CHAPTER 11

Writing Memory-Resident Software

Through its memory-management system, MS-DOS allows a program to remain resident in memory after terminating. The resident program can later regain control of the processor to perform tasks such as background printing or "popping up" a calculator on the screen. Such a program is commonly called a TSR, from the terminate-and-stay-resident function it uses to return to MS-DOS.

This chapter explains the techniques of writing memory-resident software. The first two sections present introductory material. Following sections describe important MS-DOS and BIOS interrupts and focus on how to write safe, compatible, memory-resident software. Two example programs illustrate the techniques described in the chapter. The MASM 6.1 disks contain complete source code for the two example TSR programs.

Terminate-and-Stay-Resident Programs

MS-DOS maintains a pointer to the beginning of unused memory. Programs load into memory at this position and terminate execution by returning control to MS-DOS. Normally, the pointer remains unchanged, allowing MS-DOS to reuse the same memory when loading other programs.

A terminating program can, however, prevent other programs from loading on top of it. These programs exit to MS-DOS through the terminate-and-stay-resident function, which resets the free-memory pointer to a higher position. This leaves the program resident in a protected block of memory, even though it is no longer running.

The terminate-and-stay-resident function (Function 31h) is one of the MS-DOS services invoked through Interrupt 21h. The following fragment shows how a TSR program terminates through Function 31h and remains resident in a 1000h-byte block of memory:

```
mov     ah, 31h         ; Request DOS Function 31h
mov     al, err         ; Set return code
mov     dx, 100h        ; Reserve 100h paragraphs
                        ;   (1000h bytes)
int     21h             ; Terminate-and-stay-resident
```

Note In current versions of MS-DOS, Interrupt 27h also provides a terminate-and-stay-resident service. However, Microsoft cannot guarantee future support for Interrupt 27h and does not recommend its use.

Structure of a TSR

TSRs consist of two distinct parts that execute at different times. The first part is the installation section, which executes only once, when MS-DOS loads the program. The installation code performs any initialization tasks required by the TSR and then exits through the terminate-and-stay-resident function.

The second part of the TSR, called the resident section, consists of code and data left in memory after termination. Though often identified with the TSR itself, the resident section makes up only part of the entire program.

The TSR's resident code must be able to regain control of the processor and execute after the program has terminated. Methods of executing a TSR are classified as either passive or active.

Passive TSRs

The simplest way to execute a TSR is to transfer control to it explicitly from another program. Because the TSR in this case does not solicit processor control, it is said to be passive. If the calling program can determine the TSR's memory address, it can grant control via a far jump or call. More commonly, a program activates a passive TSR through a software interrupt. The installation section of the TSR writes the address of its resident code to the proper position in the interrupt vector table (see "MS-DOS Interrupts" in Chapter 7). Any subsequent program can then execute the TSR by calling the interrupt.

Passive TSRs often replace existing software interrupts. For example, a passive TSR might replace Interrupt 10h, the BIOS video service. By intercepting calls that read or write to the screen, the TSR can access the video buffer directly, increasing display speed.

Passive TSRs allow limited access since they can be invoked only from another program. They have the advantage of executing within the context of the calling program, and thus run no risk of interfering with another process. Such a risk does exist with active TSRs.

Active TSRs

The second method of executing a TSR involves signaling it through some hardware event, such as a predetermined sequence of keystrokes. This type of TSR is "active" because it must continually search for its startup signal. The advantage of active TSRs lies in their accessibility. They can take control from any running application, execute, and return, all on demand.

An active TSR, however, must not seize processor control blindly. It must contain additional code that determines the proper moment at which to execute. The extra code consists of one or more routines called "interrupt handlers," described in the following section.

Interrupt Handlers in Active TSRs

The memory-resident portion of an active TSR consists of two parts. One part contains the body of the TSR—the code and data that perform the program's main tasks. The other part contains the TSR's interrupt handlers.

An interrupt handler is a routine that takes control when a specific interrupt occurs. Although sometimes called an "interrupt service routine," a TSR's handler usually does not service the interrupt. Instead, it passes control to the original interrupt routine, which does the actual interrupt servicing. (See the section "Replacing an Interrupt Routine" in Chapter 7 for information on how to write an interrupt handler.)

Collectively, interrupt handlers ensure that a TSR operates compatibly with the rest of the system. Individually, each handler fulfills one or more of the following functions:

- Auditing hardware events that may signal a request for the TSR
- Monitoring system status
- Determining whether a request for the TSR should be honored, based on current system status

Auditing Hardware Events for TSR Requests

Active TSRs commonly use a special keystroke sequence or the timer as a request signal. A TSR invoked through one of these channels must be equipped with handlers that audit keyboard or timer events.

