mismo ocurre en los programas de demostración donde el código y los comentarios están en inglés. En algunos casos, fue para preservar intactas las versiones originales y en otros, (en donde hice modificaciones a los programas o en los programas que yo desarrollé) por gusto personal.

De cualquier manera existe un glosario al final del escrito que contiene el significado de las palabras que no han sido traducidas o que sí lo fueron, pero que no son muy conocidas o que podría haber alguna duda de su significado real.

CONTENIDO

PREFACIO

CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

LISTADOS DE CODIGO

SINOPSIS 1

INTRODUCCION 3

Definición y orígenes 3

CAPITULO 1.
CONCEPTOS BASICOS 6

La memoria de la computadora 7

 Memoria convencional 8

 Memoria principal 8

 Upper Memory Area 8

 Memoria extendida 8

 Memoria expandida 9

 Memorias utilizables 10

El proceso de arranque (Boot)	11
De ROM a RAM	11
DOS toma el control	12
Estructura de un programa y uso de memoria	13
Características de los archivos ejecutables	13
Program Segment Prefix (PSP)	14
Asignación de memoria	17
Acceso a los recursos del sistema	17
Acceso vía hardware	18
Acceso por medio del BIOS y DOS	19
Interrupciones	19
El mecanismo de interrupción	20
Tipos de interrupciones	20
Por hardware	20
Por software	21
Excepciones del sistema	21
Aplicación práctica	21

CAPITULO 2.

PROGRAMAS TRANSITORIOS Y PROGRAMAS RESIDENTES 23

Descripciones y diferencias	23
Tipos de programas residentes en memoria	24
Device Drivers	24
Tipos y estructura interna	25
Device Header	26
Strategy Routine, Interrupt Routine	28
Funcionamiento de los device drivers	28
Instalación	28
Localización	29
Llamado	31
Virus	34
Características principales	34
Clasificación y funcionamiento	35
Programas residentes no TSRs	38

CAPITULO 3.

PROGRAMAS TSR 39

Clasificación de los TSR	39
Pasivos	40
Interactivos	41
Otros criterios de clasificación	42
Conceptos fundamentales para desarrollar un TSR	42
Modificación de la Tabla de Vectores de Interrupción	43
Chaining (Encadenamiento)	44

Reentrance (Reentrada)	45
Stacks internos de MS-DOS	45
InDos Flag	46
Interrupción 28h	47
Critical Error Flag	48
Acceso a archivos de disco	50
Modificación del PSP y DTA	51
Errores críticos	52
Control-Break	52
Funcionamiento de un TSR	53
Porción transitoria	53
Proceso de inicialización	53
Prevenir más de una instalación	53
Localizar banderas	56
Capturar vectores de interrupción	56
Proceso de instalación	57
Proceso de desinstalación	58
Porción residente	59
Proceso de activación	59
Proceso de desactivación	61
Comunicación con la porción residente	61
Lineamientos para codificar un TSR	63
Lineamientos generales	63
Lineamientos para uso del teclado	64
Lineamientos para uso del video	64

CAPITULO 4. ANALISIS DE CODIGOS

TSRs pasivos	66
Interrupt Handlers	67
TSRs interactivos	72
Porción transitoria	72
Código de inicialización	72
Determinar la versión de DOS	72
Prevenir más de una instalación	73
Localizar banderas	76
Capturar vectores de interrupción	78
Código de instalación	79
Código de desinstalación	80
Porción residente	84
Código de activación	84
Determinar el momento de activación	84
Rutina principal	88
Código de desactivación	91
Código para comunicación	92

CAPITULO 5.	
LAS NUEVAS TECNOLOGIAS	94
Memoria extendida y expandida	95
Interacción con HIMEM.SYS	95
A20 Handler	95
Upper Memory Blocks	97
Interacción con EMM386.SYS	99
Inicializar memoria expandida	100
Utilizar la memoria ya reservada	104
Liberar la memoria al final del proceso	105
Task Switchers	106
Instance Memory	106
Implementación de Instance Memory	108
Estructuras de datos	108
Inicialización de estructuras	111
Manipulación de la memoria por sesiones	113
 CAPITULO 6.	
EL FUTURO DE LOS TSR	118
Ventajas y desventajas de los TSR actuales	118
Los TSR y sistema operativo OS/2	120
Threads	122
Popup Displays	123
Device Monitors	125
 CONCLUSION	129
 APENDICES	130
A - XKBDDBUF.PAS	132
B - F11.PAS	133
C - CLOCK.ASM	134
D - BLANKER.ASM	136
E - POPCLOCK.ASM	139
F - JTRSTP.PAS	142
G - SAVESCR.PAS y SHOWSCR.PAS	145
H - TSRTMPLT.ASM	146
 GLOSARIO	154
REFERENCIAS BIBLIOGRAFICAS	156

LISTA DE TABLAS

Tabla 1-1. Desglose del PSP	15
Tabla 2-1. Interpretación de los bits del campo Device Attribute del Device Header	27
Tabla 2-2. Códigos de comando para un device driver	31
Tabla 5-1. Funciones más utilizadas del LIM EMM v3.2	99
Tabla 5-2. Estructuras de datos para manejo de Instance Memory	100

LISTA DE FIGURAS

Figura 1-1. Mapa de la memoria de una PC	7
Figura 2-1. Estructura de un Device Driver	26
Figura 2-2. Lista de drivers en memoria	29
Figura 2-3. Estructura del Request Header	30
Figura 2-4. Infección del sector de carga	36
Figura 3-1. Encadenamiento de ISRs	44
Figura 3-2. Interacciones del TSR con el sistema	62
Figura 6-1. Flujo de datos en OS/2	125

LISTADOS DE CODIGO

Listado 2-1. Driver Dispatch Routine	32
Listado 2-2. Esqueleto de un device driver	33
Listado 2-3. VIRUS.BAT	35
Listado 3-1. Forzar el uso del stack 3 de DOS	50
Listado 3-2. Uso de la INT 2Fh para localizar una copia del TSR previamente instalada	54
Listado 3-3. ISR para monitorear la interrupción 2Fh	55
Listado 4-1. XKBDDBUF.PAS	67
Listado 4-2. F11 PAS	68
Listado 4-3. Inicialización de un Interrupt Handler	69
Listado 4-4. Timer-Interrupt Handler	71
Listado 4-5. Verificar la versión de DOS	73
Listado 4-6. Prevenir más de una instalación del TSR utilizando la INT 2Fh	73
Listado 4-7. Prevenir más de una instalación del TSR utilizando rastreo de memoria	74
Listado 4-8. Prevenir más de una instalación del TSR utilizando la firma del programa	75
Listado 4-9. Localizar banderas de DOS 2.0	77
Listado 4-10. Localizar banderas de DOS 3.1 o superior	78
Listado 4-11. Captura de vectores de interrupción	79
Listado 4-12. Instalación del TSR en memoria	80
Listado 4-13. Código de desinstalación transitorio	81
Listado 4-14. Código de desinstalación residente	83
Listado 4-15. ISRs 08h, 09h, 10h, 13h y 28h	84
Listado 4-16. ISR para manejo de errores críticos	87
Listado 4-17. Inicio de la rutina principal	88
Listado 4-18. Inicio de la rutina principal (pascal)	89
Listado 4-19. Código de desactivación	91
Listado 4-20. Código de desactivación (pascal)	92
Listado 4-21. Comunicación con la porción residente	93
Listado 5-1. Localizar el XMM	96
Listado 5-2. Salvar y restaurar la línea A20	97
Listado 5-3. Salvar y restaurar el estado de los UMBs	98
Listado 5-4. Inicialización de memoria expandida	101
Listado 5-5. Localizar el EMM	102
Listado 5-6. Reservar memoria expandida	103
Listado 5-7. Mapear memoria hacia el page frame	104
Listado 5-8. Salvar y restaurar el estado del EMM	104
Listado 5-9. Liberar memoria expandida	105
Listado 5-10. Estructuras de datos para manejo de Instance Memory	110
Listado 5-11. Inicialización de estructuras para Instance Memory	111
Listado 5-12. Rutina de notificación	113

Listado 5-13. Uso de la INT 2Fh para manipulación de Instance Memory	114
Listado 5-14. Proceso de activación y desactivación corriendo bajo un task switcher	115
Listado 5-15. Rutinas para deshabilitar/habilitar la suspensión de sesiones de DOS ..	117
Listado 6-1. Uso de las funciones VIOPOPUP y VIOENDPOPUP	124
Listado 6-2. Esqueleto de un Device Monitor	126

A continuación se presenta un breve resumen del contenido de cada uno de los capítulos que forman este trabajo.

Introducción.

La introducción presenta una definición general de lo que es un 'programa residente', además proporciona una breve historia de estos programas y de su evolución desde antes de la primera versión del sistema operativo MS-DOS hasta nuestros días.

Capítulo 1. Conceptos básicos.

En este primer capítulo, se asientan las bases y se explican los conceptos sobre los que se fundamentan los programas residentes en memoria. Aunque varios de estos conceptos deben ser ya conocidos por el lector, han sido incluidos aquí en cuanto que están íntimamente relacionados con la implementación de los TSR. Un concepto especialmente importante y sobre el que se debe poner mucha atención, es el de interrupción. Se puede decir que las interrupciones son la piedra angular con la que se construye un programa residente.

Capítulo 2. Programas transitorios y programas residentes.

En este capítulo, se hace la distinción entre un programa convencional y uno residente. Para ello se describe la principal característica que hace que un programa sea considerado residente.

Esta característica distintiva está presente en tres tipos de programas: los device drivers, los virus y los TSR. El resto de este capítulo se destina a la descripción detallada de los dos primeros tipos de programas residentes.

Capítulo 3. Programas TSR.

A partir de este capítulo, todo el material presentado estará relacionado exclusivamente con los TSR. Este capítulo inicia haciendo una clasificación de estos programas, según su forma de operación y su complejidad. A continuación se describen en detalle las características generales de un TSR, y de todos sus componentes internos, así como de la forma de implementarlos.

La última sección de este capítulo asienta los lineamientos necesarios para codificar un TSR que sea estable y que pueda coexistir 'pacíficamente' con el sistema operativo y con cualquier otro programa que esté corriendo en el sistema al mismo tiempo.

Capítulo 4. Análisis de código

Este capítulo es una réplica del anterior, pero aquí se pasa de la teoría a la práctica. Para ello se analiza detalladamente el código de varios programas residentes. El análisis se realiza tomando porciones selectas de código y estudiándolas por separado para ejemplificar cada uno de los lineamientos y técnicas que se mencionaron en los capítulos anteriores. El código completo de todos los programas analizados se puede encontrar en los apéndices al final del trabajo.

Los programas presentados van aumentando en complejidad para ir mostrando paso a paso cómo es que se implementa cada uno de los requerimientos de un TSR que sea amigable con el sistema y que al mismo tiempo sea completamente estable. Para estas demostraciones se utilizan indistintamente pascal y ensamblador. El programa más complejo de este capítulo será desarrollado con una combinación de ambos.

Capítulo 5. Las nuevas tecnologías.

Este capítulo describe una serie de tecnologías, que se volvieron de uso común en 1991 con la liberación de la versión 5 de MS-DOS. Estas tecnologías influyen directamente en el diseño e implementación de un TSR, pues proporcionan más y mejores recursos que los programas residentes pueden explotar, pero al mismo tiempo exigen nuevos requisitos e imponen nuevas restricciones. Un programa residente actualizado, debe ser capaz de utilizar la memoria alta y expandida además de la memoria principal del sistema; debe saber cómo interactuar con los task switchers (conmutadores de tareas) y con ambientes gráficos tales como Windows. El estudio de estas tecnologías es el tema del presente capítulo.

Capítulo 6. El futuro de los TSRs.

Para terminar este trabajo de investigación, se describen las ventajas y desventajas de los TSR bajo MS-DOS. También se explican brevemente los requerimientos para implementar programas residentes que corran bajo el sistema operativo OS/2; para ello se proporcionan algunos extractos de código diseñado para ese ambiente. Con esto se trata de dar una visión de cómo serán los programas residentes en un futuro no muy lejano.

INTRODUCCION

Un programa común y corriente que corra bajo el sistema operativo MS-DOS, es cargado en memoria desde algún disco cada vez que es ejecutado y removido de ésta al terminar su operación. Si se desea ejecutar el programa otra vez, es necesario volver a cargarlo desde disco. Un programa residente, por el contrario, permanece en memoria aun cuando ya no está corriendo. De ahí, que los programas residentes sean llamados frecuentemente: TSR, que es la abreviatura para Terminate and Stay Resident (terminar y permanecer residente).

Un TSR puede ser reactivado en cualquier momento, aun cuando otro programa esté corriendo. La reactivación es causada por eventos específicos en la computadora, tales como: el movimiento del ratón, la hora del día almacenada en el reloj del sistema, acceso a algún periférico o muy frecuentemente al oprimir una secuencia especial de teclas.

La historia de los programas residentes es fascinante. Aparentemente los servicios del sistema operativo que posibilitan la creación de los TSR, existían con el único fin de reconfigurar el acceso a los periféricos, instalando device drivers y manejadores de interrupciones por medio de comandos, tales como MODE y PRINT. (Los device drivers son programas especiales de entrada/salida que casi no demandan recursos del sistema operativo). Nunca se pretendió que estos servicios fueran utilizados para implementar utilerías interactivas

Pero aún así, después de experimentar un par de años y de desensamblar pacientemente a MS-DOS, se descubrieron estos servicios y gradualmente se generaron estrategias para desarrollar TSRs. Estas estrategias involucraban el uso de funciones y áreas de datos del sistema operativo que no estaban documentadas oficialmente ni por Microsoft ni por IBM pero que eran bastante estables.

El sistema operativo MS-DOS tiene sus raíces en el sistema operativo CP/M escrito por Digital Research para las máquinas de 8 bits basadas en los procesadores 8080 y 8085 de Intel. La diferencia fundamental entre las computadoras de 8 bits y la entonces nueva PC de la IBM estaba en que las primeras podían direccionar un máximo de 64Kb de memoria, en cambio la PC tenía a su disposición 1024Kb.

Bajo CP/M, cualquier mejora o extensión al sistema operativo que se deseara realizar, (macros, teclas de función, reloj del sistema, etc.) debía ser incorporada directamente en el BIOS de la computadora, es decir, sólo el fabricante podía hacerlo. Además, la capacidad de implementar mejoras estaba muy limitada, debido a la poca memoria con la que se contaba, y como el fabricante era quien seleccionaba las mejoras, el usuario no podía configurar su computadora de acuerdo a sus preferencias o necesidades. Aún teniendo tan poca memoria disponible se llegaron a escribir algunos programas residentes para este sistema operativo.

Las primeras versiones de MS-DOS se parecían mucho a su antecesor y, por lo tanto, el soporte para programas residentes ya estaba incorporado en el sistema operativo desde su primera versión. Seattle Computer Products (SPC) fue el autor original de MS-DOS, aunque en aquellos primeros días del sistema operativo, su nombre era 86-DOS. La primera versión comercial fue el 86-DOS v0.3, la cual ya ofrecía funciones para soporte de TSRs. Esto sucedía más de un año antes de que Microsoft comprara el sistema operativo.

El propósito de estas funciones era el de permitir la fácil adición de extensiones al sistema operativo, y en particular la instalación de device drivers y manejadores de interrupciones escritas por el usuario. Así, las mejoras y extensiones podrían ser adicionadas a MS-DOS cargándolas en memoria, sin necesidad de reescribir el sistema operativo. Esto fue posibilitado en parte por el hecho de que las PC podían direccionar y utilizar bastante más memoria que las máquinas que utilizaban CP/M.

IBM adaptó el sistema operativo MS-DOS y lo vendió como PC-DOS. Las versiones de MS-DOS y PC-DOS anteriores a la versión 2.0, no tenían la opción de instalar device drivers (utilizando la instrucción 'DEVICE=' del CONFIG.SYS), pero los TSR proporcionaban una funcionalidad equivalente.

El único programa residente que se distribuyó como parte del sistema operativo, en sus versiones 1.xx, fue el programa MODE.COM. Con la introducción de la versión 2.0, Microsoft incluyó nuevas utilerías residentes como PRINT.COM, GRAPHICS.COM y ASSIGN.COM, ahora muy conocidas en el mundo de las PCs.

Fue PRINT.COM, (el ejemplo de TSR por excelencia) el que desencadenó el estudio y desarrollo de programas residentes. La capacidad de PRINT.COM de imprimir mientras otro programa está corriendo, mostró por vez primera la forma de simular el acceso simultáneo a dos programas.

Microsoft se negó a proporcionar detalles acerca de la implementación de sus programas residentes, pero los programadores entusiastas no tardaron mucho tiempo en desensamblarlos y descubrir los secretos para simular esta forma de multiproceso. Al principio, lo único que se intentaba hacer era duplicar la funcionalidad de los programas que Microsoft proporcionaba, por lo tanto, casi todos los TSR que se escribían, eran sencillos relojes que aparecían en alguna esquina de la pantalla o programas para imprimir, semejantes a PRINT.COM.

Microsoft contestó a estas actitudes expresando su ya famosa política de 'no soporte al concepto de TSRs'. El resultado, como ya es de todos sabido, fue que nadie

hizo caso de las amenazas de Microsoft y desde entonces ha habido un diluvio de programas residentes, los cuales han ido aumentando en complejidad, conforme se han ido refinando las técnicas necesarias para desarrollarlos.

En 1991, Microsoft por fin documentó gran parte de las funciones utilizadas para el desarrollo de programas residentes, las cuales ya estaban documentadas en todos lados menos en la documentación oficial de Microsoft, reconociendo así, de forma implícita, que había perdido la batalla en contra de los TSR.

Aunque MS-DOS fue hecho para ejecutar un solo programa a la vez, es obvio que posee todas las herramientas necesarias para desarrollar programas residentes. Esto es bueno, ya que sin ellos, las computadoras personales serían mucho menos eficientes y tal vez un poco aburridas.

CAPITULO 1 CONCEPTOS BASICOS

Antes de poder estudiar las características y el funcionamiento de los programas residentes en memoria, es necesario entender algunos conceptos fundamentales acerca de la computadora personal, desde el nivel de hardware, hasta los niveles altos representados por los diferentes lenguajes de programación. Cabe subrayar que el presente trabajo se enfoca en su mayor parte al área de software, por lo que los temas que corresponden al hardware no serán tocados muy a fondo y únicamente en los casos en donde sea indispensable.

Los TSRs, como cualquier programa, necesitan una determinada cantidad de memoria para funcionar correctamente; por lo tanto, es conveniente saber qué tipos de memoria se encuentran disponibles en la computadora y cuáles son utilizables por el programa residente.

También es necesario conocer las diferentes acciones que se llevan a cabo durante el proceso de arranque, (bootup) para así poder comprender mejor el funcionamiento de los 'Device Drivers' y de los 'Virus', los cuales son dos tipos de programas residentes que serán estudiados más a fondo en el siguiente capítulo.

La estructura interna de los programas nos es útil, porque nos proporciona pistas acerca de los artificios que son utilizados para hacer que dos o más programas estén presentes en memoria y puedan ser activados en cualquier momento, es decir, nos muestra cómo implementar una forma de procesamiento multitarea. Aquí conviene recordar que MS-DOS no es un sistema operativo multitarea.

Por último se describirán los procesos por medio de los cuales se puede hacer uso de los recursos que proporciona el sistema. En especial se describirán los diferentes tipos de interrupciones.

Si se analiza con un poco de detenimiento, se llega a la conclusión, de que por su propia naturaleza, los TSRs son en sí mismos interrupciones, ya que para activarse le quitan el control al programa que está corriendo en ese momento, es decir, lo interrumpen. De ahí que sea tan importante la comprensión de este mecanismo de comunicación entre el CPU y sus periféricos.

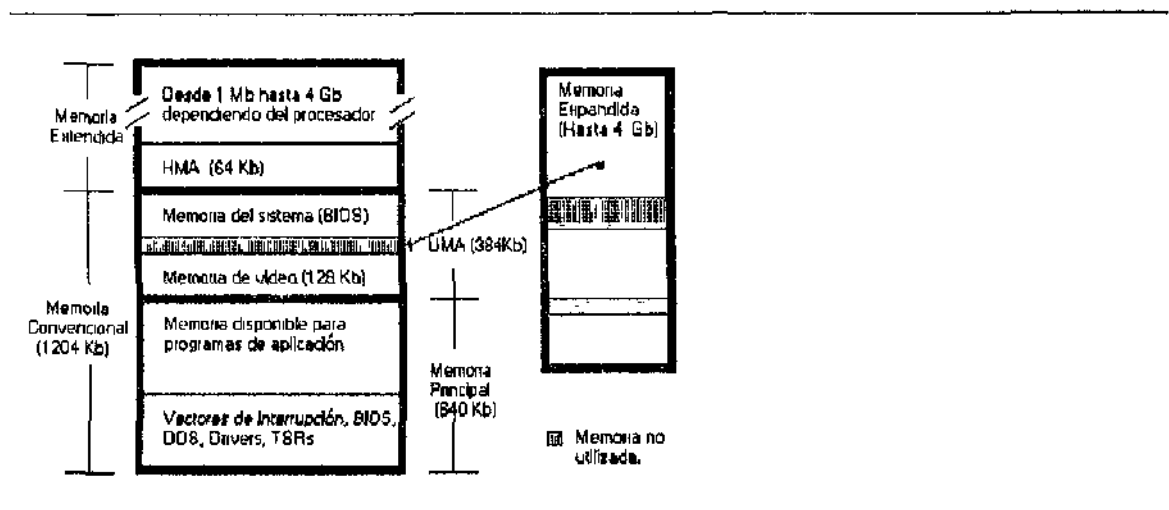
LA MEMORIA DE LA COMPUTADORA.

La memoria de la computadora consiste de un determinado número de elementos de almacenamiento, cada uno de los cuales guarda un byte de datos. A cada uno de estos elementos le asigna una dirección numérica única. Así, al primer elemento se le asigna la dirección 00h, (por convención siempre se usa la notación hexadecimal para especificar direcciones de memoria), al segundo elemento la dirección 01h y así sucesivamente hasta el último elemento, el cual tendrá una dirección que estará determinada por la cantidad total de elementos físicos de memoria instalados en la computadora.

El procesador central de las máquinas que corren bajo MS-DOS, divide la memoria de la computadora en bloques llamados 'segmentos' y cada segmento ocupa 64Kb de memoria. Las diferentes localidades de memoria son accedidas utilizando el formato segment:offset (segmento y desplazamiento), es decir, mediante la combinación de la dirección contenida dentro de alguno de los registros de segmento del procesador y el contenido de alguno de los otros registros. Por ejemplo, la 'siguiente' instrucción (de cualquier programa) a ser ejecutada por el procesador se accesa mediante la combinación de los registros CS:IP (Code Segment e Instruction Pointer respectivamente).

Esta forma de acceso a la memoria, utilizada por los procesadores 8088 y 8086, limita cada dirección a un máximo de 20 bits y, por lo tanto, la memoria accesible al procesador es de 1 Mb (2 elevado a la 20). Los procesadores más recientes (286,386,486) ya no tienen esta restricción y pueden acceder cantidades mucho más grandes de memoria, pero MS-DOS sigue limitando la cantidad de memoria accesible al mismo megabyte. La figura 1-1 muestra un mapa de la memoria de una computadora personal.

Figura 1-1 Mapa de la memoria de una PC.



Para complementar el mapa, a continuación se describe más detalladamente el contenido de cada una de las áreas y tipos de memoria.

MEMORIA CONVENCIONAL (Desde los 0Kb hasta los 1024Kb)

Por los usos para los que está destinada, esta memoria se encuentra dividida en dos áreas: memoria principal y Upper Memory Area (UMA).

Memoria principal (Desde los 0Kb hasta 640Kb)

Esta es la memoria que cualquier programa utiliza para correr, aunque no toda está disponible, ya que parte de ella es ocupada por el sistema operativo y otros programas que permanecen residentes. Desde la dirección más baja y en orden ascendente, los programas y/o datos que normalmente se encuentran en esta memoria son: la tabla de vectores de interrupción algunas partes del BIOS, el MSDOS.SYS (o el IBMDOS.COM), los device drivers integrados del sistema operativo y los que hayan sido instalados por el usuario, la porción residente del sistema operativo y los TSRs que el usuario haya instalado. El resto de los 640Kb de memoria principal queda disponible para cualquier programa de aplicación.

Upper Memory Area (UMA, desde los 640Kb hasta 1024Kb)

El nombre de esta memoria se podría traducir como Area de Memoria Superior. Casi todos los sistemas tienen 384Kb de memoria adyacentes a los 640Kb para aplicaciones. Esta memoria, aunque es parte de la memoria convencional, no se contabiliza dentro del total de memoria principal, porque no es posible que un programa almacene información en ella. Es por esta razón que a veces la UMA es considerada como parte de la memoria extendida. Este concepto está reforzado por el hecho de que la UMA y la memoria extendida generalmente son controladas por medio de un mismo tipo de programa: el 'Extended Memory Manager' o XMM (Administrador de Memoria Extendida).

Aún así, la UMA no puede ser considerada como extendida por la sencilla razón de que la memoria extendida no tiene asignada ninguna tarea en especial por default dentro del sistema, es decir, puede estar instalada en la computadora y nunca ser utilizada. En cambio la UMA sí tiene asignadas tareas específicas, ya que está destinada para dar soporte al hardware del sistema. Esta memoria por lo general contiene: porciones del BIOS de la máquina, del manejador de discos y de las tarjetas EGA/VGA, el espacio para la memoria de video y espacios no contiguos de memoria no utilizada.

MEMORIA EXTENDIDA.

Esta memoria se encuentra más allá de los 1024Kb de memoria convencional y puede haber desde 1 Mb hasta un máximo de 16Mb en una máquina con procesador 80286 o hasta 4Gb en una máquina 386 o 486.

Esta memoria no está disponible para los procesadores 8088 ni 8086 y generalmente no se puede utilizar, con ningún procesador, en forma directa, esto debido a las limitaciones impuestas por el sistema operativo. Por lo tanto, es necesario un programa administrador de memoria extendida como HIMEM de Microsoft o QRAM de

Quarterdeck Office Systems para poder utilizar esta memoria. Estos mismos programas son utilizados para controlar el uso de la UMA como ya se mencionó.

Los primeros 64Kb de esta memoria son conocidos con el nombre de 'High Memory Area' o HMA (Area de Memoria Alta). Esta memoria generalmente no es utilizada por ningún programa de aplicación. Precisamente por esto, Microsoft decidió utilizar esta memoria para ahí correr el sistema operativo y así aumentar la cantidad de memoria principal disponible para los programas de aplicación. (Esta característica está disponible únicamente con procesadores 286 o superiores).

MEMORIA EXPANDIDA.

Esta clase de memoria, generalmente se adiciona a la computadora mediante tarjetas de expansión, pero también es posible emularla a través de la memoria extendida o incluso por medio del disco duro. El acceso a esta memoria se logra a través de un programa llamado 'Expanded Memory Manager' o EMM (Administrador de Memoria Expandida), ejemplos de este tipo de programa son EMM386 de Microsoft y QEMM de Quarterdeck.

Aunque existen diferentes formas de utilizar la memoria expandida, en general todas están basadas en el proceso de 'Mapeo', el cual se describe brevemente a continuación:

1) Antes de poder utilizar la memoria expandida, el EMM debe dividirla en bloques llamados 'Páginas', cada una de las cuales equivale a 16Kb.

(Nota: Algunas implementaciones de EMMs no restringen el tamaño de la página, ya que pueden manipular simultáneamente varias páginas de distintos tamaños cada una).

2) Como ya se dijo, en los 384Kb de la UMA existen espacios de memoria no utilizados. EL EMM debe reservar 64Kb de estos espacios libres para su propio uso. En este espacio de memoria que el EMM reserva, se construye el denominado 'Page Frame' (Marco de Página). Este marco funciona como una ventana hacia la memoria expandida. Como se ve, el page frame tiene espacio suficiente para contener 4 páginas de memoria de 16Kb cada una.

Las páginas del page frame (numeradas de 0 al 3) se conocen como 'Páginas Físicas', ya que es posible para cualquier procesador, desde el 8088 hasta 80486 accederlas directamente, pues se encuentran dentro del primer megabyte de memoria. Las páginas de memoria expandida, las que se encuentran en la tarjeta de expansión, se conocen como 'Páginas Lógicas' y no pueden ser accesadas directamente por ningún procesador.

3) Por lo tanto, antes de poder utilizar una página contenida en la memoria expandida, es necesario direccionarla hacia una página del marco. El proceso de direccionar, (a través del page frame) memoria convencional (UMA) hacia memoria expandida o viceversa, se denomina mapeo de memoria.

Por ejemplo, si se desea almacenar información en la página 0 de memoria expandida, primero se mapea la página 0 hacia el page frame y después se almacena la información en el page frame.

Si todas las páginas del page frame ya están ocupadas y se mapea una nueva página de memoria expandida hacia el page frame, entonces el EMM debe mapear un página del page frame de regreso a la memoria expandida, esto con el propósito de liberar memoria para la nueva página que va a entrar. El contenido de la página que sale del page frame no es alterado en ninguna forma y se puede volver a acceder por medio de otro mapeo.

Una característica muy importante del proceso de mapeo, es que no realiza ninguna transferencia de datos (de ahí su alta velocidad), es decir, las páginas solo 'parece' que se mueven pero en realidad lo único que se modifica es la dirección de la página con la que se está trabajando. Se puede decir que el único que se mueve es el page frame el cual funciona como marco de referencia para la página que se está accediendo.

Si se desea que el TSR utilice memoria expandida, todos estos procesos deben ser controlados desde el código del programa. Afortunadamente, el EMM proporciona funciones que facilitan la implementación de estos procesos, como se verá en el capítulo 5 cuando se trate más a fondo este tema.

En resumen, la memoria expandida actúa como un banco de memoria, que bajo el control del EMM puede ser mapeado a una o varias regiones de la memoria convencional. Mapeo es el proceso mediante el cual una porción de la memoria expandida 'aparece' en alguna localidad específica de la memoria convencional, sin necesidad de que se realice ninguna transferencia de datos.

MEMORIAS UTILIZABLES

Para terminar esta sección se identificarán las memorias que pueden ser utilizadas para correr programas residentes. Como regla general, se puede decir que un programa residente puede utilizar cualquiera de las memorias antes descritas, excepto la HMA.

Para que el programa residente pueda utilizar la memoria principal no es necesaria ninguna consideración especial.

Para hacer uso de la UMA y la memoria extendida, únicamente se requiere que el programa residente sea corrido en un sistema que cuente con un procesador 386 o superior y además se debe usar algún administrador de memoria extendida.

Por último, si se desea aprovechar la memoria expandida, el TSR debe tener código interno especial que lo permita, y otra vez es necesario un procesador 386 o mejor y un programa administrador de memoria expandida. En el capítulo 5 se analiza un programa de demostración, que contiene código especial para utilizar memoria expandida.

EL PROCESO DE ARRANQUE (BOOT).

Cuando se prende la máquina (Cold Boot) o cuando se presiona la secuencia de teclas Ctrl-Alt-Del (Warm Boot) se inicia el proceso de arranque. Estas dos formas de realizar el boot, se diferencian únicamente en que el Cold Boot realiza la revisión de la memoria de la computadora, antes de ejecutar todo el proceso que a continuación se describe.

DE ROM A RAM.

El proceso de arranque se inicia con la ejecución de código en ROM, el cual verifica el correcto funcionamiento del sistema e identifica qué equipo está disponible en la computadora. Esta rutina es conocida como POST (Power On Self Test) y está almacenada en la dirección de memoria FFFFh. Al finalizar, esta rutina ejecuta una INT 19h (interrupción 19h), la cual invoca la rutina de Bootstrap almacenada también en ROM.

"La rutina de Bootstrap intenta leer el sector 1 de la pista (track) 0 del drive A y si no lo logra lo intenta con la unidad C. El sector 1, de un disco formateado con el sistema operativo MS-DOS, es conocido como el 'Boot Sector' (Sector de Carga). El sector de carga contiene código y datos que son indispensables para continuar el proceso. Para leer estos sectores se hacen llamadas a la INT 13h que se encuentra almacenada en ROM. Si ninguno de estos dos drives responde, el control es transferido al intérprete de BASIC en ROM, si está presente en la computadora, y si tampoco se encuentra, se despliega un mensaje de error.

Cuando el sector de carga es leído exitosamente, la rutina de Bootstrap pasa el control al código de ese sector. Este código verifica que los archivos IO.SYS y MSDOS.SYS (si se está utilizando PC-DOS los archivos serán IBMBIO.COM y IBMDOS.COM respectivamente) estén presentes en el directorio raíz del disco que se está utilizando para la inicialización. Si se encuentran, IO.SYS es cargado en memoria baja, utilizando llamadas a la INT 13h y se pasa el control a su rutina de inicialización. IO.SYS a su vez carga MSDOS.SYS y le transfiere el control junto con el apuntador principal a la lista que contiene todos los drivers integrados del sistema operativo. El código para implementar estos drivers se encuentra en IO.SYS." [16:p.312]

Actualmente, el sistema operativo tiene integrados drivers para los siguientes dispositivos: NUL, CON (consola), AUX (auxiliar), PRN (impresora), CLOCK\$ (reloj del sistema); para los diferentes puertos de comunicaciones COM1, COM2, LPT1, LPT2, LPT3 y para los discos flexibles y duros (si están presentes).

Al recibir el control, MSDOS.SYS almacena el apuntador de la cadena de drivers en el campo llamado 'Next Device' o 'Next Header Pointer' del NUL device (dispositivo Nulo). Este dispositivo se localiza en la Tabla Global de DOS (esta área de datos, también es conocida como la Lista de Listas). Con esta acción lo que se logra es que el NUL device se convierta en el nuevo apuntador principal de la cadena de drivers.

A continuación se inicializan los device drivers que requieren tratamiento especial como son el reloj y la consola. Después se construyen una serie de estructuras de datos que servirán para localizar los recursos globales del sistema. Entre las estructuras creadas

están: SFT (System File Table), DCB (Device Control Block), CDS (Current Directory Structure) y la lista de device drivers.

(Nota: Los campos de un driver así como la lista de device drivers, serán estudiados con más detenimiento en el siguiente capítulo, cuando se describa la estructura interna y el funcionamiento de los programas residentes en memoria denominados Device Drivers).

Para finalizar su parte del proceso de arranque, MSDOS.SYS inicializa la dirección para la interrupción 21h (que es la principal interface con el sistema operativo) y regresa el control a IO.SYS.

DOS TOMA EL CONTROL.

Ya de vuelta en la rutina de inicialización de IO.SYS, la consola, el reloj y los discos ya han sido inicializados y la interrupción 21h ya es operacional, es decir el 'Kernel' de DOS ya es capaz de ejecutar los comandos que se le soliciten. El kernel es la parte de DOS que contiene las estructuras de datos y las rutinas que dan servicio a cualquier aplicación que corra bajo MS-DOS. El kernel sirve a las aplicaciones de dos formas:

- 1) Administra la información acerca del directorio actual, de los archivos que están siendo utilizados por la aplicación y en general, de todos los dispositivos periféricos.
- 2) El kernel proporciona rutinas de servicio, (frecuentemente llamadas funciones de MS-DOS o interrupciones de MS-DOS) que son utilizadas para administrar memoria, archivos y para controlar funciones de entrada/salida de los dispositivos de la computadora (pantalla, teclado y puertos).

IO.SYS utiliza los nuevos servicios que ya se han puesto a su disposición a través de la INT 21h y con ellos abre el archivo CONFIG.SYS y ejecuta sus instrucciones. Puede haber instrucciones para: cargar device drivers especiales, para controlar el tamaño de varias tablas de datos que utiliza DOS, pedir opciones de proceso específicas y para especificar el intérprete de comandos (shell) a utilizar.

El siguiente paso que realiza esta rutina de inicialización, es hacer una llamada a la función 4Bh del DOS, la cual invoca al cargador de programas. El cargador, también llamado EXEC, es el responsable de cargar programas a memoria y pasarles el control. En este caso, la rutina de inicialización ordena a EXEC que cargue el COMMAND.COM o en su defecto, cualquier intérprete de comandos que se haya especificado en el CONFIG.SYS.

"El programa COMMAND.COM, está constituido por tres partes: una porción de inicialización, una porción residente y una porción temporal o transitoria. La porción que es cargada por la llamada a EXEC, es la porción residente. (Esta porción permanecerá cargada en memoria hasta que se apague la máquina). Este código es responsable de cargar la porción temporal, además de que contiene las rutinas para responder a los errores de entrada/salida, así como las rutinas que controlan las interrupciones 22h (Terminate Address), 23h (Ctrl-Break) y 24h (Critical Error)." [15:p.227]

La porción de inicialización es cargada inmediatamente arriba de la porción residente, y es la encargada de interpretar y ejecutar los comandos contenidos en el archivo AUTOEXEC.BAT o de pedir la fecha y hora del sistema. Después, esta porción es descartada de memoria.

Por último, la porción temporal o transitoria del COMMAND.COM es cargada en lo más alto de la memoria principal (junto al límite de los 640Kb). Esta porción es la que contiene los comandos internos del sistema operativo, la que carga y ejecuta los comandos externos y los archivos ejecutables y la que despliega el prompt del sistema (A> o C>), ésta es la señal de que el proceso de arranque ha finalizado y que el sistema operativo está listo para aceptar comandos del usuario.

Como su nombre lo sugiere, esta porción puede ser sobrescrita durante la ejecución de cualquier programa. Cuando un programa termina de correr, la porción residente determina si la porción temporal fue sobrescrita y la vuelve a cargar si es necesario.

ESTRUCTURA DE UN PROGRAMA Y USO DE MEMORIA.

Como ya es sabido, hay dos tipos de programas ejecutables que pueden correr bajo MS-DOS (sin contar a los archivos .BAT que no son programas propiamente dichos). El tipo de programa más simple está representado por los archivos .COM, la mayoría de los cuales son bastante viejos (originalmente desarrollados para CP/M) o son utilerías muy pequeñas.

Cuando MS-DOS aún era joven, el uso flexible de memoria y la capacidad de poder desocupar la memoria no utilizada no eran muy importantes, ya que generalmente era un solo programa el que corría a la vez y a él se le destinaban absolutamente todos los recursos del sistema. Pero la llegada de los programas residentes en memoria cambió todo ello. Hoy en día casi cualquier programa que se desee utilizar bajo MS-DOS, debe ser capaz de solicitar y devolver memoria conforme la vaya necesitando. Es por eso que fue creado el segundo tipo de programa, el archivo .EXE, el cual proporciona mayor flexibilidad en el manejo de la memoria.

CARACTERISTICAS DE LOS ARCHIVOS EJECUTABLES.

Todos los programas escritos bajo MS-DOS deben pasar por el proceso de linking (ligado). El linker (ligador) determina en que áreas de la memoria deberán ser cargadas cada una de las partes del programa y almacena esta información en un encabezado al principio de cada programa. Todos los programas creados de esta forma tienen una extensión .EXE, y pueden utilizar tantos segmentos de memoria como haya disponibles, además de que el código de los programas es reubicable (puede cambiar de dirección en memoria) después de haber sido cargado. Así, varios segmentos pueden estar destinados al código, datos o stack del programa.

Los archivos .EXE que cumplan con los siguientes tres requisitos podrán ser convertidos en archivos .COM:

- 1) El programa y todos sus datos deben ocupar un máximo de 64Kb (en realidad deben ocupar 64Kb menos 258 bytes).
- 2) El código, datos y stack del programa deben caber en el mismo segmento de memoria.
- 3) La primera instrucción ejecutable del programa debe tener un desplazamiento de 100h, es decir, esta primera instrucción ya tiene un lugar específico en memoria en donde debe ser cargada, por lo tanto, los archivos .COM no son reubicables y no contienen encabezado.

"MS-DOS siempre carga un archivo .COM empezando en el desplazamiento 100h (offset 100h) del 'Segmento del Programa', inmediatamente después del PSP (Ver: 'Program Segment Prefix', más adelante). La dirección del segmento del programa es almacenada en los cuatro registros de segmento del CPU y el IP es inicializado con el valor 100h. El SP (Stack Pointer) apunta a la parte más alta del segmento del programa y MS-DOS coloca 2 bytes con valor 00h en la parte superior del stack. Después de estos pasos el control es transferido a la instrucción que se encuentre contenida en la dirección CS:IP." [15:p.234]

Ahora se podrá entender por qué un programa .COM debe caber en un máximo de 64Kb menos 258 bytes y por qué la primera instrucción debe estar situada a 100h bytes del comienzo del segmento, 256 bytes (igual a 100h) corresponden al PSP del programa y los otros 2 bytes son el mínimo stack creado por MS-DOS al final del segmento.

En contraste, para cargar un programa .EXE, es necesario conocer bastante información acerca del mismo programa, como la posición y tamaño de sus segmentos de código, datos y stack. Dicha información, que está contenida en el encabezado del programa, es cargada a memoria con un desplazamiento de 100h en el segmento del programa, después del PSP. A continuación, esta información es utilizada para cargar el resto del archivo en direcciones específicas de memoria y para inicializar los registros de segmento del CPU con dichas direcciones. La ejecución comienza en CS:IP.

PROGRAM SEGMENT PREFIX (PSP).

"Para cargar cualquier programa a memoria, (inclusive para cargar COMMAND.COM) se debe utilizar la rutina EXEC del kernel de DOS (INT 4Bh). Antes de cargar el programa, EXEC localiza el segmento de memoria con la dirección más baja. Este segmento se denomina 'segmento de programa'. Empezando en el desplazamiento cero (offset 00h) de este segmento, EXEC construye el PSP (prefijo del segmento de programa). El PSP es un bloque de memoria de 256 bytes que sirve como área de comunicación entre MS-DOS y el programa que se está ejecutando." [15:p.227] Una vez que el PSP ya ha sido construido, el programa es cargado a memoria como ya se describió en la sección anterior.

La tabla 1-1 describe cada uno de los campos que forman el PSP. Varios de estos campos nunca han sido oficialmente documentados, ni por Microsoft ni por IBM, aunque la estructura utilizada para el PSP ha sido la misma desde la versión 2.0 del sistema

operativo. (Ver el glosario para una explicación del significado de la frase 'no documentado').

Tabla 1-1 Desglose del PSP [15 p 228]

Desplazamiento	Nombre del campo
00h-01h	Program Terminate Interrupción de MS-DOS para terminar programas (INT 20h)
02h-03h	Top of memory. Contiene la dirección del primer segmento de memoria que MS-DOS no asignó al programa en ejecución. Como regularmente MS-DOS da a cada programa toda la memoria disponible, este campo apunta al tope de la memoria principal (640Kb).
04h	Reservado para MS-DOS.
05h-09h	Function dispatcher. Esta es una llamada (long call) a la interrupción 21h del sistema operativo.
0Ah-0Dh	Terminate address. Contiene la dirección que recibirá el control cuando el programa termine de correr.
0Eh-11h	Ctrl-Break exit Contiene la dirección que recibe el control cuando Ctrl-Break es presionado.
12h-15h	Critical error exit. Contiene la dirección que recibe el control cuando ocurre algún error crítico.
16h-17h	PSP of parent. Contiene la dirección del segmento del PSP del programa anterior. Normalmente este campo contiene la dirección del PSP del COMMAND.COM, que es el que generalmente carga en memoria a los demás programas. (Campo no documentado).
18h-2Bh	File handle alias table. Estos 20 bytes son utilizados para almacenar los 'File Handles' (Manipuladores de Archivos) que pertenecen a este proceso. El tamaño de este campo limita el número de manipuladores a 20, aunque es posible ampliarlo utilizando los campos localizados en los desplazamientos 32h-33h y 34h-37h. (Campo no documentado).
2Ch-2Dh	Environment address. Contiene la dirección del segmento del entorno (environment) que le ha sido asignado al programa.
2Eh-31h	Reservado para MS-DOS.
32h-33h	Size of file handle alias table Almacena el tamaño del File handle alias table (Campo no documentado).

34h-37h	Address of the file handle alias table. Los primeros dos bytes contienen la dirección del desplazamiento de la file handle alias table, y los segundos la dirección del segmento.
38h-4Fh	Reservado para MS-DOS.
50h-52h	Function dispatcher return. Este campo contiene los bytes CDh 21h CBh, que son el código para una llamada a la interrupción 21h del sistema operativo, seguido de un FAR RET (far return)
53h-5Bh	Reservado para MS-DOS
5Ch-6Bh	File control block Usado si el primer argumento de la línea de comandos después del nombre del programa es el nombre de algún archivo.
6Ch-7Bh	File control block Usado si el segundo argumento de la línea de comandos después del nombre del programa es el nombre de algún archivo.
7Ch-7Fh	Reservado para MS-DOS
80h-FFh	Command tail, DTA El primer byte almacena la longitud de la línea de comandos y la línea de comandos es almacenada a partir del segundo byte. Este campo sirve además como área de transferencia para funciones de entrada/salida de los discos (Disc Transfer Area)

En resumen, el PSP contiene las direcciones de algunos servicios clave que el programa necesitará durante su ejecución (INT 20h y 21h); además contiene información que puede ser usada para determinar si el programa ha reservado suficiente memoria para su uso, y si es posible que se libere un poco de esta memoria. El PSP provee a su programa de una copia privada del entorno (Environment).

La dirección del segmento que contiene al PSP es conocida como el identificador de proceso (Process Identifier o PID). Cada programa que corre en la computadora es un 'proceso' y es identificado por un PID único. Así, aunque sólo un programa puede estar activo en un determinado momento, es posible que varios PSP, junto con los programas asociados a ellos, estén cargados en memoria al mismo tiempo, ya que cada PSP sirve para identificar a su programa como un proceso independiente.

Esto precisamente es lo que hace posible mantener activos varios programas en memoria, bajo un sistema operativo que no es multitareas. Esta característica del PSP también posibilita el desarrollo de 'Task Switchers'. Un análisis mucho más detallado de cómo manipular el PSP para implementar los programas TSR, se puede encontrar en los capítulos 3 y 4.

ASIGNACION DE MEMORIA.

Una de las tareas más importantes del sistema operativo es la administración de la memoria. DOS debe estar constantemente monitoreando qué porciones de memoria están siendo utilizadas y cuales están disponibles. Las tres tareas fundamentales que DOS realiza son: reservar memoria, liberar memoria, y cambiar el tamaño de los bloques de memoria que ya han sido reservados.

Cuando se reserva un bloque de memoria, el primer párrafo (paragraph = 16 bytes) de éste es utilizado por el MCB o Memory Control Block.

"El valor del primer byte del MCB puede ser 4Dh o 5Ah. Si es un 4Dh quiere decir que ese MCB es un miembro interno de la cadena que liga todos los MCB de un determinado proceso y si es un 5Ah, quiere decir que ese MCB es el último de la cadena.

El segundo y tercer bytes del MCB almacenan el PID del programa al que pertenecen los bloques de memoria.

El cuarto y quinto bytes, contienen el número de párrafos que le han sido asignados a ese bloque. Si se suma este número a la dirección del MCB actual, se obtiene la dirección del siguiente MCB en la cadena." [15:p.238]

Para programas convencionales, la manipulación directa de los MCB se debe dejar exclusivamente al sistema operativo. Pero cuando se desea implementar un programa residente, es necesario que el mismo programa realice esta manipulación, utilizando la información contenida en los MCB. La forma de acceder los MCB no está documentada oficialmente, pero se puede lograr a través de la función 52h de DOS. Esta función regresa un apuntador al primer bloque de memoria de la cadena asignada a un determinado proceso. Una vez que se tiene este apuntador, es posible recorrer la cadena en su totalidad.

El uso más frecuente que se le da a esta información, es para desinstalar el TSR de memoria. En el capítulo 4 se analiza el código que ejemplifica estos conceptos.

ACCESO A LOS RECURSOS DEL SISTEMA.

Los recursos del sistema en última instancia están determinados por el hardware que esté instalado en la computadora. Los dos elementos esenciales del hardware son el CPU y la memoria. Todos los demás componentes (monitores, teclados, unidades de disco, impresoras, modems, etc.) son considerados como periféricos, es decir, hardware externo a la computadora.

Todos los métodos para acceder los recursos de la computadora son implementados por medio de programas, y se pueden dar básicamente a tres niveles:

- 1) Acceso vía hardware (nivel más bajo).
- 2) Acceso por medio del BIOS.
- 3) Acceso por medio de los servicios de DOS.

En orden ascendente, cada nivel va filtrando los detalles del nivel anterior. Por ejemplo, el hardware de cada computadora es diferente según el fabricante, para evitar el

tener que reescribir un programa para cada tipo de hardware que existe, se utiliza el BIOS, el cual aísla al programa de todos los detalles para manipular el hardware. El programa de aplicación, entonces, ya no debe preocuparse del hardware específico, (tipos de circuitos integrados utilizados, sus registros, interacciones, etc.) sino únicamente de que existe una interrupción del BIOS con la cual puede controlar el subsistema deseado. En este caso se dice que el BIOS aísla las dependencias hacia el hardware.

Lo que se gana al filtrar los detalles de esta forma, es portabilidad del código hacia diferentes ambientes. Es por esto que IBM pretendía (con el diseño de la PC original) que el BIOS fuera el único medio de acceder el hardware y que las interrupciones de DOS fueran el único medio para acceder los recursos del sistema operativo. El primer objetivo nunca se logró debido a que el BIOS proporciona una funcionalidad incompleta o porque a veces es demasiado lento. El segundo sí se ha logrado en gran medida.

Como consecuencia de lo anterior, cada uno de los niveles puede o no apoyarse en algún nivel anterior para realizar sus funciones. Así, por ejemplo, Turbo Pascal tiene una instrucción para cambiar el modo de video. Esta instrucción puede utilizar la interrupción 10h del BIOS para realizar sus funciones o puede manipular el hardware (subsistema de video) directamente. Si se utiliza el BIOS, éste a su vez interactúa directamente con el hardware, que es el último nivel posible.

ACCESO VIA HARDWARE.

El hardware que más es accedido en forma directa, es la memoria de la computadora. (Memoria baja para acceso a diferentes datos que muestran el estado de la computadora, memoria de video, etc.). Por otra parte, los circuitos integrados que con mayor frecuencia son accedidos directamente son:

- Intel 8250: UART chip. (Universal Asynchronous Receiver Transmitter)
- Intel 8252 y 8253: Timer Chip
- Intel 8255: Peripheral Interface
- Intel 8259: Interrupt controler.
- Motorola 6845: CRT Controller

SideKick (el TSR más famoso) es un buen ejemplo de estos accesos. Este programa manipula directamente el subsistema de video, con el cual controla la posición y forma del cursor, utilizando los registros de datos del 6845; en forma análoga, controla el teclado por medio del 8255.

La memoria puede ser accesada utilizando las direcciones específicas que se deseen manejar. En cambio, los registros de los circuitos integrados deben ser accedidos por medio de 'puertos'. Un puerto es la ruta por medio de la cual el CPU y sus circuitos de soporte pueden intercambiar datos e información.

Se encuentran disponibles un total de 65536 (64Kb) puertos de entrada/salida. Por último, el área de puertos no forma parte de la memoria de la computadora.

Los programas de demostración incluidos en capítulos posteriores, ejemplifican ambas formas de acceso al hardware del sistema, tanto el acceso directo a memoria de video como el acceso a diferentes puertos para control del teclado, video y manipulación del 8259.

ACCESO POR MEDIO DEL BIOS Y DOS.

Los otros dos métodos de acceso a los recursos del sistema (BIOS y DOS) están basados en su totalidad en el uso de interrupciones.

Interrupciones.

La computadora debe responder a eventos asíncronos, es decir, eventos que ocurren a intervalos de tiempo aleatorios. Ejemplos de esto, son el presionar alguna tecla, acceder un disco, ejecutar comunicaciones, etc. Si el CPU tuviera que estar monitoreando continuamente cada uno de sus periféricos para ver si alguno o varios de ellos están siendo utilizados, y si lo están controlarlos, no le quedaría tiempo suficiente para realizar ningún otro proceso. Por lo tanto, para usar eficientemente el tiempo del procesador, se utilizan las interrupciones, las cuales permiten al CPU realizar sus tareas en forma normal y al mismo tiempo procesar los eventos asíncronos en el momento en el que éstos ocurren.

