Article 11
Terminate-and-Stay-Resident Utilities

The MS-DOS Terminate and Stay Resident system calls (Interrupt 21H Function 31H and Interrupt 27H) allow the programmer to install executable code or program data in a reserved block of RAM, where it resides while other programs execute. Global data, interrupt handlers, and entire applications can be made RAM-resident in this way. Programs that use the MS-DOS terminate-and-stay-resident capability are commonly known as TSR programs or TSRs.

This article describes how to install a TSR in RAM, how to communicate with the resident program, and how the resident program can interact with MS-DOS. The discussion proceeds from a general description of the MS-DOS functions useful to TSR programmers to specific details about certain MS-DOS structural elements necessary to proper functioning of a TSR utility and concludes with two programming examples.

Note: Microsoft cannot guarantee that the information in this article will be valid for future versions of MS-DOS.

Structure of a Terminate-and-Stay-Resident Utility

The executable code and data in TSRs can be separated into RAM-resident and transient portions (Figure 11-1). The RAM-resident portion of a TSR contains executable code and data for an application that performs some useful function on demand. The transient portion installs the TSR; that is, it initializes data and interrupt handlers contained in the RAM-resident portion of the program and executes an MS-DOS Terminate and Stay Resident function call that leaves the RAM-resident portion in memory and frees the memory used by the transient portion. The code in the transient portion of a TSR runs when the .EXE or .COM file containing the program is executed; the code in the RAM-resident portion runs only when it is explicitly invoked by a foreground program or by execution of a hardware or software interrupt.

TSRs can be broadly classified as passive or active, depending on the method by which control is transferred to the RAM-resident program. A passive TSR executes only when another program explicitly transfers control to it, either through a software interrupt or by means of a long JMP or CALL. The calling program is not interrupted by the TSR, so the status of MS-DOS, the system BIOS, and the hardware is well defined when the TSR program starts to execute.

In contrast, an active TSR is invoked by the occurrence of some event external to the currently running (foreground) program, such as a sequence of user keystrokes or a predefined hardware interrupt. Therefore, when it is invoked, an active TSR almost always

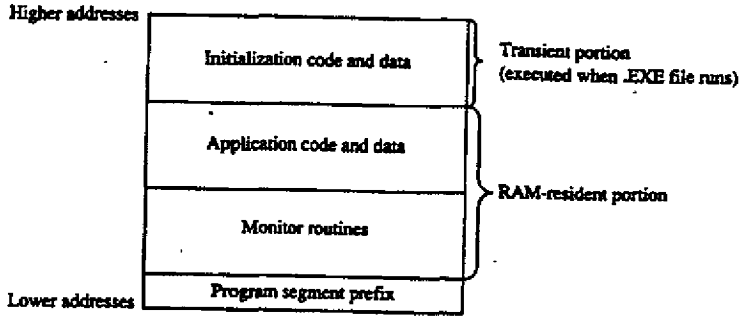

Figure 11-1. Organization of a TSR program in memory.

interrupts some other program and suspends its execution. To avoid disrupting the interrupted program, an active TSR must monitor the status of MS-DOS, the ROM BIOS, and the hardware and take control of the system only when it is safe to do so.

Passive TSRs are generally simpler in their construction than active TSRs because a passive TSR runs in the context of the calling program; that is, when the TSR executes, it assumes that it can use the calling program's program segment prefix (PSP), open files, current directory, and so on. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. It is the calling program's responsibility to ensure that the hardware and MS-DOS are in a stable state before it transfers control to a passive TSR.

An active TSR, on the other hand, is invoked asynchronously; that is, the status of the hardware, MS-DOS, and the executing foreground program is indeterminate when the event that invokes the TSR occurs. Therefore, active TSRs require more complex code. The RAM-resident portion of an active TSR must contain modules that monitor the operating system to determine when control can safely be transferred to the application portion of the TSR. The monitor routines typically test the status of keyboard input, ROM BIOS interrupt processing, hardware interrupt processing, and MS-DOS function processing. The TSR activates the application (the part of the RAM-resident portion that performs the TSR's main task) only when it detects the appropriate keyboard input and determines that the application will not interfere with interrupt and MS-DOS function processing.

Keyboard input

An active TSR usually contains a RAM-resident module that examines keyboard input for a predetermined keystroke sequence called a "hot-key" sequence. A user executes the RAM-resident application by entering this hot-key sequence at the keyboard.

The technique used in the TSR to monitor keyboard input depends on the keyboard hardware implementation. On computers in the IBM PC and PS/2 families, the keyboard coprocessor generates an Interrupt 09H for each keypress. Therefore, a TSR can monitor user keystrokes by installing an interrupt handler (interrupt service routine, or ISR) for Interrupt 09H. This handler can thus detect a specified hot-key sequence.

ROM BIOS interrupt processing

The ROM BIOS routines in IBM PCs and PS/2s are not reentrant. An active TSR that calls the ROM BIOS must ensure that its code does not attempt to execute a ROM BIOS function that was already being executed by the foreground process when the TSR program took control of the system.

The IBM ROM BIOS routines are invoked through software interrupts, so an active TSR can monitor the status of the ROM BIOS by replacing the default interrupt handlers with custom interrupt handlers that intercept the appropriate BIOS interrupts. Each of these interrupt handlers can maintain a status flag, which it increments before transferring control to the corresponding ROM BIOS routine and decrements when the ROM BIOS routine has finished executing. Thus, the TSR monitor routines can test these flags to determine when non-reentrant BIOS routines are executing.

Hardware interrupt processing

The monitor routines of an active TSR, which may themselves be executed as the result of a hardware interrupt, should not activate the application portion of the TSR if any other hardware interrupt is being processed. On IBM PCs, for example, hardware interrupts are processed in a prioritized sequence determined by an Intel 8259A Programmable Interrupt Controller. The 8259A does not allow a hardware interrupt to execute if a previous interrupt with the same or higher priority is being serviced. All hardware interrupt handlers include code that signals the 8259A when interrupt processing is completed. (The programming interface to the 8259A is described in IBM's *Technical Reference* manuals and in Intel's technical literature.)

If a TSR were to interrupt the execution of another hardware interrupt handler before the handler signaled the 8259A that it had completed its interrupt servicing, subsequent hardware interrupts could be inhibited indefinitely. Inhibition of high-priority hardware interrupts such as the timer tick (Interrupt 08H) or keyboard interrupt (Interrupt 09H) could cause a system crash. For this reason, an active TSR must monitor the status of all hardware interrupt processing by interrogating the 8259A to ensure that control is transferred to the RAM-resident application only when no other hardware interrupts are being serviced.

MS-DOS function processing

Unlike the IBM ROM BIOS routines, MS-DOS is reentrant to a limited extent. That is, there are certain times when MS-DOS's servicing of an Interrupt 21H function call invoked by a foreground process can be suspended so that the RAM-resident application can make an Interrupt 21H function call of its own. For this reason, an active TSR must monitor operating system activity to determine when it is safe for the TSR application to make its calls to MS-DOS.

MS-DOS Support for Terminate-and-Stay-Resident Programs

Several MS-DOS system calls are useful for supporting terminate-and-stay-resident utilities. These are listed in Table 11-1. *See* SYSTEM CALLS.

Table 11-1. MS-DOS Functions Useful in TSR Programs.

Function Name	Call With	Returns	Comment
Terminate and Stay Resident	AH = 31H AL = return code DX = size of resident program (in 16-byte paragraphs) INT 21H	Nothing	Preferred over Interrupt 27H with MS-DOS versions 2.x and later
Terminate and Stay Resident	CS = PSP DX = size of resident program (bytes) INT 27H	Nothing	Provided for compatibility with MS-DOS versions 1.x
Set Interrupt Vector	AH = 25H AL = interrupt number DS:DX = address of interrupt handler INT 21H	Nothing	
Get Interrupt Vector	AH = 35H AL = interrupt number INT 21H	ES:BX = address of interrupt handler	
Set PSP Address	AH = 50H BX = PSP segment INT 21H	Nothing	
Get PSP Address	AH = 51H INT 21H	BX = PSP segment	
Set Extended Error Information	AX = 5D0AH DS:DX = address of 11-word data structure: word 0: register AX as returned by Function 59H word 1: register BX word 2: register CX word 3: register DX word 4: register SI word 5: register DI word 6: register DS word 7: register ES words 8–0AH: reserved; should be 0 INT 21H	Nothing	MS-DOS versions 3.1 and later

(more)

Table 11-1. *Continued.*

Function Name	Call With	Returns	Comment
Get Extended Error Information	AH = 59H BX = 0 INT 21H	AX = extended error code BH = error class BL = suggested action CH = error locus	
Set Disk Transfer Area Address	AH = 1AH DS:DX = address of DTA INT 21H	Nothing	
Get Disk Transfer Area Address	AH = 2FH INT 21H	ES:BX = address of current DTA	
Get InDOS Flag Address	AH = 34H INT 21H	ES:BX = address of InDOS flag	

Terminate-and-stay-resident functions

MS-DOS provides two mechanisms for terminating the execution of a program while leaving a portion of it resident in RAM. The preferred method is to execute Interrupt 21H Function 31H.