A keyboard handler receives control at every keystroke. It examines each key, searching for the proper signal or "hot key." Generally, a keyboard handler should not attempt to call the TSR directly when it detects the hot key. If the TSR cannot safely interrupt the current process at that moment, the keyboard handler is forced to exit to allow the process to continue. Since the handler cannot regain control until the next keystroke, the user has to press the hot key repeatedly until the handler can comply with the request.

Instead, the handler should merely set a request flag when it detects a hot-key signal and then exit normally. Examples in the following paragraphs illustrate this technique.

For computers other than MCA (IBM PS/2 and compatible), an active TSR audits keystrokes through a handler for Interrupt 09, the keyboard interrupt:

```
Keybrd  PROC    FAR
        sti                             ; Interrupts are okay
        push    ax                      ; Save AX register
        in      al, 60h                 ; AL = key scan code
        call    CheckHotKey             ; Check for hot key
        .IF     carry?                  ; If hot key pressed,
        mov     cs:TsrRequestFlag, TRUE ;   raise flag and
        .                               ;   set up for exit
        .
        .
```

A TSR running on a PS/2 computer cannot reliably read key scan codes using this method. Instead, the TSR must search for its hot key through a handler for Interrupt 15h (Miscellaneous System Services). The handler determines the current keypress from the AL register when AH equals 4Fh, as shown here:

```
MiscServ PROC   FAR
        sti                             ; Interrupts okay
        .IF     ah == 4Fh               ; If Keyboard Intercept Service:
        call    CheckHotKey             ; Check for hot key
        .IF     carry?                  ; If hot key pressed,
        mov     cs:TsrRequestFlag, TRUE ;   raise flag and
        .                               ;   set up for exit
        .
        .
```

The example program on page 293 shows how a TSR tests for a PS/2 machine and then sets up a handler for either Interrupt 09 or Interrupt 15h to audit keystrokes.

Setting a request flag in the keyboard handler allows other code, such as the timer handler (Interrupt 08), to recognize a request for the TSR. The timer handler gains control at every timer interrupt, which occurs an average of 18.2 times per second.

The following fragment shows how a timer handler tests the request flag and continually polls until it can safely execute the TSR.

```
NewTimer PROC FAR
        .
        .
        .
        cmp     TsrRequestFlag, FALSE   ; Has TSR been requested?
        .IF     !zero?                  ; If so, can system be
        call    CheckSystem             ;   interrupted safely?
        .IF     carry?                  ; If so,
        call    ActivateTsr             ;   activate TSR
        .
        .
        .
```

Monitoring System Status

A TSR that uses a hardware device such as the video or disk must not interrupt while the device is active. A TSR monitors a device by handling the device's interrupt. Each interrupt handler simply sets a flag to indicate the device is in use, and then clears the flag when the interrupt finishes.

The following shows a typical monitor handler:

```
NewHandler PROC    FAR
        mov     cs:ActiveFlag, TRUE  ; Set active flag
        pushf                        ; Simulate interrupt by
                                     ;   pushing flags, then
        call    OldHandler           ;   far-calling original routine
        mov     cs:ActiveFlag, FALSE ; Clear active flag
        iret                         ; Return from interrupt
NewHandler ENDP
```

Only hardware used by the TSR requires monitoring. For example, a TSR that performs disk input/output (I/O) must monitor disk use through Interrupt 13h. The disk handler sets an active flag that prevents the TSR from executing during a read or write operation. Otherwise, the TSR's own I/O would move the disk head. This would cause the suspended disk operation to continue with the head incorrectly positioned when the TSR returned control to the interrupted program.

In the same way, an active TSR that displays to the screen must monitor calls to Interrupt 10h. The Interrupt 10h BIOS routine does not protect critical sections of code that program the video controller. The TSR must therefore ensure it does not interrupt such nonreentrant operations.

The activities of the operating system also affect the system status. With few exceptions, MS-DOS functions are not reentrant and must not be interrupted.

However, monitoring MS-DOS is somewhat more complicated than monitoring hardware. This subject is discussed in "Using MS-DOS in Active TSRs," later in this chapter.

Figure 11.1 illustrates the process described so far. It shows a time line for a typical TSR signaled from the keyboard. When the keyboard handler detects the proper hot key, it sets a request flag called `TsrRequestFlag`. Thereafter, the timer handler continually checks the system status until it can safely call the TSR.