Una interrupción es una señal que alerta al procesador central de la computadora, de que alguna acción se debe llevar a cabo, por ejemplo, leer una tecla del buffer de teclado. Así, el procesador central de la computadora ejecuta instrucciones en una forma ordenada y secuencial, hasta que se genera una interrupción (IRQ Interrupt Request). Cuando esto sucede, el CPU deja a un lado lo que estaba haciendo, atiende la interrupción y después continúa con el proceso secuencial anterior, desde donde fue interrumpido.

Hablando en términos un poco más técnicos, una interrupción es el mecanismo básico por medio del cual algún periférico o programa solicita los servicios que requiere.

Cada interrupción tiene asignada un número único que la identifica, el cual se conoce como 'Tipo'. Actualmente los procesadores 80x86 de Intel, reconocen los tipos del 00h hasta el FFh, es decir, 256 interrupciones diferentes.

Como ya se había mencionado, en lo más bajo de la memoria convencional (empezando en la dirección 00h) se encuentra la 'Tabla de Vectores de Interrupción'. Esta tabla ocupa 1Kb, que es el espacio necesario para almacenar 256 direcciones de memoria (cada una de 4 bytes). Cada dirección contenida en esta tabla, se conoce con el nombre de 'Interrupt Vector' (Vector de Interrupción) y cada vector apunta a una 'Interrupt Service Routine' o ISR (Rutina de Servicio de Interrupción).

Por lo tanto, a cada interrupción le corresponde una ISR específica, la cual, se encarga de proporcionar los servicios que dicha interrupción tiene asignados. Cabe recalcar que no todas las interrupciones están implementadas; es decir, no todas tienen asignada alguna ISR que responda a su número.

El mecanismo de interrupción.

El procesamiento de una interrupción se realiza de la siguiente manera:

- 1) Cuando el CPU recibe el número de la interrupción a procesar, lo multiplica por 4 para tomar en cuenta el tamaño de cada vector. El valor resultante se usa como desplazamiento dentro de la Tabla de Vectores de Interrupciones.
- 2) El procesador lee los 4 bytes que se encuentran en el desplazamiento ya calculado.
- 3) Con estos 4 bytes se construye un apuntador. Este apuntador tiene el formato segmento:desplazamiento que es estándar para los procesadores 80x86.
- 4) Después de salvar los registros de banderas, CS e IP y de haber deshabilitado el procesamiento de interrupciones, el CPU carga el CS e IP con los valores del apuntador que construyó en el paso 3 y comienza la ejecución de la ISR correspondiente. El código de la ISR es responsable de preservar los valores de todos los registros del procesador que vaya a modificar; esto es necesario para poder continuar con el proceso anterior exactamente desde donde fue interrumpido. La ISR ejecuta su código, el cual termina restaurando los registros previamente salvados y regresando el control al proceso anterior. Generalmente esto último se realiza con una instrucción IRET (Interrupt Return), la cual restaura los registro de banderas, el CS y el IP que fueron salvados antes de llamar a la rutina de interrupción y esto a su vez habilita de nueva cuenta el procesamiento de interrupciones
- 5) Por último la ejecución del proceso primario se reanuda exactamente desde el punto de interrupción.

Tipos de interrupciones

Existen tres tipos de interrupciones que pueden ocurrir en una PC: (1) Interrupciones por hardware. (2) Interrupciones por software. (3) Excepciones del procesador.

Si una interrupción puede ser habilitada y deshabilitada bajo el control de software, se dice que esa interrupción es enmascarable (maskable interrupt). Este tipo de interrupción tiene menos prioridad que una no-enmascarable (nonmaskable interrupt). Las interrupciones no enmascarables se utilizan principalmente para manejar situaciones críticas del sistema, como la falla de energía o errores de paridad de memoria.

- Interrupciones por hardware.

Todos los dispositivos de entrada/salida generan eléctricamente este tipo de interrupciones para obtener la atención del procesador. En todos los procesadores 80x86 existen dos pins que son utilizados exclusivamente para propósitos de interrupción: el INTR (maskable interrupt) y el NMI (nonmaskable interrupt). Es por medio de estos pins que el procesador recibe las peticiones de interrupción.

- Interrupciones por software.

Intel proporciona la instrucción de interrupción INT, la cual permite generar cualquiera de los 256 tipos de interrupción por medio de software. Cuando una instrucción INT es ejecutada, el procesador la trata como si fuera una interrupción de hardware INTR, con la única diferencia de que una INT tiene mayor prioridad que cualquiera de las interrupciones de hardware (INTR y NMI).

- Excepciones del sistema.

Las interrupciones por hardware pueden ser de dos tipos: externas (descritas arriba) e internas. Las excepciones del sistema son precisamente interrupciones internas del procesador, es decir, el procesador se puede interrumpir a sí mismo. Por ejemplo, cuando se le pide al procesador que ejecute una división entre cero y ante la imposibilidad de realizar dicha operación, el CPU genera una excepción de algún tipo predeterminado. (En el caso de la división entre cero se genera una INTR 0).

Funcionalmente, el procesador trata a una excepción de igual forma que a una interrupción por software, pero el tipo de la interrupción no puede ser alterado, es decir, cada excepción ya tiene asignada un número específico de interrupción.

Aplicación práctica.

Ya que cada interrupción depende de una ISR específica para proporcionar sus servicios, es posible reemplazar o interceptar esta rutina para alterar la forma de procesar la interrupción.

Tanto el reemplazo como la interceptación se llevan a cabo de la misma manera: modificando el vector correspondiente en la tabla de vectores de interrupción para que apunte a nuestra rutina y no a la rutina original.

Cuando se reemplaza una ISR, se sustituye completamente el código que recibe el control cuando se genera dicha interrupción. Una rutina que muy a menudo es reemplazada, es la ISR de la interrupción 23h (Control Break). La rutina original asignada a esta interrupción, causa que el proceso actual sea abortado inmediatamente; la rutina de sustitución en cambio, (generalmente una sola instrucción IRET) hace que se ignore dicha interrupción para que el proceso no pueda ser cancelado al presionar Ctrl-C.

En contraste, al interceptar una interrupción no se sustituye el código de su ISR, sino que únicamente se le agrega código para que realice algún proceso extra. Para lograr esto, se utiliza el proceso conocido como 'Encadenamiento', el cual será descrito detalladamente en el capítulo 3.

Una interrupción frecuentemente interceptada por los TSRs, es la interrupción 09h (ISR del teclado). Así, toda tecla presionada es inspeccionada primero por el TSR, el cual busca entre las teclas presionadas una secuencia específica. Si el TSR encuentra esa secuencia de teclas se activa y si no, pasa las teclas (mejor dicho sus códigos) a la ISR original, para que ésta las procese normalmente como si nada hubiera pasado.

En resumen, las interrupciones juegan un papel clave en el desarrollo de los programas residentes por dos razones:

- 1) Es por medio de ellas que un programa residente toma el control de los periféricos de la computadora.
- 2) Las interrupciones posibilitan la activación de los programas que se encuentran residentes en memoria.

CAPITULO 2

PROGRAMAS TRANSITORIOS Y PROGRAMAS RESIDENTES

El objetivo de este capítulo, es el de diferenciar cada uno de los tipos de programas residentes en memoria

Para iniciar, se hace la distinción entre un programa transitorio y uno residente. Esta será la última vez en que se haga referencia a un programa convencional, ya que el resto de este trabajo estará dedicado a la discusión de los programas residentes.

A continuación se identifican los tres tipos de programas residentes en memoria que existen: los device drivers, los virus y los TSRs. Este capítulo se ocupará del estudio de los dos primeros tipos únicamente, lo que nos ayudará a reconocer la importancia que tienen estos programas, y a entender por qué es que ocupan lugares tan preponderantes dentro del mundo de las computadoras personales.

Los programas TSR, por ser el tema principal de este trabajo de investigación, serán analizados mucho más a fondo en los tres capítulos siguientes.

DESCRIPCIONES Y DIFERENCIAS.

"Cuando un programa es ejecutado, el sistema operativo reserva toda la memoria principal disponible para ese programa. La mayoría de los programas retiene toda la memoria que le fue proporcionada, hasta el final de su ejecución. Para terminar, estos programas hacen una llamada directa a la interrupción 20h o a la función 4Ch del DOS. (DOS es accedido mediante la interrupción 21h). Cualquiera de estas dos llamadas instruye al sistema operativo para que libere toda la memoria del programa en cuestión y así ponerla a disposición de otros programas." [15:p.268] Este tipo de programa se conoce como 'Programa Transitorio', ya que su uso de la memoria es estrictamente temporal, es decir, únicamente por el tiempo que dure su ejecución.

De igual manera, cuando un programa residente es ejecutado, se le asigna toda la memoria disponible. Sin embargo, un programa residente termina su ejecución con una

llamada a la interrupción 27h o a la función 31h del DOS. La interrupción 27h es la forma original de permanecer en memoria y data desde el 86-DOS. La función 31h de la interrupción 21h es la forma que Microsoft 'recomienda' para hacer que un programa quede residente en memoria. Además, la función 31h es preferible porque permite especificar la cantidad de memoria que debe permanecer reservada para el programa, es decir, no liberada y puesta a disposición de cualquier otro proceso.

Es así que, la principal diferencia entre un programa transitorio y uno residente es la forma como termina cada uno su ejecución, o mejor dicho, las llamadas que realiza cada programa para terminar su ejecución.

TIPOS DE PROGRAMAS RESIDENTES EN MEMORIA.

Los procesos que se acaban de describir, son válidos exclusivamente para los programas residentes en memoria convencionales o TSR.

Ahora bien, un programa residente, como ya se dijo, es todo aquel programa que permanece en memoria aún después de haber terminado su ejecución. Esta característica hace posible que, por medio de algún mecanismo, el programa pueda ser reejecutado.

Tomando esta definición de programa residente, se puede llegar a la conclusión de que existen otros dos tipos de programas, además de los TSRs, que pueden ser considerados residentes en memoria. Cada uno de estos programas tiene sus propias características distintivas, de acuerdo con su uso y las funciones que deba desempeñar, pero ambos comparten con los TSRs la característica de poder permanecer instalados en memoria hasta el final de la sesión de trabajo.

A continuación se detallarán las características de los dos tipos de programas residentes que no son TSRs en el estricto sentido de la palabra. El primer tipo está representados por los Device Drivers, y el segundo por los programas denominados 'Virus', en cualquiera de sus variedades.

DEVICE DRIVERS.

Como ya fue mencionado, el CPU y la memoria son los dos elementos de hardware esenciales en la computadora. Todos los demás componentes del hardware, son considerados externos a la computadora. Estos componentes externos son conocidos como 'peripheral devices' (dispositivos periféricos).

La comunicación entre un dispositivo periférico (o periférico simplemente, para abreviar) y la computadora, se debe llevar a cabo de acuerdo a un conjunto de estrictos lineamientos, los cuales están determinados por la computadora y el dispositivo periférico de que se trate. Por lo tanto, por cada periférico presente en el sistema debe existir un programa responsable de controlar la comunicación entre él y la computadora.

Este tipo de programas se conoce con el nombre de 'Device Driver' (manejador de dispositivos es la traducción más apropiada).

Para entender el propósito de los device drivers, es necesario recordar que todos los programas de aplicación solicitan servicios al sistema operativo por medio de llamadas a DOS (INT 21h), como abrir un archivo de disco o leer el contenido de éste. Muchos de estos servicios facilitan grandemente el desarrollo de los programas de aplicación. Por ejemplo, un programa solicita a DOS que escriba una cadena de caracteres a disco, de la misma forma en que lo haría para mandar dicha cadena a la impresora.

Esta facilidad que nos proporciona DOS, es conocida como 'device independent I/O', (procesos de entrada/salida independientes de los dispositivos) y es de suma importancia, ya que libera a los programas de aplicación de la necesidad de conocer e interactuar con los detalles del hardware de cada periférico.

Lo que aquí se ha llamado DOS, es en realidad un conjunto de componentes diversos: el kernel, el interprete de comandos, las extensiones del sistema operativo y las extensiones del kernel.

Los device drivers son las extensiones del sistema operativo, es decir, son programas que aumentan su funcionalidad. Esto lo logran añadiéndose a DOS mediante estructuras de datos, que él mismo crea y administra, como la lista ligada de device drivers. Cuando la liga entre DOS y los device drivers ha sido creada, se puede decir que el código de estos programas ya forman parte integral del sistema operativo que se encuentra residente en memoria.

En resumen, son precisamente los device drivers los que proporcionan la independencia de los dispositivos periféricos. Esto lo logran aislando al kernel de los detalles del manejo de cada periférico. Ahora se podrá apreciar la gran importancia que tienen estos programas residentes en memoria para el correcto y fácil funcionamiento de la computadora personal.

Tipos y estructura interna.

Existen básicamente dos tipos de device drivers: drivers integrados a DOS y drivers instalables.

"Los drivers integrados del DOS contienen el código para controlar los dispositivos utilizados durante el proceso de arranque, tales como la consola y los dispositivos de disco. El incluir este código en el sistema operativo simplifica en gran medida el proceso de boot.

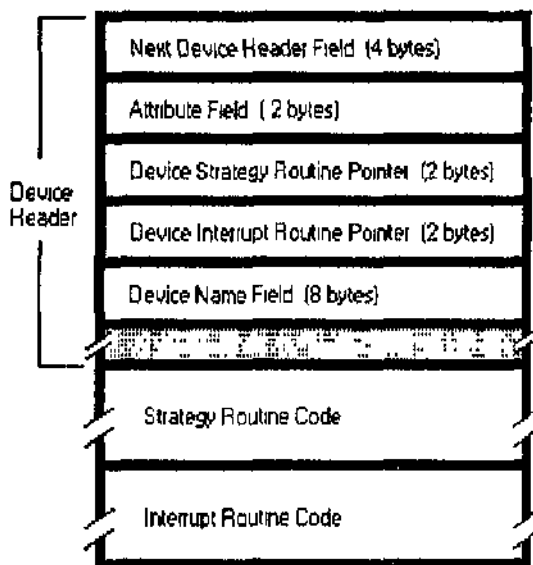
Los device drivers instalables, sirven para aumentar las capacidades de la computadora o para adaptarla a las necesidades de cada usuario. También ayudan a disminuir los requerimientos de memoria, ya que únicamente es necesario instalar los drivers que controlen el hardware específico que esté conectado a la computadora. Si no tengo un disco Bernoulli, por ejemplo, no necesito cargar su device driver. Estos device drivers son instalados mediante el uso de la instrucción DEVICE= del archivo CONFIG.SYS." [16:p.307]

Los dos tipos de device drivers, integrados o instalables, poseen la misma estructura interna, la cual consta de tres partes:

- 1) Device Header (encabezado de dispositivo).
- 2) Strategy Routine (rutina estratégica).
- 3) Interrupt Routine (rutina de interrupción).

El encabezado es una estructura de datos que el driver comparte con DOS. Las rutinas estratégica y de interrupción contienen el código y los datos que el driver utiliza para proporcionar sus servicios. La figura 2-1 muestra la estructura de un device driver.

Figura 2-1 Estructura de un Device Driver. [15 p 300]



Device Header.

El encabezado es una estructura de datos de 18 bytes de longitud, situada al comienzo de cada device driver. Esta estructura es para los drivers lo que el PSP es para cualquier programa, pues DOS lo utiliza para obtener información específica de cada driver. Esta estructura de datos está compuesta de los siguientes cinco campos: (1) Next Header Pointer, también llamado Next Device, (2) Device Attribute Field, (3) Strategy Routine Offset, (4) Interrupt Routine Offset, (5) Device Name.

- Next Header Pointer (4 bytes).

Este campo, almacena la dirección del encabezado del siguiente dispositivo. (Para ello se utiliza el formato segment:offset). Como se mencionó en el capítulo anterior, DOS crea una lista de los device drivers que están configurados en el sistema. Este primer campo sirve como apuntador al encabezado del siguiente dispositivo en la lista ligada. El último encabezado tiene un valor de -1 en cada byte de este campo.

- Device Attribute Field (2 bytes).

Este campo contiene un patrón de bits específico, el cual proporciona todas las características del dispositivo en cuestión. Este campo es muy importante ya que proporciona la información para distinguir entre dispositivos de bloque o de carácter. Además, especifica la capacidad de un driver para hacerse cargo de requerimientos especiales e identifica a los dispositivos que necesitan un tratamiento especial. La tabla 2-1 describe el significado de cada uno de los 16 bits que forman este campo.

Un dispositivo debe ser de bloque o de carácter. Los dispositivos de bloque son por lo general los discos. Éstos deben reconocer la estructura de disco impuesta por MS-DOS y las reglas para nombrar y organizar archivos. (La estructura de disco establece un lugar específico para grabar cierta información como la FAT, el árbol de directorios, etc.). Los dispositivos de carácter por otro lado, son los que manipulan cadenas de bytes, pero sólo un byte a la vez. Ejemplos de este tipo de dispositivos son el teclado, la pantalla y la impresora. (Ver: 'Funcionamiento de un device driver', más adelante).

Tabla 2-1 Interpretación de los bits del campo 'Device Attribute' del Device Header. [15:p 302]

Bit	Significado
0	1 - Es dispositivo estándar actual de entrada 0 - No es dispositivo estándar actual de entrada.
1	1 - Es dispositivo estándar actual de salida. 0 - No es dispositivo estándar actual de salida.
2	1 - Es dispositivo nulo actual. 0 - No es dispositivo nulo actual.
3	1 - Es reloj actual 0 - No es reloj actual.
4	1 - Dispositivo tiene implementada la INT 29h para E/S por consola rápidas 0 - Dispositivo no tiene implementada la INT 29h
6	1 - Dispositivo acepta los comandos. Get Logical Device y Set Logical Device. 0 - No tiene soporte para estos comandos
11	1 - Dispositivo acepta los comandos. Device Open, Device Close, Removable Media. 0 - No tiene soporte para estos comandos
12	1 - Dispositivo acepta el comando Output Til Busy 0 - No acepta el comando Output Til Busy
13	1 - Formato de discos diferente al formato IBM 0 - Formato de discos IBM (dispositivos de bloque)
14	1 - Si tiene soporte para IOCTL 0 - No tiene soporte para IOCTL
15	1 - Dispositivo de carácter. 0 - Dispositivo de bloque

- Strategy Routine Offset (2 bytes).

El tercer campo de esta estructura, almacena el desplazamiento de la rutina. El valor de este campo varía según la posición que ocupe la rutina estratégica dentro del código del driver.

- Interrupt Routine Offset (2 bytes).

Este es un campo análogo al anterior, pero contiene el desplazamiento de la rutina de interrupción.

- Device Name (8 bytes).

En los drivers de carácter, este campo almacena el nombre que el programador le haya asignado al mismo. Los bytes no usados por el nombre son llenados con caracteres en blanco. A ningún archivo de disco se le puede asignar el nombre de un dispositivo.

Los dispositivos de bloque no requieren de nombre, por lo tanto, este campo es usado para especificar el número de unidades que el driver puede controlar.

Strategy Routine, Interrupt Routine.

La única función de la rutina estratégica es la de salvar la dirección del 'Request Header' (el cual será descrito más adelante), después de lo cual regresa al sistema operativo. El verdadero trabajo, es decir, el código del driver que se encarga de proporcionar los servicios solicitados, está implementado en la rutina de interrupción. Esta rutina, recobra la dirección del request header, determina el comando a ejecutar e intenta proporcionar el servicio solicitado. De no ser posible, regresa un código de error.

Funcionamiento de los device driver.

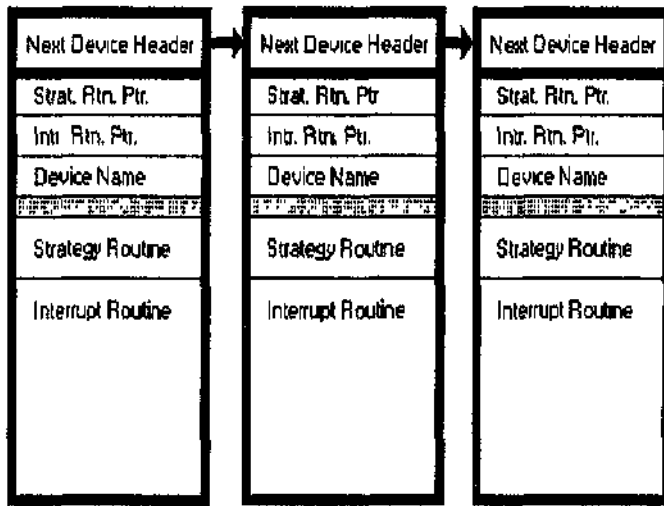
A continuación se describirán los tres procesos básicos que se deben realizar para utilizar los servicios que proporciona un device driver: (1) Instalación del driver. (2) Localización del driver ya instalado. (3) Llamado al driver para solicitarle algún servicio.

- Instalación

Ya se había mencionado que durante el proceso de arranque, DOS instala en memoria los device drivers estándar o integrados, los cuales están almacenados en el archivo IO.SYS. Como parte de este proceso, es creada una lista (simplemente ligada) la cual conecta a todos los drivers. El dispositivo nulo o NUL device es siempre el primer elemento de esta lista. A éste le siguen todos los demás drivers estándar como AUX, CON, LPT1, PRN, etc.

Después siguen los drivers instalables (que se hayan especificado en el CONFIG.SYS), los cuales son insertados en la lista, inmediatamente antes de los drivers integrados. La figura 2-2 muestra el estado de la lista después de haber insertado un driver instalable.

Figura 2-2 Lista de drivers en memoria [15:p.301]

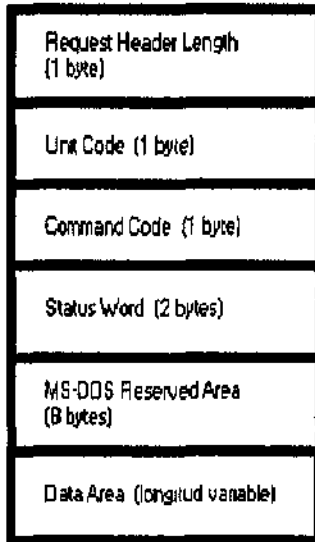


- Localización.

"Cuando un programa de aplicación necesita hacer uso de algún periférico, éste hace una llamada a la función apropiada de DOS, que a su vez busca a lo largo de la lista ligada de drivers, empezando con el NUL device, hasta que localiza el driver con el nombre que el programa solicitó. DOS siempre detiene la búsqueda cuando encuentra el primer nombre que coincida con el buscado; por lo tanto, si la lista contiene más de un driver para un dispositivo en particular, DOS siempre usará el que esté más cercano al principio de la lista. Una vez que DOS ha localizado el driver apropiado, éste debe ser informado del tipo de servicio que se requiere de él (lectura, escritura, etc.). Esta información es enviada al driver en la forma de un 'Driver Command' (comando de manejador). Existen veinte comandos diferentes, a cada uno de los cuales le es asignado un código numérico único." [15:p.303] La tabla 2-2 (ver más adelante) enumera los comandos disponibles.

Para enviar el comando, además de otros datos, DOS construye una estructura de datos llamada 'Request Header'. Esta estructura sirve como área de comunicación entre DOS y el device driver. La figura 2-3 muestra la estructura del request header.

Figura 2-3 Estructura del Request Header. [15:p.304]



El primer campo en el request header es un solo byte que almacena la longitud del request header. Como se verá más adelante, el largo de esta estructura de datos está determinado por el código de comando que haya sido enviado.

El segundo campo almacena un código de unidad (Unit Code). Este dato es válido únicamente para los dispositivos de bloque e identifica el dispositivo en particular que fue pedido. Por ejemplo, si la computadora tiene dos drives de disco instalados y ambos están controlados por el mismo device driver, entonces este campo determina cual drive ha de ser accedido.

El tercer byte del request header, contiene el código del comando a ejecutarse (Command Code o Driver Command).

El cuarto campo almacena un código de estado (Status Word), el cual indica situaciones específicas de error que el driver pudo haber encontrado cuando intentaba proporcionar los servicios solicitados. El driver construye un patrón de bits y lo almacena en este campo para comunicarse de vuelta con DOS. Cada uno de los bits de este campo tiene un significado especial, de acuerdo con el tipo de error que se haya generado.

Los siguientes 8 bytes están reservados por DOS y no están disponibles para ningún uso.

El sexto y último campo de esta estructura de datos es llamado área de datos (Data Area). El formato y el largo de esta área depende del comando que se haya especificado. Cada comando tiene asignados un conjunto específico de parámetros que deben ser llenados por el driver al efectuar sus procesos. Cuando el control es transferido del driver a DOS, éste espera encontrar cada uno de estos parámetros en localidades

específicas dentro del área de datos del request header. Es por esto que se necesita el primer campo para proporcionar la longitud real de request header. La tabla siguiente enumera los códigos de comandos que pueden solicitarse a un device driver.

Tabla 2-2 Códigos de comando para un device driver. [15:p.305]

Código	Comando
0	INIT
1	MEDIA CHECK
2	BUILD BPB (BIOS Parameter Block)
3	IOCTL INPUT
4	INPUT
5	NONDESTRUCTIVE READ
6	INPUT STATUS
7	INPUT FLUSH
8	OUTPUT
9	OUTPUT WITH VERIFY
10	OUTPUT STATUS
11	OUTPUT FLUSH
12	IOCTL OUTPUT
13	DEVICE OPEN
14	DEVICE CLOSE
15	REMOVABLE MEDIA
16	OUTPUT UNTIL BUSY
19	GENERIC I/O CONTROL
23	GET LOGICAL DEVICE
24	SET LOGICAL DEVICE

Para terminar con la sección referente a la localización del driver, es importante entender que la identificación del dispositivo apropiado, la selección del código de comando, la inicialización del request header y el llamado al driver correspondiente, son realizados por MS-DOS en respuesta a la llamada a alguna de sus funciones. El programador de un device driver no tiene que saber como es que DOS realiza estas funciones, ya que una vez hecha la llamada al driver, éste se encuentra en control y su código es el que entra en acción.

- Llamado.

Una vez que se ha localizado el driver indicado y se ha inicializado el request header, el sistema operativo almacena la dirección del device header (encabezado del dispositivo) en DS:SI y la dirección del request header en ES:BX. Al terminar estas operaciones, DOS llama a una rutina interna conocida con el nombre de 'Driver Dispatch Routine' (rutina para llamado de drivers). Todas las llamadas a un device driver pasan a través de esta rutina de DOS, excepto las solicitudes de inicialización de drivers. El listado 2-1 muestra el código de la rutina para llamado de drivers. (Este código fue desensamblado de PC-DOS 3.10)

Listado 2-1. Driver Dispatch Routine. [16.p.310]

```

;=====
; MS-DOS' Driver Dispatch Routine ;
; DS:SI - Device Header address ;
; ES BX - Request Header address ;
;=====
CallDriver proc near
    mov     ax,[si].Strat_ptr    ;Load AX with strategy offset
    mov     cs:Temp,ax          ;Save strategy offset
    mov     cs:Temp+2,ds        ; and segment
    call    cs dword ptr Temp   ;Indirect far call to
                                ; Strategy Routine

    mov     ax,[si].Int_ptr     ;Prepare for calling the
    mov     cs:Temp,ax          ; Interrupt Routine
    call    cs dword ptr Temp   ;Far call to Interrupt Routine

    ret                                ;Return to caller
CallDriver endp

```

Como lo muestra el listado, la primera acción que realiza esta rutina, es llamar a su vez a la rutina estratégica del driver.

La única función de la rutina estratégica es la de salvar el segmento y el desplazamiento del request header en dos variables locales del driver. Es responsabilidad del programador definir estas variables dentro del código del driver y de inicializarlas cuando el driver es llamado. Después de salvar el apuntador, la rutina estratégica regresa el control a MS-DOS, el cual inmediatamente llama a la rutina de interrupción.

La rutina de interrupción es la parte principal del device driver. La primera porción de esta rutina, consiste en una tabla que contiene las direcciones de las diferentes subrutinas. Estas direcciones son utilizadas por la rutina de interrupción para saber a qué parte del código hay que transferir el control, de acuerdo con el código de comando que se haya especificado en el request header. Este tipo de tablas se conocen con el nombre de 'Jump Tables'. Un ejemplo de como implementarlas, se puede encontrar en el programa de demostración TSRTMPLT.ASM (ver: apéndice H).

Como respuesta al llamado de DOS, la rutina de interrupción identifica el código de comando, busca en la tabla la subrutina adecuada y le transfiere el control. Esta subrutina procesa el comando y utiliza los campos Status Word y Data Area del request header para almacenar información relacionada con el resultado de este llamado al driver.

Cuando la rutina de interrupción completa su trabajo, el control es regresado a la rutina Driver Dispatch, la cual a su vez transfiere inmediatamente el control de vuelta a la función de DOS que la invocó. Es aquí, donde DOS determina cual fue el resultado del llamado. Esto lo hace inspeccionando el contenido del request header. Por último DOS pasa estos mismos resultados al programa de aplicación que inicialmente solicitó acceso al dispositivo periférico y le regresa el control.

Para terminar la descripción de estos programas residentes, se proporciona el listado del esqueleto de un device driver. Este código sirve únicamente como modelo, ya que es necesario expandirlo si se desea obtener un device driver que realice alguna función útil.

Listado 2-2. Esqueleto de un Device Driver. [15:306]

```

;=====
; Device Driver Skeleton ,
;=====
CSeg segment para public 'code'

skeleton proc far
    assume cs CSeg, ds.CSeg, es.CSeg

BEGIN,
;--- Device Header ---
Next_dev dd -1 ;Seg-Offs of next header
Attrib_filed dw 8000h ;Character device
Strat_ptr dw STRATEGY ;Offset of strategy routine
Int_ptr dw INTERRUPT ;Offset of interrupt routine
Nmae db 'DEMO' ;Name of driver
;--- variables ---
; Store address of Request Header
RHeader_off dw ?
RHeader_seg dw ?
;-----
; This area can be used for more variables as needed
;-----
;--- Strategy Routine ---
STRATEGY ;Call first by DOS
    mov cs,RHeader_off,bx ;Save address of Request Header
    mov cs,RHeader_seg,es
    ret ;Return to MS-DOS
;--- Interrupt Routine ---
INTERRUPT
    push ds ;Save MS-DOS registers
    push es
    push ax
    push bx
    push cx
    push dx
    push si
    push di

; Retrieve Command Code from Request Header
    mov al,es[bx+2]
;----- Real work comes here -----
; Using the Command Code in AL, branch to the
; appropriate routine and process the command.
; This area can be coded as the programmer sees
; fit, as long as the requirements for each
; command are met.
;-----
; After the command is processed, set done bit in the
; Status Word field of the Request Header and restore
; MS-DOS registers.
    mov es word ptr [bx+3],0100h ;Set done bit
    pop di ;Restore MS-DOS registers
    pop si
    pop dx
    pop cx
    pop bx
    pop ax
    pop es
    pop ds
    ret ;Return to MS-DOS
;--- End of Device Driver ---
skeleton endp
CSeg ends
end BEGIN

```

VIRUS.

Algunas de las implementaciones de los programas 'Virus' (no todas), tienen la capacidad de quedar residentes en memoria. Para lograr esto, se valen de técnicas similares a las utilizadas por los programas TSR, de ahí que el tema de los virus haya sido incluido en este trabajo de investigación. Sin embargo, no es el propósito de esta sección el realizar un análisis exhaustivo de estos programas, sino únicamente dar una breve semblanza de ellos.

(Nota: La palabra virus se utiliza en forma genérica para nombrar a cualquier variedad de programa autoreplicable. Más adelante en esta sección se hace una clasificación de los virus, en la que se le da un nombre específico a cada variante)

Características principales.

El número específico de virus, incluyendo a cada una de sus variantes, es muy difícil de precisar, ya que cada mes aparecen un promedio de 30 virus nuevos o modificados. De cualquier modo, existen muchos tipos de virus, cada uno de los cuales tiene sus propias características distintivas. Los hay que se comportan como Caballos de Troya, como Bombas de Tiempo o como Worms (gusanos). Lo que diferencia a cada uno de estos tipos de virus, es la forma en la que se introducen a las computadoras y el proceso que utilizan para activarse.

Por ejemplo, los Worms son de los virus más sofisticados pues se introducen a la máquina a través de programas contaminados, y una vez dentro de ésta permanecen siempre activos desplazándose por la memoria y los discos del sistema. Esto lo logran autocopiándose a otro lugar y después borrándose del anterior, por supuesto borrando toda la información que encuentran a su paso.

Aún así, los virus comparten ciertas características que los identifica como tales, es decir, para que un programa sea considerado virus, debe reunir un mínimo de tres requisitos, los cuales se listan a continuación:

1) Los virus son programas y poseen todas las características de un programa. Por lo tanto, un virus debe ser ejecutable. Además, como cualquier programa, el virus no puede seguir activo cuando se ha cortado la corriente a la computadora. (Los virus, a pesar de que son ejecutables, no dependen del usuario para iniciar su ejecución).

2) Aparte de ser programas extremadamente pequeños, tal vez la característica más significativa de un virus sea su capacidad de autoreplicación, es decir, poseen código que puede copiarse a sí mismo. (Reproducirse, dirían algunas personas, para hacer más explícita la analogía con los virus biológicos).

3) Los virus tienen la capacidad de modificar algún área del disco (flexible y/o duro).

Las tres áreas del disco generalmente modificadas por los virus son las siguientes: el sector de carga de un disco (descrito en la sección Proceso de Arranque del capítulo

primero), la Tabla de Particiones del disco, y los archivos ejecutables. También es posible que modifiquen archivos de datos, pero esto, generalmente sucede sólo en la fase destructiva. (El virus DEN ZUK, es capaz de infectar archivos de datos).

Las áreas y/o archivos modificados son los que posibilitan la autoreplicación. Por ejemplo, un archivo ejecutable infectado continuará infectando otros archivos cada vez que sea ejecutado.

El siguiente listado muestra el código de un archivo batch, el cuál reúne todas las características de un virus. Es ejecutable, es muy pequeño, es autoreplicable (se puede copiar a sí mismo), puede modificar otros archivos y ,además, los archivos modificados adquieren la capacidad de modificar ellos mismos a otros archivos.

Listado 2-3. VIRUS BAT

```

@echo off
rem !!! Warning.
rem This batch file contaminates all batch files in the
rem current directory
ctty nul
for %%a in (*.bat) do copy %%a + virus bat
ctty con

```

Nótese que en los requisitos de estos programas nunca se menciona que el virus deba tener capacidad destructiva. El tener capacidad destructiva es únicamente una característica muy común, pero no es necesaria para catalogar a un programa como virus. Existen virus no destructivos, así como programas convencionales que intencionalmente causan daños irreparables a la información. Otras dos características bastante comunes entre los virus actuales, pero que no son requisito indispensable, son las siguientes:

1) Los virus que pueden modificar áreas del disco por lo común también pueden reconocer los discos o archivos que ya han sido infectados, para así evitar contaminarlos más de una vez.

2) La característica de permanecer residente en memoria se encuentra en aproximadamente el 60% de los virus. Estos virus son conocidos como 'Memory Resident Infectors' o MRI.

Clasificación y funcionamiento.

Por la forma de contagiar al sistema, la Computer Virus Industry Association ha clasificado a los virus de la siguiente forma: (1) Infectores del área de carga inicial. (2) Infectores del sistema operativo. (3) Infectores de programas ejecutables.

A continuación se describe brevemente el funcionamiento de cada una de las tres clases de virus, haciendo énfasis en las implementaciones que son capaces de permanecer residentes en memoria.

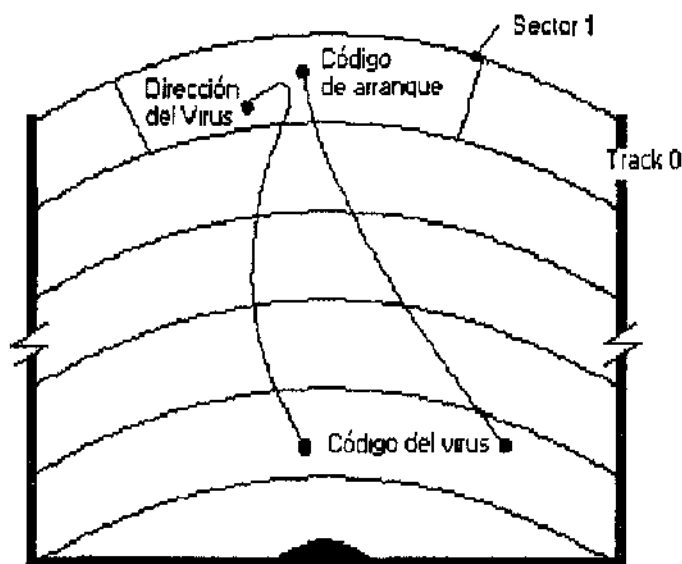
- Virus infectores del área de carga inicial.

Estos virus, ya sea que infecten el sector de carga, la tabla de particiones o ambos, toman el control de la máquina desde el mismo proceso de arranque.

Como se encuentran alojados en el sector 1 del disco, su código recibe el control inmediatamente después de haberse ejecutado la rutina POST, es decir, aún antes de que el sistema operativo se encuentre en memoria. (Ver: 'Proceso de Arranque', capítulo 1)

En realidad en el sector 1 únicamente se encuentra una dirección que apunta hacia el código del virus, en lugar de apuntar al código de carga original. Una vez que el virus se ha instalado en memoria, regresa el control al código de carga. La siguiente figura muestra gráficamente este proceso.

Figura 2-4. Infección del sector de carga



Al instalarse el virus en memoria generalmente intercepta una o varias interrupciones, por medio de las cuales efectuará sus procesos destructivos o continuará contaminando otros discos. Cabe señalar que el interceptar interrupciones es práctica común en todas las clases de virus, no sólo en los infectores del área de carga. Algunos de los virus más conocidos en el mundo de las PC, pertenecen a esta clase, entre ellos se encuentran: el Brain, el Ping Pong (virus de la pelotita), y Miguel Angel (Michaelangelo).

El famoso virus de la pelotita, por ejemplo, intercepta la interrupción 08h (Timer) para rebotar en 'paralelo' con la aplicación que esté corriendo en ese momento.

Miguel Angel, por su parte, infecta tanto el sector de carga como la tabla de particiones. Este virus se instala en lo más alto de la memoria principal (en el límite de los 640 Kb) y ocupa únicamente 2048 bytes. Entre las interrupciones que este virus intercepta está la 12h, la cual reporta la cantidad total de memoria principal que se encuentra instalada en la computadora.

Este tipo de virus conserva el control de la computadora, hasta que sea reinicializada desde algún disco no contaminado. Existen algunos virus (en particular el virus DEN ZUK), que resisten la reinicialización por medio de las teclas Ctrl-Alt-Del, lo que quiere decir que aun cuando se arranque la computadora desde un disco no contaminado, el virus permanecerá en memoria y podrá seguir infectando los discos no protegidos.

- Virus infectores del sistema operativo.

Estos virus generalmente contaminan el intérprete de comandos del sistema operativo (COMMAND.COM). Desde este programa, el virus toma el control de la computadora, ya que se adueña de comandos tales como el DIR, COPY, o ERASE. Teniendo el control de estos comandos, el virus fácilmente puede contagiar otros discos y dañar información. Dark Avenger, LeHigh y DataCrime son ejemplos de virus infectores del sistema operativo.

- Virus infectores de programas ejecutables.

Esta clase de virus se adiciona al final de algún programa ejecutable. Según el virus de que se trate, puede infectar archivos .COM, .EXE, .OVL, OVR (las dos últimas son overlays). Cuando un virus infecta un programa en disco, generalmente provoca un aumento en el tamaño del mismo, es por esto que muchos virus tiene por nombre un número, el cual representa la cantidad de bytes que le agregan al archivo que infectan).

Estos virus funcionan en forma muy semejante a los infectores del sector de carga, ya que colocan una dirección al comienzo del programa, la cual apunta al código del virus y no al inicio del programa. Después de que el programa ya se instaló en memoria, el control es transferido a la dirección original para el que proceso continúe normalmente.

El virus Jerusalén es un ejemplo muy conocido de este tipo de programas. Este virus, reemplaza la función 4Bh (EXEC) del sistema operativo, lo cual le permite tomar el control de cualquier programa al ser cargado a memoria.

Para concluir esta sección, despejaré una duda que he tenido desde que tengo conocimiento de los virus.

¿ Es posible que un virus cause daños al hardware del sistema ? La respuesta, según John McAfee (y quién mejor que él para confirmarlo) es 'sí', sin importar la clase de virus de que se trate. Un ejemplo que se me ocurre, (espero no dar malas ideas a nadie) es dañar el monitor de la computadora por medio del chip 6845 (o equivalente). Por medio de este chip se pueden alterar las frecuencias a las que trabaja el monitor, hasta llegar al punto donde el daño sea permanente.

PROGRAMAS RESIDENTES NO TSRs.

Los dos tipos de programa residente que se acaban de describir no pueden ser considerados como TSRs. La principal razón para no hacerlo, es la forma 'no interactiva' en la que cada uno de ellos es cargado en memoria.

En ninguno de estos dos casos el usuario interviene directamente en el proceso de cargado, sino que este proceso es iniciado por algún evento, diferente a un comando explícito desde el prompt del sistema para ordenar la instalación del programa en memoria.

Es verdad que existen device drivers que pueden ser instalados y desinstalados desde la línea de comandos, pero esto generalmente se realiza con la ayuda de alguna utilidad suplementaria, es decir, estos programas no poseen el código específico que les permita instalarse y removerse de memoria a sí mismos.

Por su parte, los virus sí poseen el código necesario para instalarse en memoria. Mejor dicho, poseen código para auto-instalarse en memoria, ya que el proceso de instalación es realizado sin el conocimiento, y mucho menos el consentimiento, del usuario.

Como ya se había mencionado, los últimos tres capítulos de este trabajo estarán dedicados exclusivamente al estudio de los TSRs. En este capítulo se analizarán las características que debe tener un TSR, así como su funcionamiento interno, ambos desde el punto de vista teórico.

Para iniciar este capítulo, se hará una clasificación de los TSRs de acuerdo con su forma de operación. Esto se hace con el propósito de dar un orden a la presentación de los programas de demostración que serán analizados en los siguientes capítulos.

En segundo lugar, se describirán detalladamente cuatro conceptos que son fundamentales para el desarrollo de un TSR. Después, cuando se describa el funcionamiento de estos programas, se hará evidente la relación que existe entre estos cuatro conceptos y los componentes que integran el código de un TSR.

Por último, se estudiarán los requisitos que debe reunir un TSR estable, capaz de coexistir adecuadamente con otros programas y con el sistema en general, en donde esté corriendo. Algunas de estos requisitos son indispensables para el buen funcionamiento del TSR, y otros son únicamente 'deseables', es decir, no es obligatorio incluirlos en el código del programa. Claro está, entre más características deseables tenga un TSR, más robusto será y, por lo tanto, menos probabilidades tendrá de fallar.

CLASIFICACION DE LOS TSRs.

El clasificar a los programas TSR es una tarea bastante subjetiva, por lo tanto, cada programador puede tener su propia clasificación. La forma de clasificar que se presenta a continuación ordena los TSR según su complejidad, es decir, de los más simples a los más sofisticados.

De acuerdo a la forma en que realizan sus funciones, los TSRs se pueden clasificar en: pasivos e interactivos. Esta clasificación no es absoluta, es decir, pueden

desarrollarse TSRs que en un momento dado se comporten como pasivos y en otro momento como interactivos. La frontera que diferencia un TSR pasivo de uno interactivo es, por lo tanto, flexible.

PASIVOS.

Un TSR pasivo, desempeña sus funciones en forma transparente para el usuario y no existe interacción directa con él después de que ha sido cargado en memoria. Este tipo de TSR generalmente aumenta las capacidades del sistema operativo o compensa alguna deficiencia del mismo.

Un programa que aumenta las capacidades del sistema operativo es FASTOPEN.EXE, el cual almacena información de los archivos que han sido previamente accedidos para que en caso de que se deseen acceder nuevamente, el proceso se realice con mayor rapidez.

Un ejemplo de un programa que compensa una deficiencia del sistema operativo es el programa APPEND.EXE, el cual proporciona servicios análogos a los de la instrucción PATH, pero APPEND rastrea archivos de datos, no ejecutables.

Un TSR pasivo puede ser activado de diferentes maneras, de las cuales las que más frecuentemente se utilizan son:

1) No activación. Esto parece contradictorio. ¿Cómo es posible que un programa realice cualquier función sino está activo? Pero no hay tal contradicción: un programa puede reemplazar un área de datos del hardware o del sistema operativo para hacerla más funcional. También puede mantener residente en memoria la información requerida por algún proceso en especial.

Tal es el caso del programa GRAFTABL.COM que se distribuye con MS-DOS, el cual únicamente carga a memoria una tabla con los caracteres del ASCII extendido, para que éstos puedan ser desplegados en pantalla cuando se utiliza la tarjeta CGA en modo gráfico. Sin GRAFTABL, la tarjeta CGA no puede desplegar caracteres ASCII extendidos. Este programa no necesita ser activado ya que su único propósito es el de mantener en memoria dicha tabla. El programa de demostración XKBDUF.PAS es un ejemplo de un TSR que no necesita ser activado. Este programa amplía el buffer del teclado proporcionando una área de memoria 16 veces más grande.

2) Activación por medio de una interrupción de hardware. Muy frecuentemente se utiliza la interrupción 08h, que corresponde al reloj del sistema (Timer). El programa de demostración CLOCK.ASM proporciona un ejemplo de esta técnica.

3) Activación por medio de una interrupción de software. Estos TSRs no son muy comunes y generalmente se activan por medio de las interrupciones 60h a 67h. Estas interrupciones son las únicas (de las 256 posibles) que están oficialmente disponibles para uso exclusivo de programas escritos por el usuario. El programa de demostración JTSRTP.PAS, aunque no es un TSR pasivo, utiliza estas interrupciones para establecer una línea de comunicación con el código residente en memoria.

Como se puede ver, en ningún caso el usuario interviene directamente en la activación del TSR; es decir, la activación se da por algún evento diferente a un comando explícito del usuario.

INTERACTIVOS.

Las tres características básicas que distinguen a los TSRs interactivos son:

1) Después de haber sido cargados en memoria, monitorean constantemente el teclado en busca de una secuencia de teclas específica que les indique que deben activarse. En otras palabras, esta clase de TSRs son activados únicamente mediante un comando explícito del usuario. Cabe aclarar que al presionar una tecla se genera una interrupción 09h, por lo tanto, en última instancia, estos TSRs también son activados por medio de interrupciones (generalmente de hardware), pero esta interrupción fue generada con la intervención directa del usuario.

2) Generalmente, después de haber recibido la orden de activación, el TSR toma el control del subsistema de video (memoria de video, cursor, color, etc.) para poder desplegar en pantalla algo que le indique al usuario que ya ha sido activado. En realidad el TSR también puede tomar el control de otros periféricos como el teclado, el reloj del sistema, los dispositivos de disco, etc., pero el más evidente es la pantalla.

3) Después de que el TSR ha tomado el control, acepta comandos del usuario para realizar sus tareas y permanece activo hasta que el usuario da la orden explícita para desactivarlo. Esta última característica es la que refleja el comportamiento interactivo del programa.

Los programas de demostración POPCLOCK.ASM, JTSRTP.PAS, SAVESCR.PAS y TSRTMPLT.ASM (en orden ascendente de complejidad), son ejemplos de TSR interactivos.

Como se dijo anteriormente, existen programas que poseen características de los TSRs pasivos y de los interactivos, lo cual dificulta su clasificación. Tal es el caso de los programas de demostración F11.PAS y BLANKER.ASM.

F11.PAS es un 'Interrupt Handler' (Manejador de Interrupción). Este programa intercepta la INT 09h y modifica la forma como se procesan las teclas presionadas, bajo circunstancias específicas. El TSR es activado por medio de la tecla F11, es decir, con intervención directa del usuario, por lo tanto, se debería clasificar como un TSR interactivo. Por otro lado, el programa también tiene características de TSR pasivo, ya que no toma el control de ningún periférico (aparte del teclado), ni tampoco despliega nada por sí solo, es decir, todo su proceso es transparente para el usuario.

BLANKER.PAS es un 'Screen Saver' activado por medio del reloj del sistema, es decir, sin intervención directa del usuario, por lo que se debe considerar como un TSR pasivo. Pero para restaurar la pantalla original, el usuario debe presionar alguna tecla, es decir, el usuario da la orden específica para que el TSR realice un proceso y después regrese el control. Estas son características de un TSR interactivo.

OTROS CRITERIOS DE CLASIFICACION.

Los TSRs también se pueden clasificar dependiendo de si hacen o no llamadas al sistema operativo, una vez que ya se encuentran residentes en memoria.

Cualquier TSR, ya sea pasivo o interactivo, puede hacer llamadas a DOS. Estas llamadas generalmente se hacen con el propósito de acceder archivos en disco.

El hacer llamadas a DOS aumenta la complejidad del programa. Por lo tanto, es posible que un TSR pasivo que haga llamadas a DOS sea más complejo que uno interactivo que no las haga.

Como ejemplo, se pueden comparar el programa PRINT.EXE de Microsoft, (el cual llama a DOS para abrir y leer archivos) con el TSR interactivo de demostración POPCLOCK.ASM. Aunque PRINT es pasivo, su código seguramente es bastante más complejo.

Sin embargo, como regla general, el código de un TSR interactivo es más complejo y sofisticado que el de uno pasivo.

De los programas de demostración, los que hacen llamadas a DOS, (o que contienen el código específico que lo permitiría, si así se deseara) son: JTSRTP.PAS-SAVESCR.PAS y TSRTMPLT.ASM.

A continuación se enumeran otros criterios que se pueden utilizar para clasificar a los TSRs:

- 1) El proceso utilizado por el TSR para quedar residente en memoria (INT 27h o INT 21h función 31h). De los programas de demostración el único que utiliza la INT 27h es CLOCK.ASM.
- 2) Que el TSR tenga o no la opción para desinstalarse de memoria, sin tener que reinicializar la máquina. JTSRTP.PAS-SAVESCR.PAS, TSRTMPLT.ASM ofrecen esta opción.
- 3) Que el TSR tenga o no código especial que le permita hacer uso de memoria expandida. Únicamente TSRTMPLT.ASM tiene capacidad de hacerlo.

CONCEPTOS FUNDAMENTALES PARA DESARROLLAR UN TSR.

El codificar un TSR es como una carrera de obstáculos, ya que es necesario salvar todas las restricciones que el sistema operativo antepone para el desarrollo de este tipo de programas. Esto es cierto por dos razones:

- 1) DOS es un sistema operativo que sólo puede correr un proceso a la vez. El pedirle a DOS que controle más de un programa simultáneamente está más allá de las capacidades del sistema operativo. Por lo tanto, es necesario compensar esta deficiencia para desarrollar un TSR funcional.

(Nota: "Las funciones del sistema operativo que posibilitan el desarrollo de TSR fueron originalmente diseñadas para instalar en memoria device drivers y manejadores de interrupción, no para desarrollar TSR interactivos". [22:p.50])

2) Para conseguir un TSR estable, el código del programa debe hacer uso de funciones del sistema operativo que no están oficialmente documentadas ni por Microsoft ni por IBM. Hoy en día esto ya no es un obstáculo tan grande, ya que en 1991 cuando Microsoft liberó la versión 5 de MS-DOS, también documentó las funciones más utilizadas para el desarrollo de TSRs. Esta documentación se puede encontrar en el libro 'Microsoft MS-DOS Programmer's Reference'. No obstante, varias funciones necesarias para implementar TSRs siguen sin estar documentadas.

El estudio de las restricciones asociadas con el desarrollo e implementación de los TSRs es precisamente el tema de esta sección. A continuación, se describirán cuatro áreas de consideración, las cuales son críticas para el desarrollo de un TSR estable:

- 1) Modificación de la tabla de vectores interrupción.
- 2) Problema de reentrada.
- 3) Acceso a archivos de disco desde el TSR
- 4) Manejo de errores críticos desde el TSR.

MODIFICACION DE LA TABLA DE VECTORES DE INTERRUPCION

Como se recordará, cada interrupción tiene asociada un vector de interrupción y una rutina (ISR) que proporciona los servicios de esa interrupción. El vector de interrupción apunta a la rutina ISR que le corresponde a cada interrupción en particular. Todos los vectores están almacenados en la tabla de vectores de interrupción del sistema operativo y cada uno consta de un par segment:offset (segmento:desplazamiento) con el cual se construye la dirección de su ISR.