Interrupt 21H Function 31H

When this Interrupt 21H function is called, the value in DX specifies the amount of RAM (in paragraphs) that is to remain allocated after the program terminates, starting at the program segment prefix (PSP). The function is similar to Function 4CH (Terminate Process with Return Code) in that it passes a return code in AL, but it differs in that open files are not automatically closed by Function 31H.

Interrupt 27H

When Interrupt 27H is executed, the value passed in DX specifies the number of bytes of memory required for the RAM-resident program. MS-DOS converts the value passed in DX from bytes to paragraphs, sets AL to zero, and jumps to the same code that would be executed for Interrupt 21H Function 31H. Interrupt 27H is less flexible than Interrupt 21H Function 31H because it limits the size of the program that can remain resident in RAM to 64 KB, it requires that CS point to the base of the PSP, and it does not pass a return code. Later versions of MS-DOS support Interrupt 27H primarily for compatibility with versions 1.x.

TSR RAM management

In addition to the RAM explicitly allocated to the TSR by means of the value in DX, the RAM allocated to the TSR's environment remains resident when the installation portion of the TSR program terminates. (The paragraph address of the environment is found at offset 2CH in the TSR's PSP.) Moreover, if the installation portion of a TSR program has used Interrupt 21H Function 48H (Allocate Memory Block) to allocate additional RAM, this memory also remains allocated when the program terminates. If the RAM-resident program does not need this additional RAM, the installation portion of the TSR program should free it explicitly by using Interrupt 21H Function 49H (Free Memory Block) before executing Interrupt 21H Function 31H.

Set and Get Interrupt Vector functions

Two Interrupt 21H function calls are available to inspect or update the contents of a specified 8086-family interrupt vector. Function 25H (Set Interrupt Vector) updates the vector of the interrupt number specified in the AL register with the segment and offset values specified in DS:DX. Function 35H (Get Interrupt Vector) performs the inverse operation: It copies the current vector of the interrupt number specified in AL into the ES:BX register pair.

Although it is possible to manipulate interrupt vectors directly, the use of Interrupt 21H Functions 25H and 35H is generally more convenient and allows for upward compatibility with future versions of MS-DOS.

Set and Get PSP Address functions

MS-DOS uses a program's PSP to keep track of certain data unique to the program, including command-line parameters and the segment address of the program's environment. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. To access this information, MS-DOS maintains an internal variable that always contains the location of the PSP associated with the foreground process. When a RAM-resident application is activated, it should use Interrupt 21H Functions 50H (Set Program Segment Prefix Address) and 51H (Get Program Segment Prefix Address) to preserve the current contents of this variable and to update the variable with the location of its own PSP. Function 50H (Set Program Segment Prefix Address) updates an internal MS-DOS variable that locates the PSP currently in use by the foreground process. Function 51H (Get Program Segment Prefix Address) returns the contents of the internal MS-DOS variable to the caller.

Set and Get Extended Error Information functions

In MS-DOS versions 3.1 and later, the RAM-resident program should preserve the foreground process's extended error information so that, if the RAM-resident application encounters an MS-DOS error, the extended error information pertaining to the foreground process will still be available and can be restored. Interrupt 21H Functions 59H and 5D0AH provide a mechanism for the RAM-resident program to save and restore this information during execution of a TSR application.

Function 59H (Get Extended Error Information), which became available in version 3.0, returns detailed information on the most recently detected MS-DOS error. The inverse operation is performed by Function 5D0AH (Set Extended Error Information), which can be used only in MS-DOS versions 3.1 and later. This function copies extended error information to MS-DOS from a data structure defined in the calling program.

Set and Get Disk Transfer Area Address functions

Several MS-DOS data transfer functions, notably Interrupt 21H Functions 21H, 22H, 27H, and 28H (the Random Read and Write functions) and Interrupt 21H Functions 14H and 15H (the Sequential Read and Write functions), require a program to specify a disk transfer area (DTA). By default, a program's DTA is located at offset 80H in its program segment prefix. If a RAM-resident application calls an MS-DOS function that uses a DTA, the TSR should save the DTA address belonging to the interrupted program by using Interrupt 21H Function 2FH (Get Disk Transfer Area Address), supply its own DTA address to MS-DOS using Interrupt 21H Function 1AH (Set Disk Transfer Area Address), and then, before terminating, restore the interrupted program's DTA.

The MS-DOS idle interrupt (Interrupt 28H)

Several of the first 12 MS-DOS functions (01H through 0CH) must wait for the occurrence of an expected event such as a user keypress. These functions contain an "idle loop" in which looping continues until the event occurs. To provide a mechanism for other system activity to take place while the idle loop is executing, these MS-DOS functions execute an Interrupt 28H from within the loop.

The default MS-DOS handler for Interrupt 28H is only an IRET instruction. By supplying its own handler for Interrupt 28H, a TSR can perform some useful action at times when MS-DOS is otherwise idle. Specifically, a custom Interrupt 28H handler can be used to examine the current status of the system to determine whether or not it is safe to activate the RAM-resident application.

Determining MS-DOS Status

A TSR can infer the current status of MS-DOS from knowledge of its internal use of stacks and from a pair of internal status flags. This status information is essential to the proper execution of an active TSR because a RAM-resident application can make calls to MS-DOS only when those calls will not disrupt an earlier call made by the foreground process.

MS-DOS internal stacks

MS-DOS versions 2.0 and later may use any of three internal stacks: the I/O stack (*IOStack*), the disk stack (*DiskStack*), and the auxiliary stack (*AuxStack*). In general, *IOStack* is used for Interrupt 21H Functions 01H through 0CH and *DiskStack* is used for the remaining Interrupt 21H functions; *AuxStack* is normally used only when MS-DOS has detected a critical error and subsequently executed an Interrupt 24H. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers. Specifically, MS-DOS's internal stack use depends on which MS-DOS function is being executed and on the value of the critical error flag.

The critical error flag

The critical error flag (*ErrorMode*) is a 1-byte flag that MS-DOS uses to indicate whether or not a critical error has occurred. During normal, errorless execution, the value of the critical error flag is zero. Whenever MS-DOS detects a critical error, it sets this flag to a nonzero value before it executes Interrupt 24H. If an Interrupt 24H handler subsequently invokes an MS-DOS function by using Interrupt 21H, the nonzero value of the critical error flag tells MS-DOS to use its auxiliary stack for Interrupt 21H Functions 01H through 0CH instead of using the I/O stack as it normally would.

In other words, when control is transferred to MS-DOS through Interrupt 21H, the function number and the critical error flag together determine MS-DOS stack use for the function. Figure 11-2 outlines the internal logic used on entry to an MS-DOS function to select which stack is to be used during processing of the function. As stated above, for Functions 01H through 0CH, MS-DOS uses *IOStack* if the critical error flag is zero and *AuxStack* if the flag is nonzero. For function numbers greater than 0CH, MS-DOS usually uses *DiskStack*, but Functions 50H, 51H, and 59H are important exceptions. Functions 50H and 51H use either *IOStack* (in versions 2.x) or the stack supplied by the calling program (in versions 3.x). In version 3.0, Function 59H uses either *IOStack* or *AuxStack*, depending on the value of the critical error flag, but in versions 3.1 and later, Function 59H always uses *AuxStack*.

MS-DOS versions 2.x

```
if    (FunctionNumber >= 01H and FunctionNumber <= 0CH)
      or
      FunctionNumber = 50H
      or
      FunctionNumber = 51H

then if    ErrorMode = 0
      then use IOStack
      else use AuxStack

else ErrorMode = 0
      use DiskStack
```

MS-DOS version 3.0

```
if    FunctionNumber = 50H
      or
      FunctionNumber = 51H
      or
      FunctionNumber = 62H

then use caller's stack

else if    (FunctionNumber >= 01H and FunctionNumber <= 0CH)
           or
           Function Number = 59H

      then if    ErrorMode = 0
           then use IOStack
           else use AuxStack

      else ErrorMode = 0
           use DiskStack
```

Figure 11-2. Strategy for use of MS-DOS internal stacks. *(more)*

MS-DOS versions 3.1 and later

```
if    FunctionNumber = 33H
      or
      FunctionNumber = 50H
      or
      FunctionNumber = 51H
      or
      FunctionNumber = 62H

then use caller's stack

else if     (FunctionNumber >= 01H and FunctionNumber <= 0CH)

      then if     ErrorMode = 0
            then use IOStack
            else use AuxStack

      else if FunctionNumber = 59H
            then use AuxStack
            else ErrorMode = 0
                 use DiskStack
```

Figure 11-2. Continued.