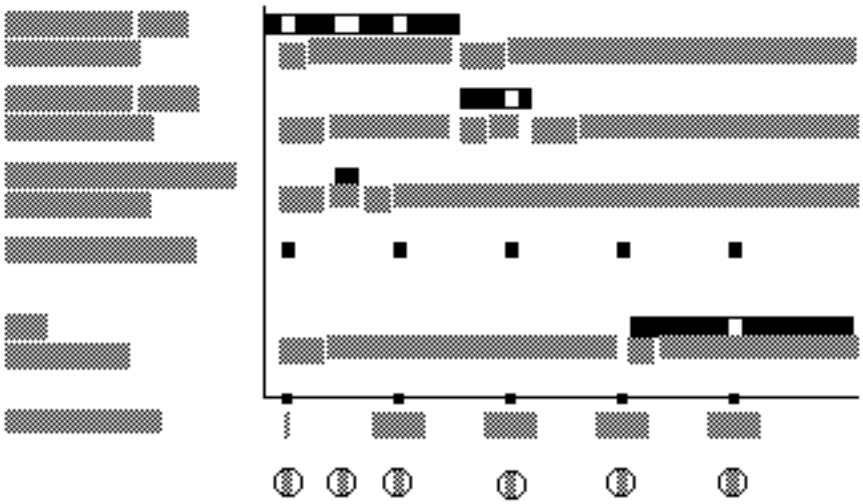

Figure 11.1 Time Line of Interactions Between Interrupt Handlers for a Typical TSR

The following comments describe the chain of events depicted in Figure 11.1. Each comment refers to one of the numbered pointers in the figure.

1. At time = t, the timer handler activates. It finds the flag `TsrRequestFlag` clear, indicating the user has not requested the TSR. The handler terminates without taking further action. Notice that Interrupt 13h is currently processing a disk I/O operation.
2. Before the next timer interrupt, the keyboard handler detects the hot key, signaling a request for the TSR. The keyboard handler sets `TsrRequestFlag` and returns.
3. At time = t + 1/18 second, the timer handler again activates and finds `TsrRequestFlag` set. The handler checks other active flags to determine if the TSR can safely execute. Since Interrupt 13h has not yet completed its disk operation, the timer handler finds `DiskActiveFlag` set. The handler therefore terminates without activating the TSR.

4. At time = t + 2/18 second, the timer handler again finds `TsrRequestFlag` set and repeats its scan of the active flags. `DiskActiveFlag` is now clear, but in the interim, Interrupt 10h has activated as indicated by the flag `VideoActiveFlag`. The timer handler accordingly terminates without activating the TSR.
5. At time = t + 3/18 second, the timer handler repeats the process. This time it finds all active flags clear, indicating the TSR can safely execute. The timer handler calls the TSR, which sets its own active flag to ensure it will not interrupt itself if requested again.
6. The timer and other interrupts continue to function normally while the TSR executes.

The timer itself can serve as the startup signal if the TSR executes periodically. Screen clocks that continuously show seconds and minutes are examples of TSRs that use the timer this way. ALARM.ASM, a program described in the next section, shows another example of a timer-driven TSR.

Determining Whether to Invoke the TSR

Once a handler receives a request signal for the TSR, it checks the various active flags maintained by the handlers that monitor system status. If any of the flags are set, the handler ignores the request and exits. If the flags are clear, the handler invokes the TSR, usually through a near or far call. Figure 11.1 illustrates how a timer handler detects a request and then periodically scans various active flags until all the flags are clear.

A TSR that changes stacks must not interrupt itself. Otherwise, the second execution would overwrite the stack data belonging to the first. A TSR prevents this by setting its own active flag before executing, as shown in Figure 11.1. A handler must check this flag along with the other active flags when determining whether the TSR can safely execute.

Example of a Simple TSR: ALARM

This section presents a simple alarm clock TSR that demonstrates some of the material covered so far. The program accepts an argument from the command line that specifies the alarm setting in military form, such as 1635 for 4:35 P.M. For simplicity, the argument must consist of four digits, including leading zeros. To set the alarm at 7:45 A.M., for example, enter the command:

```
ALARM 0745
```

The installation section of the program begins with the `Install` procedure. `Install` computes the number of five-second intervals that must elapse before the alarm sounds and stores this number in the word `CountDown`. The

procedure then obtains the vector for Interrupt 08 (timer) through MS-DOS Function 35h and stores it in the far pointer `OldTimer`. Function 25h replaces the vector with the far address of the new timer handler `NewTimer`. Once installed, the new timer handler executes at every timer interrupt. These interrupts occur 18.2 times per second or 91 times every five seconds.