El código de las ISRs, lo puede proporcionar el BIOS, DOS o un programa de aplicación escrito por el usuario, en nuestro caso, un TSR. Las dos técnicas utilizadas por los TSRs para tomar el control de las interrupciones y así poder proporcionar el nuevo código que se hará cargo de procesarlas son: el reemplazo y la intercepción.

En ambos casos, el TSR debe sustituir la dirección de la ISR que se encuentra en la tabla de vectores, con la dirección de otra rutina. Esta otra rutina forma parte del código del programa que permanecerá residente en memoria.

El sistema operativo ofrece dos funciones que permiten manipular las direcciones contenidas en la tabla de vectores de interrupción. La función 35h, con la cual se puede obtener la dirección de la ISR de una determinada interrupción, y la función 25h, que permite modificar dicha dirección.

Ya sea que se vaya a utilizar reemplazo, intercepción o ambas, es requisito indispensable que el TSR salve la dirección original que se encuentra almacenada en la tabla de vectores antes de modificar dicho valor. Esto se debe realizar por las siguientes tres razones:

1) Si el TSR puede ser desinstalado, se deberán restaurar las direcciones originales en la tabla de vectores antes de remover al TSR de memoria, esto con el fin de no dejar interrupciones sin ISRs asociadas.

2) El proceso de reemplazo que utilizan los TSRs es generalmente temporal, es decir, las ISRs se reemplazan después de que el TSR ha sido invocado y únicamente mientras está activo, pero cuando el TSR es desactivado y únicamente permanece latente en memoria, se restauran las ISRs originales.

(3) El proceso de interceptación necesita de las direcciones originales para implementar el mecanismo denominado 'Chaining' (encadenamiento), el cual evita romper la cadena de ISRs que dan servicio a una interrupción determinada.

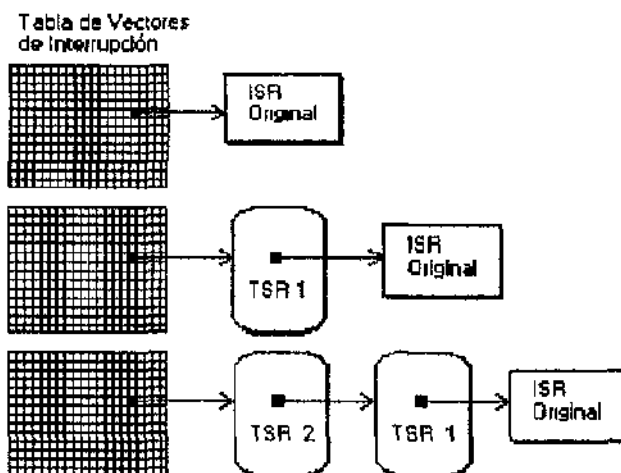
Chaining (Encadenamiento).

Encadenamiento es el proceso que se sigue para llamar a la rutina de interrupción anterior desde el interior de una nueva ISR, es decir, una ISR que llama a otra ISR. Este proceso generalmente se implementa utilizando el apuntador (a la ISR anterior) que debió haber sido salvado durante el proceso de inicialización del TSR.

Este mecanismo debe ser implementado para que cualquier número de TSRs puedan estar residentes en memoria y todos puedan reejecutarse sin importar el orden en el que sean instalados. En otras palabras, el encadenamiento es absolutamente necesario si se desea que el TSR interactúe y coexista pacíficamente con cualquier otro programa que esté corriendo, o que esté cargado en memoria al mismo tiempo.

La figura 3-1 muestra el proceso de encadenamiento, desde que se tiene la tabla de vectores original hasta después de que dos TSRs, que capturan la misma interrupción, han sido cargados en memoria.

Figura 3-1 Encadenamiento de ISRs. [15.p.274]



REENTRANCE (REENTRADA).

DOS es un sistema operativo que no puede ser 'reentrado'. Esto simplemente quiere decir que no se pueden ejecutar dos funciones de MS-DOS al mismo tiempo.

Siempre que un programa realiza una llamada a alguna función de DOS, (mediante la INT 21h) se dice que se ha 'entrado' a DOS. Si alguna interrupción es generada mientras nos encontramos dentro de DOS, y si esta interrupción realiza una nueva llamada, se dice que DOS ha sido 'reentrado'. Una reentrada puede causar que el sistema operativo 'true'ne'. Perdón por el término, pero esta palabra describe perfectamente los que le puede suceder a DOS si es reentrado en un mal momento.

Bajo condiciones normales, es decir, sin cargar TSRs a memoria, el problema de reentrada no existe. Si un solo programa está corriendo en la computadora, sólo una llamada a DOS puede ser realizada a la vez. Pero si agregamos TSRs el panorama se vuelve más complicado, ya que es totalmente posible que un TSR haga una llamada al sistema operativo al mismo tiempo que otra llamada, hecha por el proceso principal, está siendo atendida. El resultado, como ya se explicó, es el desastre.

Al principio de esta sección se dijo que DOS no puede ser reentrado. Esto no es totalmente cierto, ya que sí es posible reentrar a DOS, pero sólo en forma parcial. Por lo tanto, algunas precauciones deben ser tomadas para asegurar que DOS será reentrado únicamente bajo ciertas circunstancias, las cuales deben ser estrictamente controladas por el TSR.

Para entender mejor lo que es el problema de reentrada, así como la forma de controlarlo, es necesario conocer el funcionamiento de los stacks internos del sistema operativo.

Stacks internos de MS-DOS.

DOS mantiene tres stacks internos que utiliza para procesar las llamadas a sus funciones.

"El stack número 1 es utilizado para procesar las llamadas a las funciones 00h y de la 0Dh en adelante. Este stack también se utiliza para procesar las interrupciones 25h (Absolute Disk Read), 26h (Absolute Disk Write) y la 28h.

El stack número 2 es utilizado para procesar las llamadas a las funciones 01h hasta la 0Ch de DOS. Estas funciones se encargan de procesar la Entrada/Salida de caracteres, (Character I/O functions). En las versiones 2.x del sistema operativo, este stack también se utiliza en las llamadas a las funciones 50h (Set Process ID) y 51h (Get Process ID). Este último hecho complica aún más las cosas para los TSRs que, corriendo bajo estas versiones de DOS, deben manipular archivos de disco.

El stack número 3 es utilizado por DOS mientras determina cuál de los otros dos stacks debe utilizar para procesar alguna función. Este stack también es utilizado para procesar llamadas a la función 59h (Get Extended Error Information). Por último, el tercer stack es utilizado por DOS cuando se deben procesar errores críticos. Si DOS es

llamado desde un Critical Error Handler (rutina que se encarga de procesar errores críticos), el tercer stack es utilizado en lugar del segundo." [15:p.275]

(Nota: Los errores críticos son causados por fallas en los periféricos. Algunos errores críticos comunes son: Write Protected Diskette, Drive Not Ready, Sector Not Found, Printer Out Of Paper y General Failure).

DOS no posee ningún mecanismo para preservar el contenido de sus stack internos. Esto quiere decir que si DOS es reentrado, no se puede garantizar la integridad de los datos contenidos en ninguno de los tres stacks. Si la llamada que causa la reentrada utiliza algún stack que contiene información que DOS todavía necesita, entonces no habrá forma de regresar el control a la primera llamada, ya que la información necesaria para hacerlo habrá sido destruida por la segunda. Esto provocará una falla insalvable en el sistema y la computadora muy probablemente se congelará o empezará a comportarse en forma por demás errática.

Afortunadamente, existen tres piezas de información que permiten identificar el stack interno que está siendo utilizado por el sistema operativo en un determinado momento, esto es, permiten controlar el problema de reentrada. Estas tres piezas de información son:

- 1) La InDos Flag (Bandera InDos).
- 2) La interrupción 28h.
- 3) La Critical Error Flag (Bandera de Error Crítico).

La primera y tercera son componentes del sistema operativo, es decir, forman parte del código residente en memoria de DOS. La segunda es una interrupción que el mismo sistema operativo genera bajo ciertas circunstancias.

Como regla general, ninguna de las banderas ni la interrupción son utilizadas en forma aislada, ya que para determinar lo más acertadamente posible el estado de los stacks internos de DOS, generalmente es necesario combinar las tres informaciones.

A continuación se describen, las características individuales de las dos banderas y de la interrupción, así como la forma de combinar la información que proporciona cada una de ellas para evitar conflictos de reentrada.

InDos Flag.

Esta bandera, documentada hasta la liberación de MS-DOS 5, es encendida cada vez que se entra a DOS, es decir, cada vez que DOS está procesando alguna función. Del mismo modo, cuando se sale de DOS, es decir, cuando DOS regresa el control al programa que realizó la llamada, la InDos Flag es apagada.

El principal uso que un TSR le da a esta bandera, es para determinar si será necesario utilizar los trucos de manipulación de stacks para llamar a las funciones de DOS.

Un programa TSR puede localizar esta bandera, es decir, obtener su dirección en memoria, utilizando la función 34h de DOS. La forma de codificar esta función será descrita detalladamente en el próximo capítulo.

Es esencial que la función 34h sea llamada únicamente durante el proceso de inicialización del TSR. Por lo tanto, la dirección de la InDos Flag deberá ser almacenada en una variable interna del TSR. Así, cuando éste sea reejecutado, podrá determinar el estado de la bandera utilizando la dirección almacenada en su variable sin necesidad de llamar nuevamente a la función 34h, lo que podría ocasionar problemas de reentrada.

Existen varias complicaciones asociadas con el uso de la InDos Flag. Estas se deben a que el significado de la bandera se puede interpretar de varias maneras, según el estado en el que se encuentre la computadora en un determinado momento.

Como regla general, se puede decir que cuando la bandera está apagada, es decir, cuando tiene un valor de cero, es posible utilizar cualquier función de DOS desde el TSR sin ningún riesgo.

Ahora bien, es un hecho que la InDos Flag tiene un valor diferente de cero siempre que el prompt del sistema está desplegado en pantalla. Por lo tanto, si un TSR sólo pudiera reejecutarse cuando esta bandera tiene un valor de cero, el programa nunca podría ser activado mientras el prompt del sistema está presente en pantalla.

Esto es ilógico, ya que si el prompt del sistema está desplegado, lo más seguro es que no se esté realizando ningún proceso. En otras palabras, no deberían existir problemas de reentrada mientras el prompt del sistema está en pantalla.

De lo anterior se puede deducir que sí es posible hacer llamadas a DOS, aun cuando el valor de esta bandera no sea cero. Para aclarar la forma de utilizar la InDos Flag, es necesario entender el funcionamiento de la interrupción 28h, la cual está relacionada directamente con el valor de esta bandera, así como con la interpretación que se le pueda dar.

Interrupción 28h.

Esta interrupción (no documentada y reservada para uso exclusivo del sistema operativo) es conocida como la MS-DOS Scheduler o más frecuentemente como 'MS-DOS Idle Interrupt'. La traducción que se le podría dar es: Interrupción del sistema operativo para los ratos de ocio.

En términos un poco más técnicos, "esta interrupción es generada desde la rutina Get Input From Keyboard de DOS si, y sólo si, es seguro utilizar la INT 21h para accesar los discos en ese momento." [24:p204] Esto quiere decir que la INT 28h es generada siempre que DOS está ocioso y únicamente está esperando a que se presione alguna tecla. En realidad esta interrupción es generada siempre que DOS está procesando, alguna de las funciones 01h a 0Ch, es decir, funciones de E/S de carácter. (Get Input Form Keyboard es la función 01h de DOS).

Aparentemente, esta interrupción fue implementada en el sistema operativo con el único propósito de proveer a los TSRs de un mecanismo seguro para accesar los servicios

de DOS. Esta teoría se ve reforzada por el hecho de que el vector original de esta interrupción apunta a una simple instrucción IRET (Interrupt Return), es decir, la ISR original para esta interrupción no realiza absolutamente ninguna función.

La interacción entre la InDos Flag y la INT 28h se da de la siguiente manera: si la InDos Flag tiene un valor diferente de cero (está encendida) y DOS esta procesando alguna llamada a cualquiera de las funciones 01h a 0Ch, una interrupción 28h es generada por el mismo sistema operativo con cada 'tick' del reloj de la computadora. (Un tick es generado por el reloj, 18.2 veces cada segundo).

Como se recordará, estas funciones utilizan el stack 2 del sistema operativo, por lo tanto, una INT 28h es una señal de que los stacks 1 y 3 del sistema operativo pueden ser utilizados sin ningún riesgo, es decir, es una señal de que las funciones no relacionadas con E/S de carácter sí pueden ser utilizadas sin problemas de reentrada.

De hecho, lo que esta interrupción asegura es que no es posible utilizar el stack 2 del sistema operativo. Por lo tanto, si el TSR fue activado mediante la interrupción 28h, el hacer llamadas a cualquiera de las funciones 01h a 0Ch de DOS, generalmente provocará que la computadora se congele. (Ver: 'Funcionamiento de un TSR', más adelante, para una descripción los diferentes mecanismos de activación).

Se dice generalmente, ya que también existe un artificio para salvar este problema. Este truco está basado en el uso de la 'ErrorMode Flag' mejor conocida como la Critical Error Flag o bandera de error crítico.

Critical Error Flag.

Cuando ocurre algún error crítico, DOS enciende la Critical Error Flag y genera una interrupción 24h. La ISR asociada a esta interrupción es conocida como 'Critical Error Handler' (Manejador de Error Crítico). Esta rutina es responsable de corregir el error. La rutina default del sistema operativo simplemente despliega el famoso mensaje: Abort, Retry, Ignore, Fail? cuando se genera algún error crítico.

Si al momento de recibir la orden de activación esta bandera se encuentra encendida indicando que un error crítico está siendo procesado por DOS, el TSR no debe activarse.

Durante un error crítico, DOS utiliza el tercer stack (stack de errores críticos) para procesar el error. Al mismo tiempo, alguno de los stack para funciones (1 ó 2) también está siendo utilizado, ya que DOS estaba procesando alguna función cuando ocurrió el error. Por lo tanto, un TSR codificado apropiadamente simplemente esperará hasta que el error haya sido resuelto, para después activarse.

En contraste con la forma de localizar la InDos Flag, nunca ha existido una función documentada o no documentada que regrese la dirección de la Critical Error Flag. Pero aún así, hay tres formas de localizar esta bandera.

La primera utiliza la función 5Dh de DOS, la cual nunca ha sido documentada pero ha estado disponible desde la versión 3.1 del sistema operativo. Esta función regresa

un apuntador a un área de datos de DOS conocida con el nombre de 'DOS Swappable Data Area'. El primer byte de esta área es precisamente la Critical Error Flag, por lo tanto, la mayoría de los TSRs utilizan el apuntador regresado por la función 5Dh como la dirección absoluta de la bandera.

Con la liberación de MS-DOS 5, Microsoft documentó que la dirección de la Critical Error Flag corresponde a la del byte inmediatamente anterior a la InDos Flag, lo cual también ha sido cierto desde la versión 3.1 de DOS. Por lo tanto, la segunda forma de localizar esta bandera, es simplemente decrementar la dirección obtenida para la InDos Flag, siempre y cuando la versión del sistema operativo lo permita.

La tercera forma de localizar la Critical Error Flag es compatible con las versiones 2.0 y superiores del sistema operativo. En pocas palabras, este método registra toda la memoria utilizada por el sistema operativo, hasta encontrar la dirección de la bandera.

Específicamente, el método consiste en ir leyendo todos los bytes del segmento de código del kernel de DOS, hasta encontrar la primera ocurrencia de una serie de instrucciones específicas, las cuales apuntan a la bandera. La explicación más detallada de estos métodos se deja para el siguiente capítulo, ya que es más sencillo explicarlos cuando se está analizando el código que los implementa.

Al igual que con la InDos Flag, el proceso de localización de esta bandera, sin importar el método utilizado, se debe realizar durante la inicialización del TSR.

Ahora bien, ¿cómo es que se implementa el truco para utilizar las funciones 01h a 0Ch, aun cuando la InDos Flag esté encendida ? Para poder responder a esta pregunta, imaginemos la siguiente situación.

La InDos Flag está encendida porque el sistema operativo está esperando a que se presione alguna tecla, es decir, está procesando continuamente su función 01h. Al mismo tiempo, DOS está generando una interrupción 28h, 18.2 veces por segundo.

Entonces, el TSR es activado por medio de la INT 28h, lo que nos indica dos cosas: primera, que el stack 2 no está disponible y segunda, que ningún error crítico estaba siendo procesado por el sistema operativo en el momento que el TSR fue activado.

En este momento estamos seguros que los stacks 1 y 3 pueden ser utilizados sin peligro. Entonces el truco consiste en forzar a DOS para que utilice alguno de estos stacks, (en lugar del segundo) al procesar una nueva función de E/S de carácter. Con esta manipulación de stacks, se evita destruir la información de la primera llamada.

Si el TSR enciende 'manualmente' la Critical Error Flag, (sin provocar ningún error real) entonces DOS pensará que está procesando un error crítico y, por lo tanto, utilizará el tercer stack para procesar cualquiera de sus funciones, incluyendo las de E/S de carácter.

El código que aparece en el listado 3-1 muestra cómo forzar el uso del stack de error del sistema operativo para procesar funciones. Este código es equivalente al que se encuentra integrado en el programa de demostración TSRTMPLT.ASM en la rutina 'dospspcall'.

Listado 3-1. Forzar el uso del stack 3 de DOS [19 p.47]

```

...
doscall proc near
    assume cs:Cseg
    cmp     ah,0Ch           ;See if I/O call
    jbe     DOSCALL_2
    cmp     cs:[Dos_version],300h ;See if Dos = 3 0
    ja      DOSCALL_1
    cmp     ah,59h
    je      DOSCALL_2

DOSCALL_1.
    cmp     cs:[Dos_version],30Ah ;See if DOS < 3 1
    jae     DOSCALL_OK           ;If not, just call DOS
    cmp     ah,50h
    je      DOSCALL_2
    cmp     ah,51h
    jne     DOSCALL_OK

DOSCALL_2.
    push    ds
    push    di
    lds     di,cs.CritErr_ptr     ;Get crit err flag adr
    inc     byte ptr [di]        ;Set DOS in Crit Err State
    pop     di
    pop     ds
    int     21h                 ;Call DOS
    push    ds
    push    di
    lds     di,cs.CritErr_ptr     ;Get crit err flag adr
    dec     byte ptr [di]        ;Clear DOS Crit Err State
    pop     di
    pop     ds
    ret

DOSCALL_OK
    int     21h
    ret
doscall endp

```

Es claro que con las instrucciones que le corresponden a DOSCALL_2, se está forzando el uso del tercer stack para procesar cualquiera de las funciones 01h a 0Ch, 50h, 51h o 59h (esta última sólo si se está corriendo bajo una versión de MS-DOS anterior a la 3.1) y que cualquier otra llamada se hace sin ninguna consideración especial (DOSCALL_OK).

La razón para incluir las funciones 50h y 51h en este proceso, tiene que ver con el acceso a archivos de disco desde el TSR. Este es precisamente el tema de la siguiente sección.

ACCESO A ARCHIVOS DE DISCO.

Siempre que MS-DOS carga un programa a memoria, le asigna un 'Process ID' o identificador de proceso. (Ver: 'Program Segment Prefix', capítulo 1). En todos los casos, sin excepción, el PID no es más que el segmento de la dirección que le corresponde al

PSP del programa. El 'PID activo' es, por lo tanto, el identificador del programa actualmente en ejecución.

DOS almacena el PID activo internamente y lo utiliza cuando el programa accesa archivos mediante el uso de 'File Handles' (Manipuladores de Archivos). Como se recordará, cada programa almacena sus handles en uno de los campos de su PSP.

Para poder manipular archivos de disco, el sistema operativo debe realizar las siguientes dos acciones:

- 1) Al recibir el comando, el sistema operativo primero determina cual PID está activo, es decir, qué proceso está actualmente en ejecución.
- 2) Después de esta verificación, DOS busca en el PSP correspondiente al proceso activo el file handle del archivo que se desea acceder.

Desafortunadamente, cuando un TSR es activado DOS no actualiza el PID activo, por lo tanto, el sistema operativo considera al programa que fue interrumpido como el proceso actual. Es responsabilidad del mismo TSR informar al sistema operativo del cambio de proceso, es decir, el TSR debe actualizar el PID activo, lo que se logra mediante el proceso que se describirá a continuación.

Modificación del PSP y DTA.

1) La función 51h de DOS se utiliza para obtener el PID activo. Este es el identificador del programa que estaba corriendo cuando el TSR fue invocado. El identificador así obtenido debe ser salvado en alguna variable interna del TSR. Esta función existe desde la versión 2.0 del sistema operativo, pero nunca ha sido documentada. Por otro lado, la función 62h proporciona los mismos servicios y sí está documentada, pero fue implementada hasta la versión 3.0.

2) Después se utiliza la función 50h de DOS, (documentada hasta la liberación de DOS 5) para establecer el PID del TSR como el PID actual. Una vez que el TSR ya se ha establecido como el proceso actual, puede utilizar los manipuladores de archivo almacenados en su PSP.

3) Antes de regresar a la aplicación principal, el TSR debe restaurar el PID que estaba activo al momento de reejecutarse. Para lograr esto, se utiliza nuevamente la función 50h

Existe otro método, un poco más complicado, con el cual se puede cambiar el PSP actual. Este método recorre toda la memoria que pertenece a DOS hasta encontrar el lugar exacto donde DOS almacena el valor del PSP actual. Así, cuando sea necesario cambiar el PSP actual, únicamente habrá que modificar esta área de memoria con el nuevo valor. Este método es utilizado en el programa de demostración JTSRTP.PAS y será explicado con más detenimiento en el siguiente capítulo.

Como ya se dijo, en las versiones 2.x del sistema operativo las funciones 50h y 51h utilizan el segundo stack. Esto causa un serio problema, ya que agrupa estas

funciones junto con las funciones de E/S de carácter. Por lo tanto, cuando el TSR es activado por medio de la INT 28h, las funciones 50h y 51h también deben ser procesadas utilizando el stack de error, (stack 3) como se ve en el listado 3-1.

ERRORES CRITICOS

Además de prevenir la activación del TSR durante un error crítico, se debe considerar la forma como serán manejados los errores que se generen desde el mismo TSR. Normalmente, el control de los errores críticos se logra reemplazando la ISR asociada con la INT 24h.

La forma más sencilla de manejar errores críticos, consiste en no modificar la Critical Error Handler actual, es decir, el TSR utilizará la rutina para manejo de errores del programa que fue interrumpido. Este método debe ser suficiente si el TSR no realiza operaciones de E/S, tales como acceso a disco o a impresora.

Sin embargo, si el TSR ha de realizar este tipo de funciones, es muy recomendable implementar dentro del código del programa una rutina para manejo de errores, apropiada a sus necesidades. Como parte del proceso de activación, (no de inicialización) el TSR debe reemplazar el vector asociado a la interrupción 24h y hacer que apunte a su propia rutina, la cual permanecerá vigente hasta que el TSR sea desactivado por el usuario. Antes de regresar el control a la aplicación interrumpida, el TSR deberá restaurar el vector original. Estas acciones son necesarias, ya que la mayoría de las aplicaciones depende de su propia rutina para manejar sus errores críticos.

Una forma muy simple de implementar esta rutina es ignorando los errores críticos. Esto se logra haciendo que la nueva ISR regrese un resultado con valor cero. Esto le indica a DOS que debe continuar como si el error no hubiera existido. Como DOS ignorará todos los errores, es responsabilidad del TSR realizar las acciones correctivas necesarias para resolverlos. El programa SAVESCR.PAS ejemplifica este procedimiento.

Control-Break.

La acción de cancelar la ejecución de un programa no es un error, pero si el programa cancelado es un TSR, esta acción se vuelve crítica, ya que el TSR nunca podría regresar el control al programa que fue interrumpido y, por lo tanto, la computadora se congelaría.

Es necesario entonces, que el TSR reemplace los vectores de la INT 1Bh (Ctrl-Break) y la INT 23h (Ctrl-C). Generalmente las ISRs de estas interrupciones son reemplazadas por una simple instrucción IRET, es decir, simplemente regresan el control sin llevar a cabo ninguna acción. El proceso de reemplazo y restauración de estas ISRs se debe llevar a cabo igual que para la interrupción 24h.

FUNCIONAMIENTO DE UN TSR.

En esta sección se describirán las técnicas fundamentales para mantener un programa residente en memoria, y para lograr accederlo nuevamente con el fin de reejecutarlo.

Un programa residente consta de dos componentes:

El primer componente, inicializa el TSR y establece la forma que habrá de ser utilizada para reejecutar el TSR. Esta porción del código es ejecutada únicamente cuando el programa es cargado a memoria desde la línea de comandos.

El segundo componente es la porción del programa que es reejecutada cada vez que el TSR es invocado. Por lo tanto, el segundo componente debe permanecer accesible al sistema operativo, aún después de que haya finalizado la ejecución del programa.

PORCION TRANSITORIA.

El primer componente del TSR se conoce como 'Porción Transitoria', y como su nombre lo indica, permanecerá en memoria sólo temporalmente. Es por esto que la porción transitoria generalmente es codificada al final del programa, ya que así podrá ser descartada de memoria una vez que finalice su ejecución. Esta porción de código, realiza los siguientes procesos:

- 1) Inicialización
- 2) Instalación.
- 3) Desinstalación. (Opcional)

Proceso de Inicialización.

Esta parte del código se encarga de realizar tres funciones que son básicas para el buen funcionamiento del TSR: (1) Determinar si el TSR ha sido previamente instalado en memoria, para prevenir más de una instalación. (2) Localizar las banderas que proporcionan una entrada limpia al sistema operativo. (3) Capturar los vectores de interrupción necesarios.

- Prevenir más de una instalación.

Existen bastantes métodos para determinar si una copia del mismo TSR ha sido previamente instalada en memoria, pero todos son variantes de los tres métodos que a continuación se describen.

- 1) Este primer método utiliza la fuerza bruta, ya que registra cada segmento de memoria en uso, buscando un bloque que sea idéntico a los primeros bytes de la imagen del programa TSR. Si se encuentra dicho bloque se concluye que el TSR ya ha sido instalado y, por lo tanto, la instalación de la segunda copia debe ser detenida.

Al utilizar este método, se debe tener cuidado de no confundir la imagen del programa que se pudiera encontrar en algún buffer de disco, con la imagen de una copia real del TSR previamente instalada.

Además, como se mencionó en el primer capítulo, un TSR puede residir en la memoria denominada Upper Memory Area, es decir, por encima de la memoria principal. Por lo tanto, el proceso de búsqueda también deberá ser realizado en esta área de memoria si el programa está corriendo en un sistema que lo permita (386 o superior)

2) El segundo método teóricamente debe ser más sencillo. Este método utiliza la interrupción 2Fh de DOS, la cual se conoce como 'Interrupción Multiplex'. Esta interrupción proporciona los mecanismos estándar, tanto para localizar copias previamente instaladas de algún programa, como para accesarlas. Desafortunadamente, estos servicios sólo pueden ser utilizados a partir de la versión 3.0 del sistema operativo.

Para que un TSR pueda utilizar esta interrupción, el programador le debe asignar un identificador numérico, conocido como 'Device ID'. Existen 256 identificadores diferentes (00h - FFh), pero Microsoft se reserva el uso de los IDs desde el 00h hasta el BFh. Esto quiere decir que todos los demás programas de aplicación deben compartir los restantes 64 identificadores.

El listado 3-2, muestra el código que normalmente se utiliza durante el proceso de instalación para determinar, por medio de la interrupción multiplex de DOS, si una copia del TSR ya está instalada en memoria.

Listado 3-2 Uso de la INT 2Fh para localizar una copia del TSR previamente instalada. [19:p.42]

```

mov     ah,0DBh      ;Select device ID 0DBh
mov     al,0         ;Install check
int     2Fh
or      al,al        ;If AL <> 0, device ID found
je      TSR_NOT_FOUND
inc     Dev_Found    ;Set program installed flag
jmp     ABORT_INSTALL

```

```

TSR_NOT_FOUND,
; Continue with installation
...

```

Como se puede ver, el ID de proceso que se desea localizar se almacena en AH y se ejecuta la interrupción. De regreso de ésta, AL contendrá un valor diferente de cero si el programa fue localizado en memoria.

Es necesario entonces, que la porción residente del TSR tenga implementada dentro de su código una ISR que será insertada en la cadena de ISRs correspondientes a la INT 2Fh. Así, cuando la interrupción sea generada desde la porción transitoria de una segunda copia potencial del TSR, la porción residente en memoria podrá verificar el contenido de AH.

Si el valor de AH no coincide con el identificador del TSR, entonces la ISR de nuestro programa debe llamar a la rutina de interrupción anterior, es decir, debe permitir que el proceso continúe a lo largo de la cadena de vectores para esta interrupción. Por otro lado, si el valor de AH coincide con el ID, entonces el TSR en memoria regresará un valor diferente de cero en AL, para indicar que no se debe instalar una segunda copia.

El código de esta rutina de interrupción aparece en el listado 3-3. Este listado contiene código (NEXT_CMD) que no es utilizado para localizar una copia del TSR ya residente en memoria. Este código extra será explicado más adelante, cuando se describan las formas de comunicación con la porción residente del TSR.

Listado 3-3. ISR para monitorear la interrupción 2Fh. [19-p 42]

```

...
muxint  proc far
        assume  cs,Cseg, ds nothing, es nothing
        cmp     ah,0DBh      ,Is ID correct ?
        je      MUXINT_ID

OLD_MUXINT_JMP.
        jmp     cs:[int2fh]   ,Call old int

MUXINT_ID:
        or      al,al        ,See if install check command
        jne     NEXT_CMD
        mov     al,-1        ,Set installed flag

MUXINT_EXIT:
        iret

NEXT_CMD:
        , Check for other TSR-specific commands
.

muxint  end
.

```

El principal problema al utilizar este método es la escasez de identificadores de proceso, lo cual hace muy posible que un ID sea utilizado por más de un programa a la vez.

Por lo tanto, para estar seguros de que es nuestro TSR el que está respondiendo a la interrupción multiplex, se deben tomar algunos pasos adicionales. El procedimiento más común consiste en que la rutina de interrupción de nuestro TSR regrese algún o algunos valores adicionales, (además del valor de AL) utilizando otros registros del procesador. Estos valores adicionales serán utilizados para identificar plenamente a nuestro programa.

(Nota: Una variante de este segundo método puede utilizar cualquier otra interrupción que no sea la multiplex. Se debe tener cuidado de no interferir con el funcionamiento normal de la interrupción que se vaya a utilizar).

3) El último método de localización de copias previamente instaladas utiliza las interrupciones 60h a 67h, las cuales están reservadas para el uso exclusivo de los programas de aplicación escritos por el usuario.

Durante el proceso de instalación, el TSR captura alguno de estos vectores. El vector será reemplazado, es decir, no habrá encadenamiento. Por lo tanto, es necesario que el vector a capturar no esté siendo utilizado por ningún otro programa. Esto se puede comprobar, ya que un vector no utilizado debe tener un valor nulo (NIL).

La porción transitoria del TSR debe ir revisando uno a uno los vectores 60h al 67h, para así poder seleccionar el primero que se encuentre disponible. Cada vez que el TSR encuentre que un vector ya está ocupado, deberá asegurarse de que no es una copia previamente instalada de él mismo quien lo está utilizando.

Esto se logra mediante cadenas de caracteres llamadas 'Firmas'. La firma del TSR está contenida tanto en la copia residente, como en la segunda copia potencial que se está tratando de instalar. Si al realizar la comprobación las dos firmas coinciden, se debe detener la instalación de la segunda copia del TSR.

Por último, si ningún vector está siendo utilizado o ninguna de las firmas coincidió con la del TSR, entonces se concluye que éste aún no ha sido instalado y, por lo tanto, el proceso de instalación debe capturar el primer vector que encontró disponible.

(Nota: El proceso de ir recorriendo los vectores 60h al 67h también puede ser utilizado para comunicarse con la porción residente del TSR).

- Localizar banderas.

El proceso de inicialización debe localizar la dirección de las banderas InDos Flag y la Critical Error Flag y almacenar dichas direcciones en variables internas del TSR. Estas variables serán utilizadas por la porción residente durante el proceso de activación. La forma de realizar estos procesos ya fue descrita anteriormente, al estudiar cada una de estas banderas.

- Capturar vectores de interrupción.

Si un programa permanece residente en memoria después de haber terminado su ejecución, la única forma de reejecutarlo (activarlo) es por medio de interrupciones. Si el TSR no reemplaza o intercepta ninguna interrupción, nunca podrá ser activado. Por lo tanto, para que un TSR realice algún trabajo útil, debe capturar cuando menos una interrupción.

(Nota: Los Interrupt Handlers (Manejadores de Interrupción) son un tipo de TSR pasivo, que generalmente captura sólo una interrupción. Su propósito es simplemente el de modificar el funcionamiento de dicha interrupción. El programa de demostración F11.PAS es un Interrupt Handler).

La rutina de inicialización debe almacenar el vector original de cada interrupción capturada en una variable interna y modificar la tabla de vectores de interrupción para que los vectores capturados apunten a las ISRs implementadas dentro del código del TSR.

Un TSR sofisticado debe capturar cuando menos las siguientes tres interrupciones:

INT 08h: Timer Interrupt (reloj del sistema).
 INT 09h: Hardware Keyboard Interrupt (teclado)
 INT 28h: DOS Idle Interrupt.

Estas tres interrupciones juegan un papel clave en el proceso de activación del TSR, como se verá más adelante. Además, dependiendo de las funciones específicas que vaya a realizar el TSR, se deberán reemplazar o interceptar otras interrupciones tales como:

INT 10h: BIOS Video Interrupt.
 INT 13h: BIOS Disk Interrupt.
 INT 16h: BIOS Keyboard Interrupt.
 INT 1Bh: Keyboard Break Interrupt.
 INT 23h: Break Address Interrupt.
 INT 24h: Critical Error Interrupt.
 INT 2Fh: DOS Multiplex Interrupt.
 INT 60h-67h: Interrupciones del usuario, etc.

Las interrupciones 10h, 13h, 16h se interceptan generalmente para asegurar que el TSR no será activado mientras se esté procesando alguna de ellas. Si el TSR es activado mientras se accesa el video, por ejemplo, se podría alterar la imagen que se está desplegando en pantalla. Si la activación se da en el mismo momento de accesar un disco, esto podría confundir a la computadora y se perderían datos durante el proceso de activación.

Esta clase de monitoreo se logra implementando una rutina, para cada interrupción capturada, que simplemente prenderá una bandera al generarse la interrupción correspondiente; después llamará a la ISR original para que sea ella quien realmente realice el proceso solicitado. De regreso de la ISR original, nuestra rutina apagará la bandera y regresará el control.

Estas banderas de advertencia serán monitoreadas por las rutinas encargadas de activar el TSR. Si al momento de invocar el TSR cualquiera de estas banderas está encendida, se abortará el proceso de activación.

Las interrupciones 1Bh, 23h y 24h generalmente no son interceptadas sino reemplazadas, pero únicamente durante el periodo de actividad del TSR. Cuando el TSR permanece latente en memoria, los vectores originales son restaurados.

Proceso de Instalación.

La última acción que realiza la porción transitoria antes de terminar la ejecución del programa, es dejar residente en memoria una parte del código del TSR. Para lograr esto se utiliza la función 31h de DOS (INT 21h).

Esta función permite terminar la ejecución de un programa y al mismo tiempo especificar la cantidad de memoria, perteneciente al programa, que deberá permanecer

reservada. Ningún otro programa podrá utilizar esta memoria ya que ella contendrá la porción residente del TSR.

También es posible utilizar la INT 27h para terminar y dejar residente al programa, pero esta interrupción es menos flexible que la función 31h, como se verá en el siguiente capítulo cuando se estudie el código de ambas implementaciones.

Proceso de desinstalación. (Opcional)

Este proceso puede ser realizado por la porción transitoria de una segunda copia del TSR ejecutada desde la línea de comandos, o puede ser ejecutada por la porción residente de TSR como respuesta a un comando del usuario.

Generalmente se utiliza el primer método, ya que así se ahorra memoria, pues no es necesario mantener el código de desinstalación residente en memoria. De cualquier modo el procedimiento es el mismo:

1) Lo primero que debe hacer la rutina de desinstalación es verificar que toda la memoria que está por encima del TSR esté libre, es decir, se debe verificar que ningún programa que haya sido cargado después del TSR permanezca aún en memoria.

Esta comprobación generalmente se lleva a cabo revisando los vectores de las interrupciones que capturó el TSR. Si éstos siguen apuntando a las rutinas internas de la porción residente, se puede estar casi seguro de que ningún otro programa se encuentra cargado en memoria encima del TSR.

Si los vectores no corresponden a las rutinas de nuestro TSR, se concluye que otro programa volvió a capturar esas mismas interrupciones y que además sigue cargado en memoria.

Aunque es posible desinstalar el TSR aunque exista algún programa encima de él, esto no es recomendable, ya que se crearían huecos de memoria que DOS no podría utilizar. Por lo tanto, se toma como norma que si la memoria situada arriba del TSR no está libre, se debe abortar el proceso de desinstalación.

2) Si es posible desinstalar el TSR, se deben restaurar los vectores originales que se capturaron durante el proceso de instalación.

3) Por último, se debe liberar la memoria que está siendo ocupada por la porción residente del TSR y regresar al sistema operativo. Para realizar este último paso del proceso de desinstalación, se debe localizar en memoria el principio de la porción residente.

Generalmente, esto se logra utilizando algún método de rastreo de la memoria, ya sea buscando la firma del TSR o el primer MCB (Memory Control Block) asignado a la porción residente del programa. La descripción detallada de estos conceptos se deja para el próximo capítulo, para que la explicación del proceso se pueda apoyar en el código que lo implementa.

PORCION RESIDENTE.

El segundo componente de un TSR se conoce como 'Porción Residente', el cual permanece accesible al sistema operativo y, por lo tanto, puede ser reejecutada en cualquier momento. El código de esta porción realiza cuando menos dos procesos:

- 1) Activación.
- 2) Desactivación.

Entre estos dos procesos es donde el TSR realiza el trabajo útil para el que fue diseñado, pero no es necesario que exista tal proceso intermedio para que el TSR funcione adecuadamente.

Proceso de activación.

El proceso de activación varía ligeramente según se trate de un TSR pasivo o uno interactivo.

Si el TSR es pasivo será simplemente activado por medio de alguna interrupción, frecuentemente, la INT 08h.

Pero si el TSR es interactivo primero debe ser invocado y después activado. Generalmente, un TSR es invocado al presionar una secuencia especial de teclas. Por lo tanto, la interrupción que más frecuentemente es utilizada para invocar un TSR interactivo es la INT 09h (teclado).

La ISR para esta interrupción implementada dentro del código de la porción residente, podrá monitorear todas las teclas que sean presionadas y si detecta la secuencia especial prenderá una bandera indicando que el TSR ha sido invocado.

Esta bandera es constantemente revisada por las rutinas de interrupción que activan el TSR. Las interrupciones que más frecuentemente son utilizadas para activar un TSR, ya sea pasivo o interactivo, son las INT 08h y la 28h.

La INT 08h (reloj del sistema) proporciona una forma muy eficiente para activar el TSR. Como esta interrupción es generada 18.2 veces cada segundo, la rutina que el TSR insertó en la cadena de ISRs asociadas con esta interrupción será ejecutada continuamente. Esta rutina revisará tres cosas antes de activar el TSR: la bandera que indica que el TSR fue invocado (en caso de ser un TSR interactivo), la bandera que indica que el TSR ya está activo y el estado del sistema.

Para cumplir con este último punto la ISR del reloj debe asegurarse como mínimo de que la InDos Flag esté apagada, pero según las funciones que realice el programa revisará también la Critical Error Flag y las banderas de advertencia. (Ver: 'Capturar vectores de interrupción').

Aunque la interrupción del reloj es muy eficiente funciona únicamente durante la ejecución de algún otro programa, no así cuando se está al nivel de sistema operativo, es decir, no funciona cuando el prompt del sistema está desplegado en pantalla. Como ya se

sabe, cuando el prompt está presente DOS está ocioso y sólo está esperando a que se presione alguna tecla. Mientras esto sucede DOS llamará continuamente a la INT 28h y la InDos Flag permanecerá encendida. Como consecuencia, la ISR asociada al reloj no permitirá que el TSR sea activado.

Es por esto que el TSR necesita interceptar también la INT 28h. La ISR asociada con esta interrupción será muy parecida a la del reloj, con la única diferencia de que esta rutina sí permitirá la activación del TSR aunque la InDos Flag esté encendida.

En resumen, para permitir la activación del TSR es necesario que se cumplan los siguientes requisitos:

- 1) Que se haya invocado al TSR (si es interactivo).
- 2) El TSR no debe estar activo en el momento de ser invocado ya que el activarlo por segunda vez causaría problemas de reentrada si es que el TSR implementa su propio stack interno. (Ver: 'Lineamientos generales', más adelante).
- 3) El sistema no debe estar procesando ninguna función del BIOS porque éste tampoco puede ser reentrado.
- 4) MS-DOS debe encontrarse en un estado estable, es decir, no debe estar procesando errores críticos

Si todos los requisitos se cumplen habrá que 'Activar el TSR'. Activar el TSR quiere decir: pasar el control a la rutina principal de la porción residente. Generalmente esta rutina es la que implementa el trabajo real que ha de desempeñar el programa.

Antes de iniciar con el trabajo real esta rutina debe salvar el estado del sistema, es decir, se deben salvar todos los parámetros configurables que el TSR pueda alterar mientras está activo. Los parámetros a salvar varían según el hardware instalado en la computadora, la versión de el sistema operativo bajo la que se está corriendo y las funciones que deba desempeñar el TSR; pero como regla general se puede decir que si se va a modificar algún parámetro, su estado debe ser salvado antes de iniciar con el trabajo real.

Antes que nada la rutina principal debe cambiar al stack interno del TSR y salvar todos los registros del procesador. También si el TSR utiliza funciones de disco se debe actualizar el PSP activo y modificar la dirección de la DTA. Después, según las necesidades del TSR se pueden reemplazar las INT 1Bh, 23h y/o 24h, y se pueden salvar parámetros tales como: el estado del cursor (posición y tamaño), el estado del apuntador del ratón, el estado del teclado (bloqueo de números, de letras mayúsculas, de scroll, etc.), la información de error extendido de DOS (esta información es la que regresa la función 59h), etc.

Así como es importante salvar los parámetros que se van a modificar también es importante no salvar aquellos que no serán cambiados. Con esto se ahorra memoria, se reduce el código del TSR y se hace más eficiente su desempeño.

Proceso de desactivación.

Este proceso también está incluido en el código de la rutina principal del TSR y no requiere de mayor explicación ya que es exactamente el proceso inverso al de activación. Aquí se restauran los valores de los parámetros que estaban vigentes cuando el TSR fue invocado y que durante el proceso de activación fueron salvados para que el TSR pudiera reejecutarse sin ningún problema.

Después de ejecutar el proceso de restauración, el código de la rutina principal de la porción residente regresa el control a la aplicación que fue interrumpida.

Comunicación con la parte residente.

Es posible que en algún momento el usuario desee cambiar la forma en que opera el TSR. Por ejemplo, si se tiene un programa que habilite y deshabilite las teclas Shift-PrintScreen (el clásico pantallazo a la impresora).

El usuario puede alternar los dos estados cambiando una variable interna del código de la porción residente. Pero ¿cómo cambiarla? La respuesta es: utilizando algún método de comunicación entre una segunda porción transitoria ejecutada desde la línea de comandos y la porción que se encuentra residente.

Generalmente, el comando que se le quiere pasar a la porción residente se especifica como un parámetro desde la línea de comandos. Siguiendo con el ejemplo anterior tendríamos los siguientes dos comandos:

```
C:\>PRTSCR /on
C:\>PRTSCR /off
```

(Nota: Se supone que el TSR ya está instalado en memoria al momento de ejecutar estos comandos).

La comunicación con la porción residente se puede implementar básicamente de dos formas:

1) Utilizando la interrupción multiplex. El procedimiento para comunicarse con el TSR es básicamente el mismo que para identificar una copia previamente instalada (ver: listado 3-2). Con la única variante de que el valor almacenado en AL no será cero sino el valor del comando que se desee pasar a la porción residente. Por lo tanto, la segunda instrucción del listado 3-2 deberá ser cambiada por:

```
mov al,Cmd.
```

Esto explica el código extra que existe en la ISR implementada en el código de la porción residente, (ver: listado 3-3). Las líneas 10, 11 y de la 14 hasta el final de la rutina no serían necesarias, si no se quisiera implementar la opción de comunicación, ya que estas son las líneas de código que identifican el comando que se le envía a la porción residente y actúan de acuerdo a él.

2) La segunda forma de implementar la comunicación con el código residente utiliza las interrupciones reservadas para los usuarios (60h - 67h). El procedimiento, otra vez es muy similar al de identificación de copias ya instaladas.

Un programa que utilice este método debe revisar uno a uno los vectores del 60h al 67h y cuando se localice, por medio de la firma, el vector que fue capturado por el TSR se podrá generar la interrupción asociada con ese vector, y por medio de ella pasar el comando a la porción residente.

Por ejemplo, supongamos que el TSR capturó el vector 65h. Cuando la segunda copia del TSR (ejecutada desde el prompt del sistema) localice dicho vector podrá comunicar el comando a la porción residente utilizando las siguientes instrucciones:

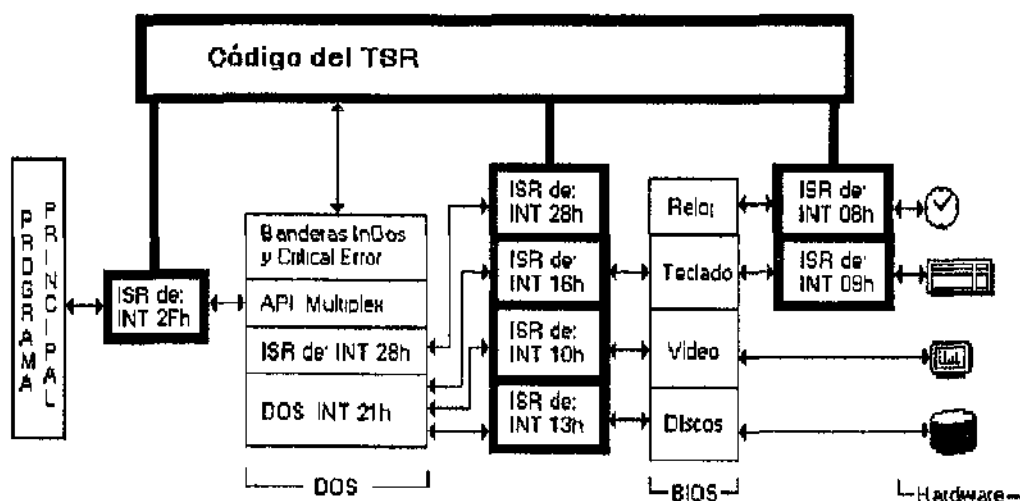
```
mov    IntNum,65h
mov    al,Cmd
int     IntNum
```

La variable Cmd contendrá el comando deseado y la variable IntNum el número de la interrupción cuyo vector fue capturado por el TSR. De forma análoga al método de la interrupción multiplex, la porción residente debe tener implementada una rutina ISR asociada con la interrupción capturada. Esta ISR será la responsable de procesar los comandos solicitados desde el prompt del sistema. El programa de demostración JTSRTP.PAS implementa una línea de comunicación de este tipo.

(Nota: En 1986, cuando empezó el auge de los TSRs se propuso a la INT 52h como interfase de comunicación con la porción residente, pero este método nunca fue adoptado por los programadores).

Para terminar la sección referente al funcionamiento de un TSR se presenta una gráfica que resume las interacciones que existen entre un TSR típico y el sistema operativo, el BIOS y el hardware.

Figura 3-2. Interacciones del TSR con el sistema. [19:p.46]



LINEAMIENTOS PARA CODIFICAR UN TSR.

Sin importar que tan complejo o sofisticado sea el TSR, la principal preocupación al desarrollarlo debe ser el tratar de crear un programa que sea estable. Para lo cual debemos asegurarnos de que el programa será capaz de ejecutar sus funciones libremente, sin privar de esta misma capacidad a los demás procesos que estén corriendo en la computadora al mismo tiempo.

Por lo tanto, un TSR debe ser totalmente transparente y no obstruir el funcionamiento normal del sistema, y al mismo tiempo debe estar siempre disponible para ser activado. Esta meta, que a primera vista parece contradictoria, es la causante de que un TSR tenga que reunir todos los requisitos que se han estudiado hasta este momento para lograr compatibilidad con un sistema operativo que no fue originalmente diseñado para acoger TSRs, cuando menos no en sus implementaciones interactivas.

Para concluir este capítulo, se describirán las áreas que más frecuentemente resultan problemáticas en el desarrollo de un TSR.

LINEAMIENTOS GENERALES.

1) "El TSR no debe romper ninguna cadena de vectores de interrupción, es decir, siempre debe encadenar la interrupción interceptada con la rutina anterior. Al mismo tiempo, el TSR debe 'confiar' que todos los demás programas harán lo mismo, por lo tanto, no debe ser indispensable para el programa que sus vectores sean los últimos en la cadena.

2) El TSR no debe capturar un vector de interrupción más de una vez. Es, por lo tanto, conveniente que el TSR pueda determinar si es que ya está instalado en memoria y de ser así no volverse a instalar." [24:p.410]

3) El uso de un stack local dentro de una rutina de interrupción o de un TSR se puede analizar desde dos puntos de vista.

Primero, el uso de un stack local impide que el TSR pueda ser reentrado, es decir, si el programa es invocado y reactivado desde dentro de sí mismo, el contenido del stack local se pierde ya que la segunda llamada almacena sus datos sobre los primeros. Por lo tanto, se debe tener cuidado de ignorar la secuencia especial de teclas que activan al TSR si es que éste ya se encuentra activo.

Segundo, si no se implementa el stack local la rutina o programa deberá utilizar el stack perteneciente al programa que fue interrumpido. Bajo estas circunstancias se corre el riesgo de desbordar dicho stack, lo cual congelaría la computadora.

Por lo tanto, el uso de un stack local dentro del TSR se deja a criterio del programador. Como regla general la mayoría de los TSRs implementan un stack interno para evitar problemas con el programa principal. Es muy conveniente entonces deshabilitar las interrupciones antes de cambiar al stack interno y volverlas a habilitar después de realizar dicho cambio.

4) "Cuando el código del TSR reemplaza al manejador de errores críticos de DOS (Critical Error Handler, INT 24h), nunca se debe hacer un salto desde dentro del código del manejador hacia otro punto del TSR.

En cambio, la nueva ISR debe asignar el número del error a alguna bandera interna la cual pueda ser verificada por las rutinas encargadas del manejo de errores críticos en el TSR. Estas rutinas serán llamadas posteriormente. Después de esto se debe regresar el control a DOS mediante una instrucción IRET, enviando un código de respuesta para que se ignore el error. Esto asegura que los diferentes stacks de sistema operativo serán limpiados apropiadamente." [24:p.411]

5) Antes de ejecutar el proceso de desinstalación el TSR debe verificar que toda la memoria por encima de él esté libre. Esta verificación se debe realizar para no dejar huecos de memoria que no podrían ser utilizados por DOS aunque el TSR ya no se encuentre ahí.

Como se dijo anteriormente, sí es posible desinstalar el TSR aunque la memoria por encima de él no esté libre; esto se logra implementando 'Interrupt Headers'. Estos encabezados permiten que un programa residente (device driver o TSR) pueda liberarse de la cadena de vectores de interrupción. Para que esta técnica funcione apropiadamente se requiere que todos los programas residentes en memoria, sin excepción, implementen los interrupt headers, ya que si alguno no lo hace la cadena de interrupciones quedaría truncada al momento de desinstalar cualquier programa. "Desafortunadamente, muy pocos programas residentes implementan esta tecnología lo cual limita grandemente su utilidad. Ni la misma Microsoft ha incorporado los interrupt headers en sus programas residentes " [19:p.51]

LINEAMIENTOS PARA USO DEL TECLADO.

1) "El TSR no debe tomar el control absoluto de la interrupción de hardware del teclado (INT 09h), es decir, no debe reemplazar a la interrupción. Lo más que se debe hacer es interceptar la interrupción y agregarle el código que requiera el TSR, pero siempre permitiendo que la rutina de interrupción original procese las entradas de teclado.

Si se utiliza esta interrupción para activar el TSR mediante una secuencia de teclas específica, se debe prender una bandera para indicar que el programa ha sido invocado, y después permitir que se continúe el proceso a lo largo de la cadena de vectores.