This scheme makes Functions 01H through 0CH reentrant in a limited sense, in that a substitute critical error (Interrupt 24H) handler invoked while the critical error flag is nonzero can still use these Interrupt 21H functions. In this situation, because the flag is nonzero, *AuxStack* is used for Functions 01H through 0CH instead of *IOStack*. Thus, if *IOStack* is in use when the critical error is detected, its contents are preserved during the handler's subsequent calls to these functions.

The stack-selection logic differs slightly between MS-DOS versions 2 and 3. In versions 3.x, a few functions—notably 50H and 51H—avoid using any of the MS-DOS stacks. These functions perform uncomplicated tasks that make minimal demands for stack space, so the calling program's stack is assumed to be adequate for them.

The InDOS flag

InDOS is a 1-byte flag that is incremented each time an Interrupt 21H function is invoked and decremented when the function terminates. The flag's value remains nonzero as long as code within MS-DOS is being executed. The value of InDOS does not indicate which internal stack MS-DOS is using.

Whenever MS-DOS detects a critical error, it zeros InDOS before it executes Interrupt 24H. This action is taken to accommodate substitute Interrupt 24H handlers that do not return control to MS-DOS. If InDOS were not zeroed before such a handler gained control, its value would never be decremented and would therefore be incorrect during subsequent calls to MS-DOS.

The address of the 1-byte InDOS flag can be obtained from MS-DOS by using Interrupt 21H Function 34H (Return Address of InDOS Flag). In versions 3.1 and later, the 1-byte critical error flag is located in the byte preceding InDOS, so, in effect, the address of both flags can be found using Function 34H. Unfortunately, there is no easy way to find the critical error flag in other versions. The recommended technique is to scan the MS-DOS segment, which is returned in the ES register by Function 34H, for one of the following sequences of instructions:

```
test    ss:[CriticalErrorFlag],0FFH        ;(versions 3.1 and later)
jne     NearLabel
push    ss:[NearWord]
int     28H
```

or

```
cmp     ss:[CriticalErrorFlag],00          ;(versions earlier than 3.1)
jne     NearLabel
int     28H
```

When the TEST or CMP instruction has been identified, the offset of the critical error flag can be obtained from the instruction's operand field.

The Multiplex Interrupt

The MS-DOS multiplex interrupt (Interrupt 2FH) provides a general mechanism for a program to verify the presence of a TSR and communicate with it. A program communicates with a TSR by placing an identification value in AH and a function number in AL and issuing an Interrupt 2FH. The TSR's Interrupt 2FH handler compares the value in AH to its own predetermined ID value. If they match, the TSR's handler keeps control and performs the function specified in the AL register. If they do not match, the TSR's handler relinquishes control to the previously installed Interrupt 2FH handler. (Multiplex ID values 00H through 7FH are reserved for use by MS-DOS; therefore, user multiplex numbers should be in the range 80H through 0FFH.)

The handler in the following example recognizes only one function, corresponding to AL = 00H. In this case, the handler returns the value 0FFH in AL, signifying that the handler is indeed resident in RAM. Thus, a program can detect the presence of the handler by executing Interrupt 2FH with the handler's ID value in AH and 00H in AL.

```
mov     ah,MultiplexID
mov     al,00H
int     2FH
cmp     al,0FFH
je      AlreadyInstalled
```

To ensure that the identification byte is unique, its value should be determined at the time the TSR is installed. One way to do this is to pass the value to the TSR program as a command-line parameter when the TSR program is installed. Another approach is to place the identification value in an environment variable. In this way, the value can be found in the environment of both the TSR and any other program that calls Interrupt 2FH to verify the TSR's presence.

In practice, the multiplex interrupt can also be used to pass information to and from a RAM-resident program in the CPU registers, thus providing a mechanism for a program to share control or status information with a TSR.

TSR Programming Examples

One effective way to become familiar with TSRs is to examine functional programs. Therefore, the subsequent pages present two examples: a simple passive TSR and a more complex active TSR.

HELLO.ASM

The "bare-bones" TSR in Figure 11-3 is a passive TSR. The RAM-resident application, which simply displays the message *Hello, World*, is invoked by executing a software interrupt. This example illustrates the fundamental interactions among a RAM-resident program, MS-DOS, and programs that execute after the installation of the RAM-resident utility.

```
;
; Name:           hello
;
; Description:    This RAM-resident (terminate-and-stay-resident) utility
;                 displays the message "Hello, World" in response to a
;                 software interrupt.
;
; Comments:       Assemble and link to create HELLO.EXE.
;
;                 Execute HELLO.EXE to make resident.
;
;                 Execute  INT 64h  to display the message.
;


TSRInt          EQU     64h
STDOUT          EQU     1

RESIDENT_TEXT   SEGMENT byte public 'CODE'
                ASSUME  cs:RESIDENT_TEXT,ds:RESIDENT_DATA

TSRAction       PROC    far

                sti                     ; enable interrupts

                push    ds              ; preserve registers
                push    ax
                push    bx
                push    cx
                push    dx
```

Figure 11-3. HELLO.ASM, a passive TSR.

(more)

```
                mov     dx,seg RESIDENT_DATA
                mov     ds,dx
                mov     dx,offset Message       ; DS:DX -> message
                mov     cx,16                   ; CX = length
                mov     bx,STDOUT               ; BX = file handle
                mov     ah,40h                  ; AH = INT 21H function 40H
                                                ; (Write File)
                int     21h                     ; display the message

                pop     dx                      ; restore registers and exit
                pop     cx
                pop     bx
                pop     ax
                pop     ds
                iret

TSRAction       ENDP

RESIDENT_TEXT   ENDS


RESIDENT_DATA   SEGMENT word public 'DATA'

Message         DB      0Dh,0Ah,'Hello, World',0Dh,0Ah

RESIDENT_DATA   ENDS


TRANSIENT_TEXT  SEGMENT para public 'TCODE'
                ASSUME  cs:TRANSIENT_TEXT,ss:TRANSIENT_STACK

HelloTSR PROC   far                     ; At entry:     CS:IP -> SnapTSR
                                        ;               SS:SP -> stack
                                        ;               DS,ES -> PSP
; Install this TSR's interrupt handler

                mov     ax,seg RESIDENT_TEXT
                mov     ds,ax
                mov     dx,offset RESIDENT_TEXT:TSRAction
                mov     al,TSRInt
                mov     ah,25h
                int     21h

; Terminate and stay resident

                mov     dx,cs           ; DX = paragraph address of start of
                                        ; transient portion (end of resident
                                        ; portion)
                mov     ax,es           ; ES = PSP segment
                sub     dx,ax           ; DX = size of resident portion
```

Figure 11-3. Continued.

(more)

```
                mov     ax,3100h        ; AH = INT 21H function number (TSR)
                                        ; AL = 00H (return code)
                int     21h

HelloTSR        ENDP

TRANSIENT_TEXT  ENDS


TRANSIENT_STACK SEGMENT word stack 'TSTACK'

                DB      80h dup(?)

TRANSIENT_STACK ENDS


                END     HelloTSR
```

Figure 11-3. Continued.

The transient portion of the program (in the segments *TRANSIENT_TEXT* and *TRANSIENT_STACK*) runs only when the file HELLO.EXE is executed. This installation code updates an interrupt vector to point to the resident application (the procedure *TSRAction*) and then calls Interrupt 21H Function 31H to terminate execution, leaving the segments *RESIDENT_TEXT* and *RESIDENT_DATA* in RAM.

The order in which the code and data segments appear in the listing is important. It ensures that when the program is executed as a .EXE file, the resident code and data are placed in memory at lower addresses than the transient code and data. Thus, when Interrupt 21H Function 31H is called, the memory occupied by the transient portion of the program is freed without disrupting the code and data in the resident portion.

The RAM containing the resident portion of the utility is left intact by MS-DOS during subsequent execution of other programs. Thus, after the TSR has been installed, any program that issues the software interrupt recognized by the TSR (in this example, Interrupt 64H) will transfer control to the routine *TSRAction*, which uses Interrupt 21H Function 40H to display a simple message on standard output.

Part of the reason this example is so short is that it performs no error checking. A truly reliable version of the program would check the version of MS-DOS in use, verify that the program was not already installed in memory, and chain to any previously installed interrupt handlers that use the same interrupt vector. (The next program, SNAP.ASM, illustrates these techniques.) However, the primary reason the program is small is that it makes the basic assumption that MS-DOS, the ROM BIOS, and the hardware interrupts are all stable at the time the resident utility is executed.