Each time it executes, `NewTimer` subtracts one from a secondary counter called `Tick91`. By counting 91 timer ticks, `Tick91` accurately measures a period of five seconds. When `Tick91` reaches zero, it's reset to 91 and `CountDown` is decremented by one. When `CountDown` reaches zero, the alarm sounds.

```
;* ALARM.ASM - A simple memory-resident program that beeps the speaker
;* at a prearranged time. Can be loaded more than once for multiple
;* alarm settings. During installation, ALARM establishes a handler
;* for the timer interrupt (Interrupt 08). It then terminates through
;* the terminate-and-stay-resident function (Function 31h). After the
;* alarm sounds, the resident portion of the program retires by setting
;* a flag that prevents further processing in the handler.

        .MODEL tiny                     ; Create ALARM.COM
        .STACK
        .CODE
        ORG     5Dh                     ; Location of time argument in PSP,
CountDown       LABEL   WORD            ;   converted to number of 5-second
                                        ;   intervals to elapse
        .STARTUP
        jmp     Install                 ; Jump over data and resident code

; Data must be in code segment so it won't be thrown away with Install code.
OldTimer        DWORD   ?               ; Address of original timer routine
tick_91         BYTE    91              ; Counts 91 clock ticks (5 seconds)
TimerActiveFlag BYTE    0               ; Active flag for timer handler

;* NewTimer - Handler routine for timer interrupt (Interrupt 08).
;* Decrements CountDown every 5 seconds. No other action is taken
;* until CountDown reaches 0, at which time the speaker sounds.

NewTimer PROC   FAR
        .IF     cs:TimerActiveFlag != 0 ; If timer busy or retired,
        jmp     cs:OldTimer             ;   jump to original timer routine
        .ENDIF
        inc     cs:TimerActiveFlag      ; Set active flag
        pushf                           ; Simulate interrupt by pushing flags,
        call    cs:OldTimer             ;   then far-calling original routine
        sti                             ; Enable interrupts
        push    ds                      ; Preserve DS register
        push    cs                      ; Point DS to current segment for
        pop     ds                      ;   further memory access
        dec     tick_91                 ; Count down for 91 ticks
```

```
        .IF     zero?                   ; If 91 ticks have elapsed,
        mov     tick_91, 91             ;   reset secondary counter and
        dec     CountDown               ;   subtract one 5-second interval
        .IF     zero?                   ; If CountDown drained,
        call    Sound                   ;   sound speaker
        inc     TimerActiveFlag         ; Alarm has sounded--inc flag
        .ENDIF                          ;   again so it remains set
        .ENDIF

        dec     TimerActiveFlag         ; Decrement active flag
        pop     ds                      ; Recover DS
        iret                            ; Return from interrupt handler
NewTimer ENDP

;* Sound - Sounds speaker with the following tone and duration:

BEEP_TONE       EQU     440             ; Beep tone in hertz
BEEP_DURATION   EQU     6               ; Number of clocks during beep,
                                        ;   where 18 clocks = approx 1 second

Sound   PROC    USES ax bx cx dx es     ; Save registers used in this routine
        mov     al, 0B6h                ; Initialize channel 2 of
        out     43h, al                 ;   timer chip
        mov     dx, 12h                 ; Divide 1,193,180 hertz
        mov     ax, 34DCh               ;   (clock frequency) by
        mov     bx, BEEP_TONE           ;   desired frequency
        div     bx                      ; Result is timer clock count
        out     42h, al                 ; Low byte of count to timer
        mov     al, ah
        out     42h, al                 ; High byte of count to timer
        in      al, 61h                 ; Read value from port 61h
        or      al, 3                   ; Set first two bits
        out     61h, al                 ; Turn speaker on
; Pause for specified number of clock ticks

        mov     dx, BEEP_DURATION       ; Beep duration in clock ticks
        sub     cx, cx                  ; CX:DX = tick count for pause
        mov     es, cx                  ; Point ES to low memory data
        add     dx, es:[46Ch]           ; Add current tick count to CX:DX
        adc     cx, es:[46Eh]           ; Result is target count in CX:DX
        .REPEAT
        mov     bx, es:[46Ch]           ; Now repeatedly poll clock
        mov     ax, es:[46Eh]           ;   count until the target
        sub     bx, dx                  ;   time is reached
        sbb     ax, cx
        .UNTIL  !carry?
```

```
        in      al, 61h                 ; When time elapses, get port value
        xor     al, 3                   ; Kill bits 0-1 to turn
        out     61h, al                 ;   speaker off
        ret
Sound   ENDP

;* Install - Converts ASCII argument to valid binary number, replaces
;* NewTimer as the interrupt handler for the timer, then makes program
;* memory-resident by exiting through Function 31h.
;*
;* This procedure marks the end of the TSR's resident section and the
;* beginning of the installation section. When ALARM terminates through
;* Function 31h, the above code and data remain resident in memory. The
;* memory occupied by the following code is returned to DOS.

Install PROC

; Time argument is in hhmm military format. Converts ASCII digits to
; number of minutes since midnight, then converts current time to number
; of minutes since midnight. Difference is number of minutes to elapse
; until alarm sounds. Converts to seconds-to-elapse, divides by 5 seconds,
; and stores result in word CountDown.
DEFAULT_TIME    EQU     3600            ; Default alarm setting = 1 hour
                                        ;   (in seconds) from present time
        mov     ax, DEFAULT_TIME
        cwd                             ; DX:AX = default time in seconds
        .IF     BYTE PTR CountDown != ' ' ; If not blank argument,
        xor     CountDown[0], '00'      ;   convert 4 bytes of ASCII
        xor     CountDown[2], '00'      ;   argument to binary

        mov     al, 10                  ; Multiply 1st hour digit by 10
        mul     BYTE PTR CountDown[0]   ;   and add to 2nd hour digit
        add     al, BYTE PTR CountDown[1]
        mov     bh, al                  ; BH = hour for alarm to go off
        mov     al, 10                  ; Repeat procedure for minutes
        mul     BYTE PTR CountDown[2]   ; Multiply 1st minute digit by 10
        add     al, BYTE PTR CountDown[3] ;   and add to 2nd minute digit
        mov     bl, al                  ; BL = minute for alarm to go off
        mov     ah, 2Ch                 ; Request Function 2Ch
        int     21h                     ; Get Time (CX = current hour/min)
```

```
        mov     dl, dh
        sub     dh, dh
        push    dx                      ; Save DX = current seconds
        mov     al, 60                  ; Multiply current hour by 60
        mul     ch                      ;   to convert to minutes
        sub     ch, ch
        add     cx, ax                  ; Add current minutes to result
                                        ; CX = minutes since midnight
        mov     al, 60                  ; Multiply alarm hour by 60
        mul     bh                      ;   to convert to minutes
        sub     bh, bh
        add     ax, bx                  ; AX = number of minutes since
                                        ;   midnight for alarm setting
        sub     ax, cx                  ; AX = time in minutes to elapse
                                        ;   before alarm sounds
        .IF     carry?                  ; If alarm time is tomorrow,
        add     ax, 24 * 60             ;   add minutes in a day
        .ENDIF

        mov     bx, 60
        mul     bx                      ; DX:AX = minutes-to-elapse-times-60
        pop     bx                      ; Recover current seconds
        sub     ax, bx                  ; DX:AX = seconds to elapse before
        sbb     dx, 0                   ;   alarm activates
        .IF     carry?                  ; If negative,
        mov     ax, 5                   ;   assume 5 seconds
        cwd
        .ENDIF
        .ENDIF
        mov     bx, 5                   ; Divide result by 5 seconds
        div     bx                      ; AX = number of 5-second intervals
        mov     CountDown, ax           ;   to elapse before alarm sounds

        mov     ax, 3508h               ; Request Function 35h
        int     21h                     ; Get Vector for timer (Interrupt 08)
        mov     WORD PTR OldTimer[0], bx  ; Store address of original
        mov     WORD PTR OldTimer[2], es  ;   timer interrupt
        mov     ax, 2508h               ; Request Function 25h
        mov     dx, OFFSET NewTimer     ; DS:DX points to new timer handler
        int     21h                     ; Set Vector with address of NewTimer

        mov     dx, OFFSET Install      ; DX = bytes in resident section
        mov     cl, 4
        shr     dx, cl                  ; Convert to number of paragraphs
        inc     dx                      ;   plus one
        mov     ax, 3100h               ; Request Function 31h, error code=0
        int     21h                     ; Terminate-and-stay-resident
Install ENDP
        END
```