2) Cuando el TSR está activo y corriendo se debe utilizar la interrupción 16h del BIOS (ROM BIOS Keyboard_IO routine), y no la INT 09h, para leer el teclado. Esto permite que todos los TSRs en la cadena puedan monitorear las teclas presionadas y así poder tomar las acciones necesarias." [24:p.410]

LINEAMIENTOS PARA USO DEL VIDEO.

1) "Para realizar la manipulación de los diferentes recursos del video (color, cursor, modo de video, etc.), las rutinas del BIOS en ROM deben ser utilizadas siempre que sea posible en lugar de realizar accesos directos al hardware del sistema." [24:p.411]

2) Cuando se utiliza el BIOS para manipular el subsistema de video, el mismo BIOS actualiza con cada cambio todas sus áreas de datos. En contraste, cuando un programa realiza cambios mediante accesos directos al hardware deja sin actualizar estas áreas, por lo tanto, el programa que realice cambios directos también es responsable de mantener la integridad de las áreas de datos del BIOS. De no hacerlo provocaría que los datos reportados por el BIOS estuvieran fuera de sincronización con el estado real en que se encuentra la computadora.

La descripción de los procesos para actualizar la áreas de datos del BIOS sale del tema de este trabajo de investigación. Es por esto que a continuación únicamente se mencionan algunos ejemplos sin entrar en mayor detalle.

(Nota: Para mayor información acerca de las áreas del BIOS que están relacionadas con el subsistema de video se puede consultar el libro: PROGRAMMER'S GUIDE TO PC & PS/2 VIDEO SYSTEMS. Ver: 'Referencias bibliográficas').

- Si se modifica directamente el modo de video se deben actualizar los valores del área de datos del BIOS desde la dirección 0040h:0049h hasta la 0040h:0066h.

- "Si se modifica el bit que indica parpadeo/intensidad mediante el puerto 3D8h (Mode Control Register), este cambio también se debe reflejar en el área de datos del BIOS que se encuentra en la dirección 0040h:0065h.

- Por lo general el cursor de hardware debe ser utilizado, pero si el programa por alguna razón lo deshabilita para utilizar un cursor generado por software, la posición de este nuevo cursor también debe ser mantenida en el área de datos del BIOS situada en 0040h:0050h - 0040h:005Fh, dependiendo de qué página de video esté siendo utilizada. Esto para que el BIOS pueda seguir reportando la posición correcta del cursor en pantalla.

- Si se modifica el juego de colores directamente (sólo para tarjetas EGA y VGA) el programa debe copiar los valores de los nuevos colores en el área de datos respectiva. Esta área se puede localizar mediante el segundo apuntador de la tabla que está situada en la dirección 0040h:00A8h." [24:p.411]

3) Aunque no es un requerimiento absolutamente necesario un TSR debería verificar que el juego de colores (cuando está corriendo en sistemas con tarjeta EGA o VGA) no ha sido modificado. En caso de que el juego de colores sea distinto al que el TSR necesita, entonces él mismo debe guardar los valores actuales y volverlo a modificar de acuerdo a sus propias necesidades. Cuando el TSR sea desactivado deberá restaurar los colores que estaban vigentes cuando fue activado. Esto es algo que SideKick (específicamente la versión 1.58a) no hace. Se puede hacer el experimento de correr Works y después activar y desactivar SideKick. SK aparecerá con sus colores originales pero los colores de Works serán modificados. Esto quiere decir que SideKick modifica los colores de acuerdo a sus necesidades, pero no se preocupa por restaurar los colores de la aplicación principal.

CAPITULO 4 ANALISIS DE CODIGO

En este capítulo se describirán los mismos procesos estudiados en el capítulo 3 pero desde el punto de vista práctico. Se presentarán básicamente las mismas secciones pero la explicación será reemplazada por el código real que desempeña las funciones deseadas.

La primera sección del capítulo será dedicada a los TSRs pasivos incluyendo a los interrupt handlers. En segundo lugar se presentará el código para cada una de las diferentes porciones que constituyen un TSR interactivo.

TSRs PASIVOS.

La primera porción de código que será presentada es en realidad un programa completo, pero es un programa tan simple que conviene estudiarlo como un todo y no en sus partes constitutivas. El programa XKBDBUF.PAS (eXtend KeyBoarD BUffer) extiende la longitud del buffer de teclado. Por lo general este buffer tiene capacidad para almacenar 16 teclas por adelantado y a menudo resultan insuficientes, de ahí la utilidad de este TSR.

En realidad el programa no extiende el buffer sino lo reemplaza y para lograrlo actualiza algunas áreas de datos del BIOS, de tal forma que el mismo BIOS y el sistema operativo reconozcan y utilicen este nuevo buffer.

La característica principal de este programa es que no necesita ser activado una vez que ya se encuentra residente, es decir, la única utilidad del programa es la de proteger la memoria asignada al nuevo buffer de teclado para que ningún otro programa pueda sobrescribir en ella. Por esta razón no se captura ninguna interrupción, ni tampoco se localiza ni salva la dirección de ninguna bandera del sistema operativo.

Listado 4-1. XKBDDBUF.PAS [9:p.652]

```

program XKbdBuf;
($M 1024,0,0)  { 1K stack, no heap }
uses  dos;

const
  BUFSZ = 512;    { Enough for 256 keys }
  PSPSZ = 256;
  CODESZ = 1000;  { Estimated code size }

var
  Regs : registers;

  KbdBufBegin   integer absolute $0000:$0480,
  KbdBufEnd     integer absolute $0000:$0482,
  KbdBufHead    integer absolute $0000:$041A,
  KbdBufTail    integer absolute $0000:$041C,
  KbdBuffer     array[1..BUFSZ] of byte;

(----- Main program -----)
begin
{ Set the offset of the buffer relative to segment 40h }
  KbdBufBegin :=
    (seg(KbdBuffer[1])-$40) * 16 + ofa(KbdBuffer[1]),

{ Adjust the rest of the pointers }
  KbdBufEnd   = KbdBufBegin + BUFSZ,

{ If head = tail then buffer is empty }
  KbdBufHead := KbdBufBegin,
  KbdBufTail := KbdBufBegin,

{ Terminate and Stay Resident }
{ Specify resident size }
  Regs.dx := (PSPSZ + BUFSZ + CODESZ) div 16 + 1,
  Regs.ax = $3100;
  mados(Regs);
end { program - XKbdBuf }

```

Sólo las tres últimas líneas del programa están relacionadas con un TSR. Estas líneas llaman a la función 31h del sistema operativo (msdos=INT 21h). Esta función espera que el registro DX contenga la cantidad de memoria que se desea dejar residente, la cual debe estar expresada en paragraphs (párrafos=16 bytes).

INTERRUPT HANDLERS

Los siguientes dos programas son un poco más complejos pues ya interceptan una o varias interrupciones. El primero de ellos sigue siendo suficientemente sencillo como para no necesitar de porciones diferenciadas de inicialización, activación, etc.

Este programa es un ejemplo de los denominados Interrupt Handlers, los cuales al instalarse en memoria modifican la forma como responde una determinada interrupción. La principal característica que distingue a un interrupt handler es que el código que realiza el trabajo real está contenido dentro de la rutina de interrupción misma (ISR), es decir, no existe una rutina principal independiente.

Específicamente, este programa intercepta la interrupción 09h, es decir, la interrupción de hardware del teclado. Esto se hace con el fin de proporcionar los caracteres áéíóúñ sin necesidad de cargar un interceptor de teclado completo como el KEYB.COM que se distribuye con el sistema operativo.

Listado 4-2. F11.PAS

```

program F11;

($M 1024,0,0 )
($I-,R-,S-,D-)

uses dos;

const
  scF11 = $57,      { F11's scan code }
  BIOSDATA = $0040,

var
  AFlag : byte,
  _OldInt09 : procedure,

{ Keyboard buffer head ptr }
  KbdBufHead : integer absolute $0000:$041A,

{$P+}
procedure newint09; interrupt;
{ New keyboard ISR }
begin
  { Call old interrupt }
  asm pushf end; { pushf to simulate interrupt }
  _OldInt09,

{ Check if F11 was pressed }
  if Port[$60] = scF11 then AFlag := 1

{ Don't act when F11 is released }
  else if AFlag = 1 then AFlag = 2
  else
    if AFlag = 2 then
      begin
        case char(mem[BIOSDATA.KbdBufHead]) of
          'a': mem[BIOSDATA.KbdBufHead] := byte('á'),
          'e': mem[BIOSDATA.KbdBufHead] := byte('é'),
          'i': mem[BIOSDATA.KbdBufHead] := byte('í'),
          'o': mem[BIOSDATA.KbdBufHead] := byte('ó'),
          'u': mem[BIOSDATA.KbdBufHead] := byte('ú'),
          'n': mem[BIOSDATA.KbdBufHead] := byte('ñ'),
          'N': mem[BIOSDATA.KbdBufHead] := byte('Ñ'),
        end, { case }

{ Don't reset flag if shift is pressed to get Ñ }
        if ((mem[BIOSDATA.$0017] and $02) <> $02) and
           ((mem[BIOSDATA.$0017] and $01) <> $01) then
          AFlag := 0,
        end,
      end, { newint09 }
{$P-}

[----- Main program -----]
begin
  AFlag := 0,

{ Save current ISR }
  getintvec($09,@_OldInt09);

{ Insert F11's ISR into keyboard's chain }
  setintvec($09,@newint09);

{ Terminate and Stay Resident }
  keep(0),
end. { program - F11 }

```

Como se puede ver, Turbo Pascal proporciona los procedimientos para realizar todas las tareas de inicialización necesarias. Así, con 'getintvec' se obtiene el vector actual de la interrupción y con 'setintvec' se modifica este valor. En este código se ha reemplazado la llamada directa a la función 31h con una llamada a la función 'keep', lo cual facilita grandemente el proceso de dejar residente el programa pues ya no se debe calcular manualmente la cantidad de memoria que ha de permanecer reservada. El precio que se paga por esta facilidad es que el tamaño del programa residente en memoria aumentará considerablemente.

El proceso de encadenamiento se realiza en las dos primeras líneas de la ISR (newint09) antes de hacer cualquier otra cosa. Es importante notar que estas dos líneas de código dependen exclusivamente de la versión 6.0 o superior de Turbo Pascal. En versiones anteriores del compilador se debe utilizar ensamblador en línea (inline) para realizar estos mismos procesos. Otro detalle importante es el hecho de que una ISR en Turbo Pascal debe ser compilada como 'far', de ahí las directivas de compilación {F+} y {F-}. El resto del código es específico para esta aplicación y, por lo tanto, no será explicado pues no está directamente relacionado con la implementación de un TSR.

A continuación se presenta el código de CLOCK.ASM, el cual es un interrupt handler que despliega un reloj digital en la esquina superior izquierda de la pantalla. Este programa es más complejo que el anterior ya que posee dos porciones de código perfectamente definidas: una de inicialización/instalación y otra para realizar el trabajo real. El listado 4-3 muestra el código de inicialización/instalación.

Listado 4-3 Inicialización de un Interrupt Handler. [8:p.148]

```

...

Interrupts    segment at 0h
               org 1Ch*4      ,1Ch is the timer interrupt

TimerInt     label word
Interrupts    ends

loadclock     proc near
, Initialization routine (Program's transient portion)

    assume ds Interrupts      ,The Interrupt Vector Table
    mov     ax,Interrupts     ,will be used as data segment
    mov     ds,ax
    cli                                ,Disable interrupts

, Get current Timer interrupt vector because
, it has to be called before our interrupt
    mov     ax,TimerInt
    mov     _OldTimerInt,ax
    mov     ax,TimerInt[2]
    mov     _OldTimerInt[2],ax

; Set Clock routine as the new Timer interrupt routine
    mov     TimerInt,offset clock
    mov     TimerInt[2],cs
    sti                                ,Enable interrupts again

, Use video services to determine # of columns
    mov     ah,15
    int     10h
    sub     ah,6      ,Subtract space for displaying the clock
    shl     ah,1      ;Times 2 (chars and attrs)
    mov     byte ptr DispPos,ah ;Set ClkDisp position

```

```

                                ;within video memory
mov    VideoPort,03BAh    ,Default = monochrome display
test   al,4                ;Is it really monochrome
jnz    GET_TIME            ,Yes
add    DiapPos,8000h        ,No, set up for color display
mov    VideoPort,03DAh

GET_TIME.
; Avoid displaying 00:00 for the first 15 seconds
call   calc

; Terminate and Stay Resident
; Do not keep transient portion resident
mov    dx,offset loadclock
int     27h
loadclock    endp

```

La característica más notable de este programa es que no utiliza ni una sola llamada al sistema operativo, es decir, todos los procesos se realizan con accesos directos a memoria o al hardware del sistema. Este hecho provoca que el código utilizado sea poco convencional y, por lo tanto, no se verá en ningún otro programa de demostración. La ventaja que se obtiene al no llamar al sistema operativo radica en que no se deben localizar la InDos Flag ni la Critical Error Flag. Como consecuencia, CLOCK.ASM es el único programa de demostración incluido en este trabajo que es capaz de correr bajo las versiones 1.x de MS-DOS.

En primer lugar se obtiene el vector actual del reloj del sistema. Esto se hace definiendo un área de datos que coincide exactamente con la Tabla de Vectores de Interrupción (Interrupts segment at 0h). Esta área de datos también define la posición, dentro de la tabla, del vector buscado (Timer = 1Ch). Después, con simples instrucciones 'mov' se obtiene la dirección actual y se modifica la tabla de vectores para que apunte a la nueva ISR implementada dentro del código de CLOCK.ASM. Nótese que la modificación se realiza en medio de las instrucciones 'cli' y 'sti', la primera de las cuales deshabilita la generación de interrupciones y la segunda la vuelve a habilitar. Estas dos instrucciones se utilizan cuando el programa ha de realizar 'Código Crítico', es decir, código que no debe ser interrumpido.

El resto de la inicialización es específica para este programa y sólo verifica el tipo de la tarjeta gráfica que está instalada en la máquina e inicializa la cadena que será desplegada por el reloj digital.

El proceso de instalación en memoria de CLOCK.ASM constituye otra de las peculiaridades de este programa ya que utiliza la INT 27h en lugar de la INT 21h función 31h. Esta interrupción necesita que CS apunte al PSP del programa actual y que DX contenga la dirección a partir de la cual puede ser cargado el siguiente programa, es decir, la dirección del último byte que ha de quedar residente + 1. La limitante que tiene esta interrupción es que sólo permite dejar residente un máximo de 64Kb.

Con esto concluye la porción de inicialización e instalación del programa. A continuación se muestra el código del manejador de la interrupción del reloj incluido en CLOCK.ASM, el cual despliega el reloj digital en pantalla.

Listado 4-4. Timer-Interrupt Handler. [8:p.148]

```

clock  proc near
; New timer interrupt (INT 1Ch)

; Save all registers used by this routine
push   ax
push   cx
push   di
push   si
push   ds
push   es

; Must push flags before calling any interrupt routine
; Call old timer interrupt
pushf
call    dword ptr _OldTimerInt

    assume ds,BiosStorage ,Change current data segment
    mov  cx,BiosStorage  ,Find the system's current time
    mov  ds,cx

; Check if time needs to be recalculated
    mov  cx,RecalcCount
    inc  cx
    cmp  cx,273          ,Recalc every 273 counts = 15 seconds
    jb   DONT_RECALC
    call  calc            ,Recalculate now
    mov  cx,0            ,Prepare for resetting RecalcCount
DONT_RECALC
    mov  RecalcCount,cx ,Store incremented or zero value

; Change extra segment to point to video memory
    assume es,VidMem
    mov  cx,VidMem
    mov  es,cx

    mov  dx,VideoPort
    mov  di,DispPos      ,Cursor location = destination
    lea  si,ClkDisp       ,Display in memory = source
    mov  cx,10           ,Move 5 chars and 5 attrs
    cli                               ;No interrupts during critical section

; Wait until it is safe to write to video memory
SCAN_LOW:
; Loop until video controller scan status is low
    in   al,dx
    test al,1
    jnz  SCAN_LOW
    mov  ah,cs [si] ,Load ah with the byte to be written

SCAN_HIGH:
; After port has gone low, it must go high before it is
; safe to write directly to video memory
    in   al,dx
    test al,1
    jz   SCAN_HIGH

    mov  es:[di],ah      ,Move the byte to video memory
    inc  di              ,One byte at a time
    inc  si
    loop SCAN_LOW        ,Repeat for next byte
    sti                               ,Interrupts enabled after critical section

; Restore all registers used by this routine
    pop  es
    pop  ds
    pop  si
    pop  di
    pop  cx
    pop  ax
    iraq                ;Interrupt return
clock  endp

```

Nótese que lo primero y lo último que se hace en la ISR es salvar y restaurar los valores de todos los registros del procesador que serán modificados durante la ejecución de la rutina. Esto se debe hacer en todas y cada una de las ISRs implementadas por el usuario.

(Nota: Los programas en Turbo Pascal, no necesitan incluir este código, ya que el mismo compilador genera automáticamente el código para salvar todos los registros. Otra vez, esto facilita la programación pero hace que el código en pascal sea más grande de lo necesario. Por ejemplo, si la ISR sólo modifica cinco registros sólo debería salvar esos registros, sin embargo Turbo Pascal salva todos sin importar cuantos sean realmente utilizados).

Después de salvar los registros se llama a la rutina de interrupción anterior (encadenamiento) y se despliega el reloj en pantalla. Nótese que para desplegar el reloj se accesa directamente la memoria de video.

TSRs INTERACTIVOS.

En esta sección se irá presentando el código para cada una de las diferentes porciones que constituyen un TSR interactivo. No se presentarán los programas completos como se hizo con los ejemplos anteriores sino únicamente las porciones del código que sean necesarias para demostrar la implementación del TSR. Además, cuando sea necesario se presentará el código de varios programas, realizando el mismo proceso, con el fin de ejemplificar los diferentes métodos de implementación.

PORCION TRANSITORIA.

Código de inicialización.

- Determinar la versión de DOS.

Antes de poder utilizar los servicios del sistema operativo es necesario que el TSR verifique bajo que versión de DOS está corriendo. Un TSR que haga llamadas al sistema operativo necesita como mínimo la versión 2.0.

Con respecto a los programas de demostración, todos corren bajo DOS 2.0 excepto JTSRTP.PAS-SAVESCR.PAS y BLANKER.ASM los cuales necesitan la versión 3.1 como mínimo.

Listado 4-5. Verificar la versión de DOS.

```

...
; Check DOS version
mov     dx,offset Err2
mov     ah,30h           ;Get MaDos version
int     21h
xchg    al,ah
cmp     ax,30Ah          ;If DOS >= 3 1, continue
jnb     ABORT_INSTALL

```

El código para verificar la versión de DOS es muy sencillo pues sólo es necesario hacer un llamado a la función 30h de DOS. La función regresa en AX la versión del sistema operativo que está corriendo en la computadora. Específicamente AL contiene la versión mayor y AH la menor. En pascal simplemente se debe llamar a la función 'dosversion', la cual regresa la versión en forma análoga a la INT 21h (ver: listado 4-8).

(Nota: Si el programa SETVER.EXE de MS-DOS 5 está instalado en memoria, esta función puede regresar información incorrecta).

- Prevenir más de una instalación.

Como ya se dijo en el capítulo anterior, existen varias formas de detectar la presencia del TSR en memoria para así prevenir múltiples instalaciones del mismo. A continuación se presentan tres diferente métodos para implementar esta detección. Los métodos utilizan: la interrupción 2Fh o Multiplex, rastreo de memoria y la firma del TSR.

Listado 4-6. Prevenir más de una instalación del TSR utilizando la INT 2Fh.

```

;=====
; New Multiplex ISR (INT 2Fh)
;=====
new2FIsr    proc far
    cmp     ax,0DD00h
    je      SET_INSTALLED_FLAG
    jmp     _Old2FIsr

SET_INSTALLED_FLAG.
    mov     ax,1992h           ;Set installed flag
    iret                    ;Return to transient portion
new2FIsr    endp

...

; Check TSR already installed
mov     dx,offset Err3
mov     ax,0DD00h           ;If ID=0DD already installed
int     2Fh                 ; abort installation
cmp     ax,1992h
jne     CONTINUE_INIT

```

Este código corresponde al programa BLANKER.ASM y como se puede ver, para implementar este método se requiere de una rutina asociada a la interrupción Multiplex, la cual debió haber sido instalada por el código de inicialización de la primera copia del TSR. Así, para realizar la detección de ésta sólo se necesita hacer una llamada a dicha interrupción. Si la llamada para la ID=DDh regresa AX=1992, alertaría a la segunda copia potencial para que no se volviera a instalar. El ID=DDh es arbitrario y se podría utilizar cualquier otro siempre y cuando no se utilicen los IDs reservados por Microsoft (ver: 'Proceso de inicialización', capítulo 3).

Este código no sigue el método tradicional que simplemente verifica que AL contenga un valor diferente de cero, que es el valor con el que se inicializa AL con la instrucción 'mov ax,0DD00h', sino que espera un valor específico (1992) como resultado de la INT 2Fh. Con esto se evitan confusiones si se da el caso de que más de un programa se haya instalado con la ID=DDh.

Listado 4-7. Prevenir más de una instalación del TSR utilizando rastreo de memoria. [19:p.49]

```

;-----
; FIND_COPY determines if the TSR is already resident
; Exit  CF - clear if copy found
;       ES - segment of installed code (if CF = 0)
;-----
find_copy    proc near
    not      word ptr [BEGIN]    ; Initialize fingerprint
    mov     bx,0A000h           ; Start scan in UMBS
    mov     ax,cs               ; Keep CS value in AX

FIND_COPY_1
    inc     bx                  ; Increment search seg value
    mov     es,bx
    cmp     ax,bx               ; Not installed if current
    je      FIND_COPY_2        ; segment is looped back to
    mov     si,offset BEGIN     ; Search this seg for ASCII
    mov     di,si               ; fingerprint
    mov     cx,16
    repe    cmpsb
    jne     FIND_COPY_1        ; Loop back if not found
    cld

FIND_COPY_EXIT
    ret

FIND_COPY_2
    stc
    ret
find_copy    endp

```

El listado 4-7 corresponde al programa TSRTMPLT.ASM (TSR TeMPLaTe). Este código implementa un ciclo que busca los primeros 16 bytes de la imagen del programa en cada uno de los segmentos de memoria. Nótese que la búsqueda inicia en los Upper Memory Blocks. El ciclo (repe) compara la cadena apuntada por ES:DI con la cadena apuntada por DS:SI. El ciclo termina si se encuentra un segmento que contenga exactamente los 16 bytes buscados, o si BX es incrementado de tal forma que regrese nuevamente al segmento inicial de la búsqueda.

La última forma de localizar un copia ya residente del mismo TSR utiliza parte de los dos métodos anteriores, ya que para implementarlo primero se debe interceptar una interrupción con la cual se establece una línea de comunicación con la porción residente, y segundo se debe buscar la firma del TSR en memoria. Este es el método que utiliza el programa de demostración JTSRTP.PAS como se muestra en el siguiente listado.

Listado 4-8 Prevenir más de una instalación del TSR utilizando la firma del programa. [13:p.169]

```

function ok2inst1 (Count:word,
                  Arg,Name:string) : boolean,
{ Check DOS version and TSR already installed }
var
  Vec,
  Dif : word,
  Ptr,
  Rtn : pointer,
  PtrArr : array[1..2] of integer absolute Ptr;
  RtnArr : array[1..2] of integer absolute Rtn;
  St : string,
  Reg : registers;
begin
  ok2inst1 := FALSE,

  {--- Check DOS version ---}
  Vec := swap(dosversion),
  if Vec < $30A then
  begin
    writeln('DOS 3.1 REQUIRED');
    exit,
  end;

  {--- Check TSR already installed ---}
  TarName := Name, { Initialize TSR signature }
  Rtn := @tar_isr,

  { Calculate the difference between the interrupt service }
  { routine and data segment in the transient program }
  Dif := DSeg - RtnArr[2],

  for Vec := $60 to $67 do
  begin
    { Get the address for the interrupt vector }
    getintvec(Vec, Ptr);

    { If the address = NIL then the TSR program has not yet }
    { been installed. Set the interrupt to point to the }
    { TSR's interrupt service routine, save the vector's }
    { number and exit. }
    if Ptr = NIL then
    begin
      setintvec(Vec, Rtn),
      TarVec := Vec,
      ok2inst1 := TRUE,
      exit,
    end;

    { Store the signature in a string and compare it to the }
    { string passed to the function. If the two strings }
    { match, the TSR is already installed. Use the offset }
    { calculated previously to access the data segment of }
    { the TSR program }
    { If TSR already resident, check command line for /u }
    { Communicate with the TSR program via it's interrupt }
    move(mem[PtrArr[2]+Dif+ofs(TarName)], St,
         length(TarName)+1);
    if St = TarName then
    begin
      if (Count > 0) and (Arg[1] = '/') and
        (upcase(Arg[2]) = 'U') then
        Reg.ax := 1 { Uninstall command }
      else
        Reg.ax := $FF, { Invalid switch }
    end;
  end;

```

```

    intr(Vec, Reg);      { Communicate command to }
    exit,                { resident portion      }
end,
end, { for }
end; { ok2inst1 }

```

Este método es un poco más complicado que los anteriores como se podrá apreciar. En esencia el método consiste de dos fases: inicialización y un ciclo de búsqueda.

En la fase de inicialización se almacena el nombre del programa (la firma) en una variable interna y se calcula la distancia que hay entre el segmento de datos y la rutina de interrupción 'tsr_isr'. Esta rutina debió haber sido instalada en alguno de los vectores de interrupción 60h a 67h durante la inicialización de la primera copia del TSR. (La ISR 'tsr_isr' será explicada más adelante cuando se trate el tema de la comunicación con la porción residente).

El ciclo de búsqueda obtiene el vector de cada una de las interrupciones ya mencionadas. Si el vector no es nulo se presupone que está apuntando a la rutina 'tsr_isr' del TSR ya residente.

Si éste es el caso, al sumar la diferencia entre DS y 'tsr_isr' (calculada en la fase de inicialización de la rutina) al segmento del vector obtenido (sólo al segmento, no al offset) obtendremos la dirección del segmento de datos del TSR ya residente. Ahora bien, si dentro de ese segmento de datos buscamos en el offset que corresponde a la variable TsrName, la cual almacena el nombre del programa, podremos comparar la cadena que se encuentra en memoria con el nombre que recibió la rutina como parámetro, es decir, comparar las firmas. Todos estos cálculos se hacen como parte de la instrucción 'move'.

Si el vector obtenido no apunta a nuestra ISR entonces todos los cálculos anteriores simplemente arrojarán como resultado una cadena que de ningún modo podrá ser igual a la firma del TSR.

Por otro lado, si se encuentra que el vector obtenido es nulo se concluye que el TSR aún no ha sido instalado y se procede a capturar dicho vector. Por último, si no existen vectores de interrupción disponibles se aborta el proceso de instalación.

- Localizar banderas.

Este proceso depende de la versión mínima de DOS que el programa acepte. En primer lugar se presentará el código estándar para localizar la InDos Flag y la Critical Error Flag cuando el TSR pueda correr bajo DOS 2.0.

Listado 4-9 Localizar banderas de DOS 2.0 [19:p.49]

```

mov     ah,34h           ,Get INDOS address from DOS
int     21h
mov     word ptr [Indos_ptr],bx
mov     word ptr [Indos_ptr+2],es
call    find_cefptra     ;Find critical error flag
jnc     CEF_FOUND
mov     dx,offset ErrMsg6 ,Crit err flag not found
jmp     short ERROR_EXIT

;-----
; FIND_CFPTR returns a pointer to the DOS ErrorMode flag
; Exit - CF - clear if ErrorMode Flag found
;       ES:BX - Segment-Offset of ErrorMode Flag
;
; For versions of DOS before 3.1, search the following
; code in DOS seg.
;
;   36 80 3E ???? 00      cmp byte ptr es:[ErrorMode],0
;   75 ??                jne near label
;   CD 28                int 28h
;
; For DOS 3.1 and later, ErrorMode sits before INDOS flag
;
;-----
find_cefptra proc near
mov     ah,34h           ,Get ptr to INDOS flag
int     21h             ,After DOS 3.1 the CEF is
dec     bx              , documented to be the byte
cmp     Dos_version,30Ah , before the INDOS flag
jnc     FIND_CEF_EXIT

mov     ax,3E80h         ,CMP opcode
mov     cx,-1           ,Max segment size
mov     di,bx           ,Start at INDOS address

FIND_CEF_1:
repne   scasb           ,Scan for CMP
jcxz    FIND_CEF_NOTFOUND ,Error if CMP not found
cmp     es,[di],ah      ,Check other half of CMP
jne     FIND_CEF_1      , opcode
cmp     byte ptr es [di+4],075h ;Check for JNE
jne     FIND_CEF_1
cmp     word ptr es [di+6],028CDh ;Check for INT 28h
jne     FIND_CEF_1      , call

FIND_CEF_FOUND:
inc     di              ,Get offset of ErrorMode flag
mov     bx,es:[di]
clc

FIND_CEF_EXIT
ret

FIND_CEF_NOTFOUND
stc
jmp     short FIND_CEF_EXIT
find_cefptra endp

```

Las primeras líneas del listado 4-9 forman parte del código de inicialización del programa TSRTMPLT.ASM. Como se puede ver para localizar a la InDos Flag simplemente se llama a la función 34h de DOS.

Por otro lado, para localizar la Critical Error Flag se utiliza una rutina que emplea el método de rastreo (ver: 'Critical Error Flag', capítulo 3). Este método implementa un

ciclo que busca una serie de instrucciones desde la dirección de la InDos Flag en adelante. Estas instrucciones deben ser buscadas no por su nombre sino por su equivalente en hexadecimal, ya que estamos buscando dentro de código ejecutable. La secuencia específica de instrucciones a buscar se puede ver en el comentario al inicio de la rutina de búsqueda. Nótese que el rastreo se realiza únicamente si la versión de DOS es menor a 3.1.

A continuación se presenta (en el mismo listado) el código de los programas BLANKER.ASM y JTSRTP.PAS ya que ambos requieren de la versión 3.1 de DOS para poder correr adecuadamente. En ambos casos el código es esencialmente es el mismo.

Listado 4-10. Localizar banderas de DOS 3.1 ó superior.

```
BLANKER: .ASM -----
;
; -----
; Locate and save DOS status flags
; -----
; InDos Flag
mov     ah,34h
int     21h
mov     InDosFlagOfs,bx      ,Save offset
mov     InDosFlagSeg,es     ,Save segment

; Critical Error Flag
dec     bx                  ,Crit Err Flag=byte before
mov     CritErrFlagOfs,bx   , InDos Flag

;
;

JTSRTP PAS -----
;
; { Save address of InDos and Critical Error Flags }
Reg.ah := $34,
mados(Reg),
DosSeg := Reg.es,
InDosFlag := Reg.bx,
CritErrFlag := pred(Reg.bx),
;
;
; -----
```

La localización de la InDos Flag es exactamente la misma que en el listado 4-9 y la gran diferencia radica en que la Critical Error Flag se localiza simplemente decrementando la dirección de la InDos Flag en uno. Este hecho como ya se dijo está documentado para la versión 5.0 y superiores de MS-DOS, pero es válido desde la versión 3.1.

- Capturar vectores de interrupción.

Para demostrar la forma en la que se deben interceptar los diferentes vectores de interrupción utilizados por un TSR se mostrará el código en ensamblador que se puede considerar estándar para estos casos.

Listado 4-11. Captura de vectores de interrupción.

```

;-----
; Insert new ISRs into interrupt chains
;-----
mov     ax,3508h           ;Save old Timer ISR
int     21h
mov     Old08Ofs,bx
mov     Old08Seg,es
mov     ax,2508h           ;Set new Timer ISR
mov     dx,offset new08_isr
int     21h

mov     ax,3509h           ;Save old Kbd ISR
int     21h
mov     Old09Ofs,bx
mov     Old09Seg,es
mov     ax,2509h           ;Set new Kbd ISR
mov     dx,offset new09_isr
int     21h

```

Este código (de BLANKER.ASM) únicamente muestra la captura de la INT 08h y 09h, pero el procedimiento es el mismo para cualquier vector que se vaya a 'interceptar'. Primero se llama a la función 35h de DOS la cual espera que AL contenga el número de la interrupción de la que deseamos obtener la dirección. La función regresa en ES el segmento del vector y BX el offset. Después de salvar la dirección en variables internas se llama a la función 25h, la cual utiliza el DS actual y el offset de la nueva ISR (éste último debe estar almacenado en DX) para construir el nuevo vector con el que se modificará la tabla de vectores de interrupción.

Este código se puede complicar si algunas interrupciones sólo se encuentran disponibles bajo ciertas versiones de DOS, en cuyo caso antes de realizar la intercepción se deberá verificar dicho dato. También se pueden desarrollar subrutinas que realicen el proceso y así evitar la repetición de código para cada interrupción (ver: TSRTMPLT.ASM, apéndice H). En pascal el procedimiento es prácticamente el mismo como se puede ver en el listado 4-2.

Código de instalación.

Ya se han estudiado en la sección de TSRs pasivos las dos formas de instalar el programa en memoria, es decir, dejarlo residente (INT 27h e INT 21h función 31h). Para complementar esos listados se presentará la forma de llamar a la función 31h utilizando ensamblador.

Listado 4-12 Instalación del TSR en memoria

```

;-----
; Install resident portion in memory
;-----
; Display installed message,
mov     dx,offset Msg2
call    putmsg

; DX = amount of memory to leave resident
mov     dx,(offset ResPortionEnd - offset CSeg + 15)
mov     cl,4
shr     dx,cl           ;Convert to paragraphs
mov     ax,3100h        ;Go TSR
int     21h
initialize     endp

```

El código mostrado en el listado 4-12 se puede considerar estándar para todos los TSRs y por lo común se encuentra al final del código de inicialización. Lo importante de este código es la línea que almacena en DX la cantidad de memoria a dejar residente. La operación, básicamente calcula la distancia entre el inicio del segmento de código del programa y una variable (ResPortionEnd) que identifica el final de la porción residente, también dentro del segmento de código.

Los 15 bytes que se suman al final no son absolutamente necesarios pues simplemente actúan como un colchón. En otros casos (ver: TSRTMPLT.ASM, apéndice H) también se puede sumar una cantidad x de bytes, los cuales serán utilizados como stack interno. En el listado 4-12 (BLANKER.ASM) no se hace esto porque el stack interno está implementado a través de la variable 'StkTop' (ver: declaraciones de variables en BLANKER.ASM, apéndice D).

Por último, la cantidad de bytes calculada es dividida entre 16 para convertirla a paragraphs y se llama la función 31h de DOS para dejar el TSR residente en memoria.

Código de desinstalación.

Como se mencionó en el capítulo anterior, este código puede formar parte de la porción transitoria del TSR o puede quedar residente.

En primer lugar se mostrará el código para desinstalar el TSR en ensamblador, el cual está contenido en la porción transitoria de TSRTMPLT.ASM. En segundo lugar se mostrará el código en pascal que permanece residente (junto con todo el demás código utilizado para implementar las ISRs, la rutina principal, etc.) del programa JTSRTP.PAS.

Cabe hacer notar que a pesar de las diferentes implementaciones, ambos procesos de desinstalación son iniciados desde la línea de comandos utilizando la opción '/u', pero con la diferencia de que el segundo método puede alterarse para que el TSR sea desinstalado mediante alguna combinación especial de teclas presionadas cuando la

rutina principal del TSR esté activa, es decir, JTSRTP.PAS puede ser modificado para poder desinstalarse sin necesidad de volver a correr el programa.

Listado 4-13. Código de desinstalación transitorio. [19 p 49]

```

,-----
, REMOVE deallocates the memory block addressed by EMS
, and restores the interrupt vectors displaced on
, installation
, Exit  CF - clear if program uninstalled
,       CP - set if can't uninstall
,-----
remove proc near
    assume  cs,CSeg, ds,CSeg
    cmp     TSRResident,0      ;See if TSR installed
    jne     REMOVE_1
    mov     dx,offset ErrMsg5
    jmp     REMOVE_ERROR_1

REMOVE_1:
    mov     es,Installed_seg
    mov     ax,5                ;Unhook notification chain
    mov     di,offset CallbackInfo
    cmp     word ptr es:[SWServices+2],0
    je      REMOVE_2
    call    es:[SWServices]

REMOVE_2:
    mov     al,08h              ;Check interrupt 08h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,09h              ;Check interrupt 09h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,10h              ;Check interrupt 10h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,13h              ;Check interrupt 13h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,16h              ;Check interrupt 16h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,28h              ;Check interrupt 28h vector
    call    check_vector
    je      REMOVE_3

JMP_REMOVE_ERROR:
    jmp     REMOVE_ERROR

REMOVE_3:
    cmp     Dos_version,300h    ;DOS 3.0 and up
    jb      SKIP_CHECK2F
    mov     al,2Fh              ;Check interrupt 2F vector
    call    check_vector
    jne     REMOVE_ERROR

SKIP_CHECK2F:
    cmp     Ems_flag,0          ;If using expanded memory
    je      SKIP_REMOVE_EMS     ; free it
    mov     dx,Ems_handle       ;Free expanded memory
    call    free_emsmem
    jne     REMOVE_ERROR

SKIP_REMOVE_EMS:
    push    ds                  ;Save DS
    assume  ds,nothing
    mov     ax,2508h            ;Restore interrupt 08h vector
    lds     dx,es:[Int08h]

```

```

int      21h

mov      ax,2509h      ,Restore interrupt 09h vector
lds      dx,es,[Int09h]
int      21h

mov      ax,2510h      ,Restore interrupt 10h vector
lds      dx,es[Int10h]
int      21h

mov      ax,2513h      ;Restore interrupt 13h vector
lds      dx,es[Int13h]
int      21h

mov      ax,2516h      ,Restore interrupt 16h vector
lds      dx,es,[Int16h]
int      21h

mov      ax,2528h      ,Restore interrupt 28h vector
lds      dx,es[Int28h]
int      21h

cmp      cs.Dos_Version,300h ;Dos 3.0 and up
jb       SKIP_RESTORE2F
mov      ax,252Fh      ;Restore interrupt 2Fh vector
lds      dx,es:[Int2Fh]
int      21h

SKIP_RESTORE2F.
pop      ds            ;Restore DS
assume   ds CSeg
not      word ptr es [BEGIN] ;Destroy fingerprint
mov      ah,49h        ,Free memory given to
int      21h          , original program block
jc       REMOVE_ERROR ;Branch on error

mov      dx,offset InfoMsg2 ,Tell user TSR is now
call     message       ; removed
cld      ;Clear CF for exit

REMOVE_EXIT
ret      ,Exit with CF intact

REMOVE_ERROR
mov      dx,offset ErrMsg4 ,Error during remove

REMOVE_ERROR_1
stc
jmp      REMOVE_EXIT ;TSR was not installed
remove   endp

;-----
; CHECK_VECTOR is called by remove to compare the segment
; pointed to by an interrupt vector against a segment
; value supplied by the caller.
; Entry: AL - interrupt number
; Exit:  ZF clear - segment do not match
;       ZF set - segment match
;-----
check_vector proc near
push     es
mov      cx,es
mov      ah,35h        ,Get vector
int      21h
mov      ax,es          ;Transfer segment to AX
cmp      ax,cx          , and compare
pop      es
ret
check_vector endp

```

Este código es bastante complejo pues se trata de un programa que puede hacer uso de los servicios de un task switcher y también puede utilizar memoria expandida (para una explicación de este código, ver: capítulo 5). En esta sección simplemente se describirá el código convencional.

Lo primero que se hace (REMOVE_2:) es verificar que el segmento de todos los vectores que el TSR capturó, sigan apuntando a las ISR de la porción residente. Este proceso se lleva a cabo en la subrutina 'check_vector'. Es importante notar que en todo momento, desde que se entra a la rutina 'remove', ES contiene el segmento de memoria en donde fue encontrada la primera copia del TSR ya instalado (ver: listado 4-7).

Si todos los vectores pasan la prueba se procede a restaurar los valores que fueron salvados durante el proceso de captura, de lo contrario se aborta el proceso de desinstalación. Por último se libera la memoria que está siendo utilizada por la porción residente. Esto se hace con la función 49h de DOS, la cual necesita que ES contenga la dirección del segmento de memoria que ha de ser liberado.

Listado 4-14. Código de desinstalación residente. [13:p.169]

```

procedure reltarmem,
{ Release TSR's memory before unloading it }
var
  Reg : registers,
begin
  while mem[EndDosMem 0] = $4D do
  begin
    { Deallocate memory if it belongs to the TSR }
    if memw[EndDosMem.1] = TerPSP then
      begin
        Reg.ah := $49,
        Reg.es := EndDosMem + 1,
        msdos(Reg);
      end,
    { Get next MCB }
    EndDosMem := EndDosMem + memw[EndDosMem.3] + 1,
  end,
end, { reltarmem }

function uninstalltsr : boolean;
{ Remove the program from memory if possible }
var
  T,K,I : pointer,
begin
  { Get current values of the interrupts the TSR }
  { has captured }
  getintvec(TIMERINT, T),
  getintvec(KBDINT, K),
  getintvec(IDLEINT, I);

  { If the current values in the interrupt vectors point }
  { to the TSR's routines then no other program have }
  { stolen the TSR's interrupts and the TSR can be }
  { unloaded safely }
  if (T = @newtimer) and (K = @newkbd) and
    (I = @newidle) then
  begin
    { Restore original interrupt handlers' vectors }
    setintvec(TIMERINT, @OldTimerVec);
    setintvec(KBDINT, @OldKbdVec),
    setintvec(IDLEINT, @OldIdleVec),
    setintvec(TerVec, NIL),
    reltarmem, { Deallocate all memory }
    uninstalltsr := TRUE;
  end
  else
    uninstalltsr := FALSE;
  end, { uninstalltsr }

```

Con lo que respecta a la verificación de vectores, el código en pascal es prácticamente igual al anterior. El proceso de liberación de memoria por otro lado es bastante diferente. En esencia el método consiste en buscar el primer MCB asignado al TSR, iniciando la búsqueda desde el final de la memoria ocupada por el sistema operativo (no tiene caso buscar en esa memoria pues el TSR no puede estar ahí). Como se recordará, el segundo y tercer campos de cada MCB (ver: 'Asignación de memoria', capítulo 1) contiene el PID del proceso al que pertenecen, el cual no es otra cosa que el PSP del programa en cuestión.

Entonces sólo se debe implementar un ciclo que busque en cada MCB el valor del PSP asignado a la primera copia del TSR y liberar todos esos MCBs utilizando, otra vez, la función 49h de DOS.

PORCIÓN RESIDENTE.

Código de activación.

Como se mencionó en el capítulo anterior, este proceso consiste en detectar el momento adecuado en el que se debe activar la rutina principal del TSR y en salvar todos los parámetros configurables que dicha rutina vaya a modificar durante su ejecución.

- Determinar el momento de activación.

La primera parte de este proceso se lleva a cabo dentro de las ISRs implementadas en la porción residente. El código de estas rutinas no se puede considerar estándar para todos los TSRs, pues varía según la aplicación para la que haya sido diseñado el programa. Es por esta razón que no serán listadas todas las ISRs de los programas de demostración pues sería demasiado código.

Sin embargo, a continuación se muestran las ISRs para las interrupciones 08h, 09h, 10h, 13h y 28h, las cuales son las más frecuentemente encontradas en un TSR.

Listado 4-15. ISRs 08h, 09h, 10h, 13h y 28h. [19 p49]

```

,=====
, New handler for int 8h (timer)
,=====
new8_hndlr  proc near
    pushf                    ,Simulate int
    call     Old8Hndlr       ,Chain to old handler

    ; Check program's HotKey
    cmp     HotKey,0         ,HotKey pressed ?
    je      HKEY_0           ,If not, exit

    ; Check peripherals
    cmp     VideoFlag,0      ,Int 10h busy ?
    jne     DECKEY           ,If yes, dec HotKey flag
    cmp     DiskFlag,0       ,Int 13h busy ?
    jne     DECKEY           ,If yes, dec HotKey flag

    push    di               ,Save registers
    push    es

    ; Check value for InDos flag
    mov     di,InDosOfs      ,Offset of flag
    mov     es,InDosSeg      ,Segment of flag

```

```

    cmp     byte ptr es.[di],0
    jne     POPSTK          ,Exit if DOS busy

; Check critical error flag
    mov     di,ErrFlagOfs   ,Offset of flag
    cmp     byte ptr es [di],0
    jne     POPSTK          ,Exit if flag set

; Everything OK, pop up the clock
    pop     es              ;Restore registers
    pop     di
    mov     HotKey,0        ,Clear HotKey flag
    call    pop_up          ,Run clock routine

HKEY_0:
    iret

POPSTK:
    pop     es
    pop     di

DECHKEY:
    dec     HotKey
    iret                ,Return to MsDos
new8_hndlr endp

;=====
; New handler for int 9h (keyboard hardware interrupt)
;=====
new9_hndlr proc near
    sti                ,Enable interrupts
    pushf              ;Simulate int
    call    Old9Hndlr

; Check keyboard for HotKey pressed
    push     ax          ,Save ax
    mov     ah,02h       ,Get toggle keys' status
    int     16h          ,Call BIOS keyboard routine
    and     al,0Fh       ,Compare toggle keys' status with HotKey
    cmp     al,06h       ;Ctrl+LeftShift pressed ?
    pop     ax           ,Restore ax
    jne     EXIT_09      ,If HotKey not pressed, exit

; HotKey was pressed
    cmp     RunningFlag,0 ,Program already running ?
    jne     EXIT_09      ,If yes, exit

; Initialize HotKey counter
    mov     HotKey,18    ,HotKey is now active

EXIT_09:
    iret                ,Return to MsDos
new9_hndlr endp

;=====
; New handler for int 10h (ROM BIOS video service)
;=====
new10_hndlr proc near
    pushf              ,Simulate int
    inc     VideoFlag
    call    Old10Hndlr
    dec     VideoFlag
    iret
new10_hndlr endp

;=====
; New handler for int 13h (ROM BIOS disk service)
;=====
new13_hndlr proc far
    pushf              ,Simulate int
    inc     DiskFlag
    call    Old13Hndlr
    pushf              ,Protect flags
    dec     DiskFlag
    popf              ,Restore flags
    ret     2           ,Return to MsDos, discard 2 bytes
new13_hndlr endp

```

```

;=====
; New handler for int 28h (DOS scheduler)
;=====
new28_hndlr  proc near
    pushf                    ;Simulate int
    call     Old28Hndlr      ;Chain to old handler

    ; Check program's HotKey
    cmp     HotKey,0          ;HotKey pressed ?
    je      EXIT_28          ;If not, exit

    ; Check peripherals
    cmp     VideoFlag,0       ;Int 10h busy ?
    jne     EXIT_28          ;If yes, exit
    cmp     DiskFlag,0        ;Int 13h busy ?
    jne     EXIT_28          ;If yes, exit

    push    di                ;Save registers
    push    es

    ; Check critical error flag
    mov     es,InDosSeg
    mov     di,ErrFlagOfs     ;Offset of flag
    cmp     byte ptr es.[di],0
    pop     es                ;Restore registers
    pop     di
    jne     EXIT_28

    mov     HotKey,0          ;Clear HotKey flag
    call    pop_up            ;Run clock routine

EXIT_28:
    iret                    ;Return to MaDos
new28_hndlr  endp

```

Nótese que todas las ISRs llaman a la ISR original como primer paso, es decir, realizan el encadenamiento de rutinas de interrupción.

Las ISRs para las INT 10h y 13h no realizan ningún proceso, simplemente encienden una bandera antes de encadenar con la ISR original para así poder detectar, dentro de las otras rutinas, que el BIOS está accediendo los periféricos asociados con estas interrupciones.

Las ISRs para las INT 08h y 28h son muy parecidas: primero revisan si la secuencia de teclas de activación ha sido presionada, de ser así, revisan que el BIOS no esté accediendo el video ni las unidades de disco (VideoFlag=0 y DiskFlag=0). Después verifican que DOS no se encuentre procesando algún error crítico (ErrFlagOfs=0). La ISR para la INT 08h además verifica que la InDos Flag también sea igual a cero. Si todos estos requisitos se cumplen llaman a la rutina principal, de lo contrario regresan el control inmediatamente.

La ISR para la INT 09h es la más interesante de todas. Esta rutina monitorea el teclado por medio de la interrupción 16h función 02h del BIOS. Si se detecta que las teclas de activación fueron presionadas la ISR verifica que el TSR no esté activo (RunningFlag=0), pues el activarlo una segunda vez causaría serios problemas de reentrada. Si no está activo, la ISR inicializa una bandera (HotKey=18) para que las ISRs 08h y 28h puedan activar la rutina principal. Esta bandera actúa como una cuenta regresiva que monitorea los intentos de activación.

Por ejemplo, si después de haber presionado las teclas de activación la ISR 08h encuentra que se está procesando un error crítico no activará a la rutina principal y regresará el control, pero antes de hacerlo decrementará la bandera (DECHKEY:). Como esta rutina es llamada 18.2 veces por segundo, la ISR 08h intentará activar el TSR durante un segundo aproximadamente. Si después de este lapso de tiempo (18 interrupciones) el error crítico siguiera sin resolverse, la bandera sería decrementada hasta el punto de apagarse y, por lo tanto, ya no se seguiría intentando la activación de la rutina principal.

Este proceso no se debe codificar en las dos ISRs (08h y 28h) pues de hacerlo, no se podría garantizar que el TSR intentará activarse durante el tiempo especificado por el valor inicial (18=un segundo, en el ejemplo).

Una interrupción que frecuentemente es reemplazada (no interceptada) por los TSRs es la INT 24h. El listado 4-16 muestra el código que implementa la ISR asociada con esta interrupción, el código se muestra en ensamblador y en pascal.

Listado 4-16. ISR para manejo de errores críticos.

```

TSRTIMPLT ASM -----[19.p.49]
...
;=====
; CRITICALERR receives control when an interrupt 24h is
; generated
;=====
criticalerr proc far
    assume cs:Cseg, ds:nothing, es:nothing
    xor     al,al                ,Default to ignore
    cmp     cs:Dos_version,30Ah  ,See if before 3.1
    jl      CRITICAL1
    add     al,3
CRITICAL1:
    iret
criticalerr endp
.

JTSRTP.PAS -----[13.p.169]
.
procedure newcrit (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
                  ES,BP,word),
    interrupt;
{ Set AX = 0 to tell DOS to ignore critical errors }
begin
    AX := 0,
end, { newcrit }
.

```

En ambas implementaciones, simplemente se regresa un valor de 0 ó 3 en AL, según la versión de DOS bajo la que se esté corriendo, para que DOS ignore el error. Si esta rutina es llamada es porque de hecho algún error crítico fue generado y, por lo tanto, es responsabilidad del TSR el resolverlo, es decir, no basta con hacer que DOS lo ignore sino que es necesario que el mismo TSR contenga código para corregir cada situación de error que él mismo pueda provocar.

- Rutina principal.

Cuando la rutina principal es ejecutada, lo primero que se debe hacer es salvar todos los parámetros configurables que vayan a ser alterados. Otra vez, este código puede ser tan simple o tan complejo como el TSR lo requiera. En este capítulo se mostrará el código de dos TSRs convencionales y se deja para el próximo capítulo el estudio del salvado y restauración de los valores asociados con las memorias extendida, expandida y los task switchers.

Listado 4-17. Inicio de la rutina principal.

```

;=====
; Main routine
;=====
scrblanker proc near
    mov     BlankerOn,1      ;Set TSR active flag
;-----
; Switch to internal stack
;-----
    cli                     ;Disable interrupts
    mov     OldSP,sp        ;Save DOS SP register
    mov     OldSS,ss        ;Save DOS SS register
    push    cs              ;Switch to internal stack
    pop     ss
    mov     sp,offset StkTop
    sti                     ;Enable interrupts
;-----
; Save system state
;-----
; Save all registers
    push    ax
    push    bx
    push    cx
    push    dx
    push    si
    push    di
    push    ds
    push    es
    push    bp

; Check screen mode
    mov     ah,0Ph          ;Get display mode
    int     10h             ;Returns BH=video page
    cmp     al,3
    jbe     SAVE_CURSOR
    cmp     al,7
    je      SAVE_CURSOR
    jmp     EXIT_SCRBLANKER ;Exit if not in text mode

SAVE_CURSOR.
; Save cursor state
    mov     ah,03h          ;Get cursor state
    int     10h
    mov     CursorPos,dx     ;Save position
    mov     CursorSz,cx     ;Save size

```

Este código corresponde al programa BLANKER.ASM, pero es bastante representativo y se puede decir que es el mínimo de código que se debe encontrar al inicio de la rutina principal de un TSR interactivo.