SNAP.ASM

The preceding assumption is a reliable one in the case of the passive TSR in Figure 11-3, which executes only when it is explicitly invoked by a software interrupt. However, the situation is much more complicated in the case of the active TSR in Figure 11-4. This program is relatively long because it makes no assumptions about the stability of the operating environment. Instead, it monitors the status of MS-DOS, the ROM BIOS, and the hardware interrupts to decide when the RAM-resident application can safely execute.

```
;
; Name:         snap
;
; Description:  This RAM-resident (terminate-and-stay-resident) utility
;               produces a video "snapshot" by copying the contents of the
;               video regeneration buffer to a disk file.  It may be used
;               in 80-column alphanumeric video modes on IBM PCs and PS/2s.
;
; Comments:     Assemble and link to create SNAP.EXE.
;
;               Execute SNAP.EXE to make resident.
;
;               Press Alt-Enter to dump current contents of video buffer
;               to a disk file.
;

MultiplexID     EQU     0CAh            ; unique INT 2FH ID value

TSRStackSize    EQU     100h            ; resident stack size in bytes

KB_FLAG         EQU     17h             ; offset of shift-key status flag in
                                        ; ROM BIOS keyboard data area

KBIns           EQU     80h             ; bit masks for KB_FLAG
KBCaps          EQU     40h
KBNum           EQU     20h
KBScroll        EQU     10h
KBAlt           EQU     8
KBCtl           EQU     4
KBLeft          EQU     2
KBRight         EQU     1

SCEnter         EQU     1Ch

CR              EQU     0Dh
LF              EQU     0Ah
TRUE            EQU     -1
FALSE           EQU     0

                PAGE
;----------------------------------------------------------------------------
;
; RAM-resident routines
;
;----------------------------------------------------------------------------

RESIDENT_GROUP  GROUP   RESIDENT_TEXT,RESIDENT_DATA,RESIDENT_STACK
```

Figure 11-4. SNAP.ASM, a video snapshot TSR.

(more)

MSC 00131482

```
RESIDENT_TEXT   SEGMENT byte public 'CODE'
                ASSUME  cs:RESIDENT_GROUP,ds:RESIDENT_GROUP

;----------------------------------------------------------------------------
; System verification routines
;----------------------------------------------------------------------------

VerifyDOSState  PROC    near            ; Returns:    carry flag set if MS-DOS
                                        ;             is busy
                push    ds              ; preserve these registers
                push    bx
                push    ax

                lds     bx,cs:ErrorModeAddr
                mov     ah,[bx]         ; AH = ErrorMode flag

                lds     bx,cs:InDOSAddr
                mov     al,[bx]         ; AL = InDOS flag

                xor     bx,bx           ; BH = 00H, BL = 00H
                cmp     bl,cs:InISR28   ; carry flag set if INT 28H handler
                                        ; is running
                rcl     bl,01h          ; BL = 01H if INT 28H handler is running

                cmp     bx,ax           ; carry flag zero if AH = 00H
                                        ; and AL <= BL
                pop     ax              ; restore registers
                pop     bx
                pop     ds
                ret

VerifyDOSState  ENDP


VerifyIntState  PROC    near            ; Returns:    carry flag set if hardware
                                        ;             or ROM BIOS unstable

                push    ax              ; preserve AX

; Verify hardware interrupt status by interrogating Intel 8259A Programmable
;  Interrupt Controller

                mov     ax,00001011b    ; AH = 0
                                        ; AL = 0CW3 for Intel 8259A (RR = 1,
                                        ; RIS = 1)
                out     20h,al          ; request 8259A's in-service register
                jmp     short L10       ; wait a few cycles
L10:            in      al,20h          ; AL = hardware interrupts currently
                                        ; being serviced (bit = 1 if in-service)
```

Figure 11-4. Continued.

(more)

```
                cmp     ah,al
                jc      L11                 ; exit if any hardware interrupts still
                                            ; being serviced

; Verify status of ROM BIOS interrupt handlers

                xor     al,al               ; AL = 00H

                cmp     al,cs:InISR5
                jc      L11                 ; exit if currently in INT 05H handler

                cmp     al,cs:InISR9
                jc      L11                 ; exit if currently in INT 09H handler

                cmp     al,cs:InISR10
                jc      L11                 ; exit if currently in INT 10H handler

                cmp     al,cs:InISR13       ; set carry flag if currently in
                                            ; INT 13H handler
L11:            pop     ax                  ; restore AX and return
                ret

VerifyIntState  ENDP


VerifyTSRState  PROC    near                ; Returns: carry flag set if TSR
                                            ;          inactive
                rol     cs:HotFlag,1        ; carry flag set if (HotFlag = TRUE)
                cmc                         ; carry flag set if (HotFlag = FALSE)
                jc      L20                 ; exit if no hot key

                ror     cs:ActiveTSR,1      ; carry flag set if (ActiveTSR = TRUE)
                jc      L20                 ; exit if already active

                call    VerifyDOSState
                jc      L20                 ; exit if MS-DOS unstable

                call    VerifyIntState      ; set carry flag if hardware or BIOS
                                            ; unstable
L20:            ret

VerifyTSRState  ENDP

                PAGE
;-----------------------------------------------------------------------------
; System monitor routines
;-----------------------------------------------------------------------------

ISR5            PROC    far                 ; INT 05H handler
                                            ; (ROM BIOS print screen)
                inc     cs:InISR5           ; increment status flag
```

Figure 11-4. Continued.

(more)

```
                pushf
                cli
                call    cs:PrevISR5     ; chain to previous INT 05H handler

                dec     cs:InISR5       ; decrement status flag
                iret

ISR5            ENDP


ISR8            PROC    far             ; INT 08H handler (timer tick, IRQ0)

                pushf
                cli
                call    cs:PrevISR8     ; chain to previous handler

                cmp     cs:InISR8,0
                jne     L31             ; exit if already in this handler

                inc     cs:InISR8       ; increment status flag

                sti                     ; interrupts are ok
                call    VerifyTSRState
                jc      L30             ; jump if TSR is inactive

                mov     byte ptr cs:ActiveTSR,TRUE
                call    TSRapp
                mov     byte ptr cs:ActiveTSR,FALSE

L30:            dec     cs:InISR8

L31:            iret

ISR8            ENDP


ISR9            PROC    far             ; INT 09H handler
                                        ; (keyboard interrupt IRQ1)
                push    ds              ; preserve these registers
                push    ax
                push    bx

                push    cs
                pop     ds              ; DS -> RESIDENT_GROUP

                in      al,60h          ; AL = current scan code

                pushf                   ; simulate an INT
                cli
                call    ds:PrevISR9     ; let previous handler execute
```

Figure 11-4. Continued. *(more)*

```
                mov     ah,ds:InISR9        ; if already in this handler ..
                or      ah,ds:HotFlag       ; .. or currently processing hot key ..
                jnz     L43                 ; .. jump to exit

                inc     ds:InISR9           ; increment status flag
                sti                         ; now interrupts are ok

; Check scan code sequence

                cmp     ds:HotSeqLen,0
                je      L40                 ; jump if no hot sequence to match

                mov     bx,ds:HotIndex
                cmp     al,[bx+HotSequence]     ; test scan code sequence
                jne     L41                 ; jump if no match

                inc     bx
                cmp     bx,ds:HotSeqLen
                jb      L42                 ; jump if not last scan code to match

; Check shift-key state

L40:            push    ds
                mov     ax,40h
                mov     ds,ax               ; DS -> ROM BIOS data area
                mov     al,ds:[KB_FLAG]     ; AH = ROM BIOS shift-key flags
                pop     ds

                and     al,ds:HotKBMask     ; AL = flags AND "don't care" mask
                cmp     al,ds:HotKBFlag
                jne     L42                 ; jump if shift state does not match

; Set flag when hot key is found

                mov     byte ptr ds:HotFlag,TRUE

L41:            xor     bx,bx               ; reinitialize index

L42:            mov     ds:HotIndex,bx      ; update index into sequence
                dec     ds:InISR9           ; decrement status flag

L43:            pop     bx                  ; restore registers and exit
                pop     ax
                pop     ds
                iret

ISR9            ENDP
```

Figure 11-4. Continued. *(more)*

```
ISR10           PROC    far             ; INT 10H handler (ROM BIOS video I/O)

                inc     cs:InISR10      ; increment status flag

                pushf
                cli
                call    cs:PrevISR10    ; chain to previous INT 10H handler

                dec     cs:InISR10      ; decrement status flag
                iret

ISR10           ENDP


ISR13           PROC    far             ; INT 13H handler
                                        ; (ROM BIOS fixed disk I/O)
                inc     cs:InISR13      ; increment status flag

                pushf
                cli
                call    cs:PrevISR13    ; chain to previous INT 13H handler

                pushf                   ; preserve returned flags
                dec     cs:InISR13      ; decrement status flag
                popf                    ; restore flags register

                sti                     ; enable interrupts
                ret     2               ; simulate IRET without popping flags

ISR13           ENDP


ISR1B           PROC    far             ; INT 1BH trap (ROM BIOS Ctrl-Break)

                mov     byte ptr cs:Trap1B,TRUE
                iret

ISR1B           ENDP


ISR23           PROC    far             ; INT 23H trap (MS-DOS Ctrl-C)

                mov     byte ptr cs:Trap23,TRUE
                iret

ISR23           ENDP


ISR24           PROC    far             ; INT 24H trap (MS-DOS critical error)

                mov     byte ptr cs:Trap24,TRUE
```