Note the following points about ALARM:

- The constant `BEEP_TONE` specifies the alarm tone. Practical values for the tone range from approximately 100 to 4,000 hertz.
- The `Install` procedure marks the beginning of the installation section of the program. Execution begins here when ALARM.COM is loaded. A TSR generally places its installation code after the resident section. This allows the terminating TSR to include the installation code with the rest of the memory it returns to MS-DOS. Since the installation section executes only once, the TSR can discard it after becoming resident.
- You can install ALARM any number of times in quick succession, each time with a new alarm setting. The timer handler does not restore the original vector for Interrupt 08 after the alarm sounds. In effect, the multiple installations remain daisy-chained in memory. The address in `OldTimer` for one installation is the address of `NewTimer` in the preceding installation.
- Until a system reboot, `NewTimer` remains in place as the Interrupt 08 handler, even after the alarm sounds. To save unnecessary activity, the byte `TimerActiveFlag` remains set after the alarm sounds. This forces an immediate jump to the original handler for all subsequent executions of `NewTimer`.
- `NewTimer` and `Sound` alter registers DS, AX, BX, CX, DX, and ES. To preserve the original values in these registers, the procedures first push them onto the stack and then restore the original values before exiting. This ensures that the process interrupted by `NewTimer` continues with valid registers after `NewTimer` returns.
- ALARM requires little stack space. It assumes the current stack is adequate and makes no attempt to set up a new one. More sophisticated TSRs, however, should as a matter of course provide their own stacks to ensure adequate stack depth. The example program presented in "Example of an Advanced TSR: SNAP," later in this chapter, demonstrates this safety measure.