En primer lugar se enciende la bandera que indica que el TSR ha sido activado (BlankerOn=1) para evitar múltiples activaciones. Inmediatamente después se habilita el stack interno. Nótese que es indispensable que este código vaya entre instrucciones 'cli' y 'sti' para evitar interrupciones durante el cambio de stack. Por último se salva el valor de todos los registros del procesador.

A partir de aquí el código ya es específico para BLANKER.ASM. Se verifica que el video de la computadora esté en modo texto de máximo de 80x25. Si esto se cumple se salva la posición y el tamaño del cursor. Después de esto (no listado) se salva el contenido de toda la pantalla.

Listado 4-18. Inicio de la rutina principal (pascal). [13:p.169]

```

...
procedure beginint,
{ Swap interrupted program's stack info and }
{ TSR stack info }
inline ( $FF/$06/StackSW/      { inc StackSW }
          $75/$10/              { jne +16 }
          $8C/$16/CurStackSS/   { mov CurStackSS,SS }
          $89/$26/CurStackSP/   { mov CurStackSP,SP }
          $8E/$16/TsrStackSS/   { mov SS,TsrStackSS }
          $8B/$26/TsrStackSP),  { mov SP,TsrStackSP }

..

procedure newTimer (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
                   ES,BP,word),
  interrupt,
{ New timer interrupt routine }
{ If DOS is not busy and pop up is pending then pop up }
begin
  cli,
  beginint,
  sti,

  calloldint(_OldTimerVec), { Call old timer interrupt }
  if PopOk and (mem[DosSeg:InDosFlag] = 0) and
    (mem[DosSeg CritErrFlag] = 0) then
  begin
    Port[$20] := $20,
    PopOk := FALSE,
    popup;
  end,

  cli,
  endint,
  sti,
end; { newtimer }

..

procedure beforepop,
var
  I . byte;
  Reg . registers,
begin
  Running := TRUE,

{ Get the DTA address of the interrupted program }
  Reg ah := $2F,
  msdos(Reg),
  CurDTASeg := Reg.es,
  CurDTAOfs := Reg.bx,

{ Set the DTA address of the TSR }
  Reg ah := $1A,
  Reg ds := TsrDTASeg,
  Reg dx := TsrDTAOfs,

```

```

      mados(Reg),
{ Get ^Break setting }
  Reg.ax := $3300;
  mados(Reg);
  CurBreak := Reg.dl,
{ Disable ^Break checking }
  Reg.ax := $3301,
  Reg.dl := 0,
  mados(Reg),
{ Get current critical error interrupt address }
  getintvec(CRITINT, @OldCritVec);
{ Install the TSR's critical error interrupt }
  setintvec(CRITINT, @newcrit);
{ Make the TSR's PSP current }
  CurPSP := PSPArray[1],
  for I := 1 to PSPCounter do
    memw[DosSeg PSPArray[I]] := TsrPSP;
end; { beforepop }

...

procedure popup,
{ PopUp interface routine }
begin
  beforepop;
  _PopRoutine,
  afterpop,
end; { popup }

```

Este código en esencia ejecuta las mismas acciones que el de ensamblador, pero debido a la naturaleza modular de pascal el código no se encuentra todo en una misma rutina sino que está dividido en diferentes subrutinas.

El cambio del stack del programa interrumpido al stack interno de la porción residente, no se realiza dentro de la rutina principal sino que se hace al entrar a cualquiera de ISRs de JTSRTP.PAS (en el listado 4-18 se muestra la ISR asociada al reloj del sistema). El cambio de stack se lleva a cabo utilizando un macro escrito en ensamblador en línea (beginint), el cual debe ejecutarse entre instrucciones 'cli' y 'sti'.

Puesto que el cambio de stack se realiza en las ISRs se debe verificar si la interrupción fue generada desde adentro o fuera del TSR. Si la bandera 'StackSW' tiene un valor de cero entonces la interrupción fue generada desde dentro y, por lo tanto, el stack actual corresponde al del TSR y no se debe volver a cambiar. Si la bandera tiene un valor menor que cero entonces si se debe cambiar al stack interno del TSR.

Después de esto la rutina principal (popup) ejecuta la subrutina 'beforepop', la cual es la que realmente se encarga de inicializar el ambiente para el TSR. Aquí el código es más completo que en el ejemplo anterior pues se actualizan la DTA y el PSP activos, lo cual permite que el TSR pueda acceder archivos de disco. También se deshabilitan las teclas Ctrl-Break para evitar que el programa sea abortado por el usuario, lo cual congelaría la computadora. Por último, se reemplaza la ISR para la interrupción 24h (ver: listado 4-16).

Terminando esta inicialización se llama a la rutina '_PopRoutine', la cual ejecuta el trabajo real del programa. '_PopRoutine' es en realidad sólo un apuntador a la rutina, la cual es ejecutada vía el apuntador y no directamente, por ejemplo, en SAVESCR.PAS '_PopRoutine' apunta a la rutina 'screensaver'.

Código de desactivación.

Cuando la rutina principal ya ha realizado el trabajo para el que fue diseñado el TSR, todos los parámetros configurables que hayan sido alterados deben ser restaurados antes de regresar el control al programa interrumpido. A continuación se muestra el código de desactivación para los dos ejemplos anteriores.

Listado 4-19 Código de desactivación.

```

;-----
; Restore system state
;-----
; Restore cursor state
mov     ah,01h           ; Restore cursor size
mov     cx,CursorSz
int     10h
mov     ah,02h           ; Restore cursor position
mov     dx,CursorPos
int     10h

EXIT_SCRBLANKER
; Restore all registers
pop     bp
pop     es
pop     ds
pop     di
pop     si
pop     dx
pop     cx
pop     bx
pop     ax

;-----
; Restore external stack
;-----
cli                     ; Disable interrupts
mov     ss,OldSS        ; Restore SS register
mov     sp,OldSP        ; Restore SP register
sti                     ; Enable interrupts

mov     BlankerOn,0     ; Reset TSR active flag
ret                               ; Return
scrblanker   endp

...
```

Es obvio que el código es exactamente el inverso de la rutina de activación. En el programa BLANKER.ASM, antes de ejecutar el código mostrado, ya se ha restaurado el contenido de la pantalla y se ha vaciado el buffer de teclado, pero como ese código es específico de la aplicación no se ha listado aquí.

Listado 4-20. Código de desactivación (pascal). [13:p.169]

```

..
procedure  endint,
{ Restore interrupted program's stack info }
  inline ( $PF/$OE/StackSW/      [ dec StackSW ]
           $7D/$08/              [ jge +8 ]
           $8E/$16/CurStackSS/   [ mov SS,CurStackSS ]
           $8B/$26/CurStackSP),  [ mov SP,CurStackSP ]
  ...

procedure  afterpop,
var
  I . byte;
  Reg . registers,
begin
  Running := FALSE,

  { Set DTA address back to the original }
  Reg.ah := $1A,
  Reg.ds := CurDTASeg,
  Reg.dx := CurDTAOfs,
  msdos(Reg),

  { Reset original ^Break setting }
  Reg.ax := $3301,
  Reg.dl := CurBreak,
  msdos(Reg),

  { Restore original critical error interrupt }
  setintvec(CRITINT, 8_OldCritVec),

  { Restore the interrupted PSP back }
  for I := 1 to PSPCounter do
    memw[DosSeg:PSPArray[I]] := CurPSP;
end; { afterpop }
..

```

En pascal, la subrutina 'afterpop' es llamada como parte de la rutina principal (ver: listado 4-18) y al igual que el ejemplo en ensamblador es el inverso del código de activación. Sin embargo, es importante notar que la restauración de otros estados (pantalla, teclado, memoria, etc.) son responsabilidad de la rutina '_PopRoutine' y deben ser realizados antes de regresar el control a la rutina 'popup' (ver: rutina 'screensaver' en SAVESCR.PAS, apéndice G).

Después de esto el control es regresado al código de la ISR que activó la rutina principal, la cual debe restaurar el stack del programa interrumpido. Esto se hace con el macro 'endint', también escrito en ensamblador en línea. Nótese que también en este código se verifica que la bandera 'StackSW' sea menor que cero antes de modificar el stack actual.

Código para comunicación.

Para terminar este capítulo se describirá el código que implementa una línea de comunicación entre el usuario y la porción residente del TSR. Como se recordará, existen básicamente dos formas de comunicarse con la porción residente: por medio de la interrupción Multiplex y por medio de las interrupciones reservadas para los programas

escritos por los usuarios (60h a 67h). Aquí sólo se listará el código para el segundo método pues el código para implementar el primero ya fue presentado en el capítulo anterior (ver: listado 3-2 y listado 3-3).

Listado 4-21. Comunicación con la porción residente. [13:p.169]

```

...
procedure   tsr_isr (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP:word),
    interrupt,
    { TSR communication interrupt    }
    { Uninstall the TSR if possible }
begin
    { Save TarSwitch before executing the interrupt }
    TarSwitch = AX,

    cli,
    beginint,
    sti;

    case TarSwitch of
        1: if uninstalltsr then
            writeln('UNINSTALLED')
        else
            writeln('UNABLE TO UNINSTALL');
        else
            writeln('ALREADY INSTALLED',NL,HotKeyMsg),
    end, { case }

    cli,
    endint,
    sti;
end, { tsr_isr }

```

El mecanismo de comunicación es implementado a través de alguna de las interrupciones 60h a 67h, a la cual se le asigna la ISR 'tsr_isr' mostrada en el listado 4-21. La ISR es llamada almacenando en AX el comando a ejecutar y generando la interrupción correspondiente, lo cual se lleva a cabo dentro de la rutina que previene múltiples instalaciones (ver: listado 4-8). Nótese que lo primero que se hace es salvar el comando contenido en AX. Esto se debe hacer porque el código en pascal (case), modificará dicho valor sin que se pueda hacer nada para evitarlo.

En el código presentado para este trabajo, la rutina 'tsr_isr' responde únicamente al comando de desinstalación (/u). Otros comandos pueden ser adicionados según la aplicación específica que se desee realizar. Por ejemplo, si se deseara desarrollar un editor de texto residente se podría aceptar la opción '/b' desde la línea de comandos la cual indicaría que el programa no deberá generar archivos .BAK. Este comando sería pasado a través de 'tsr_isr' para alterar la bandera interna que controla la creación de esos archivos.

(Nota: Para mayor información acerca del funcionamiento de las instrucciones en ensamblador y de las convenciones de llamado para las interrupciones utilizadas en este capítulo, consultar la Assembly Language Database. Ver: Referencias bibliográficas).

Hasta este momento se han estudiado las características y el funcionamiento de TSRs que pueden ser considerados estándar, pero que no reflejan completamente las necesidades de un programa residente actual, ya que conforme MS-DOS crece en complejidad el desarrollo de los TSR se vuelve más difícil.

La dificultad radica en el hecho de que los TSRs deben permanecer compatibles con las versiones anteriores del sistema operativo y al mismo tiempo debe explotar los nuevos recursos que proporciona la versión 5. Esto provoca un aumento en el número de requisitos que el TSR debe reunir, así como en el número de restricciones impuestas al programa.

Muchas de las tecnologías que se describirán en este capítulo, sino es que todas, ya existían antes de la liberación de MS-DOS 5 pero únicamente en forma aislada, ya que eran desarrolladas por diferentes casas de software y, por lo tanto, eran vendidas como paquetes independientes. Esto inevitablemente provoca que no todo el mundo tenga acceso a todas las nuevas tecnologías que se desarrollan.

Este panorama cambió con DOS 5, el cual reúne en un solo paquete gran parte de estas tecnologías. La principal consecuencia de esta 'unificación' es que las tecnologías que antes estaban disponibles sólo para un sector de la población computacional, ahora se encuentran al alcance prácticamente de todo el mundo, es decir, el uso de estas tecnologías se pueden considerar como estándar de ahora en adelante. Por lo tanto, los nuevos TSR deben estar diseñados para explotar estos recursos que ahora ya se encuentran a su disposición en forma regular.

Desafortunadamente, la adición al sistema operativo de estos servicios, trae como consecuencia que el desarrollo de los TSRs se vuelva aún más complejo de lo que ya era. De estas tecnologías, las que serán estudiadas a continuación tienen que ver principalmente con el uso de memoria y con el control de los recursos del sistema.

(Nota: Para ejemplificar los conceptos presentados en este capítulo se utilizan porciones de código del programa de demostración TSRTMPLT.ASM [19:p.49]).

MEMORIA EXTENDIDA Y EXPANDIDA.

De las adiciones al sistema operativo que más impacto tienen sobre el desarrollo de TSRs, son los drivers HIMEM.SYS y EMM386.SYS.

HIMEM.SYS es un administrador de memoria extendida (XMM), el cual implementa la especificación XMS (Extended Memory Specification) desarrollada por Lotus, Intel y Microsoft para controlar el acceso a la Upper Memory Area (Area de Memoria Superior), a la High Memory Area (Area Memoria Alta) y en general a toda la memoria extendida.

Los usos que más frecuentemente se le asignan a HIMEM son: cargar parte del sistema operativo en memoria extendida (específicamente en la HMA), mover device drivers y TSRs hacia el área de memoria superior para ahorrar memoria principal (sólo con procesadores 386 o superiores) y controlar los accesos a memoria extendida por parte de diversos programas como SMARTDRV.SYS, RAMDRIVE.SYS, etc.

EMM386.SYS también es una administrador de memoria pero éste controla el acceso a la memoria expandida. "Este EMM hace uso de los sofisticados recursos como registros de página y modo virtual-86 que proporcionan los procesadores 386 y superiores de Intel." [19:p.41]

Estos recursos hacen posible que se utilicen partes de la UMA, para realizar el proceso de mapeo de/hacia memoria expandida. (Ver: 'La memoria de la computadora', capítulo 1).

El impacto que HIMEM.SYS y EMM386.SYS tiene sobre el código del TSR se puede decir que es mínimo. En esencia lo único que se debe hacer es salvar el estado de ambos drivers al momento de activar el TSR y restaurar dichos valores al desactivarlo. Obviamente, también se debe incluir el código necesario para reservar espacio en memoria expandida. A continuación se describen los requerimientos para interactuar tanto con HIMEM.SYS como con EMM386.SYS.

INTERACCION CON HIMEM.SYS

Si el TSR está corriendo en una máquina en donde se encuentre cargado el driver HIMEM.SYS, entonces el código del TSR debe realizar los siguientes dos procesos:

- 1) Salvar y restaurar el estado del 'A20 Handler'.
- 2) Salvar y restaurar el estado de los UMBs.

A20 Handler.

Un A20 handler (también conocido como Línea A20) es un componente de la computadora que permite acceder la memoria alta (HMA). "Existen 15 líneas A20 diferentes, (1-14,16) de las cuales HIMEM utiliza sólo una para lograr el acceso a la HMA. La línea utilizada depende de la computadora en donde se esté corriendo el sistema operativo." [12:p.611]

El TSR no se debe preocupar de estos detalles ya que el mismo HIMEM detecta que línea A20 debe utilizar. Por lo tanto, lo único que debe hacer el TSR es salvar su estado para prevenir cualquier problema cuando DOS esté corriendo desde HMA.

Para lograr esto, primero se debe localizar la dirección del XMM. Por medio de esta dirección se pueden acceder los servicios que se necesitan para salvar y restaurar el estado de la línea A20. El listado 5-1 muestra como implementar este código, el cual debe estar contenido en la porción transitoria junto con todo el código de inicialización estándar del TSR.

Listado 5-1 Localizar el XMM

```

;-----
; Determine if extended memory manager active
;-----
mov     ax,4300h           ;Look for HIMEM.SYS
int     2Fh
or      al,al
jne     NOXMS
mov     ax,4310h           ;Get entry point for XMM
int     2Fh
mov     word ptr [Xms_service],bx
mov     word ptr [Xms_service+2],es
push    cs
pop     es
inc     Xms_flag

```

NOXMS

Como se ve, la interrupción multiplex es utilizada para verificar que el driver ya esté cargado en memoria y para solicitar la dirección del XMM, la cual es almacenada en una variable interna (Xms_service). Esta dirección servirá para salvar y restaurar el estado de la línea A20.

Es muy importante notar que este código es válido únicamente para localizar la presencia de HIMEM, es decir, este código no sirve para localizar otros XMMs (como QRAM) Esto por la sencilla razón de que el ID que se utiliza en la interrupción multiplex (ID=43h) es exclusivo de HIMEM y además está dentro del rango de IDs reservados por Microsoft. (Ver: 'Funcionamiento de un TSR', capítulo 3).

A continuación se presenta el código para salvar y restaurar el estado de la línea A20. Estos procesos se realizan en una forma muy similar al llamado de una interrupción convencional y deben ser llevados a cabo durante los procesos de activación y desactivación respectivamente.

Listado 5-2. Salvar y restaurar la línea A20

```

; Save state of A20 line
;-----
    cmp     Xma_flag,0
    je      SKIP_XMS_SAVE
    mov     ah,7                ,Get state of A20 line
    call    [Xms_service]
    mov     A20_state,ax

SKIP_XMS_SAVE

;-----
; Restore A20 line
;-----
    cmp     Xma_flag,0
    je      SKIP_XMS_RESTORE
    mov     ah,5                ,Assume local disable A20
    cmp     A20_state,0
    je      XMS_RESTORE_1
    inc     ah

XMS_RESTORE_1
    call    [Xms_service]      ,Restore A20 line state
    mov     A20_state,ax

SKIP_XMS_RESTORE

```

El código para estos procesos es bastante sencillo. Únicamente se obtiene el estado de la línea por medio de la función 7 del XMM y se salva en una variable interna. Al desactivarse el TSR se asume que la porción residente modificó dicha línea y simplemente se utiliza la función 5 ó 6 del XMM para restaurar el estado anterior.

Upper Memory Blocks (UMBs).

Como ya se mencionó, HIMEM puede utilizar la UMA para almacenar drivers y/o TSRs. Lo que realmente se utiliza de esta área de memoria son los Upper Memory Blocks o UMBs (Bloques de Memoria Superior). Los UMBs son los espacios de memoria de la UMA que no están siendo utilizados y que generalmente se encuentran por debajo o entre la memoria ocupada por el BIOS.

Para poder hacer uso de estos nuevos recursos MS-DOS 5 tiene integradas las funciones que permiten manipular los UMBs. Estas funciones se utilizan por medio la interrupción estándar para obtener servicios del sistema operativo, la INT 21h. De estas funciones, las que el programa de demostración TSRTMPLT.ASM utiliza son las siguientes:

- 5800h (Get Memory Allocation Strategy).
- 5801h (Set Memory Allocation Strategy).
- 5802h (Get Upper-Memory Link).
- 5803h (Set Upper-Memory Link).

"Cuando un programa es ejecutado los UMBs no están ligados a la cadena de memoria reservable" [19:p.42] ,es decir, un programa no puede utilizar las funciones estándar del sistema operativo para reservar o liberar dinámicamente la memoria localizada en el área superior.

Por lo tanto, "para poder utilizar los UMBs, el TSR debe ligarlos a la cadena de memoria reservable. Además, es necesario instruir a DOS para que busque dentro de esta memoria, un bloque libre adecuado a las necesidades que el TSR pudiera tener en un momento dado." [19:p.42]

Antes de poder realizar el proceso de ligado es necesario, como siempre, salvar el estado actual. En primer lugar se debe salvar la dirección actual de la rutina de manejo de memoria, después se podrá salvar el estado de ligado de los UMBs y modificar dicho estado si, y sólo si, se está corriendo bajo DOS 5 o alguna versión posterior. Para esto se utilizan las funciones 5800h, 5802h y 5803h respectivamente. Por último, antes de que el TSR se desactive debe restaurar los valores anteriores a la activación, como lo muestra el listado 5-3. Otra vez, los procesos de salvado y restauración se realizan durante la activación y desactivación del TSR.

Listado 5-3 Salvar y restaurar el estado de los UMBs.

```

...
;-----
; Save UMB link and mem allocation state
; If DOS 5, link UMBs
;-----
mov     ax,5800h           ;Get DOS mem alloc strategy
int     21h
mov     Mem_alloc,ax
cmp     Dos_version,500h   ;If DOS 5, save UMB link
jb      SKIP_UMBLINK      ; state. Then link UMBs
mov     ax,5802h
int     21h
mov     LinkFlag,al        ;Save UMB link state
mov     ax,5803h
mov     bx,1               ;Link UMBs
int     21h

SKIP_UMBLINK

...
;-----
; Restore UMB link state, link and allocation start
;-----
mov     ax,5801h           ;Set DOS mem alloc strategy
mov     bx,Mem_alloc
int     21h

cmp     Dos_version,500h   ;If DOS 5, restore UMB
jb      SKIP_UMBLINK_RESTORE ; link state
mov     ax,5803h           ;Set UMB link state
xor     bh,bh
mov     bl,LinkFlag
int     21h

SKIP_UMBLINK_RESTORE:

```

INTERACCION CON EMM386.SYS.

Para poder utilizar memoria expandida, el programa primero debe reservarla. Es posible para cualquier programa reservar desde una sola página hasta el total de páginas disponibles. A cada programa que reserva memoria expandida le es asignado un 'Expanded Memory Handle'. No existe una traducción literal satisfactoria para este concepto, pero quiere decir que el EMM identifica a cada programa que le solicita memoria expandida mediante su Handle, es decir, el handle es simplemente un identificador.

Más específicamente, "el handle es un número que el EMM asocia con las páginas de memoria expandida que reservó. Así, cada handle puede tener su propio conjunto de páginas." [13:p.113] Por ejemplo, el handle 1 puede tener asignadas tres páginas las cuales estarán numeradas de 0 a 2. Al mismo tiempo el handle 2 puede tener asignadas cuatro páginas numeradas de 0 a 3. Como se ve ambos handles tienen páginas 0,1,2. Como resultado, las páginas de memoria expandida no se pueden referenciar únicamente por su número sino que es necesario especificar también su handle.

Para reservar memoria, obtener un handle, mapear la memoria hacia dentro y fuera del page frame y en una palabra para manipular memoria expandida, se utilizan las funciones que proporciona el EMM. Estas funciones pueden ser accesadas a través de la interrupción 67h, la cual es capturada por el EMM durante su instalación en memoria

(Nota: No es recomendable que ningún programa modifique el vector de la interrupción 67h ya que esto podría interferir con el correcto funcionamiento del EMM).

Aunque existen varios tipos de administradores de memoria expandida y cada tipo ha pasado por diferentes revisiones, el estándar actual para EMMs está dado por la especificación LIM EMS v4.0 (Lotus-Intel-Microsoft Expanded Memory Specification). Los EMM que son compatibles con esta versión proporcionan 30 funciones para la manipulación de memoria expandida, de las cuales la tabla 5-1 describe las más frecuentemente utilizadas.

Tabla 5-1. Funciones más utilizadas del LIM EMM v3.2. [13 p.115]

AH	Función
40h	Get Status Determina si el EMM está cargado en memoria y si el hardware está funcionando adecuadamente. Además de otros valores específicos para cada función, el EMM siempre regresa en el registro AH el estatus de la función que acaba de realizar. AH=0, es la indicación de que la función fue procesada exitosamente. De lo contrario se concluye que hubo algún error durante el proceso y, por lo tanto, la información que se obtenga de dicha llamada se debe considerar como inválida.
41h	Get Page Frame Address. Localiza la dirección del page frame. Regresa. BX=segmento del page frame.

42h	Get Unallocated Page Count. Determina la cantidad de memoria disponible. Regresa: DX=número total de páginas instaladas; BX=número de páginas disponibles.
43h	Allocate Pages. Reserva páginas de memoria. Enviar: BX=páginas deseadas. Regresa DX=Handle.
44h	Map Handle Page Mapea una página de memoria expandida hacia el page frame. Enviar. AL=página destino del page frame; BX=página de memoria expandida deseada; DX=Handle correspondiente.
45h	Deallocate Pages Libera todas las páginas de memoria asociadas a un handle Enviar. DX=Handle.
46h	Get EMM Versión Regresa la versión del EMM. Los cuatro bits altos de AL contienen la versión y los bits bajos los decimales de la revisión.
47h	Save Mapping Context Salva el estado del EMM. Enviar. DX=Handle
48h	Restore Mapping Context. Restaura el Estado del EMM. Enviar: DX=Handle

El uso de memoria expandida por parte del TSR, y en realidad de cualquier programa, se puede dividir en tres fases fundamentales:

- 1) Inicializar memoria expandida.
- 2) Utilizar la memoria ya reservada.
- 3) Liberar la memoria al final del proceso.

Inicializar memoria expandida.

El código para implementar esta fase del uso de memoria expandida se debe poner en la porción transitoria, junto con el código de inicialización estándar del TSR. El listado 5-4 muestra la porción de código que controla la fase de inicialización de la memoria expandida. Se puede decir que este código es de nivel medio pues únicamente llama a otras subrutinas para que sean ellas las que realicen el trabajo real.

Los procesos que esta rutina realiza son: verificar que el usuario haya solicitado acceso a memoria expandida, localizar el EMM, salvar la dirección del page frame y la versión del EMM, reservar una página de memoria y por último mapear esa página de memoria hacia el page frame.

Listado 5-4. Inicializar memoria expandida.

```

, Set up EMS memory if necessary
-----
cmp     Ems_flag,0           ,See if expanded memory
je      NO_EMSMEM           , requested

call    check4ems           ,See if EMS driver installed
jnc     INSTALL_1
mov     dx,offset ErrMsg2    ,EMS driver error
jmp     ERROR_EXIT

INSTALL_1.
mov     Ems_segment,bx      ,Save seg of EMS page frame
mov     Ems_version,dx      ,Save EMS version

mov     bx,1                ,Allocate 1 page
call    get_emamem
mov     Ems_handle,dx       ,Save EMS handle

xor     al,al                ,Map to EMS page 0
xor     bx,bx
call    map_emamem

NO_EMSMEM:

```

Lo primero que se debe hacer antes de utilizar cualquiera de las funciones del EMM es determinar si el driver de memoria expandida está instalado. Para este propósito no se debe utilizar la función 40h (Get Status), ya que si el EMM no está instalado lo más probable es que la INT 67h no esté implementada, es decir, no habrá ninguna ISR que responda a los llamados de dicha interrupción.

Por lo tanto, el llamar a la interrupción 67h sin un EMM instalado provocará en el mejor de los casos que el TSR obtenga información inválida, pero muy probablemente congelará la computadora.

"La forma correcta de localizar un EMM en memoria es por medio de su firma. Cuando el EMM está cargado en memoria deja la firma 'EMMXXXX0' en el offset 0Ah del segmento en donde se encuentra instalado." [13:p.114] Esta firma corresponde al campo Device Name del driver de memoria expandida, (ver 'Device Drivers', capítulo 2).

La firma, junto con el hecho de que el EMM captura la INT 67h, nos permiten determinar si el driver está instalado en memoria. Para ello necesitamos conocer el vector asociado a la INT 67h. Como ya se sabe la función 35h de DOS regresa en ES el segmento y en BX el desplazamiento de la ISR asociada con cualquier interrupción. Si el EMM está instalado, entonces al pedir el vector de la INT 67h, ES contendrá el segmento donde se encuentra instalado y, por lo tanto, la dirección ES:00A0h contendrá la firma del driver.

Si el driver se encuentra en memoria, se deberá verificar que tanto el driver como la memoria estén funcionando normalmente. Otro requisito indispensable para poder mapear memoria expandida, es conocer la dirección del page frame, ya que sin esta dirección no sabríamos en donde se deben almacenar los datos que han de ser mapeados a memoria expandida.

El código del listado 5-5 realiza estos tres procesos (búsqueda del EMM, verificación de su estado y localización del page frame) y además verifica la versión del EMM. Esta última acción únicamente será necesaria si el TSR ha de utilizar funciones que dependan de alguna versión específica de memoria expandida. Todos estos procesos se realizan a través de llamadas a las funciones correspondientes del EMM.

Listado 5-5. Localizar el EMM.

```

;-----
; CHECK4EMS determines if an expanded memory driver is
; loaded.
; Exit  BX - segment of EMS page frame if CF clear
;       DL - EMS driver version number
;       CF - clear if EMS driver found
;-----
check4ems    proc near
    push     es
    push     di
    mov     ax,3567h           ,Get EMS vector
    int     21h
    mov     di,0Ah
    mov     si,offset Ems_header ,Using the segment from
                                , the 67h, look at offset
    mov     cx,8               ; 0Ah Compare the next 8
                                , bytes with the expected
    cld
    repe    cmpsb              ,EMS header If they are
    pop     di                 , the same, EMS driver
    pop     es                  , found
    je      CHECK4EMS_FOUND

CHECK4EMS_ERROR:
    stc
    jmp     short CHECK4EMS_EXIT

CHECK4EMS_FOUND:
    mov     ah,40h             ,Check status
    int     67h
    or      ah,ah
    jne     CHECK4EMS_ERROR

    mov     ah,41h             ,Get page frame segment
    int     67h
    or      ah,ah
    jne     CHECK4EMS_ERROR

    mov     ah,46h             ,Get EMM version number
    int     67h
    or      ah,ah
    jne     CHECK4EMS_ERROR
    mov     dl,al               ,Copy version number

CHECK4EMS_EXIT
    ret
check4ems    endp
...

```

La variable Ems_header contiene la firma del EMM la cual será comparada con la firma que se encuentre en memoria. La dirección del page frame y la versión del EMM se dejan en los registros BX y DL respectivamente para que la rutina que llamó a CHECK4EMS los guarde en las variables internas correspondientes. Si esta rutina encuentra algún error encenderá la bandera de carry (stc) para que se aborte el proceso de inicialización de memoria expandida.

(Nota: Existe otro método para detectar la presencia del EMM en memoria. En pocas palabras, este método trata de abrir (en modo sólo-lectura) el archivo 'EMMXXXX0' por medio de la función 3Dh de DOS. Si la función logra abrir el archivo exitosamente, entonces se debe verificar que en realidad es el driver y no un archivo de disco. Esto se logra mediante la función 4400h de DOS la cual determina si el file handle pertenece a un archivo o a un device driver. Si al regresar de esa función el bit 7 del registro DX tiene un valor de 1, el handle la pertenece a un driver y, por lo tanto, se concluye que el EMM si está instalado en memoria).

Una vez que ya se ha localizado el driver de memoria expandida ya es posible reservar memoria para uso del TSR (ver: listado 5-4). Esto se puede hacer desde la porción transitoria, desde la porción residente o desde ambas. Como regla general, si el TSR reserva memoria desde la porción residente, éste deberá liberar dicha memoria como parte del proceso de desactivación, es decir, la memoria será utilizada únicamente durante el período de actividad del TSR. Esto con el fin de no monopolizar la memoria expandida y permitir que otros programas la utilicen durante los períodos de inactividad del TSR.

El programa de demostración TSRTMPLT.ASM reserva memoria desde la porción transitoria utilizando el código que se muestra en el siguiente listado.

Listado 5-6 Reservar memoria expandida

```

;-----
; GET_EMSMEM allocates pages of memory from the Expanded
; memory manager
; Entry BX - number of 16K pages to request
; Exit DX - handle to pages
; ZF - Set if call successful
;-----
get_emsmem    proc near
    mov     ah,43h                ,Allocate EMS pages
    int     67h
    or      ah,ah
    ret
get_emsmem    endp

```

Esta subrutina recibe en BX el número de páginas de memoria a reservar. Por lo demás la rutina es simplemente una llamada a la función Allocate Pages del EMM. La subrutina regresa en DX el EMM Handle asociado con la memoria que se acaba de reservar.

Como última acción de la fase de inicialización de memoria expandida, el programa puede mapear la memoria reservada o parte de ella hacia el page frame. Este proceso es tan sencillo como el de reservar memoria. La subrutina que se muestra a continuación, recibe en DX el handle, en BX la página fuente y en AL la página destino.

Listado 5-7. Mapear memoria hacia el page frame.

```

;-----
; MAP EMSMEM maps a page of expanded memory to the EMS
; page frame.
; Entry   AL - Physical page in page frame
;         BX - logical page to map
;         DX - handle to memory page
; Exit    ZF - set if call successful
;-----
map_emsmem  proc near
    mov     ah,44h                ,Map EMS page
    int     67h
    or      ah,ah
    ret
map_emsmem  endp

```

Utilizar la memoria ya reservada.

Para que la porción residente del TSR pueda utilizar su memoria expandida, es necesario que durante el proceso de activación se mapee nuevamente dicha memoria hacia el page frame. Este proceso se debe repetir, aunque durante la inicialización en la porción transitoria ya se haya realizado, ya que cualquier otro programa pudo haber hecho lo mismo con sus propias páginas, es decir, al activarse el TSR no tiene la seguridad de que su memoria siga en el page frame

Al hacer esto, se modifica el estado del EMM y, por lo tanto, el TSR debe salvarlo y restaurarlo antes y después de utilizar su memoria expandida. Las funciones del EMM para salvar y restaurar su estado funcionan en forma análoga a las instrucciones push y pop de ensamblador como se muestra a continuación.

Listado 5-8. Salvar y restaurar el estado del EMM

```

;-----
; If using EMS memory, save EMS mapping context and
; map our page
;-----
    cmp     Ems_flag,0
    je      SKIP_EMS_SAVE
    mov     ah,47h                ,Save mapping context
    mov     dx,Ems_handle
    int     67h
    or      ah,ah
    jne     CLEAN_UP
    xor     ax,ax
    mov     bx,ax
    call    map_emsmem            ,Map our page
    jne     CLEAN_UP

SKIP_EMS_SAVE:

```

```

;-----
; If using EMS memory, restore EMS mapping context
;-----
    cmp     Ems_flag,0
    je      SKIP_EMS_RESTORE
    mov     ah,48h          ;Restore mapping context
    mov     dx,Ems_handle
    int     67h

```

SKIP_EMS_RESTORE:

..

Es muy importante que el proceso de activación del TSR verifique que la función de salvado (47h) fue exitosa (AH=0) pues es posible que el EMM sea incapaz de salvar su estado bajo ciertas circunstancias. Si el proceso de salvado no es exitoso, se debe detener la activación del TSR (en el código se transfiere el control a CLEAN_UP).

Liberar la memoria al final del proceso.

El código para este proceso puede encontrarse tanto en la porción transitoria como en la residente dependiendo de las necesidades del TSR. Como ya se había mencionado, es aconsejable que si la porción residente reserva memoria durante el periodo de actividad del TSR, esta misma memoria sea liberada como parte del proceso de desactivación. Por lo tanto, este código generalmente se pone en la porción residente.

En el caso de TSRTMPLT.ASM este código es residente (igual que el de las subrutinas 'get_emsmem' y 'map_emsmem') aunque el programa en sí no reserva ni libera memoria estando residente.

Listado 5-9. Liberar memoria expandida.

```

;-----
; FREE_EMSMEM frees pages of memory from expanded memory
; manager.
; Entry:  DX - handle to memory pages to free
; Exit:   ZF - Set if call successful
;-----
free_emsmem  proc near
    mov     ah,45h          ;Free EMS
    int     67h
    or      ah,ah
    ret
free_emsmem  endp

```

..

(Nota: Los conceptos acerca de la memoria expandida presentados hasta aquí, son validos para cualquier EMM compatible con la especificación LIM EMS v3.2 o superior y no únicamente para EMM386.SYS).

TASK SWITCHERS.

Los 'Task Switchers' (Conmutadores de Tareas) son otra tecnología que tiene gran impacto en el desarrollo de TSRs, y siendo que estos programas son tan populares hoy en día conviene estudiar la forma en que un TSR debe interactuar con ellos. Esta sección específicamente trata de dos de los task switchers más populares en el mercado: el programa DOSSHELL, incluido a partir de la versión 4 del sistema operativo MS-DOS y el ambiente gráfico Windows.

Un task switcher es un programa que permite cargar en memoria varios programas para después activarlos y desactivarlos, uno a la vez, en cualquier orden según los comandos del usuario. Esta forma de operación debe parecer muy familiar ya que es la misma forma de operar que tienen los TSRs. La diferencia radica en que el TSR se controla a sí mismo y el task switcher controla a otros programas.

Aunque es innegable la utilidad de estos programas, el hecho de que puedan mantener varias copias de MS-DOS residentes en memoria simultáneamente (una para cada tarea) complica la programación de los TSRs. Cada una de las copias de MS-DOS que se encuentran residentes se conocen con el nombre de 'Sesión de DOS', así, dentro de cada sesión estará corriendo un proceso (tarea) independiente.

"Si el TSR es cargado después del task switcher, este último crea una nueva sesión de DOS y carga el TSR dentro de ella. Bajo estas condiciones no existe ningún problema porque el task switcher está controlando todos los llamados al sistema operativo y para el TSR todo el proceso de cargado fue transparente, es decir, el TSR no sabe que está instalado dentro de una sesión de DOS.

Sin embargo, si el TSR es cargado antes de que el task switcher esté activo, el programa residente será instalado en memoria global (como normalmente se hace). El problema con este orden de instalación es que el TSR podrá ser activado desde varias sesiones de DOS. Si el TSR es invocado desde una sesión mientras continúa activo en otra, seguramente se generarán problemas de reentrada.

Afortunadamente, tanto el task switcher de MS-DOS 5 como Windows tiene integrada una API que permite controlar estos problemas, es decir, poseen un conjunto de funciones que permiten a los programas de aplicación, no sólo a TSRs, controlar los diversos aspectos relacionados con la conmutación de tareas en el sistema." [19:p.43]

(Nota: API significa 'Application Program Interface' o Interface para Programas de Aplicación, es decir, una API especifica la forma en que deben interactuar dos programas. En esencia, la API de un programa define estructuras de datos y funciones que pueden ser utilizadas por otros programas para comunicarse con él. Por ejemplo, el sistema operativo implementa su API a través de la interrupción 21h, así cualquier programa puede interactuar (pedir servicios) con DOS por medio de dicha interrupción).

INSTANCE MEMORY.

"Una forma de controlar el problema de las múltiples sesiones simultaneas de DOS, consiste simplemente no permitir que dos sesiones puedan llamar a un mismo TSR al mismo tiempo. Esta solución es por demás ineficiente ya que forza al usuario a salir de

la sesión actual, regresar a la sesión donde se encuentra activo el TSR, desactivar el TSR, volver a la sesión original y activar nuevamente el TSR desde esa sesión." [19:p.43] En algunos casos, el no permitir llamados desde varias sesiones no es una solución. Por ejemplo, un editor de línea de comandos como DOSKEY debe poder ser accesado desde cualquier sesión de DOS en cualquier momento, pues de lo contrario el programa pierde toda su utilidad.

"Una mejor solución, aunque bastante más complicada, se basa en el uso de 'Instance Memory' (Memoria por Instancia o Memoria por Sesión). Como su nombre lo indica, la memoria por instancia es independiente para cada sesión, es decir, el task switcher se encarga de no mezclar los datos de una sesión con los de otra. Esto le permite al TSR mantener una réplica de sus datos (stack interno, banderas, etc.) en cada sesión en donde se encuentre activo. Por lo tanto, el TSR adquiere la capacidad de ser reentrado ya que utiliza un segmento de datos diferente para cada sesión." [19:p.43]

Como ya se dijo, tanto el task switcher utilizado por DOSSHELL como Windows proporcionan el soporte necesario para controlar la memoria por sesiones. Desafortunadamente, los mecanismos para crear y mantener los datos por sesión que utilizan cada uno de ellos, no son 100% compatibles entre sí. Esto debido a que la API del task switcher de MS-DOS 5 es 'completa' pero la de Windows 3.0 o anteriores no lo es.

Como ya se sabe, Windows 3.0 puede correr en tres modos de operación: Real, Estándar, y 386-Enhanced (Mejorado). Las funciones de Windows únicamente pueden mantener memoria por sesión cuando están corriendo en modo mejorado. Además, bajo los modos real y estándar, los programas diseñados para DOS no son notificados de que va haber o hubo algún cambio de sesión. La única notificación que reciben estos programas es que Windows va a iniciar o terminar su ejecución.

Bajo estas condiciones, los TSRs que tengan que correr bajo Windows 3.0 en modo real o estándar tienen tres opciones: (1) El TSR sólo debe permitir ser activado por una sesión a la vez. (2) El código del TSR se puede hacer reentrable, eliminando los stacks internos. (3) El TSR puede interceptar el mensaje que Windows envía a todo el sistema cuando va a ser ejecutado, y así impedir su ejecución.

Ahora que Windows 3.1 ya fue liberado, estos problemas ya no existen. En primer lugar Windows 3.1 únicamente posee dos modos de operación: Estándar y Enhanced. El modo estándar utiliza exactamente las mismas funciones que el task switcher de MS-DOS 5. El modo mejorado sigue utilizando la funciones de la versión 3.0 para mantenerse compatible.

En resumen, el TSR que necesite coexistir con DOSSHELL y con Windows debe tener el código para implementar memoria por sesiones. Para lograr esto, debe ser capaz de interactuar con ambos programas, lo que significa que debe poder controlar dos diferentes APIs. La API del task switcher de MS-DOS 5, la cual sirve para DOSSHELL y para Windows 3.1 en modo estándar y la API de Windows 3.1 en modo mejorado.

IMPLEMENTACION DE INSTANCE MEMORY.

Para implementar la memoria por sesiones es necesario que el TSR contenga el código para realizar los siguientes procesos:

- 1) Se requiere de cuatro estructuras de datos.
- 2) Inicializar las estructuras de datos.
- 3) La manipulación propiamente dicha de la memoria por sesiones.

Estructuras de datos.

En primer lugar se describirán los campos de cada una de estas estructuras (ver: tabla 5-2) y después se presentará el código para implementarlas.

Las estructuras SWSTARTUPINFO y SWINSTANCEINFO son idénticas tanto para el task switcher de MS-DOS como para Windows en modo mejorado. Estas dos primeras estructuras sirven para crear las réplicas del bloque de memoria que se necesita mantener en cada una de las sesiones en las que el TSR se encuentra activo.

Las otras dos estructuras, SWCALLBACKINFO y SWSWAPINFO son exclusivas de la API de DOS y de Windows en modo estándar y sirven como medio de comunicación entre el TSR y el task switcher. (Por esta razón se dice que la API de Windows en modo mejorado no es completa).

Tabla 5-2 Estructuras de datos para manejo de Instance Memory [19.p 44]

Estructura	Campo	Tipo y Descripción
SWSTARTUPINFO -----		
	sisVersion	dw 3 Este campo le sirve únicamente a Windows y es utilizado para identificar la estructura de datos
	sisNextDev	dd ? Apuntador a la estructura SWSTARTUPINFO anterior. Un programa obtiene este apuntador por medio de la interrupción multiplex.
	sisVirtDevFile	dd 0 Al igual que el primer campo, éste es ignorado por DOS. Únicamente le sirve a Windows para saber que device driver virtual ha de ser cargado a memoria cuando Windows inicie su ejecución.
	sisReferenceData	dd ? Este valor será pasado al device driver virtual que Windows cargará y puede contener cualquier valor.
	sisInstanceData	dd ? Este es un apuntador a una lista de estructuras SWINSTANCEITEM. Para que sea posible identificar el final de la lista, se debe colocar una estructura SWINSTANCEITEM nula (vacía) al final de la lista.

SWINSTANCEITEM -----	
iisPtr	dd ? Apuntador al bloque de memoria que ha de ser utilizado para crear la memoria por sesión.
iisSize	dw ? Tamaño del bloque de memoria.
SWCALLBACKINFO -----	
scbiNext	dd ? Apuntador a la estructura SWCALLBACKINFO anterior, pasada al programa por la interrupción multiplex.
scbiEntryPoint	dd ? Apuntador al inicio de la rutina de notificación del programa.
scbiReserved	dd ? Valor reservado para usos posteriores.
scbiAPI	dd ? Apuntador a una lista de estructuras SWSWAPINFO. La lista es terminada con una estructura SWSWAPINFO nula
SWSWAPINFO -----	
aisLength	dw 10 Longitud de la estructura.
aisAPI	dw ? Identifica la API asíncrona que el programa reconoce, puede ser: 0001 - NetBIOS 0002 - 802.2 0003 - TCP/IP 0004 - LAN Manager pipes 0005 - NetWare IPX
aisMajor	dw ? Número de la versión reconocida.
aisMinor	dw ? Número decimal de la versión reconocida
aisSupport	dw ? Nivel de soporte proporcionado: 0001 - Soporte mínimo. (No conmutación) 0002 - Soporte a nivel de API. 0003 - Compatibilidad con el conmutador de tareas. 0004 - Total compatibilidad.

El listado 5-10 muestra las declaraciones en ensamblador de las estructuras utilizadas en TSRTMPLT.ASM para implementar memoria por sesiones. Las estructuras están declaradas como sigue:

SWSTARTUPINFO	-->	StartupInfo
SWINSTANCEITEM	-->	InstItem1
SWCALLBACKINFO	-->	CallBackInfo
SWSWAPINFO	-->	SwapInfo

SwapInfo está declarada únicamente porque el último campo de la estructura SWCALLBACKINFO necesita un apuntador a este tipo de estructura, pero el programa de demostración TSRTMPLT.ASM no tiene implementado soporte para ninguna API asíncrona.

Este listado también muestra la tabla SWNotifyJT que contiene las direcciones de las diferentes subrutinas de notificación implementadas en el TSR, las cuales serán descritas más adelante.

Listado 5-10 Estructuras de datos para manejo de Instance Memory.

```

, Jump table used by switcher notification routine
SWNotifyJT    dw offset SWNInit
               dw offset SWNQSuspend
               dw offset SWNSuspend
               dw offset SWNAActivate
               dw offset SWNActive
               dw offset SWNCSession
               dw offset SWNDSession
               dw offset SWNExit
;-----
, Switcher global data structures
;-----
StartupInfo = $
  sisVersion    dw 3      , Switcher structure ID
  sisNextDev     dd 0      , Ptr to prev startup struct
  sisVirtDevFile dd 0      , Ptr to name of opt dev driver
  sisReferenceData dd 0    , Data for Win device driver
  sisInstData    dd 0      , Ptr to instance mem list

InstItem1 dd 0      , Ptr to instance data
InstSize1 dw 0      , Size of instance data
InstItem2 dd 0      , Ptr to instance stack
InstSize2 dw 0      , Size of instance stack
           dd 0      , Ptr to next block = 0 to
           dw 0      , terminate list

;=====
, Instance data
;=====
TSRInstData = $

;-----
, Switcher instance data structure
;-----
SWService      dd 0      , Ptr to switcher service rtn
CallbackInfo   = $
  acbiNext      dd 0      , Ptr to prev callback struct
  acbiEntryPoint dd 0      , Ptr to local callback proc
  acbiReserved  dd 0      , Reserved
  acbiAPI       dd 0      , Ptr to info structure

SwapInfo       = $
               dw 0      , Zero swap info structure
               dw 0      , since no API support
               dw 0      , implemented by TSR
               dw 0
               dw 0
...

TSRInstDataEnd = $
...
```

Además de las estructuras que se necesitan para implementar memoria por sesiones, se debe delimitar el bloque de datos que va a ser instanciado en cada sesión, es decir, se deben delimitar todas las variables que necesitan estar presentes en cada sesión para evitar problemas de reentrada. En TSRTMPLT.ASM el bloque está definido por las

variables: 'TSRInstData' y 'TSRInstDataEnd'. Así, todas las variables que se encuentren entre estos dos identificadores serán reproducidas en cada sesión en donde el TSR sea activado.

Inicialización de estructuras.

Estas estructuras deben ser inicializadas durante la ejecución de la porción transitoria del TSR. El listado 5-11 muestra las siete fases de este proceso, separando cada fase con un renglón en blanco.

Listado 5-11. Inicialización de estructuras de Instance Memory.

```

-----
, Initialize task switcher structures and
, see if task switcher active
-----
    cmp     word ptr Dos_version,300h    ;DOS 3.0 and up
    jnb     NOWIN
    mov     ax,offset TSRInstData      ,Init instance memory
    mov     word ptr [InstItem1],ax    , structure
    mov     word ptr [InstItem1+2],cs
    mov     ax,offset TSRInstDataEnd - offset TSRInstData
    mov     [InstSize1],ax

    mov     ax,offset ResCodeEnd      ,Init instance stack
    mov     word ptr [InstItem2],ax    ; structure
    mov     word ptr [InstItem2+2],cs
    mov     ax,RESSTACKSIZE
    mov     [InstSize2],ax

    mov     ax,offset InstItem1      ,Init ptr to inst
    mov     word ptr [sisInstData],ax  , mem struct
    mov     word ptr [sisInstData+2],cs

    mov     ax,offset SWNotify      ,Init ptr to
    mov     word ptr [scbiEntryPoint],ax , notification
    mov     word ptr [scbiEntryPoint+2],cs ; routine

    mov     ax,4B02h                ,See if switcher active
    int     2Ph                      ,Use multiplex interrupt
    or      ax,ax
    jnb     NOSWITCHER
    mov     word ptr [SWService],di    ;Save ptr to service
    mov     word ptr [SWService+2],es , routine

    mov     ax,4                    ,Hook notification chain
    push    cs                       ,ES BX point to callback
    pop     es                        , structure
    mov     di,offset CallbackInfo
    call    SWService

NOSWITCHER
-----
, Check for Windows active
-----
    mov     ax,1600h                ,See if enhanced mode Windows
    int     2Ph
    or      al,al
    je      NOWIN
    inc     Win_enhanced              ,Set enhanced mode flag

NOWIN:

```

1 y 2) En primer lugar la lista de estructuras SWINSTANCEITEM (InstItem1) debe ser inicializada. En el caso de TSRTMPLT.ASM la lista consta únicamente de dos estructuras: la primera apunta al bloque de variables que ha de ser instanciado, es decir, apunta hacia TSRInstData. La segunda estructura apunta hacia el stack interno del TSR (512 bytes).

3) El campo sisInstanceData de la estructura SWSTARTUPINFO (StartupInfo) es inicializado con la dirección de la lista de estructuras SWINSTANCEITEM.

4) El campo scbiEntryPoint de la estructura SWCALLBACKINFO (CallbackInfo) es inicializado para que apunte a la rutina de notificación implementada dentro del TSR.

TSRTMPLT.ASM reconoce siete tipos de notificaciones las cuales son procesadas por medio de la rutina SWNotify. Esta rutina utiliza la tabla SWNotifyJT (ver: listado 5-10) para llamar a su vez a alguna de las siguientes subrutinas según el tipo de notificación recibido: SWNInti, SWNQSuspend, SWNSuspend, SWNActivate, SWNActive, SWNCSession, SWNDSession, SWNExit. (Todas estas rutinas deben permanecer residentes en memoria, ver: listado 5-12).

La rutina de notificación SWNotify será llamada por el task switcher cada vez que necesite 'notificar' al TSR, que se va a realizar alguna acción (como crear, suspender o destruir alguna sesión). Cuando el TSR recibe la notificación tiene la oportunidad de impedir que la acción se lleve a cabo. Por ejemplo, si mientras el TSR está ejecutando código crítico llega la notificación de que la sesión será suspendida, la rutina SWNSuspend responderá con un código que le indique al task switcher que no debe suspender la sesión en ese momento. Cabe señalar que cada aplicación debe implementar sus propias rutinas de notificación de acuerdo a sus necesidades.

En esencia, las rutinas de notificación deben regresar un valor de cero en AX si están de acuerdo en que se realice la acción y cualquier otro número si la acción debe ser abortada. En TSRTMPLT.ASM todas las subrutinas simplemente regresan AX=0 sin ejecutar ningún proceso adicional. Las rutinas SWNQSuspend y SWNSuspend son una excepción, pues éstas regresan el valor de una bandera interna (No_Switch), la cual permanece encendida mientras el TSR esté ejecutando código crítico. El código para la rutina principal de notificación y el de dos subrutinas se muestra en el listado 5-12. (Las demás subrutinas son iguales a alguna de las dos presentadas, ver: TSRTMPLT.ASM, apéndice H).

5) Aquí se verifica que el task switcher esté cargado en memoria. Si éste se encuentra instalado regresará la dirección de la rutina que presta los servicios para controlar la conmutación de tareas, la cual deberá ser salvada para uso posterior.