Figure 11-4. Continued. *(more)*

```
                xor     al,al           ; AL = 00H (MS-DOS 2.x):
                cmp     cs:MajorVersion,2 ; ignore the error
                je      L50

                mov     al,3            ; AL = 03H (MS-DOS 3.x):
                                        ; fail the MS-DOS call in which
                                        ; the critical error occurred
L50:            iret

ISR24           ENDP


ISR28           PROC    far             ; INT 28H handler
                                        ; (MS-DOS idle interrupt)
                pushf
                cli
                call    cs:PrevISR28    ; chain to previous INT 28H handler

                cmp     cs:InISR28,0
                jne     L61             ; exit if already inside this handler

                inc     cs:InISR28      ; increment status flag

                call    VerifyTSRState
                jc      L60             ; jump if TSR is inactive

                mov     byte ptr cs:ActiveTSR,TRUE
                call    TSRapp
                mov     byte ptr cs:ActiveTSR,FALSE

L60:            dec     cs:InISR28      ; decrement status flag

L61:            iret

ISR28           ENDP


ISR2F           PROC    far             ; INT 2FH handler
                                        ; (MS-DOS multiplex interrupt)
                                        ; Caller:  AH = handler ID
                                        ;          AL = function number
                                        ; Returns for function 0:  AL = 0FFH
                                        ; for all other functions:  nothing

                cmp     ah,MultiplexID
                je      L70             ; jump if this handler is requested

                jmp     cs:PrevISR2F    ; chain to previous INT 2FH handler
```

Figure 11-4. Continued.

(more)

```
L70:            test    al,al
                jnz     MultiplexIRET   ; jump if reserved or undefined function

; Function 0:   get installed state

                mov     al,0FFh         ; AL = 0FFH (this handler is installed)

MultiplexIRET:  iret                    ; return from interrupt

ISR2F           ENDP

                PAGE
;
;
; AuxInt21--sets ErrorMode while executing INT 21H to force use of the
;       AuxStack instead of the IOStack.
;
;

AuxInt21        PROC    near            ; Caller:     registers for INT 21H
                                        ; Returns:    registers from INT 21H

                push    ds
                push    bx
                lds     bx,ErrorModeAddr
                inc     byte ptr [bx]   ; ErrorMode is now nonzero
                pop     bx
                pop     ds

                int     21h             ; perform MS-DOS function

                push    ds
                push    bx
                lds     bx,ErrorModeAddr
                dec     byte ptr [bx]   ; restore ErrorMode
                pop     bx
                pop     ds
                ret

AuxInt21        ENDP

Int21v          PROC    near            ; perform INT 21H or AuxInt21,
                                        ; depending on MS-DOS version

                cmp     DOSVersion,30Ah
                jb      L80             ; jump if earlier than 3.1

                int     21h             ; versions 3.1 and later
                ret
```

Figure 11-4. Continued.

(more)

```
L80:            call    AuxInt21        ; versions earlier than 3.1
                ret

Int21v          ENDP

                PAGE
;-----------------------------------------------------------------------------
; RAM-resident application
;-----------------------------------------------------------------------------

TSRapp          PROC    near

; Set up a safe stack

                push    ds              ; save previous DS on previous stack

                push    cs
                pop     ds              ; DS -> RESIDENT_GROUP

                mov     PrevSP,sp       ; save previous SS:SP
                mov     PrevSS,ss

                mov     ss,TSRSS        ; SS:SP -> RESIDENT_STACK
                mov     sp,TSRSP

                push    es              ; preserve remaining registers
                push    ax
                push    bx
                push    cx
                push    dx
                push    si
                push    di
                push    bp

                cld                     ; clear direction flag

; Set break and critical error traps

                mov     cx,NTrap
                mov     si,offset RESIDENT_GROUP:StartTrapList

L90:            lodsb                   ; AL = interrupt number
                                        ; DS:SI -> byte past interrupt number

                mov     byte ptr [si],FALSE     ; zero the trap flag

                push    ax              ; preserve AX
                mov     ah,35h          ; INT 21H function 35H
                                        ; (get interrupt vector)
                int     21h             ; ES:BX = previous interrupt vector
                mov     [si+1],bx       ; save offset and segment ..
                mov     [si+3],es       ;   .. of previous handler
```

Figure 11-4. Continued.

(more)

MSC 00131490

MS-PCA 1186447
CONFIDENTIAL

```
                pop     ax              ; AL = interrupt number
                mov     dx,[si+5]       ; DS:DX -> this TSR's trap
                mov     ah,25h          ; INT 21H function 25H
                int     21h             ; (set interrupt vector)
                add     si,7            ; DS:SI -> next in list

                loop    L90

; Disable MS-DOS break checking during disk I/O

                mov     ax,3300h        ; AH = INT 21H function number
                                        ; AL = 00H (request current break state)
                int     21h             ; DL = current break state
                mov     PrevBreak,dl    ; preserve current state

                xor     dl,dl           ; DL = 00H (disable disk I/O break
                                        ; checking)
                mov     ax,3301h        ; AL = 01H (set break state)
                int     21h

; Preserve previous extended error information

                cmp     DOSVersion,30Ah
                jb      L91             ; jump if MS-DOS version earlier
                                        ; than 3.1
                push    ds              ; preserve DS
                xor     bx,bx           ; BX = 00H (required for function 59H)
                mov     ah,59h          ; INT 21H function 59H
                call    Int21v          ; (get extended error info)

                mov     cs:PrevExtErrDS,ds
                pop     ds
                mov     PrevExtErrAX,ax ; preserve error information
                mov     PrevExtErrBX,bx ; in data structure
                mov     PrevExtErrCX,cx
                mov     PrevExtErrDX,dx
                mov     PrevExtErrSI,si
                mov     PrevExtErrDI,di
                mov     PrevExtErrES,es

; Inform MS-DOS about current PSP

L91:            mov     ah,51h          ; INT 21H function 51H (get PSP address)
                call    Int21v          ; BX = foreground PSP

                mov     PrevPSP,bx      ; preserve previous PSP

                mov     bx,TSRPSP       ; BX = resident PSP
                mov     ah,50h          ; INT 21H function 50H (set PSP address)
                call    Int21v
```

Figure 11-4. Continued. *(more)*

```
; Inform MS-DOS about current DTA (not really necessary in this application
; because DTA is not used)

                mov     ah,2Fh          ; INT 21H function 2FH
                int     21h             ; (get DTA address) into ES:BX
                mov     PrevDTAoffs,bx
                mov     PrevDTAseg,es

                push    ds              ; preserve DS
                mov     ds,TSRPSP
                mov     dx,80h          ; DS:DX -> default DTA at PSP:0080H
                mov     ah,1Ah          ; INT 21H function 1AH
                int     21h             ; (set DTA address)
                pop     ds              ; restore DS

; Open a file, write to it, and close it

                mov     ax,0E07h        ; AH = INT 10H function number
                                        ; (write teletype)
                                        ; AL = 07H (bell character)
                int     10h             ; emit a beep

                mov     dx,offset RESIDENT_GROUP:SnapFile
                mov     ah,3Ch          ; INT 21H function 3CH
                                        ; (create file handle)
                mov     cx,0            ; file attribute
                int     21h
                jc      L94             ; jump if file not opened

                push    ax              ; push file handle
                mov     ah,0Fh          ; INT 10H function 0FH (get video status)
                int     10h             ; AL = video mode number
                                        ; AH = number of character columns
                pop     bx              ; BX = file handle

                cmp     ah,80
                jne     L93             ; jump if not 80-column mode

                mov     dx,0B800h       ; DX = color video buffer segment
                cmp     al,3
                jbe     L92             ; jump if color alphanumeric mode

                cmp     al,7
                jne     L93             ; jump if not monochrome mode

                mov     dx,0B000h       ; DX = monochrome video buffer segment

L92:            push    ds
                mov     ds,dx
                xor     dx,dx           ; DS:DX -> start of video buffer
                mov     cx,80*25*2      ; CX = number of bytes to write
                mov     ah,40h          ; INT 21H function 40H (write file)
```