Using MS-DOS in Active TSRs

This section explains how to write active TSRs that can safely call MS-DOS functions. The material explores the problems imposed by the nonreentrant nature of MS-DOS and explains how a TSR can resolve those problems. The solution consists of four parts:

- Understanding how MS-DOS uses stacks
- Determining when MS-DOS is active
- Determining whether a TSR can safely interrupt an active MS-DOS function
- Monitoring the Critical Error flag

Understanding MS-DOS Stacks

MS-DOS functions set up their own stacks, which makes them nonreentrant. If a TSR interrupts an MS-DOS function and then executes another function that sets up the same stack, the second function will overwrite everything placed on the stack by the first function. The problem occurs when the second function returns and the first is left with unusable stack data. A TSR that calls an MS-DOS function must not interrupt any function that uses the same stack.

MS-DOS versions 2.0 and later use three internal stacks: an I/O stack, a disk stack, and an auxiliary stack. The current stack depends on the MS-DOS function. Functions 01 through 0Ch set up the I/O stack. Functions higher than 0Ch (with few exceptions) use the disk stack, as do Interrupts 25h and 26h. MS-DOS normally uses the auxiliary stack only when it executes Interrupt 24h (Critical Error Handler).

Determining MS-DOS Activity

A TSR's handlers can determine when MS-DOS is active by consulting a 1-byte flag called the InDos flag. Every MS-DOS function sets this flag upon entry and clears it upon termination. During installation, a TSR locates the flag through Function 34h (Get Address of InDos Flag), which returns the address as ES:BX. The installation portion then stores the address so the handlers can later find the flag without again calling Function 34h.

Theoretically, a TSR can wait to execute until the InDos flag is clear, thus sidestepping the entire issue of interrupting MS-DOS. However, several low-order functions — such as Function 0Ah (Get Buffered Keyboard Input) — wait idly for an expected keystroke before they terminate. If a TSR were allowed to execute only after MS-DOS returned, it too would have to wait for the terminating event.

The solution lies in determining when the low-order functions 01 through 0Ch are active. MS-DOS provides another service for this purpose: Interrupt 28h, the Idle Interrupt.

Interrupting MS-DOS Functions

MS-DOS continually calls Interrupt 28h from the low-order polling functions as they wait for keyboard input. This signal says that MS-DOS is idle and that a TSR may interrupt provided it does not overwrite the I/O stack. Put another way, a TSR can safely interrupt MS-DOS Functions 01 through 0Ch provided it does not call them.

An active TSR that calls MS-DOS must monitor Interrupt 28h with a handler. When the handler gains control, it checks the TSR request flag. If the flag indicates the TSR has been requested and if system hardware is inactive, the handler executes the TSR. Since control must eventually return to the idle MS-DOS function which has stored data on the I/O stack, the TSR in this case must not call any MS-DOS function that also uses the I/O stack. Table 11.1 shows which functions set up the I/O stack for various versions of MS-DOS.

Table 11.1 MS-DOS Internal Stacks

Function	Critical Error flag	MS-DOS 2.x	MS-DOS 3.0	MS-DOS 3.1+
01–0Ch	Clear	I/O*	I/O	I/O
	Set	Aux*	Aux	Aux
33h	Clear	Disk*	Disk	Caller*
	Set	Disk	Disk	Caller
50h–51h	Clear	I/O	Caller	Caller
	Set	Aux	Caller	Caller
59h	Clear	n/a*	I/O	Disk
	Set	n/a	Aux	Disk
5D0Ah	Clear	n/a	n/a	Disk
	Set	n/a	n/a	Disk
62h	Clear	n/a	Caller	Caller
	Set	n/a	Caller	Caller
All others	Clear	Disk	Disk	Disk
	Set	Disk	Disk	Disk

* I/O=I/O stack, Aux = auxiliary stack, Disk = disk stack, Caller = caller's stack, n/a = function not available.

TSRs that perform tasks of long or indefinite duration should themselves call Interrupt 28h. For example, a TSR that polls for keyboard input should include an **INT 28h** instruction in the polling loop, as shown here:

```
poll:      int     28h          ; Signal idle state
           mov     ah, 1
           int     16h          ; Key waiting?
           jnz     poll         ; If not, repeat polling loop
           sub     ah, ah
           int     16h          ; Otherwise, get key
```

This courtesy gives other TSRs a chance to execute if the InDos flag happens to be set.

Monitoring the Critical Error Flag

MS-DOS sets the Critical Error flag to a nonzero value when it detects a critical error. It then invokes Interrupt 24h (Critical Error Handler) and clears the flag when Interrupt 24h returns. MS-DOS functions higher than 0Ch are illegal during critical error processing. Therefore, a TSR that calls MS-DOS must not execute while the Critical Error flag is set.