6) Ya habiendo inicializado la estructura SWCALLBACKINFO (fase 4) se debe encadenar con las demás estructuras del mismo tipo, que el task switcher utiliza para comunicarse con cada una de las aplicaciones que corren en las diferentes sesiones de DOS. Esta fase es esencial pues es la que en realidad habilita la comunicación entre el

task switcher y el TSR. Esto concluye la inicialización requerida por la API del task switcher de DOS y de Windows 3.1 en modo estándar.

7) Por último, se verifica si Windows está corriendo en modo mejorado y de estarlo, únicamente se prende una bandera de alerta.

Listado 5-12. Rutina de notificación

```

,=====
; SWNOTIFY - Parse switcher notification messages
;=====
SWNotify  proc far
    assume  cs:Cseg, ds:nothing, es:nothing
    push    si
    push    ds
    push    cs
    pop     ds
    assume  ds:Cseg
    mov     word ptr [SWService],di    ,Save ptr to service
    mov     word ptr [SWService+2],es  , routine
    cmp     ax,7
    ja      SWN1                        ;If unknown type, skip
    mov     si,ax
    shl     si,1
    call    [SWNotifyJT+si] ,Call proper notification func
    mov     ax,0

SWN1:
    pop     ds
    pop     si
    ret

;-----
; SWINIT - Switcher initialization notification
;-----
SWNInit  proc near
    assume  cs:Cseg, ds:Cseg
    mov     ax,0                        ,Allow switcher init
    ret
SWNInit  endp

;-----
; SWQSUSPENDINIT - Switcher query suspend notification
;-----
SWNQSuspend  proc near
    assume  cs:Cseg, ds:Cseg
    mov     ax,No_switch                ,Switch only if not in
    ret                                         , critical section
SWNQSuspend  endp
...

```

Manipulación de la memoria por sesiones.

Después de haber inicializado las estructuras, se deben habilitar los mecanismos que permitirán la creación de la instance memory y harán posible la comunicación entre el TSR y el task switcher. Los procesos que a continuación se describen deben ser implementados en la porción residente del TSR.

Como ya se había mencionado, las estructuras SWCALLBACKINFO y SWSWAPINFO sirven para establecer un medio de comunicación entre el TSR y el task

switcher. Pero éste no es el único mecanismo de comunicación disponible. La interrupción multiplex también es utilizada para los mismos fines y en el caso de Windows en modo mejorado, constituye el único medio de comunicación. Por lo tanto, el TSR debe contar con una ISR encadenada a la interrupción multiplex (2Fh). A continuación se muestra el código para esta rutina de interrupción.

Listado 5-13 Uso de la INT 2Fh para manipulación de Instance Memory.

```

; =====
; MUXINT processes DOS Multiplex interrupt (2Fh)
; =====
muxint proc far
    assume cs CSeg, ds nothing, es nothing

    cmp     ax,1605h           ,See if Windows start
    je      INIT_WIN
    cmp     ax,4B05h           ,See if switcher get
    je      INIT_INSTANCE     , instance data
    cmp     ax,4B01h           ,See if switcher build chain
    je      CHAIN_INIT

MUXINT_JMP:
    jmp     cs.[Int2Fh]        ,Call old int

INIT_WIN
    test    dx,01h             ,See if enhanced mode Windows
    jne     INIT_INSTANCE
    inc     cs.Win_enhanced

INIT_INSTANCE
    pushf
    call    cs.[Int2Fh]        ,Call old int
    mov     word ptr cs.[sisNextDev],bx
    mov     word ptr cs.[sisNextDev+2],es
    push    cs                 ,ES BX point to switcher struct
    pop     es
    mov     bx,offset StartupInfo
    jmp     short MUXINT_EXIT

CHAIN_INIT
    pushf
    call    cs.[Int2Fh]        ,Call old int
    mov     word ptr cs.[scbiNext],bx
    mov     word ptr cs.[scbiNext+2],es
    push    cs                 ,ES BX point to switcher struct
    pop     es
    mov     bx,offset CallbackInfo

MUXINT_EXIT
    iret                     ,Return
muxint endp

```

En primer lugar la rutina verifica que el llamado a la interrupción multiplex haya sido hecho por el task switcher de DOS o por Windows en modo mejorado. Si no fue hecho por ninguno de ellos, el control debe ser transferido a la siguiente ISR en la cadena, de tal forma que el control ya no regrese a nuestra rutina (MUXINT_JMP).

"Cuando el task switcher de DOS es ejecutado, genera una interrupción multiplex con la ID=4B05h, con el propósito de identificar a los bloques de memoria por sesiones que requieren de sus servicios. Esto mismo ocurre cuando Windows es ejecutado pero

con la diferencia que la ID es 1605h." [19:p.45] Si nuestra ISR detecta algunos de estos IDs se deben realizar las siguientes acciones:

1) Si el llamado fue hecho por Windows entonces la rutina verifica si esa nueva sesión de Windows está iniciando en modo mejorado y si es así, simplemente prende una bandera que lo indique. Si el bit 0 de DX está apagado entonces Windows está iniciando en modo mejorado (INIT_WIN).

2) "En ambos casos nuestra rutina debe llamar a la siguiente ISR en la cadena. Cuando el control regrese, ES:BX contendrá la dirección de una estructura SWSTARTUPINFO perteneciente a otro programa que también requiera de los servicios del task switcher. Nuestra rutina debe almacenar esta dirección en el campo sisNextDev de la estructura SWSTARTUPINFO del TSR y la dirección de ésta última debe ser almacenada en ES:BX." [19:p.45] En esencia, lo que se está haciendo es armar una lista de estructuras. Con este proceso el task switcher tiene acceso a la información que necesita para crear la memoria por sesiones, para cada programa que haya respondido a la interrupción multiplex original (INIT_INSTANCE).

Cuando el task switcher de DOS desea construir una cadena de rutinas de notificación, es decir, habilitar este mecanismo de comunicación; genera una interrupción multiplex con una ID=4B01h. En este caso, se sigue el mismo procedimiento descrito en el paso 2), con la diferencia de que la estructura utilizada es SWCALLBACKINFO.

Este proceso también se realizó durante la inicialización del TSR. La razón para repetirlo aquí, es para dar soporte a los programas que ya se encuentren residentes al momento de ejecutar el task switcher.

Después de estos procedimientos tanto la memoria por sesiones como las rutinas de notificación ya están habilitadas y, por lo tanto, la porción residente ya puede activarse en varias sesiones de DOS simultáneamente sin peligro. Sólo resta por implementar el código de activación y desactivación. La única función de este código es la de prevenir que la sesión donde el TSR se encuentra activo sea suspendida, si es que la porción residente está ejecutando código crítico.

Generalmente, el único código crítico que siempre está presente en un TSR es el que cambia al stack interno del mismo. Si se llegara a generar alguna interrupción mientras se está haciendo el cambio el resultado sería desastroso, pues el sistema operativo quedaría totalmente confundido y no sabría que stack usar. Por lo tanto, es indispensable evitar cualquier interrupción. Para terminar con el tema de los task switcher se presenta el código de activación y desactivación implementado en TSRTMPLT.ASM.

Listado 5-14. Proceso de activación y desactivación corriendo bajo un task switcher.

```

...
,=====
, MAIN
,=====
main proc far
    assume cs:CSeg, ds:nothing, es:nothing
    cli
    inc     cs:No_switch      ;Don't allow switch

```

```

inc     cs:[Main_active] ,Set program status flag
mov     cs:SS_register,ss ,Save SS and SP registers
mov     cs:SP_register,sp
mov     ax,cs             ,Switch to internal stack
mov     bx,Resident_SP
mov     ss,ax
mov     sp,bx
dec     cs:No_switch
sti     ,Enable interrupts
call    save_regs        ,Save all registers
assume  ds:CSeg, es:nothing
;
;-----
; Restore register values, switch back to original stack
; and return to caller
;-----
MAIN_EXIT:
call    restore_regs     ,Restore registers
assume  ds:nothing
cli     ,Make sure ints are off
inc     cs:No_switch     ,Don't allow task switch
mov     ss,cs:SS_register ,Switch to original stack
mov     sp,cs:SP_register
dec     cs:[Main_active] ,Clear program status flag
dec     cs:No_switch     ,Allow task switch
sti     ,Interrupts on
ret     ,Return to interrupt routine
main    endp

```

Como lo muestra el código, además de deshabilitar las interrupciones (cli), el cual es un paso estándar en cualquier TSR, se enciende la bandera No_Switch. Como se recordará, esta bandera es monitoreada por las subrutinas de notificación SWNQSuspend y SWNSuspend para prevenir que la sesión sea suspendida mientras se realiza algún proceso que no puede ser interrumpido. En este caso, dicho proceso es el cambio de stack representado por las líneas de código que se encuentran entre los 'cli' y 'sti' en los procesos de activación y desactivación respectivamente.

El código mostrado en el listado 5-14 únicamente previene que el task switcher de DOS suspenda la sesión, pero no evita que Windows en modo mejorado lo haga. Para evitar que ambos programas suspendan la sesión, TSRTMPLT.ASM implementa dos funciones (ver: listado 5-15). Si se desean utilizar estas funciones es necesario modificar ligeramente el código del listado 5-14 como sigue:

```

sustituir las líneas:  inc cs:No_Switch
por:                  call stop_switch
y
sustituir las líneas:  dec cs:No_Switch
por:                  call go_switch

```

Listado 5-15. Rutinas para deshabilitar/habilitar la suspensión de sesiones de DOS.

```

;-----
; STOP_SWITCH signals the task switcher (if active) to not
; switch this session
;-----
stop_switch  proc near
    assume  cs,CSeg
    inc     cs.No_switch      ,Set no switch flag for T.S.
    cmp     Win_enhanced,0
    jne     STOP_SWITCH_1

STOP_SWITCH_EXIT
    ret

STOP_SWITCH_1
    mov     ax,1681h          ;Win begin critical section
    int     2Fh
    jmp     short STOP_SWITCH_EXIT
stop_switch  endp

;-----
; GO_SWITCH signals the task switcher (if active) it is
; OK to switch this session
;-----
go_switch    proc near
    assume  cs,CSeg
    dec     cs.No_switch      ,Clear switch flag for T.S.
    cmp     Win_enhanced,0
    jne     GO_SWITCH_1

GO_SWITCH_EXIT
    ret

GO_SWITCH_1
    mov     ax,1682h          ;Win end critical section
    int     2Fh
    jmp     short GO_SWITCH_EXIT
go_switch    endp

```

Estas rutinas, aparte de encender la bandera No_Switch, verifican si Windows está corriendo en modo mejorado, utilizando para ello la bandera Win_enhanced la cual se enciende durante la inicialización o por medio de la interrupción multiplex. Si esta bandera está encendida, se genera una interrupción multiplex con un ID=1681 para deshabilitar la suspensión de sesiones y con un ID=1682 para volverla a habilitar.

PRECAUCION. "Varias de las tecnologías presentadas aquí dependen específicamente de la versión 5 (o superiores) de MS-DOS. Generalmente, la versión de DOS se puede determinar por medio de la función 30h (Get Version Number), pero la adición al sistema operativo del programa SETVER.EXE trae como consecuencia que ya no se pueda confiar en la versión reportada por esta función. Para resolver este problema, se ha añadido a DOS la función 3306h (Get DOS Version) la cual siempre regresa la verdadera versión del sistema operativo que se está utilizando." [19:p.42]

CAPITULO 6 EL FUTURO DE LOS TSR

Una cosa es segura dentro del mundo de las computadoras: nada permanece estático. Todas las tecnologías deben renovarse y mejorar. Los programas residentes no son la excepción y, por lo tanto, en éste último capítulo se habla del posible futuro de los TSR.

VENTAJAS Y DESVENTAJAS DE LOS TSR ACTUALES.

Para saber en que áreas es probable que los TSR cambien, es necesario hacer una evaluación de sus ventajas y desventajas. Es en las desventajas de éstos, en donde podemos esperar que haya más cambios en un futuro.

Entre las ventajas que ofrecen los programas residentes podemos mencionar que son muy útiles, aumentan la eficiencia con la que interactuamos con la máquina, aumentan nuestra productividad, nos facilitan muchas tareas, etc. Muchas de estas ventajas están basadas en el hecho de que estos programas están disponibles con sólo presionar una tecla.

Tal vez la ventaja más grande que tienen los programas residentes en memoria, es que permiten simular un ambiente multiproceso. Esto es cierto desde el punto de vista del usuario, ya que éste tiene a su disposición inmediata y no importando en donde esté, una gran variedad de utilerías como relojes, calculadoras, programas para impresión simultánea, editores de texto, programas para revisar ortografía y muchos más.

Por otro lado, las desventajas más serias que tienen los programas residente en memoria están relacionadas con su misma naturaleza como se ve a continuación:

- 1) El TSR consume y retiene memoria principal que podría se utilizada por otro programa.

2) Los TSR interceptan interrupciones. Generalmente, cada vez que se carga un TSR, se interceptan las mismas interrupciones una y otra vez. Esto crea una cadena de ISRs que entre más larga sea más inestabilidad introduce en el sistema.

3) El instalar demasiados TSR en memoria degrada el desempeño de la computadora. Es justo decir que esto no ocurre en todos los casos.

La presencia de un solo TSR en memoria rara vez causa dificultades, pero cuando múltiples programas luchan por tener el control de las mismas interrupciones (del reloj del sistema o del teclado por ejemplo) su interacción puede provocar que el sistema se comporte en forma impredecible.

Aún si todos los programas pueden coexistir pacíficamente, se tiene el problema de que se ha creado una cadena muy larga de ISRs que están monitoreando cada una el teclado en busca de su secuencia específica de teclas para poder activarse. Los múltiples monitoreos roban tiempo de procesamiento y esto hace que la máquina se vuelva lenta. Otro problema relacionado también con el teclado se da cuando dos o más TSR responden a la misma secuencia de teclas para activarse.

Si a todo lo anterior se le agrega que a veces es imposible correr un programa que requiera de mucha memoria debido a que están instalados varios TSR que ya la están ocupando, entonces tenemos un escenario en donde los TSR causan muchos problemas que no se compensan con los beneficios que pudieran aportar.

Ray Duncan (afamado programador y autor de libros que son considerados biblias en el mundo de las PC) alguna vez escribió: "Los TSR han sido víctimas de su propio éxito" [22:p.50], lo cual es cierto. Conforme los TSR han ido desarrollando sus capacidades se han vuelto más populares y los usuarios quieren tener un número cada vez mayor de ellos instalados en memoria, provocándose así un aumento en las exigencias impuestas al sistema. Con esto, lo único que se ha logrado es que las interacciones entre los programas de aplicación, los TSR y la computadora sean cada vez más complicadas y en determinadas ocasiones hasta imposibles.

"Como consecuencia, una nueva clase de programas TSR ha nacido con el propósito de resolver o cuando menos atenuar estos problemas. Con estos nuevos programas es posible activar, desactivar y desinstalar de memoria cualquier TSR de acuerdo a los comandos del usuario. Estos programas también actúan como mediadores entre los diversos TSR que se encuentren cargados en memoria al mismo tiempo, evitando así posibles conflictos entre ellos." [22:p.49] Como ejemplos podemos mencionar Software Carousel y DoubleDos.

"Existen además varias propuestas encaminadas a la interacción pacífica entre TSRs. Tenemos por ejemplo: la API de SideKick Plus de Borland International, la Metro de Lotus Development y por último una especificación del dominio público llamada RingMaster." [22:p.49]

Utilizando estas propuestas se eliminaría el problema de que cada TSR capturara las mismas interrupciones una y otra vez, ya que todos los TSR dependerían de un solo punto de entrada controlado por el un TSR maestro. Este programa al ser activado desplegaría un menú conteniendo todos los servicios disponibles y en donde cada TSR

correspondería a un servicio. También el problema de la memoria desaparecería porque sólo el TSR maestro estaría instalado y éste la administraría de acuerdo a las diferentes peticiones de servicios.

Desafortunadamente estas propuestas fueron hechas desde 1987 y nunca tuvieron mucha aceptación y probablemente ya nunca la tendrán. Ahora en 1992, la forma de simular multiproceso es mediante el uso del ambiente Windows, pero este ambiente es un parche más al sistema operativo MS-DOS.

LOS TSR Y EL SISTEMA OPERATIVO OS/2.

Es tiempo de un cambio radical. Si los TSR son una forma de compensar las deficiencias de MS-DOS, ¿por qué no cambiar entonces el sistema operativo en su totalidad en vez de seguir parchándolo ?

Esto fue lo mismo que pensaron IBM y Microsoft en 1987 cuando liberaron su entonces nuevo sistema operativo, monousuario, multiproceso: OS/2. Hasta la fecha OS/2 no ha sido ampliamente aceptado por la comunidad computacional y ahora que se ha anunciado la próxima liberación de Windows NT (Windows New Technology) tal vez ya nunca lo sea, el tiempo lo dirá. Entonces por qué investigar la forma cómo desarrollar un TSR que corra bajo OS/2 y no bajo Windows NT. La razón es muy sencilla; a la fecha de realización de este trabajo, Microsoft no ha liberado públicamente ninguna documentación relacionada con los aspectos técnicos de este nuevo sistema operativo y sin ella es prácticamente imposible desarrollar cualquier programa.

Sin embargo, es conveniente conocer los conceptos fundamentales para desarrollar este tipo de programas bajo OS/2, en primer lugar por la importancia que tiene este sistema operativo por sí solo y en segundo lugar por las grandes similitudes que existen entre él y Windows NT. Así, un programa desarrollado para cualquier sistema operativo debería correr en el otro sin necesidad de realizar demasiadas modificaciones al código.

Al cambiar al sistema operativo OS/2 se asume que las deficiencias que los TSRs compensaban fueron eliminadas y, por lo tanto, estos programas ya no tienen ninguna razón para existir. Por ejemplo, no hay más necesidad de simular multitareas cuando el mismo sistema operativo es multitareas por naturaleza. En contraste con este razonamiento, nos encontramos con la API de OS/2, la cual proporciona todos los servicios necesarios para desarrollar estos programas pero en forma más sencilla y al mismo tiempo mucho más robusta.

Bajo OS/2 muchos de los problemas que aquejaban a los TSRs son irrelevantes ya que el sistema operativo realiza la mayor parte del trabajo que se necesita para dejar un programa residente en memoria. A continuación se describen tres de estos problemas ya resueltos:

1) La manipulación de la memoria que se debía realizar al interactuar con los task switchers, por ejemplo, bajo OS/2 es innecesaria, pues él mismo proporciona la protección de memoria (Memoria Protegida) y el sistema de niveles de privilegio, es decir, procesamiento de acuerdo a las prioridades de cada tarea.

2) El problema de la escasez de memoria se ha resuelto ya que OS/2 no utiliza memoria segmentada y como el sistema operativo debe correr en un procesador 80286 como mínimo, la memoria accesible puede llegar hasta 1Gb corriendo en modo protegido. Esto sin necesidad de tener físicamente instalada tal cantidad de memoria, pues OS/2 puede utilizar memoria virtual (en disco).

3) El que OS/2 soporte multitareas se traduce en que OS/2 es totalmente reentrante, así un TSR puede hacer llamadas al sistema operativo en cualquier momento sin necesidad de implementar ninguna clase de truco para evitar conflictos. Sucede lo mismo con la activación del TSR, ya que éste no tiene que preocuparse por salvar el estado del programa que esté corriendo en ese momento, es OS/2 quien se encarga de manejar todos esos detalles.

Que el sistema operativo sea quien se encargue de los detalles que antes le correspondían al TSR, no es casualidad sino necesidad. Bajo MS-DOS cualquier programa y un TSR con más razón, puede acceder directamente todos los recursos del sistema. En la práctica si existen algunas restricciones pero nada que no se pueda evadir de una u otra manera.

Bajo OS/2 el panorama cambia totalmente ya que el sistema operativo es el único que puede y debe tener acceso directo a los recursos del sistema, para así poder garantizar la coexistencia pacífica entre los diversos procesos concurrentes. En OS/2 no se necesita que cada programa cuide de no interferir con los demás, sino que el sistema operativo es quien actúa como árbitro entre ellos. Así, cuando un programa genera algún error fatal, únicamente el proceso asociado con él será suspendido sin que esto afecte a los demás procesos.

Por lo tanto, un programa de aplicación debe acceder los recursos por medio del sistema operativo, es decir, a través de llamadas a los diferentes servicios de OS/2. Si el acceso a estos recursos se hiciera en forma directa se pondría en peligro la integridad de todas las tareas que estén corriendo simultáneamente en la computadora. Además, si OS/2 ya tiene el código para controlar todas las tareas de bajo nivel, sería tonto que un programa tratara de duplicar todo ese trabajo, mejor utilizar lo que ya está hecho.

Antes de iniciar con la explicación de los conceptos fundamentales para desarrollar un TSR diseñado para el sistema operativo OS/2, cabe hacer la aclaración que bajo este sistema operativo el nombre TSR no es totalmente correcto, pues un programa no necesita terminar su ejecución, permanecer en memoria y después realizar varios trucos para reactivarse. Como OS/2 es multitarea varios programas pueden estar ejecutándose en forma concurrente en memoria, sin que por ello se genere ningún conflicto entre estos programas.

(Nota: Los fragmentos de código que se presentan en esta sección no se pueden considerar 'funcionales', simplemente son un ejemplo de las grandes facilidades que proporciona el sistema operativo OS/2 para desarrollar esta clase de programas. El

desarrollar TSRs y en general cualquier programa trabajo de investigación completo e independiente).

El sistema operativo OS/2 ofrece muchos servicios para realizar programas de todas Hclases, pero ~~resiste~~ ^{existen} que son de Hpecial importancia cuando se desea

- 1) Threads.
- 2) Popup Displays.
- 3) Device Monitors.

TTHREADS.

"Al igual que otros sistemas operativos multitarea, OS/2 utiliza el concepto de 'Proceso'. Cada proceso representa dos cosas: la ejecución de un programa de aplicación y la asignación de recursos (archivos, memoria, etc.) a dicha ejecución. Cada proceso puede crear otros procesos y puede intercambiar datos con ellos a través de los mecanismos de comunicación 'Interproceso'.

Además de estos conceptos OS/2 maneja también la noción de 'Thread' (Hilo de Ejecución). Un thread representa un punto de ejecución dentro de un proceso, en otras palabras, un thread es la unidad mínima de ejecución." [22:p.51] Cada thread es controlado por el 'System Scheduler', es decir, por la parte del sistema operativo que se encarga de repartir el tiempo del procesador entre las diferentes tareas activas en memoria, de acuerdo a sus prioridades.

Así, cada proceso tiene un thread primario que recibe el control de OS/2 cuando el programa en cuestión es ejecutado. "Es posible para el thread primario iniciar la ejecución de otros threads dentro del mismo proceso sin necesidad de crear nuevos procesos independientes " [22:p 51]

"Al mantener múltiples threads dentro de un mismo proceso se obtienen los siguientes beneficios:

- 1) Los threads podrán correr en forma asíncrona entre sí, es decir, cada thread se ejecuta en forma independiente de los demás.
- 2) Cada thread podrá tener diferente prioridad de ejecución y podrá manipular las prioridades de los demás threads.
- 3) Los threads pueden compartir el acceso a la misma memoria, archivos, queues (listas de espera para impresoras, comunicaciones, etc.) y 'Semáforos'." [22:p.51]

Un ejemplo de esta forma de ejecución, se puede apreciar en un procesador de palabras. El procesador en sí sería el thread principal. Cuando el usuario ordene salvar el archivo modificado a disco, el programa principal creará un nuevo thread, el cual se encargará de realizar el proceso de salvado. El beneficio que se obtiene con esta forma de operación, es que ambos threads pueden ejecutarse en forma concurrente, es decir, simultáneamente. Tal vez se debería dar mayor prioridad al thread principal (proceso de

palabra) para que el usuario no note una degradación en la velocidad de respuesta del programa. Al terminar de salvar el archivo a disco, el thread secundario será destruido para que no siga consumiendo tiempo del procesador.

Este concepto de thread se ajusta muy bien a la forma como operan los TSRs. Por ejemplo, un programa como SAVESCR.PAS pero diseñado para correr bajo OS/2 tendría tres threads. El primero, y el de mayor prioridad, deberá monitorear el teclado. Este será el único thread activo hasta que el usuario presione la secuencia de teclas específica del TSR. Los otros dos threads serían para vaciar la pantalla a disco y para controlar los errores de manejo de archivos.

POPUP DISPLAYS.

Como ya se sabe, bajo MS-DOS cuando el TSR desea desplegar algo en pantalla, debe implementar el código para salvar y restaurar el contenido de la misma. Si esto se tuviera que hacer para todos los modos de video y para todas las tarjetas gráficas significaría bastante trabajo para el programador.

Para evitar todo este trabajo se limita la funcionalidad del TSR y se acepta que el programa corra únicamente bajo los modos de video más comunes.

"Bajo OS/2 estos problemas no existen ya que el mismo sistema operativo proporciona los servicios para manipular el video, los cuales son independientes de la tarjeta gráfica que esté instalada en la computadora. Un TSR puede llamar a la función VIOPOPUP para obtener el control de la pantalla. Esta función realiza los siguientes procesos:

- 1) Salva el contenido de la pantalla actual, así como el modo de video.
- 2) Envía el programa en 'Foreground' hacia el 'Background', es decir, asigna al proceso principal un estatus de proceso secundario, en preparación para dejar al TSR como principal.
- 3) Cambia a modo texto (80x25) y limpia la pantalla
- 4) Regresa el control al TSR para que éste pueda disponer de la pantalla de acuerdo a sus necesidades.

Cuando el TSR sea desactivado o cuando ya no necesite hacer uso del monitor, simplemente se deberá llamar a la función complementaria: VIOENDPOPUP, con la cual pedimos a OS/2 que restaure el contenido de la pantalla, el modo de video y el programa anterior." [22:p.51] El fragmento de código que se muestra en el listado 6-1, realiza precisamente estos procesos.

Listado 6-1. Uso de las funciones VIOPOPUP y VIOENDPOPUP. [22:p.50]

```

..
;-----
; Data declaration section
;-----

PopFlags dw 1      ,1=Wait when window not available
              ,0=Don't wait.

Msg       db 'This is a pop up window'
MsgLen    equ $-Msg

;-----
; More variable definitions as needed
;-----

..

;-----
; Display pop up window
;-----
        , Push function arguments
        , address of pop-up flags
        push ds      , offset dgroup:PopFlags
        push 0       , VIO handle (reserved)
        call vpopup  , Call OS/2
        or ax,ax      , If error
        jnz ERROR    , abort

;-----
; Display message
;-----
        , Push function arguments
        , message address
        push ds      , offset dgroup:Msg
        push MsgLen  , message length
        push 12      , y coord
        push (80-MsgLen)/2 , x coord (centered)
        push 0       , VIO handle (reserved)
        call viowrtcharstr , Call OS/2
        or ax,ax      , If error
        jnz ERROR    , abort

; Pause for 5 seconds
        push 0
        push 5000
        call dosleep

;-----
; Close pop up window
;-----
        , Pus function arguments
        , VIO handle (reserved)
        push 0
        call vioendpopup , Call OS/2
        or ax,ax      , If error
        jnz ERROR    , abort

```

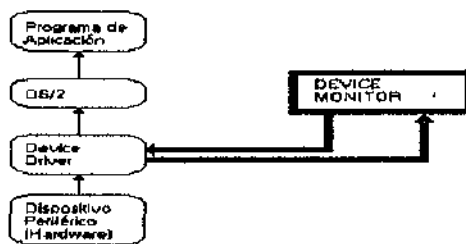
Como se puede ver, el código es muy sencillo, pues el TSR simplemente realiza llamadas al sistema operativo para solicitar los diferentes servicios que requiere sin necesidad de implementar internamente el código específico para cada una de las acciones.

DEVICE MONITORS.

Un 'Device Monitor' es un programa que monitorea, valga la redundancia, la actividad de algún device driver específico. (Al igual que MS-DOS, OS/2 también utiliza los device drivers para aislar al usuario de los detalles del manejo del hardware, ver 'Device Drivers', capítulo 2).

Normalmente, el flujo de datos desde los periféricos hasta los programas de aplicación sigue un camino estrictamente definido a través del sistema. Como se puede ver en la figura 6-1 el dato se genera en el periférico, pasa por el device driver correspondiente, pasa a través del sistema operativo y llega al programa de aplicación.

Figura 6-1. Flujo de datos en OS/2. [22:p.51]



Cuando el dato llega a la aplicación ya ha pasado por varios procesos de filtrado y validación. Para lograr el acceso a estos datos antes de que se realice dicha filtración, es necesario que se implemente un device monitor. Con estos programas se logran interceptar los datos al nivel más bajo permitido por el sistema operativo. Como ya se mencionó, bajo OS/2 corriendo en modo protegido ningún programa tiene acceso directo a los recursos del sistema y en especial existen restricciones para el acceso directo al hardware.

Para que un programa sea considerado un device monitor debe realizar los siguientes procesos:

- 1) "Llamar a la función DOSMONOPEN, pasando como argumento el nombre del dispositivo que se desea monitorear.
- 2) En este momento el kernel de OS/2 actúa como intermediario entre el dispositivo y el programa. En primer lugar, OS/2 verifica que el dispositivo solicitado soporte monitoreos y de ser así, el sistema operativo regresa un 'Monitor Handle' con el cual el programa queda identificado como un device monitor.
- 3) Si se realizaron los 2 primeros pasos con éxito, el programa debe registrarse en la cadena de device monitors asociada al dispositivo en cuestión. Para ello se utiliza la función DOSMONREG y se envían como argumentos: el handle regresado por la primera llamada, la dirección de dos buffers de datos que serán utilizados para

procesar las funciones de Entrada y Salida de los 'Paquetes' de datos del device driver y por último un valor que indique en que posición, dentro de la cadena de device monitors desea estar el programa (primer lugar, último o no importa)." [5:p.173]

Si dos o más programas solicitan ser el primero (o el último) en la cadena, siempre se respeta el orden original, es decir, el programa que lo solicitó primero tiene prioridad después del cual vendrá el segundo en haberlo solicitado, etc.

Si el paso 3 se realiza exitosamente, el programa se convierte automáticamente en un device monitor y adquiere la responsabilidad de pasar los datos a lo largo de la cadena de device monitors, para lo cual utilizará sus buffers de datos. Este proceso es muy similar al encadenamiento de los vectores de interrupción que se debe realizar bajo MS-DOS.

Para lograr esto se debe hacer uso de las funciones DOSMONREAD y DOSMONWRITE, las cuales como sus nombres lo indican, son utilizadas para leer/escribir un paquete de datos de/hacia el device driver. El programa realiza la validación o la modificación de los datos que contiene el paquete después de llamar a la función de lectura y antes de llamar a la función de escritura.

Por ejemplo, con una llamada a la función DOSMONREAD se puede obtener un paquete de datos del teclado, el paquete contendrá el caracter de la tecla, su scan code (código de rastreo), la hora en que fue presionada la tecla, etc. Al obtener esta información el device monitor puede verificar si la tecla presionada fue su tecla de activación. Si lo fue entonces retendrá el paquete de datos, es decir, no lo transmitirá a lo largo de la cadena de device monitors. Si no fuera su tecla de activación, el device monitor simplemente llamaría a la función DOSMONWRITE, la cual regresa el paquete de datos al device driver para que éste lo pueda seguir transmitiendo a los demás device monitors en la cadena.

"Como regla general, el TSR implementará un ciclo de lectura/escritura y lo asignará a un thread con alta prioridad, esto es con el fin de no degradar el desempeño del sistema en general." [22:p.51]

Por último, cuando el device monitor ya no sea necesario, es decir, cuando se desee retirar al programa de la cadena de device monitors del periférico que estemos monitoreando, simplemente se debe hacer un llamado a la función DOSMONCLOSE enviando como argumento el handle que nos fue asignado por el sistema operativo cuando llamamos a la función DOSMONOPEN. Después de esto, el programa puede detener su ejecución sin ninguna consideración adicional. Para terminar, se presenta el esqueleto de un device monitor que puede ser incorporado al código de un TSR real.

Listado 6-2 Esqueleto de un Device Monitor [22:p.52]

```

,-----
, Data declaration section
,-----

```

```

;-----
; Device monitor variables (Keyboard Monitor)
;-----
KbdName    db  '\DEV\KBD$',0    ;Keyboard logical device name
KbdHandle  dw  ?                ;Handle for DOSMONOPEN

KbdMonIn   dw  128,64 dup(0)    ;Buffers for the kbd monitor
KbdMonOut  dw  128,64 dup(0)

KbdPacket  db  128 dup(0)       ;Hold kbd data packet
KbdPktLen  dw  ?                ;Length of data in KbdPacket

ScrGroup   dw  ?                ;Current screen group number

;-----
; Define more variables as needed
;-----

.

;-----
; Open monitor connection
;-----
push        ds                    ;Push function arguments
push        offset dgroup,KbdName ; device name
push        ds                    ; address to receive
push        offset dgroup:KbdHandle ; handle
call        dosmonopen            ;Call OS/2
or          ax,ax                  ;If error
jnz         ERROR                 ; abort

;-----
; Register device monitor
;-----
push        KbdHandle              ;Push function arguments
push        ds                    ; monitor handle
push        offset dgroup:KbdMonIn ; input buffer address
push        ds                    ; output buffer address
push        offset dgroup:KbdMonOut
push        1                      ; request front of list
push        ScrGroup
call        dosmonread             ;Call OS/2
or          ax,ax                  ;If error
jnz         ERROR                 ; abort

.

;-----
; More initialization code as needed
;-----

...

;-----
; Actual keyboard monitor loop
; Must be very tight to avoid degrading keyboard
; response
;-----
NEXTCHAR:

; Read next packet

mov         KbdPktLen,KbdPktLen-KbdPacket ;Push function arguments
push        ds                    ; max length of read
push        offset dgroup:KbdMonIn ; address of input buffer
push        0                      ; wait until data available
push        ds                    ; address to receive packet
push        offset dgroup:KbdPacket
push        ds
push        offset dgroup:KbdPktLen
call        dosmonread             ;Call OS/2
or          ax,ax                  ;If error
jnz         ERROR                 ; abort

; Compare keyboard data against TSR's hotkey
cmp         byte ptr KbdPacket+3,HotKey
jz          ACTIVATE_TSR           ;Activate if HotKey pressed.

```

```

, If Hotkey not pressed, pass data on
; Push function arguments
push    ds                ; output buffer address
push    offset dgroup,KbdMonOut
push    ds                ; kbd packet address
push    offset dgroup,KbdPacket
push    KbdPktLen         ; length of kbd packet
call    dosmonwrite       ; Call OS/2
or      ax,ax             ; If error
jnz     ERROR             ; abort

jmp     NEXTCHAR          ; End of monitor loop

```

```
...
```

```

;-----
, Close monitor connection
;-----

```

```

DONE,
; Push function arguments
push    KbdHandle         ; monitor handle
call    dosmonclose       ; Call OS/2
or      ax,ax             ; If error
jnz     ERROR             ; abort

```

```
...
```

CONCLUSION

MS-DOS es un sistema operativo poderoso capaz de proveer al usuario de muchos servicios, pero es la naturaleza humana querer más entre más se tiene; por consiguiente siempre encontramos limitantes en lo que ya tenemos y buscamos formas de eliminarlas.

Los TSR son precisamente eso: una forma de adaptación a las limitaciones impuestas por el sistema operativo DOS. La utilidad de los programas residentes es innegable, además para determinadas aplicaciones estos programas llegan a ser más que convenientes, son absolutamente indispensables. Tanto es así, que la cantidad de programas TSR que se desarrollan con propósitos comerciales es enorme.

Es innegable también que las técnicas utilizadas para desarrollar TSRs son bastante complejas, lo cual dificulta su desarrollo, y si a esto le agregamos el hecho de que también existen algunas desventajas asociadas con el uso de estos programas, sobre todo cuando se utilizan en exceso, se llega a la conclusión de que el uso de los TSR está limitado a un grupo específico de aplicaciones y no es aconsejable tratar de sobrepasar estos límites.

Los programas residentes por algún tiempo fueron y todavía son, una forma adecuada de compensar las deficiencias de MS-DOS; pero el universo de las PC se vuelve cada vez más complejo y cada vez es más difícil mantener el paso de las nuevas tecnologías.

Por lo tanto es lógico pensar que llegaran nuevas tecnologías que eventualmente reemplazarán la funcionalidad de los TSR. Quizás no serán sustituidos del todo, sino que únicamente mejorará la forma de implementarlos o tal vez cambiará la forma en que interactúan con la computadora. Lo único que sí es seguro es que cambiarán.

APENDICES

A continuación se anexan los listados completos de todos los programas de demostración mencionados en el texto de este trabajo. Los listados están ordenados según el grado de complejidad del programa (de menos a más):

<u>APENDICE</u>	<u>PROGRAMA</u>
A	XKBDBUF.PAS Programa para extender el buffer del teclado. <Para: DOS 2.0 o superior>
B	F11.PAS Este programa puede reemplazar al KEYB.COM de DOS cuando se carga el teclado en español (SP). Proporciona los caracteres áéíóúñ a través de la tecla F11. <Para: DOS 3.1 o superior>
C	CLOCK.ASM Reloj digital constante en la esquina superior derecha, (modo texto). <Para: DOS 1.0 o superior>
D	BLANKER.ASM Un limpiador de pantalla para períodos de inactividad, (modo texto). <Para: DOS 3.1 o superior>
E	POPCLOCK.ASM Reloj digital en una ventana tipo PopUp, (modo texto). <Para: DOS 2.0 o superior>

- F** **JTRSTP.PAS**
 Unidad (TPU) que proporciona los servicios para convertir un programa normal en un programa residente en memoria.
 <Para: DOS 3.1 o superior>
- G** **SAVESCR.PAS**
 Este programa utiliza la unidad JTSRTP.TPU para obtener los servicios necesarios para permanecer residente en memoria. El programa salva pantallas (modo texto) a disco. Se salvan tanto los caracteres como los atributos de video.
 <Para: DOS 3.1 o superior>
SHOWSCR.PAS (NO residente)
 Utilería para desplegar las pantallas salvadas.
 <Para: DOS 2.0 o superior>
- H** **TSRTMPLT.ASM**
 Programa de demostración. Este programa únicamente hace sonar la bocina de la computadora cuando se presionan sus teclas de activación y desactivación. TSRTMPLT.ASM es sólo el esqueleto de un programa real, al cual se le debe agregar código para que realice alguna función útil.
 <Para: DOS 3.1 o superior>

```

*****
* system      This is a Demo
* Program     XKBDDBUF.PAS
* Description  TSR program that increases the size of the
*             keyboard type-ahead buffer by modifying low
*             memory pointers that hold the default
*             keyboard buffer
*             Reference: Norton Guides -
*             Assembly Language Database
*             This program was compiled with
*             Turbo Pascal v6.0
* Author      Kris Jamsa ("DOS Power User's Guide")
* Modified by José Luis Infante Ortega
* Revisions   (01-DEC-92) Version 1.0
*****

```

```

program XKbdBuf;

{$M 1024,0,0} { 1K stack, no heap }

uses dos;

const
  BUFSZ = 512, { Enough for 256 keys }
  PSPSZ = 256,
  CODESZ = 1000, { Estimated code size }

var
  Regs registers;

{ Memory location: 0000 0480h = Keyboard Buffer }
{ Start Offset Pointer This word contains the }
{ offset of the start of the keyboard buffer. }
{ This allows you to move the buffer, increase its }
{ size, or decrease its size. There is one catch, }
{ however. The address here is an offset starting at }
{ address 0 400h. For example, the default 32-byte }
{ buffer at 0 41eh forces this offset address to be }
{ 001eh. Therefore, if the buffer is to be moved to }
{ another location, the buffer starting offset must }
{ be relative to 0 400h }
  KbdBufBegin integer absolute $0000:$0480,

{ Memory location: 0000 0482h = Keyboard Buffer End }
{ Offset Pointer This word contains the offset of the }
{ end of the keyboard buffer plus one. This allows you }
{ to move the buffer, increase its size, or decrease its }
{ size. There is one catch, however. The address here is }
{ an offset starting at address 0 400h. For example, the }
{ default 32 byte buffer at 0 41eh forces this offset }
{ address to be 003eh (20h for the 32-byte buffer itself). }
{ Therefore, if the buffer is to be moved to another }
{ location, the buffer ending offset must be relative to }
{ 0 400h }
  KbdBufEnd integer absolute $0000:$0482,

{ Memory location: 0000 041ah = Keyboard Buffer Head }
{ Pointer This word points to the next character code }
{ scan code to be removed from the keyboard buffer. If }
{ the head pointer is equal to the tail pointer (0 41ch) }
{ then the buffer is empty }
  KbdBufHead integer absolute $0000:$041A,

{ Memory location: 0000 041ch = Keyboard Buffer Tail }
{ Pointer This word points to the location in the }
{ keyboard buffer at which the next character typed will }
{ be placed. After a character is placed in the keyboard }
{ at this location, the tail pointer is incremented and }
{ adjusted if it needs to wrap to the beginning of the }
{ keyboard buffer. If the tail pointer is equal to the }
{ head pointer (0 41ah), the buffer is empty }
  KbdBufTail integer absolute $0000:$041C,

{ Memory location: 0000 041eh = Keyboard Buffer }
{ (32 bytes long) This buffer holds the ASCII character }
{ code followed by the scan code for each key typed from }
{ the keyboard. These 32 bytes are actually 16 2-chars }
{ entries. Each entry holds the ASCII character code }
{ followed by the scan code for the key typed }
{ New type-ahead buffer }
  KbdBuffer array[1..BUFSZ] of byte;

{----- Main program -----}
begin
  { Set the offset of the buffer relative to segment 40h }
  KbdBufBegin =
    (seg[KbdBuffer[1]]-$40) * 16 + ofs(KbdBuffer[1]);

  { Adjust the rest of the pointers }
  KbdBufEnd = KbdBufBegin + BUFSZ;

  { If head = tail then buffer is empty }
  KbdBufHead = KbdBufBegin;
  KbdBufTail = KbdBufBegin;

  { Terminate and Stay Resident }
  { Specify resident size }
  Regs.dx = (PSPSZ + BUFSZ + CODESZ) div 16 + 1;
  Regs.ax = $3100;
  mados(Reg);
end { program - XKbdBuf }

```

```

*****
* System      Thesis Demo
* Program     F11 PAS
* Description  New interrupt 09h handler
*             Provides accented characters without
*             loading KEYB.SF
*             Usage   Press F11 and [a|e|i|o|u|n|N]
*                   To get [á|é|í|ó|ú|ñ|Ñ]
*             Reference Norton Guides -
*                   Assembly Language Database
*                   Turbo Assembler's THELP.DOC
*             This program was compiled with
*             Turbo Pascal v6.0
* Author      José Luis Inigo Ortega.
* Revisions   (01-DEC-92) Version 1.0
*****

```

```

program F11,
($M 1024,0,0)
($I-,R-,S-,D-)

uses dos,

const
  acF11 = $57,      { F11's scan code }
  BIOSDATA = $0040,

var
  AFlag byte,
  _OldInt09 procedure,

{ Keyboard buffer head ptr }
  KbdBufHead integer absolute $0000:$041A,

{ $P+ }
procedure newint09, interrupt,
{ New keyboard ISR }
begin
  { Call old interrupt }
  asm pushf end, { pushf to simulate interrupt }
  _OldInt09.

{ Check if F11 was pressed }
  if Port[$60] = acF11 then AFlag = 1

{ Don't act when F11 is released }
  else if AFlag = 1 then AFlag = 2
  else
    if AFlag = 2 then
      begin
        case char(mem[BIOSDATA.KbdBufHead]) of
          'a' mem[BIOSDATA.KbdBufHead] = byte('á'),
          'e' mem[BIOSDATA.KbdBufHead] = byte('é'),
          'i' mem[BIOSDATA.KbdBufHead] = byte('í'),
          'o' mem[BIOSDATA.KbdBufHead] = byte('ó'),
          'u' mem[BIOSDATA.KbdBufHead] = byte('ú'),
          'n' mem[BIOSDATA.KbdBufHead] = byte('ñ'),
          'N' mem[BIOSDATA.KbdBufHead] = byte('Ñ'),
        end, { case }

{ Don't reset flag if shift is pressed to get Ñ }
        if ((mem[BIOSDATA.$0017] and $02) <> $02) and
           ((mem[BIOSDATA.$0017] and $01) <> $01) then
          AFlag = 0,
        end,
      end, { newint09 }
{ $P- }

{----- Main program -----}
begin
  AFlag = 0,

{ Save current ISR }
  getintvec($09,@_OldInt09),

{ Insert F11's ISR into keyboard's chain }
  setintvec($09,@newint09),

{ Terminate and stay resident }
  keep(0),
end { program - F11 }

```

```

>>
;*****
; System      Thesis Demo
; Program     CLOCK ASM
; Description  TSR program for displaying a digital clock
;             on the screen's top right corner
;             Text mode only
;             This TSR shows the use of INT 27h instead
;             of INT 21h function 31h to stay resident
;             This program was build with
;             Turbo Assembler v2.01 and
;             Turbo Link v3.01
;             To build CLOCK.COM use the following or
;             equivalent commands
;             tasm clock.asm
;             tlink clock.obj /t
; Author      S. Holzener ('Desktop Workshop')
; Revisions   (01-DEC-92) Version 1.0
;*****

;===== Data segments =====
Interrupts segment at 0h
org 1Ch*4           ;1Ch is the timer interrupt

TimerInt label word
Interrupts ends

BiosStorage segment at 40h
org 6Ch             ;Timer data is stored at 40 6C and 40 6E

TimerLo dw ?
TimerHi dw ?
BiosStorage ends

VidMem segment at 0B000h ;Dummy segment to use as
VidMem ends           ;the Extra Segment

;=====
CSeq segment
assume cs CSeq
org 100h              ;Standard for COM programs

ENTRY_POINT
jmp loadclock

;--- Variable declaration section ---
_OldTimerInt dw 2 dup(?) ;Original INT 1Ch vector
RecalcCount dw 0         ;Update clock flag
VideoPort dw 7           ;Video status port (Check
                        ;it before writing directly
                        ;to video memory)

DispPos dw 0             ;Location of the cursor on screen
ClkDisp dw 5 dup(703Ah) ;Initial clock value
                        ;70h = reverse video
                        ;3Ah = ' ' char

;-----
clock proc near
    New timer interrupt (INT 1Ch)

    Save all registers used by this routine
    push ax
    push cx
    push di
    push si
    push ds
    push es

    ; Must push flags before calling any interrupt routine
    ; Call old timer interrupt
    pushf
    call dword ptr _OldTimerInt

    assume ds BiosStorage ;Change current data segment
    mov cx,BiosStorage ;Find the system's current time
    mov ds,cx

    ; Check if time needs to be recalculated
    mov cx,RecalcCount
    inc cx
    cmp cx,273 ;Recalc every 273 counts = 15 seconds
    jb DONT_RECALC
    call calc ;Recalculate now
    mov cx,0 ;Prepare for resetting RecalcCount

DONT_RECALC
    mov RecalcCount,cx ;Store incremented or zero value

    ; Change extra segment to point to video memory
    assume es VidMem
    mov cx,VidMem
    mov es,cx

    mov dx,VideoPort
    mov di,DispPos ;Cursor location = destination
    lea si,ClkDisp ;Display in memory = source
    mov cx,10 ;Move 5 chars and 5 attrs
    cld ;No interrupts during critical section

    ; Wait until it is safe to write to video memory
    SCAN_LOW

; Loop until Video controller scan status is low
    in al,dx
    test al,1
    jnz SCAN_LOW

    mov ah,cs [si] ;Load ah with the byte to be written
    SCAN_HIGH
    ; After port has gone low, it must go high before it is
    ; safe to write directly to video memory
    in al,dx
    test al,1
    jz SCAN_HIGH

    mov es [di],ah ;Move the byte to video memory
    inc di ;One byte at a time
    inc si

    loop SCAN_LOW ;Repeat for next byte
    sti ;interrupts enabled after critical section

; Restore all registers used by this routine
    pop es
    pop ds
    pop si
    pop di
    pop cx
    pop ax

    iret ;Interrupt return
clock endp

;-----
calc proc near
; Time calculation routine
; Load ClkDisp variable with current time

; Save all registers used by this routine
    push ax
    push bx
    push cx
    push dx
    push ds

    assume ds BiosStorage ;Change current data segment
    mov cx,BiosStorage ;Find the system's current time
    mov ds,ax

; Get the high byte of timer, already in hours
    mov ax,TimerHi

CHECK12
    cmp ax,12
    jle VALID_TIME ;if hours <= 12 time is valid
    sub ax,12 ;else keep subtracting 12 until

VALID_TIME
    ; Convert ax (hours) to BCD format
    ; Convert ax into 2 displayable
    ; ASCII characters
    lea bx,ClkDisp
    mov cx,[bx],ah ;Move first hours digit into
    ; ClkDisp
    mov cx,[bx+2],al ;Move second digit
    mov ax,TimerLo ;Get low byte of timer in counts
    ; (Not minutes)
    mov cx,60 ;Multiply TimerLo by 60
    shr ax,cx
    ; Convert ax (minutes) to BCD
    ; format
    add ax,3030h ;Convert ax into 2 displayable
    ; ASCII characters
    mov cx,[bx+6],ah ;Move them to ClkDisp
    mov cx,[bx+8],al

; Restore all registers used by this routine
    pop ds
    pop bx
    pop cx
    pop dx
    pop ax

    ret
calc endp

;-----
loadclock proc near
; Initialization routine (Program's transient portion)

    assume ds Interrupts ;The Interrupt Vector Table
    mov ax,Interrupts ;will be used as data segment
    mov ds,ax
    cli ;Disable interrupts

    ; Get current Timer interrupt vector because
    ; it has to be called before our interrupt
    mov ax,TimerInt
    mov _OldTimerInt,ax
    mov _OldTimerInt[2],ax
    mov _OldTimerInt[3],ax

```

```

>>
, Set Clock routine as the new Timer interrupt routine
mov     TimerInt,offset clock
mov     TimerInt[2],cs
sti                                     ,Enable interrupts again

, Use video services to determine # of columns
mov     ah,15
int     10h
sub     ah,6      ,Subtract space for displaying the clock
shl     ah,1      Times 2 (chars and attrs)
mov     byte ptr Disppos,ah      ,Set ClkDisp position
                                     ,within video memory
mov     VideoPort,03BAh      ,Default = monochrome display
test    al,4        ,Is it really monochrome
jnz     GET_TIME      ,Yes
add     Disppos,8000h      ,No, set up for color display
mov     VideoPort,03DAh

GET_TIME
, Avoid displaying 00 00 for the first 15 seconds
call    calc

, Terminate and Stay Resident
, Do not keep transient portion resident
mov     dx,offset loadclock
int     27h
loadclock endp

C$eq     ends

end     ENTRY_POINT      ,Program starts at ENTRY_POINT

```

```

>>
System (no system)
Program BLANKER.ASM
Description A screen saver
MS-DOS v3.1 or later is required
This program was built with
Turbo Assembler v 2.01 and
Turbo Link v3.01
To build BLANKER.COM use the following
or equivalent commands
C>asm popclock.asm
C>link popclock.obj /t /x
Author José Luis Inigo Ortega
Revisions (01-DEC-92) Version 1.0
*****

Resident portion
*****
Cseg segment para public 'code'
assume cs Cseg
org 100h

ENTRY_POINT
jmp INIT

*****
Resident data
*****

Old interrupt vectors
-----
Old08Irr label dword ,Old INT 08h ISR
Old08ofs dw ?
Old08Seg dw ?
Old09Irr label dword ,Old INT 09h ISR
Old09ofs dw ?
Old09Seg dw ?
Old10Irr label dword ,Old INT 10h ISR
Old10ofs dw ?
Old10Seg dw ?
Old13Irr label dword ,Old INT 13h ISR
Old13ofs dw ?
Old13Seg dw ?
Old28Irr label dword ,Old INT 28h ISR
Old28ofs dw ?
Old28Seg dw ?
Old2Firr label dword ,Old INT 2Fh ISR
Old2Fofs dw ?
Old2FSeg dw ?

TSR status vars
-----
Disabled db 0 ,TSR disabled flag
TimeLimit dw 0 ,Max time before blanking
,1092 ticks = 1 minute
VideoActive db 0 ,Int 10h active flag
DiskActive db 0 ,Int 13h active flag
BlankerOn db 0 ,TSR active flag

DOS status flags
-----
InDosFlagOfs dw ? ,InDos Flag offset
InDosFlagSeg dw ? ,InDos Flag segment
CritErrFlagOfs dw ? ,Critical Error Flag offset

Interrupted program's state vars
-----
CursorPos dw ? ,Save cursor's position
CursorSz dw ? ,Save cursor's size
OldSP dw ? ,Save DOS SP register
OldSS dw ? ,Save DOS SS register
ScrBuf dw 2000 dup(?) ,Save screen buffer

Internal stack
-----
db 255 dup(??)
stkTop db (??) ,Top of internal stack

*****
New Timer ISR (INT 08h)
*****
new08Irr proc near
pushf
call _Old08Irr
,Call old ISR

Check TSR's status vars
cmp Disabled,0 ,If TSR disabled
jne EXIT_08 ,exit
cmp BlankerOn,1 ,If scr already blanked
je EXIT_08 ,exit
cmp TimeLimit,1092 ,If on time limit (1 minute)
je CHECK_08 ,blank screen
inc TimeLimit ,else increment counter
jmp short EXIT_08 ,and exit

CHECK_08:
, Check BIOS activity
cmp VideoActive,1 ,If video active
je EXIT_08 ,exit
cmp DiskActive,1 ,If disk active
je EXIT_08 ,exit