Figure 11-4. Continued. *(more)*

```
                int     21h
                pop     ds

L93:            mov     ah,3Eh          ; INT 21H function 3EH (close file)
                int     21h

                mov     ax,0E07h        ; emit another beep
                int     10h

; Restore previous DTA

L94:            push    ds              ; preserve DS
                lds     dx,PrevDTA      ; DS:DX -> previous DTA
                mov     ah,1Ah          ; INT 21H function 1AH (set DTA address)
                int     21h
                pop     ds

; Restore previous PSP

                mov     bx,PrevPSP      ; BX = previous PSP
                mov     ah,50h          ; INT 21H function 50H
                call    Int21v          ; (set PSP address)

; Restore previous extended error information

                mov     ax,DOSVersion
                cmp     ax,30Ah
                jb      L95             ; jump if MS-DOS version earlier than 3.1
                cmp     ax,0A00h
                jae     L95             ; jump if MS OS/2-DOS 3.x box

                mov     dx,offset RESIDENT_GROUP:PrevExtErrInfo
                mov     ax,5D0Ah
                int     21h             ; (restore extended error information)

; Restore previous MS-DOS break checking

L95:            mov     dl,PrevBreak    ; DL = previous state
                mov     ax,3301h
                int     21h

; Restore previous break and critical error traps

                mov     cx,NTrap
                mov     si,offset RESIDENT_GROUP:StartTrapList
                push    ds              ; preserve DS

L96:            lods    byte ptr cs:[si] ; AL = interrupt number
                                        ; ES:SI -> byte past interrupt number

                lds     dx,cs:[si+1]    ; DS:DX -> previous handler
                mov     ah,25h          ; INT 21H function 25H
                int     21h             ; (set interrupt vector)
```

Figure 11-4. Continued.

(more)

```
                add     si,7                    ; DS:SI -> next in list
                loop    L96
                pop     ds                      ; restore DS

; Restore all registers

                pop     bp
                pop     di
                pop     si
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                pop     es

                mov     ss,PrevSS               ; SS:SP -> previous stack
                mov     sp,PrevSP
                pop     ds                      ; restore previous DS

; Finally, reset status flag and return

                mov     byte ptr cs:HotFlag,FALSE
                ret

TSRapp          ENDP

RESIDENT_TEXT   ENDS

RESIDENT_DATA   SEGMENT word public 'DATA'

ErrorModeAddr   DD      ?               ; address of MS-DOS ErrorMode flag
InDOSAddr       DD      ?               ; address of MS-DOS InDOS flag

NISR            DW      (EndISRList-StartISRList)/8 ; number of installed ISRs

StartISRList    DB      05h             ; INT number
InISR5          DB      FALSE           ; flag
PrevISR5        DD      ?               ; address of previous handler
                DW      offset RESIDENT_GROUP:ISR5

                DB      08h
InISR8          DB      FALSE
PrevISR8        DD      ?
                DW      offset RESIDENT_GROUP:ISR8

                DB      09h
InISR9          DB      FALSE
PrevISR9        DD      ?
                DW      offset RESIDENT_GROUP:ISR9

                DB      10h
InISR10         DB      FALSE
```

Figure 11-4. Continued.

(more)

```
PrevISR10       DD      ?
                DW      offset RESIDENT_GROUP:ISR10

                DB      13h
InISR13         DB      FALSE
PrevISR13       DD      ?
                DW      offset RESIDENT_GROUP:ISR13

                DB      28h
InISR28         DB      FALSE
PrevISR28       DD      ?
                DW      offset RESIDENT_GROUP:ISR28

                DB      2Fh
InISR2F         DB      FALSE
PrevISR2F       DD      ?
                DW      offset RESIDENT_GROUP:ISR2F

EndISRList      LABEL   BYTE

TSRPSP          DW      ?               ; resident PSP
TSRSP           DW      TSRStackSize    ; resident SS:SP
TSRSS           DW      seg RESIDENT_STACK
PrevPSP         DW      ?               ; previous PSP
PrevSP          DW      ?               ; previous SS:SP
PrevSS          DW      ?

HotIndex        DW      0               ; index of next scan code in sequence
HotSeqLen       DW      EndHotSeq-HotSequence   ; length of hot-key sequence

HotSequence     DB      SCEnter         ; hot sequence of scan codes
EndHotSeq       LABEL   BYTE

HotKBFlag       DB      KBAlt           ; hot value of ROM BIOS KB_FLAG
HotKBMask       DB      (KBIns OR KBCaps OR KBNum OR KBScroll) XOR 0FFh
HotFlag         DB      FALSE

ActiveTSR       DB      FALSE

DOSVersion      LABEL   WORD
                DB      ?               ; minor version number
MajorVersion    DB      ?               ; major version number

; The following data is used by the TSR application:

NTrap           DW      (EndTrapList-StartTrapList)/8   ; number of traps

StartTrapList   DB      1Bh
Trap1B          DB      FALSE
PrevISR1B       DD      ?
                DW      offset RESIDENT_GROUP:ISR1B

                DB      23h
```

Figure 11-4. Continued.

(more)

```
Trap23          DB      FALSE
PrevISR23       DD      ?
                DW      offset RESIDENT_GROUP:ISR23

                DB      24h
Trap24          DB      FALSE
PrevISR24       DD      ?
                DW      offset RESIDENT_GROUP:ISR24

EndTrapList     LABEL   BYTE

PrevBreak       DB      ?               ; previous break-checking flag

PrevDTA         LABEL   DWORD           ; previous DTA address
PrevDTAoffs     DW      ?
PrevDTAseg      DW      ?

PrevExtErrInfo  LABEL   BYTE            ; previous extended error information
PrevExtErrAX    DW      ?
PrevExtErrBX    DW      ?
PrevExtErrCX    DW      ?
PrevExtErrDX    DW      ?
PrevExtErrSI    DW      ?
PrevExtErrDI    DW      ?
PrevExtErrDS    DW      ?
PrevExtErrES    DW      ?
                DW      3 dup(0)

SnapFile        DB      '\snap.img'     ; output filename in root directory

RESIDENT_DATA   ENDS

RESIDENT_STACK  SEGMENT word stack 'STACK'

                DB      TSRStackSize dup(?)

RESIDENT_STACK  ENDS

                PAGE
;----------------------------------------------------------------------------
;
; Transient installation routines
;
;----------------------------------------------------------------------------

TRANSIENT_TEXT  SEGMENT para public 'TCODE'
                ASSUME  cs:TRANSIENT_TEXT,ds:RESIDENT_DATA,ss:RESIDENT_STACK

InstallSnapTSR  PROC    far             ; At entry:  CS:IP -> InstallSnapTSR
                                        ;            SS:SP -> stack
                                        ;            DS,ES -> PSP
```

Figure 11-4. Continued.

(more)

```
; Save PSP segment

                mov     ax,seg RESIDENT_DATA
                mov     ds,ax           ; DS  -> RESIDENT_DATA

                mov     TSRPSP,es       ; save PSP segment

; Check the MS-DOS version

                call    GetDOSVersion   ; AH = major version number
                                        ; AL = minor version number

; Verify that this TSR is not already installed
;
;       Before executing INT 2FH in MS-DOS versions 2.x, test whether INT 2FH
;       vector is in use.  If so, abort if PRINT.COM is using it.
;
;       (Thus, in MS-DOS 2.x, if both this program and PRINT.COM are used,
;       this program should be made resident before PRINT.COM.)

                cmp     ah,2
                ja      L101            ; jump if version 3.0 or later

                mov     ax,352Fh        ; AH = INT 21H function number
                                        ; AL = interrupt number
                int     21h             ; ES:BX = INT 2FH vector

                mov     ax,es
                or      ax,bx           ; jump if current INT 2FH vector ..
                jnz     L100            ; .. is nonzero

                push    ds
                mov     ax,252Fh        ; AH = INT 21H function number
                                        ; AL = interrupt number
                mov     dx,seg RESIDENT_GROUP
                mov     ds,dx
                mov     dx,offset RESIDENT_GROUP:MultiplexIRET

                int     21h             ; point INT 2FH vector to IRET
                pop     ds
                jmp     short L103      ; jump to install this TSR

L100:           mov     ax,0FF00h       ; look for PRINT.COM:
                int     2Fh             ; if resident, AH = print queue length;
                                        ; otherwise, AH is unchanged

                cmp     ah,0FFh         ; if PRINT.COM is not resident ..
                je      L101            ; .. use multiplex interrupt

                mov     al,1
                call    FatalError      ; abort if PRINT.COM already installed
```

Figure 11-4. Continued. *(more)*

```
L101:           mov     ah,MultiplexID  ; AH = multiplex interrupt ID value
                xor     al,al           ; AL = 00H
                int     2Fh             ; multiplex interrupt

                test    al,al
                jz      L103            ; jump if ok to install

                cmp     al,0FFh
                jne     L102            ; jump if not already installed

                mov     al,2
                call    FatalError      ; already installed

L102:           mov     al,3
                call    FatalError      ; can't install

; Get addresses of InDOS and ErrorMode flags

L103:           call    GetDOSFlags

; Install this TSR's interrupt handlers

                push    es              ; preserve PSP segment

                mov     cx,NISR
                mov     si,offset StartISRList

L104:           lodsb                   ; AL = interrupt number
                                        ; DS:SI -> byte past interrupt number
                push    ax              ; preserve AX
                mov     ah,35h          ; INT 21H function 35H
                int     21h             ; ES:BX = previous interrupt vector
                mov     [si+1],bx       ; save offset and segment ..
                mov     [si+3],es       ; .. of previous handler

                pop     ax              ; AL = interrupt number
                push    ds              ; preserve DS
                mov     dx,[si+5]
                mov     bx,seg RESIDENT_GROUP
                mov     ds,bx           ; DS:DX -> this TSR's handler
                mov     ah,25h          ; INT 21H function 25H
                int     21h             ; (set interrupt vector)
                pop     ds              ; restore DS
                add     si,7            ; DS:SI -> next in list
                loop    L104

; Free the environment

                pop     es              ; ES = PSP segment
                push    es              ; preserve PSP segment
                mov     es,es:[2Ch]     ; ES = segment of environment
```

Figure 11-4. Continued.