MS-DOS versions 3.1 and later locate the Critical Error flag in the byte preceding the InDos flag. A single call to Function 34h (Get Address of InDos Flag) thus effectively returns the addresses of both flags. For earlier versions of MS-DOS or for the compatibility version of MS-DOS in OS/2 1.*x*, a TSR must call Function 34h and then scan the segment returned in the ES register for one of the two following sequences of instructions:

```
; Sequence of instructions in DOS Versions 2.0 - 3.0
        cmp     ss:[CriticalErrorFlag], 0
        jne     @F
        int     28h

; Sequence of instructions in DOS compatibility version for OS/2 1.x
        test    [CriticalErrorFlag], 0FFh
        jnz     @F
        push    ss:[ ? ]
        int     28h
```

The question mark inside brackets in the preceding **PUSH** statement indicates that the operand for the **PUSH** instruction can be any legal operand.

In either version of MS-DOS, the operand field in the first instruction gives the flag's offset. The value in ES determines the segment address. "Example of an Advanced TSR: SNAP," later in the chapter, presents a program that shows how to locate the Critical Error flag with this technique.

Preventing Interference

This section describes how an active TSR can avoid interfering with the process it interrupts. Interference occurs when a TSR commits an error or performs an action that affects the interrupted process after the TSR returns. Examples of interference range from relatively harmless, such as moving the cursor, to serious, such as overrunning a stack.

Although a TSR can interfere with another process in many different ways, protection against interference involves only three steps:

1. Recording a current configuration
2. Changing the configuration so it applies to the TSR
3. Restoring the original configuration before terminating

The example program described on page 293 demonstrates all the noninterference safeguards described in this section. These safeguards by no means exhaust the subject of noninterference. More sophisticated TSRs may require more sophisticated methods. However, noninterference methods generally fall into one of the following categories:

- Trapping errors
- Preserving an existing condition
- Preserving existing data

Trapping Errors

A TSR committing an error that triggers an interrupt must handle the interrupt to trap the error. Otherwise, the existing interrupt routine, which belongs to the underlying process, would attempt to service an error the underlying process did not commit.

For example, a TSR that accepts keyboard input should include handlers for Interrupts 23h and 1Bh to trap keyboard break signals. When MS-DOS detects CTRL+C from the keyboard or input stream, it transfers control to Interrupt 23h (CTRL+C Handler). Similarly, the BIOS keyboard routine calls Interrupt 1Bh (CTRL+BREAK Handler) when it detects a CTRL+BREAK key combination. Both routines normally terminate the current process.

A TSR that calls MS-DOS should also trap critical errors through Interrupt 24h (Critical Error Handler). MS-DOS functions call Interrupt 24h when they encounter certain hardware errors. The TSR must not allow the existing interrupt routine to service the error, since the routine might allow the user to abort service and return control to MS-DOS. This would terminate both the

TSR and the underlying process. By handling Interrupt 24h, the TSR retains control if a critical error occurs.

An error-trapping handler differs in two ways from a TSR's other handlers:

1. It is temporary, in service only while the TSR executes. At startup, the TSR copies the handler's address to the interrupt vector table; it then restores the original vector before returning.
2. It provides complete service for the interrupt; it does not pass control on to the original routine.

Error-trapping handlers often set a flag to let the TSR know the error has occurred. For example, a handler for Interrupt 1Bh might set a flag when the user presses CTRL+BREAK. The TSR can check the flag as it polls for keyboard input, as shown here:

```
BrkHandler PROC FAR               ; Handler for Interrupt 1Bh
        .
        .
        .
        mov     cs:BreakFlag, TRUE ; Raise break flag
        iret                       ; Terminate interrupt

BrkHandler ENDP
        .
        .
        .
        mov     BreakFlag, FALSE   ; Initialize break flag
poll:   .
        .
        .
        cmp     BreakFlag, TRUE    ; Keyboard break pressed?
        je      exit               ; If so, break polling loop
        mov     ah, 1
        int     16h                ; Key waiting?
        jnz     poll               ; If not, repeat polling loop
```

Preserving an Existing Condition

A TSR and its interrupt handlers must preserve register values so that all registers are returned intact to the interrupted process. This is usually done by pushing the registers onto the stack before changing them, then popping the original values before returning.

Setting up a new stack is another important safeguard against interference. A TSR should usually provide its own stack to avoid the possibility of overrunning the current stack. Exceptions to this rule are simple TSRs such as the sample program ALARM that make minimal stack demands.

A TSR that alters the video configuration should return the configuration to its original state upon return. Video configuration includes cursor position, cursor shape, and video mode. The services provided through Interrupt 10h enable a TSR to determine the existing configuration and alter it if necessary.