```

```

, Check DOS status flags
push di
push es

mov di,InDosFlagOfs ,Check InDos Flag
mov es,InDosFlagSeg
cmp byte ptr es[di],0
jne ABORT_BLNK ,Exit if DOS busy

mov di,CritErrFlagOfs ,Check Crit Err Flag
cmp byte ptr es[di],0
jne ABORT_BLNK ,Exit on error

, Everything OK, blank the screen
pop es
pop di
call scrblanker ,Blank the screen
jmp short EXIT_08

ABORT_BLNK
pop es
pop di

EXIT_08
iret
new08Irr endp

*****
New Hardware Keyboard ISR (INT 09h)
*****
new09Irr proc near
sti ,Enable interrupts
pushf
call _Old09Irr
,Call old ISR

, Reset pop counter and toggle Disabled flag if
F12 key was pressed

mov TimeLimit,0 ,Always reset pop counter

push ax
in al,60h ,If F12 not pressed
cmp al,58h
jne EXIT_09 ,exit
not Disabled ,else toggle Disabled flag

EXIT_09
pop ax
iret
new09Irr endp

, Return to DOS

*****
New BIOS Video ISR (INT 10h)
*****
new10Irr proc near
pushf
inc VideoActive ,Call old ISR
,Flag set while processing
call _Old10Irr
dec VideoActive ,Reset flag after process
iret
new10Irr endp

*****
New BIOS Disk ISR (INT 13h)
*****
new13Irr proc far
pushf
inc DiskActive ,Call old ISR
,Flag set while processing
call _Old13Irr
pushf
dec DiskActive ,Protect flags
,Reset flag after process
cld ,Restore flags
popf
ret 2
new13Irr endp

*****
New DOS Idle ISR (INT 28h)
*****
new28Irr proc near
pushf
call _Old28Irr
,Call old ISR

, Check TSR's status vars
cmp Disabled,0 ,If TSR disabled
jne EXIT_28 ,exit
cmp BlankerOn,1 ,If Screen already blanked
je EXIT_28 ,exit
cmp TimeLimit,1092 ,If not on time limit
jne EXIT_28 ,exit

, Check BIOS activity
cmp VideoActive,1 ,If video active
je EXIT_28 ,exit
cmp DiskActive,1 ,If disk active
je EXIT_28 ,exit

, Check DOS Critical Error Flag
push di
push es
mov es,InDosFlagSeg
mov di,CritErrFlagOfs
cmp byte ptr es[di],0

```

```

>>
pop     es
pop     di
jne     EXIT_28      ,Exit on error

, Everything OK, blank the screen
call    scrblanker

EXIT_28
iret

new2Fier endp

, =====
, New Multiplex ISR (INT 2Fh)
, =====
new2Fier proc far
cmp     ax,0DD00h
je      SET_INSTALLED_FLAG
jmp     _Old2Fier

SET_INSTALLED_FLAG
mov     ax,1992h      ,Set installed flag
iret     ,Return to transient portion
new2Fier endp

, =====
, Main routine
, =====
scrblanker proc near
mov     BlankerOn,1    ,Set TSR active flag
, -----
, Switch to internal stack
, -----
cli     ,Disable interrupts
mov     oldsp,sp        ,Save DOS SP register
mov     oldss,ss        ,Save DOS SS register
push    cs              ,Switch to internal stack
pop     ss
mov     sp,offset StkTop
sti     ,Enable interrupts
, -----
, Save system state
, -----
, Save all registers
push    ax
push    bx
push    cx
push    dx
push    si
push    di
push    de
push    es
push    bp

, Check screen mode
mov     ah,0Fh          ,Get display mode
int     10h             ,Returns BH=video page
cmp     al,3
jbe     SAVE_CURSOR
cmp     al,7
je      SAVE_CURSOR
jmp     EXIT_SCRBLANKER ,Exit if not in text mode

SAVE_CURSOR
, Save cursor state
mov     ah,03h          ,Get cursor state
int     10h
mov     CursorPos,dx     ,Save position
mov     CursorSz,cx      ,Save size
, -----
, BLANKER
, -----
, Hide cursor
mov     ah,01h
mov     cx,1000h
int     10h

, ----- Save screen -----
mov     ah,02h          ,gotoxy(0,0)
xor     dx,dx           ,x=0,y=0
int     10h
push    cs              ,Point to start of ScrBuf
pop     es
mov     di,offset ScrBuf
mov     cx,25           ,Save 25 rows

SAVE_SCR_1
push    cx              ,Save outer loop counter
mov     cx,80           ,Save 80 columns

SAVE_SCR_2
cld
mov     ah,09h          ,Go forward
int     10h             ,Read char and attr at cursor
stosw    ,Save them in ScrBuf

inc     di              ,gotoxy(x+1,y)
mov     ah,02h
int     10h
loop    SAVE_SCR_2      ,Save next pair

xor     di,di           ,gotoxy(0,y+1)
inc     dh
mov     ah,02h
int     10h

pop     cx              ,Restore outer loop counter
loop    SAVE_SCR_1      ,Save next row

, Clear entire screen
push    bx
mov     ax,0600h
mov     bh,07h
xor     cx,cx
mov     di,79
mov     dh,24
int     10h
pop     bx              ,Restore video page

, ----- Ping Pong -----
xor     dx,dx           ,x=0,y=0

PP_WRITE
push    cx              ,Save inc/dec flag
mov     ah,02h
int     10h
mov     ax,09DCh        ,Display '_' White on Black
mov     bl,07h
mov     cx,1
int     10h
mov     cx,100          ,Set delay counter
mov     ah,02h

PP_DELAY
int     10h
loop    PP_DELAY

, Delete '_'
mov     ax,09DCh
mov     bl,00h
mov     cx,1
int     10h
pop     cx              ,Restore inc/dec flag

, Calculate ball's new position
cmp     dl,79
jne     PP_SET_CL
xor     cl,cl
jmp     short PP_RESET_CH

PP_SET_CL
cmp     dl,0
jne     PP_RESET_CH
mov     cl,1

PP_RESET_CH
cmp     dh,24
jne     PP_SET_CH
xor     ch,ch
jmp     short PP_INC_DL

PP_SET_CH
cmp     dh,0
jne     PP_INC_DL
mov     ch,1

PP_INC_DL
cmp     cl,1
jne     PP_DEC_DL
inc     dl
jmp     short PP_INC_DH

PP_DEC_DL
dec     dl

PP_INC_DH
cmp     ch,1
jne     PP_DEC_DH
inc     dh
jmp     short PP_CHECK_KBD

PP_DEC_DH
dec     dh

PP_CHECK_KBD
mov     ah,11h
int     16h
jz      PP_WRITE        ,If not keypressed
, Keep bouncing

, ----- Restore screen -----
mov     ah,10h
int     16h            ,Flush keyboard

mov     ah,02h
xor     dx,dx           ,gotoxy(0,0)
int     10h           ,x=0,y=0
push    cs              ,Point to start of ScrBuf
pop     ds
mov     si,offset ScrBuf
mov     cx,25           ,Restore 25 rows

RESTORE_SCR_1
push    cx              ,Save loop counter
mov     cx,80           ,Restore 80 columns

RESTORE_SCR_2
cld
lodaw   ,Get char and attr
mov     bl,ah           ,Attr byte
mov     ah,09h          ,Write char/attr at cursor
push    cx              ,Save inner loop counter
mov     cx,1            ,Write them just once
int     10h

```

```

>>
inc     dl                      ,gotoxy(x+1,y)
mov     ah,02h
int     10h
pop     cx                      ,Restore inner loop counter
loop    RESTORE_SCR_2          ,Restore next pair

xor     dl,dl                   ,gotoxy(0,y+1)
inc     dh
mov     ah,02h
int     10h
pop     cx                      ,Restore outer loop counter
loop    RESTORE_SCR_1          ,Restore next row

; Restore system state
; Restore cursor state
mov     ah,01h                  ,Restore cursor size
mov     cx,CursorSz
int     10h
mov     ah,02h                  ,Restore cursor position
mov     dx,CursorPos
int     10h

EXIT_SCRBLANKER.
; Restore all registers
pop     bp
pop     es
pop     ds
pop     di
pop     si
pop     dx
pop     cx
pop     bx
pop     ax

; Restore external stack
cli                      ,Disable interrupts
mov     es,OldSS           ,Restore SS register
mov     sp,OldSP           ,Restore SP register
sti                      ,Enable interrupts

mov     BlenkerOn,0        ,Reset TSR active flag
ret

scrblanker  endp

; =====
; End of TSR's resident portion
; =====
ResPortionEnd db '$'

; =====
; Transient portion
; =====
; Transient data
Mag1 db 'Blanker v1.0 Copyright (c) 1992 '
      db 'J L Higgins'
Mag2 db 'INSTALLED' [F12=Toggle On/Off]$

Err1 db 'Incompatible BIOS revision$'
Err2 db 'DOS 3.1 or later required$'
Err3 db 'Already installed$'

; =====
; Initialization routine
; =====
initialize proc near
assume  ds CSeg

INIT
mov     dx,offset Mag1        ,Display copyright notice
call    putmsg

; Check
; - BIOS date If model=FCh and submodel=01 or 02, OK
; - DOS version If DOS >= 3.1, OK
; - If TSR already installed

; Check BIOS
mov     dx,offset Err1
mov     ah,0C0h
int     15h
cmp     ah,0                  ,If INT 15h not valid
jne     ABORT_INSTALL         ,exit
cmp     byte ptr es [bx+2],0FCh ,Chk machine model
jne     ABORT_INSTALL
cmp     byte ptr es [bx+3],1    ,Chk machine submdl
jb      ABORT_INSTALL

; Check DOS version
mov     dx,offset Err2
mov     ah,10h                ,Get MaDos version
int     21h
xchg    al,ah
cmp     ax,30Ah               ,If DOS >= 3.1, continue
jb      ABORT_INSTALL

; Check TSR already installed
mov     dx,offset Err3
mov     ax,0DD00h             ,If ID=DD already installed
int     2Fh                  ,abort installation
cmp     ax,1992h
jne     CONTINUE_INIT

ABORT_INSTALL.
call    putmsg
mov     ax,4C01h              ,Exit with return code=1
int     21h                  ,Return to MaDos

CONTINUE_INIT.
; Locate and save DOS status flags

InDos Flag
mov     ah,34h
int     21h
mov     indosFlagOfs,bx        ,Save offset
mov     indosFlagSeg,es        ,Save segment

; Critical Error Flag
dec     bx
mov     CritErrFlagOfs,bx      ,Crit Err Flag=byte before
                                ,InDos Flag

; Insert new ISRs into interrupt chains

mov     ax,3508h              ,Save old Timer ISR
int     21h
mov     Old08Ofs,bx
mov     Old08Seg,es
mov     ax,2508h              ,Set new Timer ISR
mov     dx,offset new08isr
int     21h

mov     ax,3509h              ,Save old Kbd ISR
int     21h
mov     Old09Ofs,bx
mov     Old09Seg,es
mov     ax,2509h              ,Set new Kbd ISR
mov     dx,offset new09isr
int     21h

mov     ax,3510h              ,Save old BIOS Video ISR
int     21h
mov     Old10Ofs,bx
mov     Old10Seg,es
mov     ax,2510h              ,Set new BIOS Video ISR
mov     dx,offset new10isr
int     21h

mov     ax,3513h              ,Save old BIOS Disk ISR
int     21h
mov     Old13Ofs,bx
mov     Old13Seg,es
mov     ax,2513h              ,Set new BIOS Disk ISR
mov     dx,offset new13isr
int     21h

mov     ax,3520h              ,Save old DOS Idle ISR
int     21h
mov     Old20Ofs,bx
mov     Old20Seg,es
mov     ax,2520h              ,Set new DOS Idle ISR
mov     dx,offset new20isr
int     21h

mov     ax,352Fh              ,Save old Multiplex ISR
int     21h
mov     Old2FOfs,bx
mov     Old2FSeg,es
mov     ax,252Fh              ,Set new Multiplex ISR
mov     dx,offset new2Fisr
int     21h

; Install resident portion in memory

; Display installed message.
mov     dx,offset Mag2
call    putmsg

; DX = amount of memory to leave resident
mov     dx,(offset ResPortionEnd - offset CSeg + 15)
mov     cl,4
shr     dx,cl                  ,Convert to paragraphs
mov     ax,3100h              ,Go TSR
int     21h

initialize endp

; =====
; Display a message
; Entry DX - offset Message
; =====
NL db '13.10.$'

putmsg  proc near
mov     ah,09h
int     21h
mov     dx,offset NL          ,New Line
mov     ah,09h
int     21h
ret
putmsg  endp

CSeg  ends
end     ENTRY_POINT

```

```

>>
*****
System      : Thesis Demo
Program     : POPCLOCK.ASM
Description : A memory resident program that provides
              a pop up clock MS-DOS v2.0 or later is
              required.
              This program was built with
              Turbo Assembler v2.01 and
              Turbo Link v1.01.
              To build POPCLOCK.COM use the following
              or equivalent commands:
              C>task popclock.asm
              C>link popclock.obj /t /x
Author      : Steven Simin ('MSDOS Bible')
Modified by : José Luis Ifigo Ortega
Revisions   : (15-APR-92) Version 1.0
              (01-DEC-92) Version 1.1
              Hot Key modified to avoid
              conflicts with Turbo Pascal
              IDE keyboard assignments
*****

Cseg segment para public 'code'
assume cs,Cseg
org 100h                ; Required for COM programs

BEGIN
    jmp     INIT

;-----
; Variable declaration section
;-----
old9Hndlr    label    dword    old int 9h handler
old9Ofs      dw       ?
old9Seg      dw       ?
old9Hndlr    label    dword    ,Old int 9h handler:
old9Ofs      dw       ?
old9Seg      dw       ?
old10Hndlr   label    dword    ,Old int 10h handler
old10Ofs     dw       ?
old10Seg     dw       ?
old13Hndlr   label    dword    ,Old int 13h handler
old13Ofs     dw       ?
old13Seg     dw       ?
old28Hndlr   label    dword    ,Old int 28h handler
old28Ofs     dw       ?
old28Seg     dw       ?
;-----
HotKey       db       0        ,If HotKey pressed > 0
VideoFlag    db       0        ,Int 10h flag
DiskFlag     db       0        ,Int 13h flag
RunningFlag   db       0        ,If program running = 1
;-----
InDosOfs     dw       ?        ,Offset of InDos flag
InDosSeg     dw       ?        ,Segment of InDos flag
ErrFlagOfs   dw       ?        ,Offset of critical error flag
;-----
CurPos      dw       ?        ,Save current cursor's position
CurSz       dw       ?        ,Save current cursor's size
DosSP        dw       ?        ,Store MaDos stack pointer
DosSS        dw       ?        ,Store MaDos SS register
ScrBuf       dw       174 dup(?) ,Save screen buffer
;-----
stkTop       db       255 dup(' ') ,Local stack
              db       (' ')      ,Top of local stack
;-----
LoadMsg      db       * PopUp-Clock installed*,0dh,0ah,07h
              db       * Ctrl+Left_Shift to activate *$
;-----
BrkMsg       db       *Any key to continue*
TimeMsg      db       *Current time is *
Hour10       db       ?        ,String for storing time
Hour         db       ?
Min10        db       ?
Min          db       ?
Sec10        db       ?
Sec          db       ?
;-----
Dos1Msg      db       *DOS 2.x or later required*,0dh,0ah,*$

;-----
; New handler for int 8h (timer)
;-----
new8_hndlr   proc near
    pushf
    call    old8Hndlr
    ; Simulate int
    ; Chain to old handler

    ; Check program's HotKey
    cmp     HotKey,0
    je      HKEY_0
    ; HotKey pressed ?
    ; If not, exit

    ; Check peripherals
    cmp     VideoFlag,0
    jne     DECHKKEY
    ; Int 10h busy ?
    ; If yes, dec HotKey flag
    cmp     DiskFlag,0
    jne     DECHKKEY
    ; Int 13h busy ?
    ; If yes, dec HotKey flag

    push    di
    push    es
    ; Save registers

    ; Check value for InDos flag
    mov     di,InDosOfs
    ; Offset of flag
    mov     es,InDosSeg
    ; Segment of flag
    cmp     byte ptr es,[di],0
    jne     POPSTK
    ; Exit if DOS busy

    ; Check critical error flag
    mov     di,ErrFlagOfs
    ; Offset of flag
    cmp     byte ptr es,[di],0
    jne     POPSTK
    ; Exit if flag set

    ; Everything OK, pop up the clock
    pop     es
    ; Restore registers
    pop     di
    mov     HotKey,0
    ; Clear HotKey flag
    call    PopUp
    ; Run clock routine

HKEY_0
    iret

POPSTK
    pop     es
    pop     di

DECHKKEY
    dec     HotKey
    ; Return to MaDos
    iret

new8_hndlr   endp

;-----
; New handler for int 9h (keyboard hardware interrupt)
;-----
new9_hndlr   proc near
    sti
    ; Enable interrupts
    pushf
    ; Simulate int
    call    old9Hndlr

    ; Check keyboard for HotKey pressed
    push    ax
    ; Save ax
    mov     ah,02h
    ; Get toggle keys' status
    int     16h
    ; Call BIOS keyboard routine
    and     al,0Fh
    ; Compare toggle keys' status with HotKey
    cmp     al,06h
    ; Ctrl+LeftShift pressed ?
    pop     ax
    ; Restore ax
    jne     EXIT_09
    ; If HotKey not pressed, exit

    ; HotKey was pressed
    cmp     RunningFlag,0
    ; Program already running ?
    jne     EXIT_09
    ; If yes, exit

    ; Initialize HotKey counter
    mov     HotKey,18
    ; HotKey is now active

EXIT_09
    iret

new9_hndlr   endp

;-----
; New handler for int 10h (ROM BIOS video service)
;-----
new10_hndlr  proc near
    pushf
    ; Simulate int
    inc     VideoFlag
    call    old10Hndlr
    dec     VideoFlag
    iret

new10_hndlr  endp

;-----
; New handler for int 13h (ROM BIOS disk service)
;-----
new13_hndlr  proc far
    pushf
    ; Simulate int
    inc     DiskFlag
    call    old13Hndlr
    ; Protect flags
    pushf
    dec     DiskFlag
    ; Restore flags
    popf
    ret     2
    ; Return to MaDos, discard 2 bytes

new13_hndlr  endp

;-----
; New handler for int 28h (DOS scheduler)
;-----
new28_hndlr  proc near
    pushf
    ; Simulate int
    call    old28Hndlr
    ; Chain to old handler

    ; Check program's HotKey
    cmp     HotKey,0
    je      EXIT_28
    ; HotKey pressed ?
    ; If not, exit

    ; Check peripherals
    cmp     VideoFlag,0
    ; Int 10h busy ?
    jne     EXIT_28
    ; If yes, exit
    cmp     DiskFlag,0
    ; Int 13h busy ?
    jne     EXIT_28
    ; If yes, exit

    push    di
    push    es
    ; Save registers

    ; Check critical error flag
    mov     es,InDosSeg
    ; Offset of flag
    mov     di,ErrFlagOfs
    ; Segment of flag
    cmp     byte ptr es,[di],0
    jne     POPSTK
    ; Exit if DOS busy

    ; Check critical error flag
    mov     di,ErrFlagOfs
    ; Offset of flag
    cmp     byte ptr es,[di],0
    jne     POPSTK
    ; Exit if flag set

    ; Everything OK, pop up the clock
    pop     es
    ; Restore registers
    pop     di
    mov     HotKey,0
    ; Clear HotKey flag
    call    PopUp
    ; Run clock routine

EXIT_28
    iret

new28_hndlr  endp

```

```

>>
cmp     byte ptr es:[di],0
pop     es
pop     di
jne     EXIT_28

mov     HotKey,0
call    pop_up
; Restore registers
; Clear HotKey flag
; Run clock routine

EXIT_28:
iret
new28_hndlr endp

; =====
; Clock routine -- Main portion of PopUp-Clock
; =====
pop_up proc near
    mov     RunningFlag,1
    ; Set Running flag to true

    ; Set up local stack and save DOS registers
    cld
    mov     D0asp,sp
    mov     D0asp,ss
    push    cs
    pop     ss
    mov     sp,offset StkTop
    sti
    ; Enable interrupts

    ; Save MS-DOS registers on local stack
    push    ax
    push    bx
    push    cx
    push    dx
    push    si
    push    di
    push    ds
    push    es
    push    bp

    ; Check display mode, exit if in graphics mode
    mov     ah,0fh
    int     10h
    cmp     al,3
    jbe     GETCURSOR
    cmp     al,7
    jbe     GETCURSOR
    je      GETCURSOR

    ; In graphics mode, so exit
EXITPOPIT:
    pop     bp
    pop     es
    pop     ds
    pop     di
    pop     si
    pop     cx
    pop     bx
    pop     ax

    ; Disable interrupts
    mov     ss,D0ssp
    mov     sp,D0asp
    sti
    mov     RunningFlag,0
    ret

    ; Display in text mode, so continue
GETCURSOR:
    mov     ah,03h
    int     10h
    mov     CurPos,dx
    mov     CurSz,cx

    ; Save underlying contents of clock's window
    mov     ah,02h
    mov     dl,17
    mov     dh,6
    int     10h
    push    cs
    pop     es
    mov     di,offset ScrBuf
    mov     cx,6

    ; Make es local
    ; Start of screen buffer
    ; Save 6 rows

SAVELOOP_1:
    push    cx
    mov     cx,29
    ; Save loop counter
    ; Save 29 columns

SAVELOOP_2:
    cld
    mov     ah,08h
    int     10h
    stosw
    ; Read char and attr at cursor
    ; Store in buffer

    inc     di
    mov     ah,02h
    int     10h
    loop    SAVELOOP_2
    ; Move cursor to next column
    ; Cursor positioning service
    ; Save next char and attr

    mov     di,17
    inc     dh
    mov     ah,02h
    int     10h
    loop    SAVELOOP_1
    ; Move cursor to starting column
    ; of next row
    ; Cursor positioning service

    ; Make clock's window border, window(17,6,45,10)
    push    bx
    mov     ax,0700h
    ; Save page number
    ; Blank a window

    mov     bh,70h
    mov     cl,17
    mov     ch,6
    mov     dl,45
    mov     dh,10
    int     10h
    ; Black on white
    ; X1 coord
    ; Y1 coord
    ; X2 coord
    ; Y2 coord

    ; Clear window's interior, window(18,7,44,9)
    mov     ax,0700h
    mov     bh,07h
    mov     cl,18
    mov     ch,7
    mov     dl,44
    mov     dh,9
    int     10h
    ; Blank a window
    ; Yellow on red
    ; X1 coord
    ; Y1 coord
    ; X2 coord
    ; Y2 coord

    ; Display window contents
    pop     bx
    mov     ah,02h
    mov     dl,21
    mov     dh,10
    int     10h
    ; Restore page number
    ; Cursor positioning service
    ; X coord gotoxy(21,
    ; Y coord 10)

    mov     ah,01h
    mov     cx,1000h
    int     10h
    ; Set cursor size service
    ; Turn cursor off

    push    cs
    pop     ds
    mov     si,offset BrkMsg
    mov     cx,19
    cld
    ; Make ds local
    ; Set the quit prompt
    ; Display 19 chars
    ; Forward direction

WINLOOP_1:
    lodsb
    mov     ah,0Ah
    push    cx
    mov     cx,1
    int     10h
    ; Byte to al
    ; Write char at cursor service
    ; Save loop counter
    ; Output char just one time

    pop     cx
    inc     dl
    mov     ah,02h
    int     10h
    loop    WINLOOP_1
    ; Restore loop counter
    ; Increment X coord (column)
    ; Cursor positioning service
    ; Display another char

    mov     ah,02h
    mov     dl,19
    mov     dh,8
    int     10h
    mov     cx,16
    ; Cursor positioning service
    ; X coord gotoxy(19,
    ; Y coord 8)
    ; Display 16 chars 'Current time is'

WINLOOP_2:
    lodsb
    mov     ah,0Ah
    push    cx
    mov     cx,1
    int     10h
    ; Byte to al
    ; Write char at cursor service
    ; Save loop counter
    ; Output char just one time

    pop     cx
    inc     dl
    mov     ah,02h
    int     10h
    loop    WINLOOP_2
    ; Restore loop counter
    ; Increment X coord (column)
    ; Cursor positioning service
    ; Display another char

    ; Display time until key pressed
TIMELoop_1:
    call    get_time
    mov     ah,02h
    mov     dl,35
    mov     dh,8
    int     10h
    ; Get current time
    ; Cursor positioning service
    ; X coord gotoxy(35,
    ; Y coord 8)

    mov     si,offset Hour10
    mov     cx,8
    ; Start of time string
    ; 8 chars to display

TIMELoop_2:
    lodsb
    mov     ah,0Ah
    push    cx
    mov     cx,1
    int     10h
    ; Byte to al
    ; Write char at cursor service
    ; Save loop counter
    ; Display char just one time

    pop     cx
    inc     dl
    mov     ah,02h
    int     10h
    loop    TIMELoop_2
    ; Restore loop counter
    ; Increment X coord
    ; Cursor positioning service
    ; Display time string's 8 chars

    mov     ah,01h
    int     16h
    jz      TIMELoop_1
    ; Check keyboard input status
    ; ROM BIOS keyboard service
    ; Loop if no key was pressed

    mov     ah,00h
    int     16h
    ; Read keyboard service
    ; Empty the keyboard's buffer

    ; Restore underlying screen and exit
    mov     ah,02h
    mov     dl,17
    mov     dh,6
    int     10h
    ; Set cursor position
    ; Left column of clock's window (X1)
    ; Upper row of clock's window (Y1)

    mov     si,offset ScrBuf
    mov     cx,6
    ; Start of screen buffer
    ; Restore 6 rows

RESTORELOOP_1:
    push    cx
    ; Save loop counter

```

```

>> mov     cx,29             ,Restore 29 columns
RESTORELOOP_2
  cld                     ,Clear direction flag
  lodsw                  ,Get char and attr
  mov     bl,ah            ,Attr byte
  mov     ah,09h           ,Write char and attr at cursor
  push    cx               ,Save inner loop counter
  mov     cx,1             ,Write just one char and attr
  int     10h              ,Write just one char and attr
  pop     cx               ,Restore inner loop counter

  inc     di               ,Move cursor to next column
  mov     ah,02h           ,Cursor positioning service
  int     10h              ,Cursor positioning service
  loop    RESTORELOOP_2    ,Restore next char and attr

  mov     di,17            ,Move cursor to start
  inc     di               ,Of next row
  mov     ah,02h           ,Cursor positioning service
  int     10h              ,Cursor positioning service
  pop     cx               ,Restore outer loop counter
  loop    RESTORELOOP_1

, Restore cursor to its size and position prior to call
  mov     ah,01h           ,Set cursor size service
  mov     cx,CurSz
  int     10h              ,Set cursor size service

  mov     ah,02h           ,Cursor positioning service
  mov     dx,CurPos
  int     10h              ,Cursor positioning service

  jmp     EXITPOPIT        ,Go to exit routine
pop_up  endp

, =====
, Get time routine (used by pop_up)
, =====
get_time  proc near
  mov     ah,2ch           ,Get system time function
  int     21h              ,DOS service calls

, Hours returned in ch, minutes in cl and seconds in dh
, Convert this to ascii values and store them in the
, time string

  mov     bl,10            ,Divisor for div function

  xor     ah,ah            ,Convert hours to ascii
  mov     al,ch
  div     bl
  or      ax,3030h
  mov     Hour10,al
  mov     Hour,ah

  xor     ah,ah            ,Convert minutes to ascii
  mov     al,cl
  div     bl
  or      ax,3030h
  mov     Min10,al
  mov     Min,ah

  xor     ah,ah            ,Convert seconds to ascii
  mov     al,dh
  div     bl
  or      ax,3030h
  mov     Sec10,al
  mov     Sec,ah

  ret

get_time  endp

, This is the last byte that is going to remain resident
, in memory
LastByte  db    '*'

, =====
, Initialize routine (inita PopUp-Clock)
, =====
initialise  proc near
  assume ds,CSeg           ,All vars in code segment

INIT

, Locate InDos Flag
  mov     ah,34h
  int     21h
  mov     InDosOfs,bx       ,Offset address of flag
  mov     InDosSeg,es       ,Segment of address of flag

, Locate critical error flag's address
, (Not available in DOS version 1.x)
  mov     ah,30h           ,Get MeDos version
  int     21h
  cmp     al,2
  jg      USE_5D            ,Function 5Dh implemented, so use it
  jg      CALC              ,5Dh not available, so calc address

, Running DOS 1.x, so exit
  mov     dx,offset DosMsg  ,Start error message
  mov     ah,09h           ,Print string function
  int     21h              ,DOS service calls
  int     20h              ,Terminate program and return to MeDos

, Running under DOS 2.x, so calculate error flag's address
CALC
  mov     si,bx            ,bx has InDos flag
  inc     si
  jmp     SAVEIT

, Running under DOS 3.x, so locate error flag using
, function 5Dh
USE_5D
  mov     ah,5Dh           ,MeDos error function
  mov     al,6             ,Return flag address sub service
  int     21h

, Save critical error flag's address
SAVEIT
  push    cs               ,Reset ds
  pop     ds
  mov     ErrFlagOfs,si

, Insert new handlers into interrupt chain
, Insert new int 08h handler
  mov     ax,3508h         ,Get int 08h
  int     21h
  mov     Old8Ofs,bx       ,Save int 08h
  mov     Old8Seg,es

  mov     ax,2508h         ,Set int 08h

, Assign new address to int 08h
  mov     dx,offset new8_hndlr
  int     21h

, Insert new int 09h handler
  mov     ax,3509h         ,Get int 09h
  int     21h
  mov     Old9Ofs,bx       ,Save int 09h
  mov     Old9Seg,es

  mov     ax,2509h         ,Set int 09h

, Assign int 09h its new address
  mov     dx,offset new9_hndlr
  int     21h

, Insert new int 10h handler
  mov     ax,3510h         ,Get int 10h
  int     21h
  mov     Old10Ofs,bx      ,Save int 10h
  mov     Old10Seg,es

  mov     ax,2510h         ,Set int 10h

, Assign int 10h its new address
  mov     dx,offset new10_hndlr
  int     21h

, Insert new int 13h handler
  mov     ax,3513h         ,Get int 13h
  int     21h
  mov     Old13Ofs,bx      ,Save int 13h
  mov     Old13Seg,es

  mov     ax,2513h         ,Set int 13h

, Assign int 13h its new address
  mov     dx,offset new13_hndlr
  int     21h

, Insert new int 28h handler
  mov     ax,3528h         ,Get int 28h
  int     21h
  mov     Old28Ofs,bx      ,Save int 28h
  mov     Old28Seg,es

  mov     ax,2528h         ,Set int 28h

, Assign int 28h its new address
  mov     dx,offset new28_hndlr
  int     21h

, Display installed message,
, then terminate but stay resident
  mov     dx,offset LoadMsg ,Start of LoadMsg
  mov     ah,09h           ,Print string function
  int     21h              ,DOS service calls

, Calculate amount of memory to retain in dx for the TSR
  mov     dx,(offset LastByte - offset CSeg + 15)
  mov     cl,4
  shr     dx,cl            ,Convert to paragraphs
  mov     ax,3100h         ,TSR function
  int     21h              ,DOS service calls

initialize  endp

CSeg  ends

end  BEGIN

```

```

>>
{*****
* System      Thesis Demo
* Program     JTSRTP.PAS
* Description  This unit gives basic support for TSR
*             programs
*             Reference: Norton Guides -
*             Assembly Language Database
*             This unit was compiled with
*             Turbo Pascal v6.0
* Author      Stephen K O'Brien  ('Turbo Pascal,
*             Advanced Programmers Guide')
* Modified by  José Luis (Rigo Ortega
* Revisions   (18-APR-92) Version a0.1
*             (04-MAY-92) Simplified interface
*             Version a0.5
*             (01-DEC-92) Version 1.0.
*****}

unit JTSRTP.

{$D-,V-,I-,R-,F+,B-}      { DO NOT CHANGE }

{*****}
interface

uses crt, dos,

type
  TProc = procedure,

{----- Public routines -----}
function ok2instl (Count:word,
                  ArgName:string) boolean,

procedure installtr (_PopRtn:TProc;
                    HotKey:word, HKMsg:string),

{*****}
implementation

{----- Constants -----}
const
  NL = 'M^J',

{ Interrupt number definitions }
  TIMERINT = $1C,
  KBDINT = $09,
  IDLEINT = $28,
  CRITINT = $24;

{----- Initialized variables -----}
{ AVOID defining local variables inside an interrupt
{ routine, because they would be allocated inside the
{ interrupted program's stack

const
{--- DO NOT MODIFY THE CONTENTS OF THIS VARIABLES ---}
{ SS and SP saved areas of the interrupted program }
  CurStackSS . word = 0,
  CurStackSP . word = 0,

{ SS and SP saved areas of the TSR program }
  TsrStackSS . word = 0,
  TsrStackSP . word = 0,

{ Stack switch indicates whether the INT was issued }
{ from inside or outside the TSR program }
{ -1 = Outside this TSR, switch the stack }
{ 0 = Inside this TSR, don't switch the stack }
{ more than once }
  StackSW . integer = -1,

{-----}
const
{ Control vars }
  TsrName . string[15] = '', { TSR Signature }
  EndDosMem . word = 0,
  PopOk . boolean = FALSE, { TRUE = Ok to pop up }
  Running . boolean = FALSE, { TRUE = pop routine
                              currently active }

{ Hot-Key combination to activate the TSR }
  HScanCode . byte = 0,
  HShiftStatus . byte = 0,

{----- Variables -----}
var
{ Original BIOS and DOS info }
  _OldTimerVec, { Save original ISRs' vectors }
  _OldKbdVec,
  _OldIdleVec,
  _OldCritVec . TProc,
  DosSeg, { DOS Segment }
  InDosFlag, { Save DOS busy flag }
  CritErrFlag . word, { Save DOS Critical Error flag }

```

```

{ Interrupted program's info }
  CurDTASeg, { Save interrupted program's DTA }
  CurDTAOfs,
  CurPSP . word, { Interrupted program's PSP }
  CurBreak . byte, { ^Break Flag }

{ TSR's info }
  TsrDTASeg, { Address of the TSR's DTA }
  TsrDTAOfs,
  TsrPSP . word, { TSR's Program Segment Prefix }
  TsrSwitch, { Tsr, var variable (can't be local) }
  TsrResetKbd, { Used for resetting the keyboard
               before pop }
  TsrVec . byte, { TSR's communication interrupt }
  _Poproutine . TProc, { Address of pop up routine }
  HotKeyMsg . string, { TSR's hotkey message }

```

```

{ Misc vars }
  PSPArray . array [1..2] of word,
  PSPCounter . byte;

```

```

{----- Private routines -----}
procedure popup, forward,

```

```

{--- Inline macros ---}
procedure beginint,
{ Swap interrupted program's stack info and }
{ TSR stack info }
  inline ( $PF/$06/StackSW/ { inc StackSW }
           $75/$10/ { jne +16 }
           $8C/$16/CurStackSS/ { mov CurStackSS,SS }
           $89/$26/CurStackSP/ { mov CurStackSP,SP }
           $8E/$16/TsrStackSS/ { mov SS,TsrStackSS }
           $8B/$26/TsrStackSP); { mov SP,TsrStackSP }

```

```

procedure endint,
{ Restore interrupted program's stack info }
  inline ( $PF/$06/StackSW/ { dec StackSW }
           $7D/$08/ { jge +8 }
           $8E/$16/CurStackSS/ { mov SS,CurStackSS }
           $8B/$26/CurStackSP); { mov SP,CurStackSP }

```

```

procedure cli,
{ Disable interrupts (for critical sections) }
  inline ( $FA );

```

```

procedure sti,
{ Enable interrupts }
  inline ( $FB );
{-----}

```

```

procedure calloldint (_Isr:TProc),
{ Call an old interrupt service routine }
begin
  asm pushf end, { Simulate INT }
  _Isr,
end, { calloldint }

```

```

{--- JTSRTP ISRs (Interrupt Service Routines) ---}
procedure newtimer (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
                   ES,BP:word),
  interrupt,
{ New timer interrupt routine }
{ If DOS is not busy and pop up is pending then pop up }
begin
  cli,
  beginint,
  sti,

  calloldint(_OldTimerVec), { Call old timer interrupt }
  if PopOk and (mem[DosSeg InDosFlag] = 0) and
  (mem[DosSeg CritErrFlag] = 0) then
  begin
    Port[$20] := $20,
    PopOk := FALSE,
    popup,
  end,

  cli,
  endint,
  sti,
end, { newtimer }

```

```

procedure newkbd (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
                 ES,BP:word),
  interrupt,
{ New keyboard interrupt routine If hot key is pressed }
{ set PopOk so newidle and newtimer can detect it }
begin
  cli,
  beginint,
  sti,

  { Check the keyboard for the correct scancode and shift }
  { status }
  if (Port[$60] = HScanCode) and
  (mem[0000:$0417] and HShiftStatus) = HShiftStatus)
  then

```

```

>>
begin
[ Reset the keyboard ]
  TarResetKbd := Port[$61],
  Port[$61] := TarResetKbd or $80,
  Port[$61] := TarResetKbd,
[ Signal end of interrupt ]
  cli,
  Port[$20] := $20, ( End of interrupt )
  sti,
[ If TSR is not active set pop up flag to true ]
  if not Running then PopOk := TRUE,
  end
else
[ Call old keyboard ISR to handle real work ]
  calloldint(@OldKbdVec),

  cli,
  endint,
  sti,
end, { newkbd }

procedure newidle (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
  ES,BP word);
begin
  interrupt,
[ If DOS is idling and if pop up is pending then pop up ]
begin
  cli,
  beginint,
  sti,

  calloldint(@OldIdleVec), ( Call old interrupt 20 ISR )
  if PopOk and (mem[DosSeg InDosFlag] <> 0) and
    (mem[DosSeg CritErrFlag] = 0) then
  begin
    PopOk := FALSE,
    popup,
    end,

    cli,
    endint,
    sti,
  end, { newidle }

procedure newcrit (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
  ES,BP word);
begin
  interrupt,
[ Set AX = 0 to tell DOS to ignore critical errors ]
begin
  AX := 0,
end, { newcrit }
{-----}

procedure reltsmem,
[ Release TSR's memory before unloading it ]
var
  Reg registers,
begin
  while mem[EndDosMem 0] = $4D do
  begin
[ Deallocate memory if it belongs to the TSR ]
    if memw[EndDosMem 1] = TarPSP then
    begin
      Reg ah := $49,
      Reg es := EndDosMem + 1,
      mados(Reg),
      end,

[ Get next MCB ]
      EndDosMem := EndDosMem + memw[EndDosMem 3] + 1,
      end,
    end, { reltsmem }

function uninstallter : boolean,
[ Remove the program from memory if possible ]
var
  T,K,I pointer,
begin
[ Get current values of the interrupts the TSR ]
[ has captured ]
  getintvec(TIMERINT, TI),
  getintvec(KBDINT, KI),
  getintvec(IDLEINT, I),

[ If the current values in the interrupt vectors point ]
[ to the TSR's routines then no other program have ]
[ stolen the TSR's interrupts and the TSR can be ]
[ unloaded safely ]

  if (T = @newtimer) and (K = @newkbd) and
    (I = @newidle) then
  begin
[ Restore original interrupt handlers' vectors ]
    setintvec(TIMERINT, @OldTimerVec),
    setintvec(KBDINT, @OldKbdVec),
    setintvec(IDLEINT, @OldIdleVec),
    setintvec(TarVec, NIL),
    reltsmem, ( Deallocate all memory )
    uninstallter := TRUE,
  end
  else
    uninstallter := FALSE,
  end, { uninstallter }

procedure tar_isr (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
  ES,BP word);
begin
  interrupt;
[ TSR communication interrupt ]
[ Uninstall the TSR if possible ]
begin
[ Save TarSwitch before executing the interrupt ]
  TarSwitch := AX,

  cli,
  beginint,
  sti,

  case TarSwitch of
    1 if uninstallter then
      writein('UNINSTALLED');
    else
      writein('UNABLE TO UNINSTALL');

  else
    writein('ALREADY INSTALLED',NL,HotKeyMsg);
  end, { case }

  cli,
  endint,
  sti,
end, { tar_isr }

procedure beforepop,
var
  I byte,
  Reg registers,
begin
  Running := TRUE,

[ Get the DTA address of the interrupted program ]
  Reg ah := $2F,
  mados(Reg),
  CurDTASeg := Reg es;
  CurDTAOfs := Reg bx,

[ Set the DTA address of the TSR ]
  Reg ah := $1A,
  Reg ds := TarDTASeg,
  Reg dx := CurDTAOfs,
  mados(Reg),

[ Get ^Break setting ]
  Reg ax := $3300,
  mados(Reg);
  CurBreak := Reg dl,

[ Disable ^Break checking ]
  Reg ax := $3301,
  Reg dl := 0,
  mados(Reg),

[ Get current critical error interrupt address ]
  getintvec(CRITINT, @OldCritVec),

[ Install the TSR's critical error interrupt ]
  setintvec(CRITINT, @newcrit),

[ Make the TSR's PSP current ]
  CurPSP := PSPArray[1],
  for I = 1 to PSPCounter do
    memw[DosSeg.PSPArray[I]] := TarPSP,
  end, { beforepop }

procedure afterpop,
var
  I byte,
  Reg registers,
begin
  Running := FALSE,

[ Set DTA address back to the original ]
  Reg ah := $1A,
  Reg ds := CurDTASeg,
  Reg dx := CurDTAOfs,
  mados(Reg),

[ Reset original ^Break setting ]
  Reg ax := $3301,
  Reg dl := CurBreak,
  mados(Reg),

[ Restore original critical error interrupt ]
  setintvec(CRITINT, @OldCritVec),

[ Restore the interrupted PSP back ]
  for I = 1 to PSPCounter do
    memw[DosSeg.PSPArray[I]] := CurPSP,
  end, { afterpop }

procedure popup,
[ Popup interface routine ]
begin
  beforepop,
  _popRoutine,
  afterpop,
end, { popup }

```

```

>>

{----- Public routines -----}
function ok2inst1 (Count word;
                  Arg,Name string; boolean,
{ Check DOS version and TSR already installed }
var
  Vec,
  Dif word,
  Ptr,
  Rtn pointer,
  PtrArr array[1..2] of integer absolute Ptr,
  RtnArr array[1..2] of integer absolute Rtn,
  St string,
  Reg registers,
begin
  ok2inst1 = FALSE,

{--- Check DOS version ---}
  Vec = swap(dosversion);
  if Vec < $30A then
    begin
      writeln('DOS 3.1 REQUIRED!');
      exit;
    end;

{--- Check TSR already installed ---}
  TarName = Name; { Initialize TSR signature }
  Rtn := @tar_isr,

{ Calculate the difference between the interrupt service }
{ routine and data segment in the transient program }
  Dif := DSeg - RtnArr[2],

  for Vec = $60 to $67 do
    begin
      { Get the address for the interrupt vector }
      getintvec(Vec, Ptr),

      { If the address = NIL then the TSR program has not yet }
      { been installed. Set the interrupt to point to the }
      { TSR's interrupt service routine, save the vector's }
      { number and exit }
      if Ptr = NIL then
        begin
          setintvec(Vec, Rtn),
          TarVec = Vec,
          ok2inst1 = TRUE,
          exit;
        end;

      { Store the signature in a string and compare it to the }
      { string passed to the function. If the two strings }
      { match, the TSR is already installed. Use the offset }
      { calculated previously to access the data segment of }
      { the TSR program }
      { If TSR already resident, check command line for /u }
      { Communicate with the TSR program via it's interrupt }
      move(mem[PtrArr[2]+Dif ofs TarName], St,
           length(TarName)+1),
      if St = TarName then
        begin
          if (Count > 0) and (Arg[1] = '/') and
             (uppercase(Arg[2]) = 'U') then
            Reg ax = 1 { Uninstall command }
          else
            Reg ax = $FF, { Invalid switch }
            intri(Vec, Reg), { Communicate command to }
            exit, { resident portion }
          end;
        end, [ for ]
      end, [ ok2inst1 ]

procedure instaltr (_PopRtn TProc,
                   HotKey word, HKMsg string),
{ Get DOS busy flag address Save the DTA address }
{ Get current interrupts }
var
  ADR word,
  Reg registers,
begin
  HScanCode := hi(HotKey),
  HShiftStatus := lo(HotKey),
  _PopRoutine = _PopRtn,
  HotKeyMsg = HKMsg + NL + 'TSR switch [/Uninstall]',
  CheckBreak = FALSE,

{ Save the TSR's stack seg and sp To be used to swap }
{ stack in beginint }
  TarStackS := sseg, { Save stack segment }
  TarStackSP := sptr, { Save stack pointer }

{ Save address of InDos and Critical Error Flags }
  Reg ah := $14,
  mados(Reg),
  DosSeg = Reg.es,
  InDosFlag = Reg.bx,
  CritErrFlag = pred(Reg.bx),

{ Save address of DTA }
  Reg ah := $2F,
  mados(Reg),
  TarDTASeg := Reg.es,
  TarDTAOfs := Reg.bx,

{ Save segment of PSP }
  Reg ah := $51,
  mados(Reg),
  TarPSP := Reg.bx,

{ Locate 1st memory control block (MCB) to determine the }
{ end of DOS memory }
  Reg ah := $52,
  mados(Reg),
  EndDosMem := memw[Reg es Reg.bx-2],

  PSPCounter = 0,
  ADR := 0,
  while (PSPCounter < 2) and
        ((DosSeg shl 4) + ADR) < (EndDosMem shl 4) do
    begin
      if memw[DosSeg ADR] = TarPSP then
        begin
          Reg ah := $50,
          Reg.bx := TarPSP + 1,
          mados(Reg),
          if memw[DosSeg ADR] = TarPSP + 1 then
            inc(PSPCounter),
            PSPArray[PSPCounter] = ADR,
            Reg ah := $50,
            Reg.bx := TarPSP,
            mados(Reg),
          end,
          inc(ADR),
        end;
      end;

{ Get current interrupt vectors }
  getintvec(TIMERINT, @OldTimerVec), { Timer }
  getintvec(KBDINT, @OldKbdVec), { Keyboard }
  getintvec(IDLEINT, @OldIdleVec), { DOS idle }

{ Show that the TSR is now active }
  writeln('INSTALLED',NL,HotKeyMsg),

{ Install JTSRTP interrupts handlers }
  setintvec(TIMERINT, @newtimer), { TSR's timer }
  setintvec(KBDINT, @newkbd), { TSR's keyboard }
  setintvec(IDLEINT, @newidle), { TSR's DOS idle }

{ Restore DOS ^Break interrupt }
  if StackSW = -1 then
    setintvec($1B, SaveInt1B),

{ Terminate and Stay Resident }
  keep(0),
end, [ instaltr ]

{===== Initialization =====}
begin
end { unit - JTSRTP }

```

```

>>
*****
* System      Thesis Demo
* Program     SAVESCR.PAS
* Description  JTerTP demo program
*             Screen capture utility
*             This program is restricted to text mode
*             screens (80x25)
*             Portions Copyright Borland International
*             This program was compiled with
*             Turbo Pascal v6 0
* Author      José Luis Iñigo Ortega
* Revisions   (01-DEC-92) Version 1 0
*****

program SaveScr;

($M 6000, 0, 0)
{$V-,I-,R-,S-,D-,F+,X+}

uses crt, dos, JTerTP;

type
( Enough space for chars and attrs )
  ScrLine = array[1..80] of word;
  VideoPage = array[1..25] of ScrLine;

( External assembly routines from Borland's TCalc )
($L TCMVSMEM OBJ)
procedure movetoscreen (var Source, Dest,
                        Len word); external;

procedure movefromscreen (var Source, Dest,
                          Len word); external;

procedure screensaver;
( Actual TSR routine )
const
  Count byte = 0;
var
  Attr,X,Y byte;
  CrtMode byte absolute $0000 $0449;
  FName string[12];
  Scr VideoPage;
  vpFile file of VideoPage;
begin
( Continue if in text mode 80x25 )
  if CrtMode in {BW80,C080,MONO} then
  begin
( Capture screen )
    if CrtMode = MONO then
      movefromscreen(ptr($B000,0000)^,Scr,4000)
    else
      movefromscreen(ptr($B800,0000)^,Scr,4000);
    Attr = TextAttr;
    X = whereX;
    Y = whereY;
( Build file name )
    str(Count,FName);
    FName = 'SCREEN' + FName;
    inc(Count);
    TextAttr = 79;
    gotoxy(1,1);
( Save screen )
    assign(vpFile,FName);
    rewrite(vpFile);
    if ioresult <> 0 then
      write(' Error opening file ')
    else
      begin
        write(vpFile,Scr);
        if ioresult <> 0 then
          write(' Error writing file ')
        else
          begin
            writeln('-----');
            writeln(' Screen saved. Press any key. ');
            write('-----');
            end;
          close(vpFile);
        end;
      end;
( Restore screen )
    readkey;
    TextAttr = Attr;
    gotoxy(X,Y);
    if CrtMode = MONO then
      movetoscreen(Scr,ptr($B000,0000)^,4000)
    else
      movetoscreen(Scr,ptr($B800,0000)^,4000);
    end;
  else
    write('^C'); ( Error, not in 80x25 text mode )
  and, { screensaver }

(----- Main program -----)
begin
( If already installed, check for /u switch )
( HotKey
  ( Scan Code) $1F = 'S'
  ( Shift Status) $06 = Ctrl+Left-Shift )

  if ok2instl(paramcount,paramstr(1),'SaveScr')
  then
    installtr (screensaver, $1F06,
              'Press Ctrl+LeftShift+S to activate');
end ( program - SaveScr )

```

```

*****
* System      Thesis Demo
* Program     SHOWSCR.PAS
* Description  Utility program to display screens
*             captured with SAVESCR.PAS
*             Portions Copyright Borland International.
*             This program was compiled with
*             Turbo Pascal v6 0
* Author      José Luis Iñigo Ortega
* Revisions   (01-DEC-92) Version 1 0.
*****

program ShowScr;

($M 1024,0,0)
{$I-,B-,R-,D-,X+}

uses crt;

type
( Enough space for chars and attrs )
  ScrLine = array[1..80] of word;
  VideoPage = array[1..25] of ScrLine;

var
  Scr VideoPage;
  vpFile file of VideoPage;
  CrtMode byte absolute $0000 $0449;

( External assembly routines from Borland's TCalc )
($L TCMVSMEM OBJ)
procedure movetoscreen (var Source, Dest;
                        Len word); external;

procedure movefromscreen (var Source, Dest,
                          Len word); external;

procedure abort (FmtMsg string);
( Show error message and halt execution )
begin
  writeln(FmtMsg);
  halt(1);
and, { abort }

(----- Main program -----)
begin
  if paramcount > 0 then
  begin
    assign(vpFile, paramstr(1));
    reset(vpFile);
    if ioresult <> 0 then
      abort('Unable to open input file ');
    read(vpFile,Scr);
    if ioresult <> 0 then
      abort('Error reading input file ');
    close(vpFile);
    if CrtMode in {BW80,C080,MONO} then
      begin
        if CrtMode = MONO then
          movetoscreen(Scr,ptr($B000,0000)^,4000)
        else
          movetoscreen(Scr,ptr($B800,0000)^,4000);
        gotoxy(80,25);
        readkey;
        clrscr;
      end;
    else
      abort('^G');
    end;
  else
    abort('Usage SHOWSCR <filespec>');
  end ( program - ShowScr )