(more)

```
                mov     ah,49h          ; INT 21H function 49H
                int     21h             ; (free memory block)

; Terminate and stay resident

                pop     ax              ; AX = PSP segment
                mov     dx,cs           ; DX = paragraph address of start of
                                        ; transient portion (end of resident
                                        ; portion)
                sub     dx,ax           ; DX = size of resident portion

                mov     ax,3100h        ; AH = INT 21H function number
                                        ; AL = 00H (return code)
                int     21h

InstallSnapTSR  ENDP

GetDOSVersion   PROC    near            ; Caller:   DS = seg RESIDENT_DATA
                                        ;           ES = PSP
                                        ; Returns:  AH = major version
                                        ;           AL = minor version
                ASSUME  ds:RESIDENT_DATA

                mov     ah,30h          ; INT 21H function 30H:
                                        ; (get MS-DOS version)
                int     21h
                cmp     al,2
                jb      L110            ; jump if versions 1.x

                xchg    ah,al           ; AH = major version
                                        ; AL = minor version
                mov     DOSVersion,ax   ; save with major version in
                                        ; high-order byte
                ret

L110:           mov     al,00h
                call    FatalError      ; abort if versions 1.x

GetDOSVersion   ENDP
GetDOSFlags     PROC    near            ; Caller:   DS = seg RESIDENT_DATA
                                        ; Returns:  InDOSAddr -> InDOS
                                        ;           ErrorModeAddr -> ErrorMode
                                        ; Destroys: AX,BX,CX,DI
                ASSUME  ds:RESIDENT_DATA

; Get InDOS address from MS-DOS

                push    es

                mov     ah,34h          ; INT 21H function number
                int     21h             ; ES:BX -> InDOS
```

Figure 11-4. Continued. *(more)*

```
                mov     word ptr InDOSAddr,bx
                mov     word ptr InDOSAddr+2,es

; Determine ErrorMode address

                mov     word ptr ErrorModeAddr+2,es     ; assume ErrorMode is
                                                        ; in the same segment
                                                        ; as InDOS

                mov     ax,DOSVersion
                cmp     ax,30Ah
                jb      L120            ; jump if MS-DOS version earlier
                                        ; than 3.1 ..
                cmp     ax,0A00h
                jae     L120            ; .. or MS OS/2-DOS 3.x box

                dec     bx              ; in MS-DOS 3.1 and later, ErrorMode
                mov     word ptr ErrorModeAddr,bx       ; is just before InDOS
                jmp     short L125

L120:                                   ; scan MS-DOS segment for ErrorMode

                mov     cx,0FFFFh       ; CX = maximum number of bytes to scan
                xor     di,di           ; ES:DI -> start of MS-DOS segment

L121:           mov     ax,word ptr cs:LF2  ; AX = opcode for INT 28H

L122:           repne   scasb           ; scan for first byte of fragment
                jne     L126            ; jump if not found

                cmp     ah,es:[di]              ; inspect second byte of opcode
                jne     L122                    ; jump if not INT 28H

                mov     ax,word ptr cs:LF1 + 1  ; AX = opcode for CMP
                cmp     ax,es:[di][LF1-LF2]
                jne     L123                    ; jump if opcode not CMP

                mov     ax,es:[di][(LF1-LF2)+2] ; AX = offset of ErrorMode
                jmp     short L124              ; in DOS segment

L123:           mov     ax,word ptr cs:LF3 + 1  ; AX = opcode for TEST
                cmp     ax,es:[di][LF3-LF4]
                jne     L121                    ; jump if opcode not TEST

                mov     ax,es:[di][(LF3-LF4)+2] ; AX = offset of ErrorMode

L124:           mov     word ptr ErrorModeAddr,ax

L125:           pop     es
                ret
```

Figure 11-4. Continued. *(more)*

```
; Come here if address of ErrorMode not found

L126:           mov     al,04h
                call    FatalError

; Code fragments for scanning for ErrorMode flag

LFnear          LABEL   near                ; dummy labels for addressing
LFbyte          LABEL   byte
LFword          LABEL   word
                                            ; MS-DOS versions earlier than 3.1
LF1:            cmp     ss:LFbyte,0         ; CMP ErrorMode,0
                jne     LFnear
LF2:            int     28h
                                            ; MS-DOS versions 3.1 and later
LF3:            test    ss:LFbyte,0FFh      ; TEST ErrorMode,0FFH
                jne     LFnear
                push    ss:LFword
LF4:            int     28h

GetDOSFlags     ENDP

FatalError      PROC    near                ; Caller:   AL = message number
                                            ;           ES = PSP
                ASSUME  ds:TRANSIENT_DATA

                push    ax                  ; save message number on stack

                mov     bx,seg TRANSIENT_DATA
                mov     ds,bx

; Display the requested message

                mov     bx,offset MessageTable
                xor     ah,ah               ; AX = message number
                shl     ax,1                ; AX = offset into MessageTable
                add     bx,ax               ; DS:BX -> address of message
                mov     dx,[bx]             ; DS:BX -> message
                mov     ah,09h              ; INT 21H function 09H (display string)
                int     21h                 ; display error message

                pop     ax                  ; AL = message number
                or      al,al
                jz      L130                ; jump if message number is zero
                                            ; (MS-DOS versions 1.x)

; Terminate (MS-DOS 2.x and later)

                mov     ah,4Ch              ; INT 21H function 4CH
                int     21h                 ; (terminate process with return code)
```

Figure 11-4. Continued. *(more)*

```
; Terminate (MS-DOS 1.x)

L130            PROC    far

                push    es              ; push PSP:0000H
                xor     ax,ax
                push    ax
                ret                     ; far return (jump to PSP:0000H)

L130            ENDP

FatalError      ENDP


TRANSIENT_TEXT  ENDS

                PAGE
;
;
; Transient data segment
;
;

TRANSIENT_DATA  SEGMENT word public 'DATA'

MessageTable    DW      Message0        ; MS-DOS version error
                DW      Message1        ; PRINT.COM found in MS-DOS 2.x
                DW      Message2        ; already installed
                DW      Message3        ; can't install
                DW      Message4        ; can't find flag

Message0        DB      CR,LF,'TSR requires MS-DOS 2.0 or later version',CR,LF,'$'
Message1        DB      CR,LF,'Can''t install TSR:  PRINT.COM active',CR,LF,'$'
Message2        DB      CR,LF,'This TSR is already installed',CR,LF,'$'
Message3        DB      CR,LF,'Can''t install this TSR',CR,LF,'$'
Message4        DB      CR,LF,'Unable to locate MS-DOS ErrorMode flag',CR,LF,'$'

TRANSIENT_DATA  ENDS

                END     InstallSnapTSR
```

Figure 11-4. Continued.

When installed, the SNAP program monitors keyboard input until the user types the hot-key sequence Alt-Enter. When the hot-key sequence is detected, the monitoring routine waits until the operating environment is stable and then activates the RAM-resident application, which dumps the current contents of the computer's video buffer into the file SNAP.IMG. Figure 11-5 is a block diagram of the RAM-resident and transient components of this TSR.