```

```

>>
;*****
; System      Thesis Demo
; Program     TSRTMPLT.ASM
; Description  A template for a TSR program
;              This program was build with
;              Turbo Assembler v2.01 and
;              Turbo Link v3.01
;              To build TSRTMPLT.COM use the following or
;              equivalent commands
;              C> tasm /m2 tsrtmplt.asm
;              C> tlink tsrtmplt.obj /t /x
; Author      Douglas Boling (*MSJ*)
; Modified by  José Luis Inigo Ortega
; Revisions   (1991) Version 1.0
;              (01-DEC-92) Version 1.1
;              Bug fix, Get EMS Version
;*****

;-----
; BIOS data segment
;-----
Bios_data segment at 40h
    org 17h
Shift_state db ? ; State of keyboard shift keys
    org 42h
Video_Buffoff dw ? ; Offset of video buffer
    org 63h
Video_IORega dw ? ; I/O addr of video controller
    org 84h
Video_rows db ? ; Number of rows on screen
Bios_data ends

;-----
; Code segment
;-----
CSeg segment para public 'code'
    assume cs CSeg

POPTIME equ 18 ; Ticks allowed until pop
RESSTACKSIZE equ 512 ; Size of resident stack

    org 80h
Command_tail db ?

    org 100h
BEGIN
    jmp initialize

Program db "TSRTMPLT v1.0, (c) 1991 Douglas Bolings",26

;-----
; Memory locations requiered for system overnead
;-----
HotShift db 06h ; Shift combination for TSR
                <Ctrl><Left shift>
                bit key
                0 R Shift
                1 L Shift
                2 Ctrl
                3 Alt

Dos_version dw 0 ; Dos version number
Mouse_flag db 0 ; Mouse driver present
Resident_SP dw 0 ; Resident stack pointer

Ems_flag db 0 ; Use expanded memory
Ems_segment dw 0 ; EMS page frame segment
Ems_version db 0 ; Version of EMS manager
Ems_handle dw 0 ; EMS handle used

Xms_flag db 0 ; Extended memory available
Xms_service dd -1 ; Entry pt for XMS manager
Xms_version db 0 ; Version of XMS manager

Indos_ptr dd -1 ; Pointer to INDOS flag
CritErr_ptr dd -1 ; Pointer to DOS crit err flag

Int08h dd -1 ; Int 08 vector (Timer)
Int09h dd -1 ; Int 09 vector (Keyboard HW)
Int10h dd -1 ; Int 10 vector (Video BIOS)
Int13h dd -1 ; Int 13 vector (Disk BIOS)
Int16h dd -1 ; Int 16 vector (BIOS Kbd)
Int28h dd -1 ; Int 28 vector (DOS Idles)
Int2Fh dd -1 ; Int 2F vector (Multiplex)

Int08_active db 0 ; Interrupt active flag
Int09_active db 0 ; Interrupt active flag
Int10_active db 0 ; Interrupt active flag
Int13_active db 0 ; Interrupt active flag
Int16_active db 0 ; Interrupt active flag
Int28_active db 0 ; Interrupt active flag
Int2F_active db 0 ; Interrupt active flag

; Jump table used by switcher notification routine
SWNNotifyJT dw offset SWNInit
            dw offset SWNSuspend
            dw offset SWNResume
            dw offset SWNActivate
            dw offset SWNActive
            dw offset SWNCSession
            dw offset SWNDeSession
            dw offset SWNExit

; Switcher global data structures

StartupInfo = $
    dw 3 ; Switcher structure ID
    dd 0 ; Ptr to prev startup struct
    dd 0 ; Ptr to name of opt dev driver
    dd 0 ; Date for Win device driver
    dd 0 ; Ptr to instance mem list

    dd 0 ; Ptr to instance data
    dw 0 ; Size of instance data
    dd 0 ; Ptr to instance stack
    dw 0 ; Size of instance stack
    dd 0 ; Ptr to next block = 0 to
    dw 0 ; terminate list

;-----
; Instance data
;-----
TSRInstData = $

;-----
; Switcher instance data structure
;-----
SWService dd 0 ; Ptr to switcher service rtn

CallbackInfo = $
    dw 0 ; Ptr to prev callback struct
    dd 0 ; Ptr to local callback proc
    dd 0 ; Reserved
    dd 0 ; Ptr to info structure

SwapInfo = $
    dw 0 ; Zero swap info astructure
    dw 0 ; since no API support
    dw 0 ; implemented by TSR
    dw 0

;-----
; DOS state information
;-----
Saved_DTA dd 0 ; Saved ptr to curr DTA
Saved_PSP dw 0 ; Saved segment of curr PSP
SS_register dw 0 ; SS register
SP_register dw 0 ; SP register

Shift_save db 0 ; Keyboard shift lock state
A20_state dw 0 ; Saved state of A20 line

Mem_alloc dw 0 ; DOS mem allocation start
LinkFlag db 0 ; DOS UMS link flag

Curr_disk db 0 ; Default disk at popup
Curr_dir db 64 dup (0) ; Default directory at popup

ERRINFOARRAY
errAX dw 0 ; Saved extended error info
errBX dw 0
errCX dw 0
errDX dw 0
errSI dw 0
errDI dw 0
errDS dw 0
errES dw 0
dw 3 dup (0) ; Reserved error table bytes

Vector1Bh dd 0 ; Int 1B vector (Break)
Vector23h dd 0 ; Int 23 vector (Ctrl-C)
Vector24h dd 0 ; Int 24 vector (Ctrl-Err)

;-----
; TSR state information
;-----
PopFlag db 0 ; Request flag/timer
Main_active db 0 ; TSR active flag
No_switch dw 0 ; Flag indicating switch OK
Ret_addr dw 0 ; Saved return addr for calls
Win_enhanced db 0 ; Enhanced mode Windows flag

TSRInstDataEnd = $

;-----
; VIDEOINT processes BIOS video services interrupt [10h]
;-----
videoint proc far
    assume cs CSeg, ds nothing, es nothing
    inc cs Int10_active
    pushf
    call cs [Int10h] ; Call old int
    dec cs Int10_active
    iret
videoint endp

;-----
; DISKINT receives control when an interrupt 13h is
; generated
;-----
diskint proc far
    assume cs CSeg, ds nothing, es nothing
    pushf ; Save flags register
    inc cs Int13_active ; Set disk access flag
    call cs [Int13h]
    pushf ; Save flags again
    dec cs Int13_active ; Reset disk access flag
    popf ; Restore flags

```

```
>> ret 2 ,Exit with flags intact
diskint endp
```

```
;=====
; BIOSKEYINT processes BIOS keyboard services (16h)
;=====
```

```
bioskeyint proc far
    assume cs:CSeg, ds:nothing, es:nothing
    pushf ,Save flags register
    inc cs:INT16_active ,Set kbd active flag
    call cs:[INT16h]
    pushf ,Save flags again
    dec cs:INT16_active ,Reset kbd active flag
    popf ,Restore flags
    ret 2 ,Exit with flags intact
bioskeyint endp
```

```
;=====
; TIMERINT processes timer interrupt (08h)
;=====
```

```
timerint proc far
    assume cs:CSeg, ds:nothing, es:nothing
    pushf
    call cs:[INT08h] ,Call old int

    cmp cs:INT08_active,0 ,See if we are in this
    jne TIMERINT_EXIT , routine already

    cmp cs:PopFlag,0 ,See if we need to try
    jne , to pop up
```

```
TIMERINT_EXIT
    iret ,Return
```

```
TIMER_CHECK
    push ax
    inc cs:INT08_active ,Set int active flag
    call check_system ,See if system OK to popup
    or ax,ax
    jne TIMERINT_DEC
    call main ,Call the TSR
    mov cs:PopFlag,1
```

```
TIMERINT_DEC
    dec cs:PopFlag
    dec cs:INT08_active ,Clear int active flag
    pop ax
    jmp short TIMERINT_EXIT
timerint endp
```

```
;=====
; KEYINT processes keyboard interrupt (09h)
;=====
```

```
keyint proc far
    assume cs:CSeg, ds:nothing, es:nothing
    pushf
    call cs:[INT09h] ,Call old int

    push ax
    push ds
    mov ax,40h
    mov ds,ax ,Set DS to BIOS data segment
    assume ds:BIOS_data
    mov al,ds:[Shift_state]
    and al,0Fh ,Mask lock bits
    cmp al,cs:[HotShift] ,Check for hot key
    pop ds
    pop ax
    je KEYINT_HOTKEY
```

```
KEYINT_EXIT
    iret ,Return
```

```
KEYINT_HOTKEY
    mov cs:PopFlag,POPTIME ,Set timer to pop up
    jmp short KEYINT_EXIT
keyint endp
```

```
;=====
; IDLEINT processes DOS idle interrupt (28h)
;=====
```

```
idleint proc far
    assume cs:CSeg, ds:nothing, es:nothing
    pushf
    call cs:[INT28h] ,Call old int

    cmp cs:INT28_active,0 ,See if we are in this
    jne IDLEINT_EXIT , routine already

    cmp cs:PopFlag,0 ,See if we need to try
    jne IDLE_CHECK , to pop up
```

```
IDLEINT_EXIT
    iret ,Return
```

```
IDLE_CHECK
    push ax
    inc cs:INT28_active ,Set int active flag
    call check_system ,See if OK to pop up Ignore
    or al,al ,INDOS flag since in idle
    jne IDLEINT_EXIT_1
    mov cs:PopFlag,0 ,Clear pop flag
    call main ,Call the TSR
```

```
IDLEINT_EXIT_1
    dec cs:INT28_active ,Clear int active flag
    pop ax
    jmp short IDLEINT_EXIT
idleint endp
```

```
;=====
; MUXINT processes DOS Multiplex interrupt (2Fh)
;=====
```

```
muxint proc far
    assume cs:CSeg, ds:nothing, es:nothing

    cmp ax,1605h ,See if Windows start
    je INIT_WIN
    cmp ax,4B05h ,See if switcher get
    je INIT_INSTANCE , instance data
    cmp ax,4B01h ,See if switcher build chain
    je CHAIN_INIT
```

```
MUXINT_JMP
    jmp cs:[INT2Fh] ,Call old int
```

```
CLOSE_WIN
    test dx,01h ,See if enhanced mode Windows
    jne MUXINT_JMP
    dec cs:Win_enhanced ,Clear enhanced mode flag
    jmp short MUXINT_JMP
```

```
INIT_WIN
    test dx,01h ,See if enhanced mode Windows
    jne INIT_INSTANCE
    inc cs:Win_enhanced
```

```
INIT_INSTANCE
    pushf
    call cs:[INT2Fh] ,Call old int
    word ptr cs:[a1aNextDev],bx
    mov word ptr cs:[a1aNextDev+2],es
    push cs ,ES BX point to switcher struct
    pop es
    mov bx,offset StartupInfo
    jmp short MUXINT_EXIT
```

```
CHAIN_INIT
    pushf
    call cs:[INT2Fh] ,Call old int
    word ptr cs:[acbiNext],bx
    mov word ptr cs:[acbiNext+2],es
    push cs ,ES BX point to switcher struct
    pop es
    mov bx,offset CallbackInfo
```

```
MUXINT_EXIT
    iret ,Return
muxint endp
```

```
;=====
; CHECK_SYSTEM determines if the system is in a state
; compatible with TSRs
; Exit CF - Clear if OK to pop up
;=====
```

```
check_system proc near
    assume cs:CSeg, ds:nothing, es:nothing
    push bx
    push ds
    xor ax,ax
    or al,cs:INT10_active ,Check BIOS video int
    or al,cs:INT13_active ,Check BIOS disk int
    or al,cs:INT16_active ,Check BIOS keyboard int
    lds bx,cs:CritErr_ptr ,Check DOS critical error
    or al,byte ptr ds:[bx] , flag
    lds bx,cs:INDOS_ptr ,Check INDOS flag
    mov ah,byte ptr ds:[bx]
```

```
CHECK_SYS_EXIT
    pop ds
    pop bx
    ret
```

```
check_system endp
```

```
;=====
; CRITICALERR receives control when an interrupt 24h is
; generated
;=====
```

```
criticalerr proc far
    assume cs:CSeg, ds:nothing, es:nothing
    xor al,al ,Default to ignore
    cmp cs:Dos_version,30Ah ,See if before 3.1
    jl CRITICAL1
    add al,3
```

```
CRITICAL1
    iret
criticalerr endp
```

```
;=====
; MAIN
;=====
```

```
main proc far
    assume cs:CSeg, ds:nothing, es:nothing
    cli
    inc cs:No_switch ,Don't allow switch
    inc cs:[Main_active] ,Set program status flag
```

```

>> mov     cs,SS_register,ss    ,Save SS and SP registers
mov     cs,SP_register,sp
mov     ax,cs                  ,Switch to internal stack
mov     bx,Resident_SP
mov     ss,ax
mov     sp,bx
dec     cs,No_switch
sti
call    save_regs              ,Enable interrupts
assume  ds,CSeg, es,nothing

; Point the interrupt 18h, 23h, 24h vectors to internal
; handlers
mov     ax,351Bh               ,Get and save 1Bh
int     21h
mov     word ptr Vector1Bh,bx
mov     word ptr Vector1Bh[2],es
mov     ax,2*1Bh               ,Point interrupt to IRET
mov     dx,offset IDLEINT_EXIT
int     21h

mov     ax,3523h               ,Get and save 23h
int     21h
mov     word ptr Vector23h,bx
mov     word ptr Vector23h[2],es
mov     ax,2*23h               ,Set to IRET
mov     dx,offset IDLEINT_EXIT
int     21h

mov     ax,3524h               ,Get and save 24h
int     21h
mov     word ptr Vector24h,bx
mov     word ptr Vector24h[2],es
mov     ax,2*24h               ,Point int to internal
mov     dx,offset criticalerr   ,critical error handler
int     21h

; Save and switch to internal PSP
mov     ah,51h                 ,Get current PSP
call    dospapi               ,Beware DOS 2.0 - 3.0
mov     Saved_PSP,bx           ,Save it
push    cs
push    bx
mov     ah,50h                 ,Set internal PSP
call    dospapi

; Save and switch to internal DTA
mov     ah,2Fh                 ,Get current DTA
int     21h
mov     word ptr Saved_DTA,bx   ,Save it
mov     word ptr Saved_DTA[2],es
mov     dx,offset Command_tail ,Use PSP for DTA
mov     ah,1Ah                 ,Set DTA
int     21h

; If DOS 3.x save extended error information
cmp     word ptr Dos_version,030Ah
jb      SKIP_ERR_SAVE
push    ds                     ,Save DS
xor     ax,ax                  ,Clear registers
mov     bx,ax
mov     cx,ax
mov     dx,ax
mov     di,ax
mov     si,ax
mov     ah,59h                 ,Extended error info
int     21h
mov     cs,[errDS].ds          ,Call DOS
mov     ds,[errDS].ds          ,Saved returned DS
pop     ds                     ,Restore DS
push    bx
mov     bx,offset ERRINFOARRAY ,Save data in regs
mov     [bx],ax                ,in this order
pop     [bx+2]
mov     [bx+4],cx
mov     [bx+6],dx
mov     [bx+8],si
mov     [bx+10],di
mov     [bx+14],es

SKIP_ERR_SAVE
; If using EMS memory, save EMS mapping context and
; map our page
cmp     Ems_flag,0
ja      SKIP_EMS_SAVE
mov     ah,47h                 ,Save mapping context
mov     dx,Ems_handle
int     67h
or      ah,ah
jne     CLEAN_UP
xor     ax,ax
mov     bx,ax
call    map_emamem             ,Map our page
jne     CLEAN_UP

SKIP_EMS_SAVE
; Save default directory and drive
mov     ah,19h                 ,Get current disk
int     21h
mov     Curr_disk,al

```

```

mov     ah,47h                 ,Get current directory
xor     di,di
mov     si,offset Curr_dir
int     21h

; Save state of A20 line
cmp     Xms_flag,0
je      SKIP_XMS_SAVE
mov     ah,7                    ,Get state of A20 line
call    [Xms_service]
mov     A20_state,ax

SKIP_XMS_SAVE
; Save UMB link and mem allocation state
; If DOS 5, link UMBS
mov     ax,5800h               ,Get DOS mem alloc strategy
int     21h
mov     Mem_alloc,ax
cmp     Dos_version,500h       ,If DOS 5, save UMB link
jb      SKIP_UMB_LINK          ,state Then link UMBS
mov     ax,5802h
int     21h
mov     LinkFlag,al            ,Save UMB link state
mov     ax,5803h
mov     bx,1                    ,Link UMBS
int     21h

SKIP_UMB_LINK
; Save mouse state
cmp     Mouse_flag,0
je      SKIP_MOUSE_SAVE
mov     ax,17h
mov     dx,offset ResCodeEnd   ,Save mouse state
push    cs
pop      es
int     33h

SKIP_MOUSE_SAVE
; Save keyboard shift state
mov     ah,2                    ,Get keyboard shift state
int     16h
mov     Shift_save,al

; Do real work here
; Beep the speaker
mov     ax,0E07h
int     10h

WORK_1
mov     ah,01h                 ,See if key available
int     16h
jne     WORK_2
call    car_idle                ,Indicate TSR idle
jmp     short WORK_1

WORK_2
xor     ah,ah
int     16h
cmp     al,27                    ,See if <ESC> key
jne     WORK_1                  ,No, continue to wait

mov     ax,0E07h                 ,Beep the speaker again
int     10h

; Clean up DOS for return to foreground task
CLEAN_UP
push    cs
pop      ds
assume  ds,CSeg, es,nothing

; Restore keyboard shift state
mov     ax,Bios_data            ,Point ES to BIOS data seg
mov     es,ax
assume  es,Bios_data
mov     al,Shift_save
and     al,0F0h                 ,Look only at lock bits
mov     es,[Shift_state],al
push    cs
pop      es
assume  es,CSeg

; Restore mouse state
cmp     Mouse_flag,0
je      SKIP_MOUSE_RESTORE
mov     ax,17h
mov     dx,offset ResCodeEnd   ,Restore mouse state
int     33h

SKIP_MOUSE_RESTORE
; Restore UMB link state, link and allocation state
mov     ax,5801h               ,Set DOS mem alloc strategy
mov     bx,Mem_alloc

```

```

>>
int 21h

cmp Dos_version,500h ,If DOS 5, restore UMB
jb SKIP_UMBLINK_RESTORE , link state
mov ax,5803h ,Set UMB link state
xor bh,bh
mov bx,linkflag
int 21h

SKIP_UMBLINK_RESTORE
-----
; Restore A20 line
-----
cmp Xms_flag,0
je SKIP_XMS_RESTORE
mov ah,5 ,Assume local disable A20
cmp A20_state,0
je XMS_RESTORE_1
inc ah

XMS_RESTORE_1
call [Xms_service] ,Restore A20 line state
mov A20_state,ax

SKIP_XMS_RESTORE
-----
; If using EMS memory, restore EMS mapping context
-----
cmp Ems_flag,0
je SKIP_EMS_RESTORE
mov ah,48h ;Restore mapping context
mov dx,Ems_handle
int 67h

SKIP_EMS_RESTORE
-----
; Restore default directory and drive
-----
mov ah,0Bh ;Set default disk
mov dl,Curr_disk
int 21h
mov ah,3Bh ;Set current directory
mov dx,offset Curr_dir
int 21h

; Restore extended error info
-----
cmp word ptr Dos_version,30Ah ,DOS 3. and up
jb SKIP_ERR_RESTORE
mov ax,5D0Ah ,Restore ext error info
mov dx,offset ERRINFOARRAY ,Point to saved info
int 21h

SKIP_ERR_RESTORE
-----
; Restore PSP and DTA
-----
mov bx,Saved_PSP
mov ah,50h ,Restore PSP
call dospascal
push ds
lds dx,[Saved_DTA]
mov ah,1Ah ,Restore DTA
int 21h

; Reset the displaced interrupt 1Bh, 23h, 24h vectors
-----
mov ax,2524h ,Reset int 24h vector
lds dx,cs [Vector24h]
int 21h
mov ax,2523h ,Reset int 23h vector
lds dx,cs [Vector23h]
int 21h
mov ax,251Bh ;Reset int 1Bh vector
lds dx,cs [Vector1Bh]
int 21h
pop ds

; Restore register values, switch back to original stack
; and return to caller
-----
MAIN_EXIT
call restore_regs ,Restore registers
assume ds,nothing
cli
mov ax,cs.No_switch ,Make sure inta are off
mov ss,cs.SS_register ,Don't allow task switch
mov sp,cs.SP_register ,Switch to original stack
dec cs[Main_active] ,Clear program status flag
dec cs.No_switch ,Allow task switch
sti ,Interrupts on
ret ,Return to interrupt routine
main endp
=====

; DOSPSCALL modifies critical error flag on PSP calls to
; DOS if using 2.x
-----
dospascal proc near
assume cs,CSeg
cmp cs[Dos_version],30Ah ,See if DOS < 3.1
jbe DOSPSCALL_OK ,No, just call DOS
push ds
push di

```

```

lds di,cs.CritErr_ptr ,Retrieve crit err flag addr
inc byte ptr [di] ,Set DOS in crit err state
pop di
pop ds
int 21h ,Call DOS
push ds
push di
lds di,cs.CritErr_ptr ;Retrieve crit err flag addr
dec byte ptr [di] ,Reset DOS crit err state
pop di
pop ds
ret

```

```

DOSPSCALL_OK
int 21h
ret
dospascal endp

```

```

; STOP_SWITCH signals the task switcher (if active) to not
; switch this session

```

```

stop_switch proc near
assume cs,CSeg
inc cs.No_switch ,Set no switch flag for T 8
cmp Win_enhanced,0
jne STOP_SWITCH_1

```

```

STOP_SWITCH_EXIT
ret

```

```

STOP_SWITCH_1
mov ax,1681h ,Win begin critical section
int 2Fh
jmp short STOP_SWITCH_EXIT
stop_switch endp

```

```

; GO_SWITCH signals the task switcher (if active) it is
; OK to switch this session

```

```

go_switch proc near
assume cs,CSeg
dec cs.No_switch ,Clear switch flag for T 8
cmp Win_enhanced,0
jne GO_SWITCH_1

```

```

GO_SWITCH_EXIT
ret

```

```

GO_SWITCH_1
mov ax,1682h ,Win end critical section
int 2Fh
jmp short GO_SWITCH_EXIT
go_switch endp

```

```

; TSR_IDLE signals the system that the TSR is idle

```

```

tsr_idle proc near
assume cs,CSeg
int 28h ,Old DOS idle
cmp Dos_version,300h ,No multiplex interrupt for
jb TSR_IDLE_EXIT ,DOS 2.x
mov ax,1680h ,Win-OS/2 release timeslice
int 2Fh

```

```

TSR_IDLE_EXIT
ret
tsr_idle endp

```

```

=====
; NOTIFY - Parse switcher notification messages
=====

```

```

SWNotify proc far
assume cs,CSeg,ds,nothing,es,nothing
push si
push ds
push cs
pop ds
assume ds,CSeg
mov word ptr [SWService],di ,Save ptr to service
mov word ptr [SWService+2],ax ; routine
cmp ax,7
ja SWN1 ,If unknown type, skip
mov si,ax
shl si,1
call [SWNotifyJP+si] ,Call proper notification func
mov ax,0

```

```

SWN1
pop ds
pop si
ret

```

```

; SWINIT - Switcher initialization notification

```

```

SWInit proc near
assume cs,CSeg,ds,CSeg
mov ax,0 ,Allow switcher init
ret
SWInit endp

```

```

; SWSUSPENDINIT - Switcher query suspend notification

```

```

>> SWNSuspend proc near
    assume cs:Cseg, ds:Cseg
    mov ax,No_switch ,Switch only if not in
    ret , critical section
SWNSuspend endp

, SWNSUSPEND - Switcher suspend notification

SWNSuspend proc near
    assume cs:Cseg, ds:Cseg
    mov ax,No_switch ,Switch only if not in
    ret , critical section
SWNSuspend endp

, SWACTIVATE - Switcher going active notification

SWNActivate proc near
    assume cs:Cseg, ds:Cseg
    mov ax,0 ,Allow session activate
    ret
SWNActivate endp

, SWACTIVE - Switcher session active notification

SWNActive proc near
    assume cs:Cseg, ds:Cseg
    mov ax,0 ,Allow session activate
    ret
SWNActive endp

, SWNCSESSION - Switcher create session notification

SWNCSession proc near
    assume cs:Cseg, ds:Cseg
    mov ax,0 ,Allow session create
    ret
SWNCSession endp

, SWNDSESSION - Switcher destroy session notification

SWNDSession proc near
    assume cs:Cseg, ds:Cseg
    mov ax,0 ,Allow session destroy
    ret
SWNDSession endp

, SWNEXIT - Switcher exit notification

SWNExit proc near
    assume cs:Cseg, ds:Cseg
    mov ax,0 ,Allow switcher exit
    ret
SWNExit endp

SWNNotify endp
, =====

, GET_EMSMEM allocates pages of memory from the Expanded
, memory manager
, Entry BX - number of 16K pages to request
, Exit DX - handle to pages
, ZF - Set if call successful

get_emsmem proc near
    mov ah,30h ,Allocate EMS pages
    int 67h
    or ah,ah
    ret
get_emsmem endp

, FREE_EMSMEM frees pages of memory from expanded memory
, manager
, Entry DX - handle to memory pages to free
, Exit ZF - Set if call successful

free_emsmem proc near
    mov ah,45h ,Free EMS
    int 67h
    or ah,ah
    ret
free_emsmem endp

, MAP_EMSMEM maps a page of expanded memory to the EMS
, page frame
, Entry AL - Physical page in page frame
, BX - logical page to map
, DX - handle to memory page
, Exit ZF - Set if call successful

map_emsmem proc near
    mov ah,44h ,Map EMS page
    int 67h
    or ah,ah
    ret
map_emsmem endp

, SAVE_REGS saves all the registers used in the interrupt
, routines and set DS

save_regs proc near
    cld [Ret_addr] ,Get address to return to
    push ax ,Save all registers
    push bx
    push cx
    push dx
    push bp
    push si
    push di
    push ds
    push ea
    pop ds ,Set DS = CS
    assume ds:Cseg
    jmp word ptr [Ret_addr] ,Return
save_regs endp

, RESTORE_REGS restores register values

restore_regs proc near
    pop Ret_addr ,save return address
    pop es ,Restore registers
    pop ds
    assume ds:nothing
    pop di
    pop si
    pop bp
    pop dx
    pop cx
    pop bx
    pop ax
    jmp word ptr cs [Ret_addr] ,Return
restore_regs endp

, =====
, End of resident code Resident stack and mouse save
, area sits just above resident code
, =====
ResCodeEnd = $

, =====
, Non-resident data
, =====
ErrMsg1 db "Usage TSRTMPLT [/u[/e]s"
ErrMsg2 db "Expanded memory driver not founds"
ErrMsg3 db "DOS 2.0 or greater requireds"
ErrMsg4 db "Not able to uninstalls"
ErrMsg5 db "TSRTMPLT not installeds"

InfoMsg1 db "TSRTMPLT installed",13,10
InfoMsg2 db "Press Ctrl-LeftShift to activate.$"
InfoMsg3 db "TSRTMPLT uninstalls"

ErrMsg6 db "Can't installs"

Installed_seg dw 0 ,segment of installed code
TSRResident db 0 ,TSR already installed flag
Def_disk db ? ,Default disk
Ems_header db "EMMXXXX0" ,EMS driver header
Mouse_SaveSize dw 0 ,Size of mouse save area

Cmd_switches db "eu" ,Letter of valid commands
Cmd_switch_end = $

, Jump table to routines that implement the cmd line args
Cmd_jumptbl dw offset remove
dw offset use_expanded

, =====
, INITIALIZE
, =====
initialize proc near
    assume cs:Cseg, ds:Cseg
    cld ,String operation UP
    mov dx,offset Program ,Print copyright
    call message

, Get DOS version, determine if TSR already installed

    mov ah,30h ,Get DOS version
    int 21h
    xchg al,ah ,Put version in proper order
    mov Dos_version,ax ,Save DOS version
    cmp ax,200h ,See if at least DOS ver 2.0
    ja INIT_0
    mov dx,offset ErrMsg3
    jmp short ERROR_EXIT

INIT_0
    call find_copy ,See if TSR already installed
    jc INIT_1
    inc TSRResident ,Set already installed flag
    mov Installed_seg,es ,Save installed code segment

INIT_1
    push cs
    pop es

, Parse the command line for switches

    mov di,offset Command_tail ,Point DI to command
    , line text

    xor cx,cx
    or cl,[di] ,Get length of command line
    jz INIT_3 ,If 0, skip parse
    inc di

```

>>

```
INIT_2
mov     al, '/'
repne   scasb      ;Put switch in AL
jne     INIT_3     ;Scan for cmd line switches
mov     al, [di]
or      al, 20h    ;Make lower case
push    cx
di      di, offset Cmd_switches
mov     cx, offset Cmd_switch_end - offset Cmd_switches
repne   scasb
mov     bx, cx     ;Copy index into list
pop     di
pop     cx
mov     dx, offset ErrMsg1 ;Command not found msg
jne     ERROR_EXIT
shl     bx, 1      ;Convert to word offset
push    di
push    cs, [bx+Cmd_jmptbl] ;Call command routine
pop     di
pop     cx
jnc     ERROR_EXIT
jnz     INIT_3
jmp     ashort INIT_2
```

IF TSR not installed, install it

```
INIT_3,
cmp     TSRresident, 0 ;Is TSR installed ?
je      INSTALL
mov     ax, 4C00h      ;No, terminate with RC = 0
int     21h
```

Display error message and exit with Return Code = 1

```
ERROR_EXIT,
call    message
mov     es, Installed_seg ;Point ES to installed code
dec     es [Main_active] ;Enable background task
mov     ax, 4C01h        ;Exit with RC = 1
int     21h
```

Install Get address of INDOS and DOS Critical Error flags

```
INSTALL
mov     ah, 34h        ;Get INDOS address from DOS
int     21h
mov     word ptr [Indos_ptr], bx
mov     word ptr [Indos_ptr+2], es
call    find_cerrorptr ;Find critical error flag
jnc     CEP_FOUND
mov     dx, offset ErrMsg6 ;Crit err flag not found
jmp     ashort ERROR_EXIT
```

```
CEP_FOUND
mov     word ptr CritErr_ptr, bx ;Store it
mov     word ptr CritErr_ptr[2], es
```

Initialize task switcher structures and see if task switcher active

```
cmp     word ptr Dos_version, 300h ;DOS 3.0 and up
jb      NOWIN
mov     ax, offset TSRInstData ;Init instance memory
mov     word ptr [InstItem1], ax ; structure
mov     word ptr [InstItem1+2], cs
mov     ax, offset TSRInstDataEnd - offset TSRInstData
mov     [InstSize1], ax

mov     ax, offset ResCodeEnd ;Init instance stack
mov     word ptr [InstItem2], ax ; structure
mov     word ptr [InstItem2+2], cs
mov     ax, RESSTACKSIZE
mov     [InstSize2], ax

mov     ax, offset InstItem1 ;Init ptr to inst
mov     word ptr [saiInstData], ax ; mem struct
mov     word ptr [saiInstData+2], cs

mov     ax, offset SWNNotify ;Init ptr to
mov     word ptr [schiEntryPoint], ax ; notification
mov     word ptr [schiEntryPoint+2], cs ; routine

mov     ax, 4B02h ;See if switcher active
int     2Fh
or      ax, ax
jne     NOWITCHER
mov     word ptr [SWService], di ;Save ptr to service
mov     word ptr [SWService+2], es ; routine

mov     ax, 4
push    cs ;ES BX point to callback
pop     es ; structure
mov     di, offset callbackInfo
call    SWService
```

NOWITCHER

Check for windows active

```
mov     ax, 1600h ;See if enhanced mode Windows
int     2Fh
or      al, al
je      NOWIN
inc     Win_enhanced ;Set enhanced mode flag
```

NOWIN

Determine if extended memory manager active

```
mov     ax, 4300h ;Look for XMEM SYS
int     2Fh
or      al, al
jne     NOXMS
mov     ax, 4310h ;Get entry point for XMM
int     2Fh
mov     word ptr [Xms_service], bx
mov     word ptr [Xms_service+2], es
push    cs
pop     es
inc     Xms_flag
```

NOXMS

Set up EMS memory if necessary

```
cmp     Ems_Flag, 0 ;See if expanded memory requested
je      NO_EMSMEM
call    check4ems ;See if EMS driver installed
jnc     INSTALL_1
mov     dx, offset ErrMsg2 ;EMS driver error
jmp     ERROR_EXIT
```

INSTALL_1.

```
mov     Ems_segment, bx ;Save seg of EMS page frame
mov     Ems_version, dl ;Save EMS version
mov     bx, 1 ;Allocate 1 page
call    get_emsmem
mov     Ems_handle, dx ;Save EMS handle
xor     al, al ;Map to EMS page 0
xor     bx, bx
call    map_emsmem
```

NO_EMSMEM.

See if mouse driver loaded

```
xor     ax, ax
int     33h
or      ax, ax
je      NO_MOUSE
inc     Mouse_flag ;Set mouse found flag, get
mov     ax, 15h ;size of mouse driver,
int     33h ;save context area
mov     Mouse_SaveSize, bx
```

NO_MOUSE

Set interrupts used by TSR

```
mov     al, 08h ;Get/set timer interrupt
mov     dx, offset timerint
mov     di, offset Int08h
call    set_interrupt

mov     al, 09h ;Get/set keyboard interrupt
mov     dx, offset keyint
mov     di, offset Int09h
call    set_interrupt

mov     al, 10h ;Get/set video interrupt
mov     dx, offset videoint
mov     di, offset Int10h
call    set_interrupt

mov     al, 13h ;Get/set disk interrupt
mov     dx, offset diskint
mov     di, offset Int13h
call    set_interrupt

mov     al, 16h ;Get/set BIOS keyboard int
mov     dx, offset bioskeyint
mov     di, offset Int16h
call    set_interrupt

mov     al, 28h ;Get/set DOS idle interrupt
mov     dx, offset idleint
mov     di, offset Int28h
call    set_interrupt

cmp     Dos_version, 300h ;DOS 3.0 and up
jb      SKIP_HOOK2F
mov     al, 2Fh ;Get/set DOS multiplex int
mov     dx, offset muxint
mov     di, offset Int2Fh
call    set_interrupt
```

SKIP_HOOK2F

Deallocate the program's environment block

```
mov     ax, ds [2Ch] ;Get environment segment
mov     es, ax
```

```

>> mov     ah,49h           ,Free it
    int     21h
    mov     dx,offset InfoMsg1 ;Tell user TSR is now
    call    message         , installed

; Terminate and Stay Resident with ERRORLEVEL = 0
    mov     ax,3100h
    mov     dx,offset ResCodeEnd-offset CSeg+RESSTACKSIZE
    add     dx,Mouse_SaveSize
    mov     Resident_SP,dx
    add     dx,15
    mov     cl,4
    shr     dx,cl
    int     21h
initializa endp
;=====
; USE_EXPANDED sets a flag to use expanded memory
use_expanded proc near
    mov     Ems_flag,1
    ret
use_expanded endp

; REMOVE deallocates the memory block addressed by EMS
; and restores the interrupt vectors displaced on
; installation
; EXIT CF - clear if program uninstalled
; CF - set if can't uninstall
remove proc near
    assume  cs CSeg, ds CSeg
    cmp     TSRResident,0 ;See if TSR installed
    jne     REMOVE_1
    mov     dx,offset ErrMsg5
    jmp     REMOVE_ERROR_1

REMOVE_1
    mov     es,Installed_seg
    mov     ax,5 ;Unhook notification chain
    mov     di,offset CallbackInfo
    cmp     word ptr es [SWService+2] 0
    jne     REMOVE_2
    call    es [SWService]

REMOVE_2
    mov     al,08h ;Check interrupt 08h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,09h ;Check interrupt 09h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,10h ;Check interrupt 10h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,13h ;Check interrupt 13h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,16h ;Check interrupt 16h vector
    call    check_vector
    jne     JMP_REMOVE_ERROR

    mov     al,28h ;Check interrupt 28h vector
    call    check_vector
    jne     REMOVE_3

JMP_REMOVE_ERROR
    jmp     REMOVE_ERROR

REMOVE_3
    cmp     Dos_version,300h ;DOS 3.0 and up
    jb     SKIP_CHECK2F
    mov     al,2Fh ;Check interrupt 2F vector
    call    check_vector
    jne     REMOVE_ERROR

SKIP_CHECK2F
    cmp     Ems_flag,0 ;If using expanded memory
    jne     SKIP_REMOVE_EMS
    mov     dx,Ems_handle ;Free expanded memory
    call    free_emamem
    jne     REMOVE_ERROR

SKIP_REMOVE_EMS
    push     ds ;Save DS
    assume  ds nothing
    mov     ax,2508h ;Restore interrupt 08h vector
    lds     dx,es [Int08h]
    int     21h

    mov     ax,2509h ;Restore interrupt 09h vector
    lds     dx,es [Int09h]
    int     21h

    mov     ax,2510h ;Restore interrupt 10h vector
    lds     dx,es [Int10h]
    int     21h

    mov     ax,2513h ;Restore interrupt 13h vector
    lds     dx,es [Int13h]
    int     21h

    mov     ax,2516h ;Restore interrupt 16h vector
    lds     dx,es [Int16h]
    int     21h

    mov     ax,2528h ;Restore interrupt 28h vector
    lds     dx,es [Int28h]
    int     21h

    cmp     ca Dos_version,300h ;Dos 3.0 and up
    jb     SKIP_RESTORE2F
    mov     ax,252Fh ;Restore interrupt 2Fh vector
    lds     dx,es [Int2Fh]
    int     21h

SKIP_RESTORE2F
    pop     ds ;Restore DS
    assume  ds CSeg
    not     word ptr es:[BEGIN] ;Destroy fingerprint
    mov     ah,49h ;Free memory given to
    int     21h ;original program block
    jc     REMOVE_ERROR ;Branch on error

    mov     dx,offset InfoMsg2 ;Tell user TSR is now
    call    message         , removed
    cld ;Clear CF for exit

REMOVE_EXIT
    ret ;Exit with CF intact

REMOVE_ERROR
    mov     dx,offset ErrMsg4 ;Error during remove

REMOVE_ERROR_1
    stc
    jmp     REMOVE_EXIT ;TSR was not installed
remove endp

; CHECK_VECTOR is called by remove to compare the segment
; pointed to by an interrupt vector against a segment
; value supplied by the caller
; Entry AL - interrupt number
; Exit ZF clear - segment do not match
; ZF set - segment match
check_vector proc near
    push     es
    mov     cx,es
    mov     ah,35h ;Get vector
    int     21h
    mov     ax,es ;Transfer segment to AX
    cmp     ax,cx ;and compare
    pop     es
    ret
check_vector endp

; SET_INTERRUPT get and sets an interrupt
; Entry AL - interrupt number
; DX - pointer to new interrupt routine
; DI - pointer to storage location for old
; interrupt vector
set_interrupt proc near
    assume  cs CSeg, ds CSeg, es nothing
    push     es
    push     ax
    mov     ah,35h ;DOS get interrupt
    int     21h
    pop     ax
    mov     word ptr [di],bx ;Save old vector
    mov     word ptr [di+2],es
    mov     ah,25h ;DOS set interrupt
    int     21h
    pop     es
    ret
set_interrupt endp

; FIND_COPY determines if the TSR is already resident
; Exit CF - clear if copy found
; ES - segment of installed code (if CF = 0)
find_copy proc near
    not     word ptr [BEGIN] ;Initialize fingerprint
    mov     bx,0A000h ;Start scan in UMBs
    mov     ax,ca ;Keep CS value in AX

FIND_COPY_1
    inc     bx ;Increment search seg value
    mov     es,bx
    cmp     ax,bx ;Not installed if current
    jne     FIND_COPY_2 ;segment is looped back to
    mov     si,offset BEGIN ;Search this seg for ASCII
    mov     di,si ;fingerprint
    mov     cx,16
    repe     cmpsb
    jne     FIND_COPY_1 ;Loop back if not found
    cld

FIND_COPY_EXIT
    ret

FIND_COPY_2
    stc
    ret

```

```

>>
find_copy    endp

;-----
; FIND_CFPTR returns a pointer to the DOS ErrorMode flag
; Exit  CF clear if ErrorMode Flag found
;       ES:BX - Segment Offset of ErrorMode Flag
;
; For versions of DOS before 3.1, search the following
; code in DOS seg
;
;       36, B0 3E >??> 00      cmp byte ptr es [ErrorMode],0
;       75 ??                  jne near label
;       CD 28                  int 28h
;
; For DOS 3.1 and later, ErrorMode sits before INDOS flag
;-----

find_cfptr   proc near
    mov     ah,34h              ;Get ptr to INDOS flag
    int     21h                ;After DOS 3.1 the CFP is
    dec     bx                  ;documented to be the byte
    cmp     Dos.version,30Ah    ;before the INDOS flag
    jnc     FIND_CbF_EXIT

    mov     ax,3E80h            ;CMP opcode
    mov     cx,1                ;Max segment size
    mov     di,bx               ;Start at INDOS address

FIND_CFP_1
    repne   scasb               ;Scan for CMP
    jcxz    FIND_CFP_NOTFOUND    ;Error if CMP not found
    cmp     es[di],ah           ;Check other half of CMP
    jne     FIND_CFP_1          ;opcode
    cmp     byte ptr es[di+4],075h ;Check for JNE
    jne     FIND_CFP_1
    cmp     word ptr es[di+6],028CDh ;Check for INT 28h
    jne     FIND_CFP_1          ;call

FIND_CFP_FOUND
    inc     di                  ;Get offset of ErrorMode flag
    mov     bx,es[di]

FIND_CFP_EXIT
    ret

FIND_CFP_NOTFOUND.
    stc
    jmp     short FIND_CFP_EXIT
find_cfptr   endp

;-----
; CHECK4EMS determines if an expanded memory driver is
; loaded
; Exit  BX - segment of EMS page frame if CF clear
;       DL - EMS driver version number
;       CF - clear if EMS driver found
;-----

check4ema    proc near
    push    es
    push    di
    mov     ax,3567h           ;Get EMS vector
    int     21h
    mov     di,0Ah             ;Using the segment from
    mov     si,offset Ema_header ;the 67h, look at offset
    mov     cx,8               ;0Ah Compare the next 8
    cld                          ;bytes with the expected
    repe    cmpsb               ;EMS header. If they are
    pop     di                  ;the same, EMS driver
    pop     es                  ;found
    je      CHECK4EMS_FOUND

CHECK4EMS_ERROR
    stc                          ;Get not found flag
    jmp     short CHECK4EMS_EXIT

CHECK4EMS_FOUND
    mov     ah,40h              ;Check status
    int     67h
    or      ah,ah
    jne     CHECK4EMS_ERROR

    mov     ah,41h              ;Get page frame segment
    int     67h
    or      ah,ah
    jne     CHECK4EMS_ERROR

    mov     ah,46h              ;Get EMU version number
    int     67h
    or      ah,ah
    jne     CHECK4EMS_ERROR
    mov     dl,al               ;Copy version number

CHECK4EMS_EXIT
    ret
check4ema    endp

;-----
; MESSAGE print a message to screen
; Entry,  DX - offset of '$' terminated message
;-----
CRLF$ db 13,10,'$'           ;New line characters

message      proc near
    assume   cs,Cseg,ds,Cseg
    mov     ah,09h             ;Print string
    int     21h

```

API: Application Program Interface. Conjunto de especificaciones, estructuras de datos y funciones que permiten la interacción entre un programa de aplicación (el que provee la API) y otros programas escritos por desarrolladores profesionales o por usuarios finales.

DEVICE DRIVER: Programa generalmente diseñado para hacerse cargo de los detalles de hardware relacionados con algún periférico, de tal forma que el manejo de ese periférico por parte del programador sea lo más transparente posible.

SERVICIOS DE MS-DOS NO DOCUMENTADOS PERO ESTABLES: los servicios (algunas funciones de la INT 21h, las banderas de DOS, etc) que han ido apareciendo en las diferentes versiones del sistema operativo y que siguen operando en la misma forma desde la primera versión, pero por no estar documentadas, Microsoft no se compromete a darles mantenimiento ni soporte. Si se utilizan estas funciones existe el riesgo de desarrollar programas que no serán capaces de correr en futuras versiones del sistema operativo que modifiquen la forma de utilizar estos servicios.

ISR: Interrupt Service Routine. También conocida como Interrupt Handler o manejador de interrupción. Es la rutina asociada a una determinada interrupción, la cual realiza los procesos que dicha interrupción tiene asignados.

MCB: Memory Control Block. Estructura de datos de 5 bytes de longitud, situada en el primer paragraph de todos los bloques de memoria reservada mediante el uso de funciones del sistema operativo.

PARAGRAPH: Párrafo de memoria, equivalente a 16 bytes.

PID: Process ID o identificador de proceso. Es simplemente el PSP del programa, el cual es utilizado por DOS para identificar al programa que está ejecutándose. Si varios programas están cargados en memoria al mismo tiempo, cada uno de ellos es considerado como un proceso independiente y por lo tanto cada uno tendrá asignado un PSP distinto, el cual servirá como PID para diferenciarlo de los demás.

PSP: Program Segment Prefix. Estructura de datos de 256 bytes de longitud, que sirve como área de comunicación entre el sistema operativo y el programa que está ejecutándose. MS-DOS crea un PSP para cada programa que ejecuta. Como su nombre lo indica es un "prefijo del programa", es decir, el PSP es cargado en memoria inmediatamente antes que el código del programa en cuestión.

PROGRAMA ROBUSTO: se refiere a la resistencia que tiene un programa contra las fallas. Entre más robusto menos posibilidad de fallar tendrá el programa.

REENTRADA: El llamar alguna función del sistema operativo mientras otra llamada está siendo procesada. MS-DOS no tiene capacidad para ser reentrado debido a la forma como maneja sus stacks internos.

STACK: Generalmente traducido como pila. Un stack sirve como almacenamiento temporal de datos, los cuales son manipulados de acuerdo con la convención LIFO (Last In First Out).

TASK SWITCHER: Conmutador de tareas. Programa que permite mantener cargados en memoria varios programas a la vez, de tal forma que es posible cambiar de un programa a otro rápidamente y sin tener que terminar la ejecución del anterior.

TSR: Terminate and Stay Resident. Con este nombre se conoce a los programas que terminan su ejecución y permanecen residentes en memoria.

UART: Universal Asynchronous Receiver Transmitter. Con este nombre se designa al circuito integrado, fabricado por Intel, que se utiliza para implementar aplicaciones de comunicaciones seriales con las computadoras.

VECTOR DE INTERRUPCION: Es la dirección en memoria (Segment:Offset) de la ISR asociada a la interrupción en cuestión. Todos los vectores se encuentran almacenados en la Tabla de Vectores de Interrupción, localizada en lo más bajo de la memoria convencional de la computadora.

WINDOWS NT: Windows New Technology. Sistema operativo multiusuario, multitareas de Microsoft, será liberado a principios de 1993.

(): Dentro de los listados presentados en este trabajo, los tres puntos reemplazan porciones de código no mostradas.

REFERENCIAS BIBLIOGRAFICAS

- [1] Borland Intl. TURBO PASCAL 6.0 PROGRAMMER'S GUIDE.
(USA., Scotts Valley, CA.: Borland International, 1990).
- [2] Borland Intl. TURBO PASCAL 6.0 LIBRARY REFERENCE.
(USA., Scotts Valley, CA.: Borland International, 1990).
- [3] Borland Intl. TURBO ASSEMBLER 2.0 USER'S GUIDE.
(USA., Scotts Valley, CA.: Borland International, 1990).
- [4] Borland Intl. TURBO ASSEMBLER 2.0 REFERENCE GUIDE.
(USA., Scotts Valley, CA.: Borland International, 1990).
- [5] Ezzell, B. WRITING OS/2 RESIDENT UTILITIES.
(USA., Reading, MA.: Addison Wesley, 1989).
- [6] Ferreyra, G. VIRUS EN LAS COMPUTADORAS.
(México, DF.: Macrobít Editores S.A. de C.V., 1991.).
- [7] Hogan, T. THE PROGRAMMER'S PC SOURCEBOOK
(USA., Redmond, WA.: Microsoft Press, 1988).
- [8] Holzener, S. DESKTOP WORKSHOP.
(USA., New York.: Prentice Hall, 1987).

- [9] Jamsa, K. DOS: POWER USER'S GUIDE.
(USA., Berkeley, CA.: Osborne McGraw-Hill, 1988).
- [10] Jourdain, R. PROGRAMMER'S PROBLEM SOLVER FOR THE IBM PC, XT & AT. (USA., New York.: Brady Books, 1986).
- [11] McAfee J. y Haynes, C. COMPUTER VIRUSES, WORMS, DATA DIDDERS, KILLER PROGRAMAS AND OTHER THREATS TO YOUR SYSTEM. (USA., New York.: St. Martin's Press, 1989).
- [12] Microsoft Corp. MICROSOFT MS-DOS. USER'S GUIDE AND REFERENCE. (USA., Redmond, WA.: Microsoft Corporation, 1991).
- [13] O'Brien, S.K. TURBO PASCAL: ADVANCED PROGRAMMER'S GUIDE. (USA., Berkeley, CA.: Borland Osborne McGraw-Hill, 1988).
- [14] Scanlon, L.J. 8086/8088/80286 ASSEMBLY LANGUAGE.
(USA., New York.: Brady Books, 1988).
- [15] Simrin, S. THE WAITE GROUP'S MS-DOS BIBLE.
(USA., Carmel, IN.: Howard W. Sams & Company, 1990).
- [16] Varios. THE WAITE GROUP'S MS-DOS PAPERS.
(USA., Carmel, IN.: Howard W. Sams & Company, 1990).
- [17] Wilton, R. PROGRAMMER'S GUIDE TO PC & PS/2 VIDEO SYSTEMS. (USA., Redmond, WA.: Microsoft Press, 1987).

Publicaciones periodicas.

- [18] Alford, R.C. "HOW INTERRUPTS WORK."
BYTE Febrero de 1992, p. 249-256.
- [19] Boling, D. "STRATEGIES AND TECHNIQUES FOR WRITING STATE-OF-THE-ART TSRs THAT EXPLOIT MS-DOS 5."
Microsoft Systems Journal Enero/Febrero de 1992, p. 41-59.

- UPAEP**
BIBLIOTECA CENTRAL
TESTES
DOCUMENTACIÓN EN SALA
- 
- [20] Boye, S.R. y Kain, P. "THE APPLICATION PROGRAM INTERFACE."
BYTE Edición Bonus de 1987, p. 19-23.
- [21] Claff, W.J. "WRITING ASSEMBLY LANGUAGE INTERRUPT
ROUTINES.", BYTE , Edición Extra de 1986, p. 249-262.
- [22] Duncan, R. "TSRs PAST AND FUTURE."
BYTE Edición Extra de 1987, p. 49-54.
- [23] Ladd, S.R. "A TURBO TSR."
BYTE Julio de 1988, p. 301-304.
- [24] Varios. "BEST OF BIX: IBM PC AND COMPATIBLES CONFERENCE
y PASCAL CONFERENCE.", BYTE Septiembre de 1986, p. 398-411.
- [25] Yancich, D.M. "USING EXPANDED MEMORY."
BYTE Edición Especial IBM de 1989, p. 123-128.

Otras referencias.

- [26] Borland Intl. THELP.DOC
Documentación en línea para la utilería residente Turbo Help
(THELP.COM). Borland International, 1990.
- [27] CECAFI INFORME VIRUS INFORMATICOS.
(México, DF.: Centro de Cálculo de la Facultad de Ingeniería.,
UNAM, 1989).
- [28] Radcliffe, M. JC. DESQview/X A TECHNICAL PERSPECTIVE.
(USA., Santa Monica, CA.: Quarterdeck Office Systems, 1990).
- [29] Varios. ASSEMBLY LANGUAGE DATABASE (ASM.NG).
Documentación en línea utilizando el programa residente Norton Guides.
(NG.EXE), Peter Norton Computing, Inc., 1987.