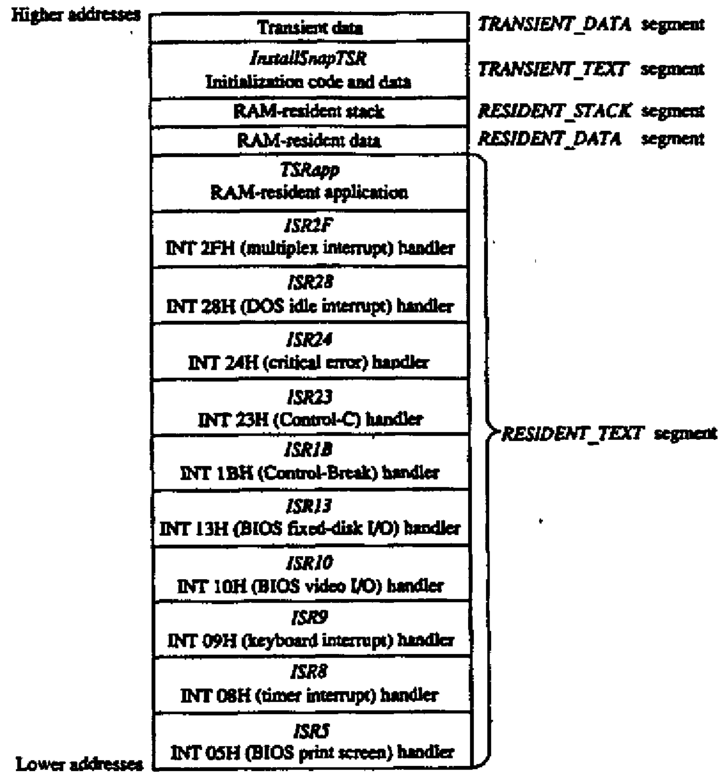

Figure 11-5. Block structure of the TSR program SNAP.EXE when loaded into memory. (Compare with Figure 11-1.)

Installing the program

When SNAP.EXE is run, only the code in the transient portion of the program is executed. The transient code performs several operations before it finally executes Interrupt 21H Function 31H (Terminate and Stay Resident). First it determines which MS-DOS version is in use. Then it executes the multiplex interrupt (Interrupt 2FH) to discover whether the resident portion has already been installed. If an MS-DOS version earlier than 2.0 is in use or if the resident portion has already been installed, the program aborts with an error message.

Otherwise, installation continues. The addresses of the InDOS and critical error flags are saved in the resident data segment. The interrupt service routines in the RAM-resident portion of the program are installed by updating all relevant interrupt vectors. The transient code then frees the RAM occupied by the program's environment, because the resident portion of this program never uses the information contained there. Finally, the transient portion of the program, which includes the *TRANSIENT_TEXT* and *TRANSIENT_DATA* segments, is discarded and the program is terminated using Interrupt 21H Function 31H.

Detecting a hot key

The SNAP program detects the hot-key sequence (Alt-Enter) by monitoring each keypress. On IBM PCs and PS/2s, each keystroke generates a hardware interrupt on IRQ1 (Interrupt 09H). The TSR's Interrupt 09H handler compares the keyboard scan code corresponding to each keypress with a predefined sequence. The TSR's handler also inspects the shift-key status flags maintained by the ROM BIOS Interrupt 09H handler. When the predetermined sequence of keypresses is detected at the same time as the proper shift keys are pressed, the handler sets a global status flag (*HotFlag*).

Note how the TSR's handler transfers control to the previous Interrupt 09H ISR before it performs its own work. If the TSR's Interrupt 09H handler did not chain to the previous handler(s), essential system processing of keystrokes (particularly in the ROM BIOS Interrupt 09H handler) might not be performed.

Activating the application

The TSR monitors the status of *HotFlag* by regularly testing its value within a timer-tick handler. On IBM PCs and PS/2s, the timer-tick interrupt occurs on IRQ0 (Interrupt 08H) roughly 18.2 times per second. This hardware interrupt occurs regardless of what else the system is doing, so an Interrupt 08H ISR a convenient place to check whether *HotFlag* has been set.

As in the case of the Interrupt 09H handler, the TSR's Interrupt 08H handler passes control to previous Interrupt 08H handlers before it proceeds with its own work. This procedure is particularly important with Interrupt 08H because the ROM BIOS Interrupt 08H handler, which maintains the system's time-of-day clock and resets the system's Intel 8259A Programmable Interrupt Controller, must execute before the next timer tick can occur. The TSR's handler therefore defers its own work until control has returned after previous Interrupt 08H handlers have executed.

The only function of the TSR's Interrupt 08H handler is to attempt to transfer control to the RAM-resident application. The routine *VerifyTSRState* performs this task. It first examines the contents of *HotFlag* to determine whether a hot-key sequence has been detected. If so, it examines the state of the MS-DOS InDOS and critical error flags, the current status of hardware interrupts, and the current status of any non-reentrant ROM BIOS routines that might be executing.

If *HotFlag* is nonzero, the InDOS and critical error flags are both zero, no hardware interrupts are currently being serviced, and no non-reentrant ROM BIOS code has been interrupted, the Interrupt 08H handler activates the RAM-resident utility. Otherwise, nothing happens until the next timer tick, when the handler executes again.

While *HotFlag* is nonzero, the Interrupt 08H handler continues to monitor system status until MS-DOS, the ROM BIOS, and the hardware interrupts are all in a stable state. Often the system status is stable at the time the hot-key sequence is detected, so the RAM-resident application runs immediately. Sometimes, however, system activities such as prolonged disk reads or writes can preclude the activation of the RAM-resident utility for several seconds after the hot-key sequence has been detected. The handler could be designed to detect this situation (for example, by decrementing *HotFlag* on each timer tick) and return an error status or display a message to the user.

A more serious difficulty arises when the MS-DOS default command processor (COMMAND.COM) is waiting for keyboard input. In this situation, Interrupt 21H Function 01H (Character Input with Echo) is executing, so InDOS is nonzero and the Interrupt 08H handler can never detect a state in which it can activate the RAM-resident utility. This problem is solved by providing a custom handler for Interrupt 28H (the MS-DOS idle interrupt), which is executed by Interrupt 21H Function 01H each time it loops as it waits for a keypress. The only difference between the Interrupt 28H handler and the Interrupt 08H handler is that the Interrupt 28H handler can activate the RAM-resident application when the value of InDOS is 1, which is reasonable because InDOS must have been incremented when Interrupt 21H Function 01H started to execute.

The interrupt service routines for ROM BIOS Interrupts 05H, 10H, and 13H do nothing more than increment and decrement flags that indicate whether these interrupts are being processed by ROM BIOS routines. These flags are inspected by the TSR's Interrupt 08H and 28H handlers.

Executing the RAM-resident application

When the RAM-resident application is first activated, it runs in the context of the program that was interrupted; that is, the contents of the registers, the video display mode, the current PSP, and the current DTA all belong to the interrupted program. The resident application is responsible for preserving the registers and updating MS-DOS with its PSP and DTA values.

The RAM-resident application preserves the previous contents of the CPU registers on its own stack to avoid overflowing the interrupted program's stack. It then installs its own handlers for Control-Break (Interrupt 1BH), Control-C (Interrupt 23H), and critical error (Interrupt 24H). (Otherwise, the interrupted program's handlers would take control if the user pressed Ctrl-Break or Ctrl-C or if an MS-DOS critical error occurred.) These handlers perform no action other than setting flags that can be inspected later by the RAM-resident application, which could then take appropriate action.

The application uses Interrupt 21H Functions 50H and 51H to update MS-DOS with the address of its PSP. If the application is running under MS-DOS versions 2.x, the critical error flag is set before Functions 50H and 51H are executed so that *AuxStack* is used for the call instead of *IOStack*, to avoid corrupting *IOStack* in the event that InDOS is 1.

The application preserves the current extended error information with a call to Interrupt 21H Function 59H. Otherwise, the RAM-resident application might be activated immediately after a critical error occurred in the interrupted program but before the interrupted program had executed Function 59H and, if a critical error occurred in the TSR application, the interrupted program's extended error information would inadvertently be destroyed.

This example also shows how to update the MS-DOS default DTA using Interrupt 21H Functions 1AH and 2FH, although in this case this step is not necessary because the DTA is never used within the application. In practice, the DTA should be updated only if the RAM-resident application includes calls to Interrupt 21H functions that use a DTA (Functions 11H, 12H, 14H, 15H, 21H, 22H, 27H, 28H, 4EH, and 4FH).

After the resident interrupt handlers are installed and the PSP, DTA, and extended error information have been set up, the RAM-resident application can safely execute any Interrupt 21H function calls except those that use *IOStack* (Functions 01H through 0CH). These functions cannot be used within a RAM-resident application even if the application sets the critical error flag to force the use of the auxiliary stack, because they also use other non-reentrant data structures such as input/output buffers. Thus, a RAM-resident utility must rely either on user-written console input/output functions or, as in the example, on ROM BIOS functions.

The application terminates by returning the interrupted program's extended error information, DTA, and PSP to MS-DOS, restoring the previous Interrupt 1BH, 23H, and 24H handlers, and restoring the previous CPU registers and stack.

Richard Wilton