However, some applications set video parameters by directly programming the video controller. When this happens, BIOS remains unaware of the new configuration and consequently returns inaccurate information to the TSR. Unfortunately, there is no solution to this problem if the controller's data registers provide write-only access and thus cannot be queried directly. For more information on video controllers, refer to Richard Wilton, *Programmer's Guide to the PC & PS/2 Video Systems*. (See "Books for Further Reading" in the Introduction.)

Preserving Existing Data

A TSR requires its own disk transfer area (DTA) if it calls MS-DOS functions that access the DTA. These include file control block functions and Functions 11h, 12h, 4Eh, and 4Fh. The TSR must switch to a new DTA to avoid overwriting the one belonging to the interrupted process. On becoming active, the TSR calls Function 2Fh to obtain the address of the current DTA. The TSR stores the address and then calls Function 1Ah to establish a new DTA. Before returning, the TSR again calls Function 1Ah to restore the address of the original DTA.

MS-DOS versions 3.1 and later allow a TSR to preserve extended error information. This prevents the TSR from destroying the original information if it commits an MS-DOS error. The TSR retrieves the current extended error data by calling MS-DOS Function 59h. It then copies registers AX, BX, CX, DX, SI, DI, DS, and ES to an 11-word data structure in the order given. MS-DOS reserves the last three words of the structure, which should each be set to zero. Before returning, the TSR calls Function 5Dh with AL = 0Ah and DS:DX pointing to the data structure. This call restores the extended error data to their original state.

Communicating Through the Multiplex Interrupt

The Multiplex interrupt (Interrupt 2Fh) provides the Microsoft-approved way for a program to verify the presence of an installed TSR and to exchange information with it. MS-DOS version 2.*x* uses Interrupt 2Fh only as an interface for the resident print spooler utility PRINT.COM. Later MS-DOS versions standardize calling conventions so that multiple TSRs can share the interrupt.

A TSR chains to the Multiplex interrupt by setting up a handler. The TSR's installation code records the Interrupt 2Fh vector and then replaces it with the address of the new multiplex handler.

The Multiplex Handler

A program communicates with a multiplex handler by calling Interrupt 2Fh with an identity number in the AH register. As each handler in the chain gains control, it compares the value in AH with its own identity number. If the handler finds that it is not the intended recipient of the call, it passes control to the previous handler. The process continues until control reaches the target handler. When the target handler finishes its tasks, it returns via an **IRET** instruction to terminate the interrupt.

The target handler determines its tasks from the function number in AL. Convention reserves Function 0 as a request for installation status. A multiplex handler must respond to Function 0 by setting AL to 0FFh, to inform the caller of the handler's presence in memory. The handler should also return other information to provide a completely reliable identification. For example, it might return in ES:BX a far pointer to the TSR's copyright notice. This assures the caller it has located the intended TSR and not another TSR that has already claimed the identity number in AH.

Identity numbers range from 192 to 255, since MS-DOS reserves lesser values for its own use. During installation, a TSR must verify the uniqueness of its number. It must not set up a multiplex handler identified by a number already in use. A TSR usually obtains its identity number through one of the following methods:

- The programmer assigns the number in the program.
- The user chooses the number by entering it as an argument in the command line, placing it into an environment variable, or by altering the contents of an initialization file.
- The TSR selects its own number through a process of trial and error.

The last method offers the most flexibility. It finds an identity number not currently in use among the installed multiplex handlers and does not require intervention from the user.

To use this method, a TSR calls Interrupt 2Fh during installation with AH = 192 and AL = 0. If the call returns AL = 0FFh, the program tests other registers to determine if it has found a prior installation of itself. If the test fails, the program resets AL to zero, increments AH to 193, and again calls Interrupt 2Fh. The process repeats with incrementing values in AH until the TSR locates a prior installation of itself—in which case it should abort with an appropriate message to the user—or until AL returns as zero. The TSR can then use the value in AH as its identity number and proceed with installation.

The SNAP.ASM program in this chapter demonstrates how a TSR can use this trial-and-error method to select a unique identity number. During installation, the program calls Interrupt 2Fh to verify that SNAP is not already installed. When

deinstalling, the program again calls Interrupt 2Fh to locate the resident TSR in memory. SNAP's multiplex handler services the call and returns the address of the resident code's program-segment prefix. The calling program can then locate the resident code and deinstall it, as explained in "Deinstalling a TSR," following.

Using the Multiplex Interrupt Under MS-DOS Version 2.*x*

A TSR can use the Multiplex interrupt under MS-DOS version 2.*x*, with certain limitations. Under version 2.*x*, only MS-DOS's print spooler PRINT, itself a TSR program, provides an Interrupt 2Fh service. The Interrupt 2Fh vector remains null until PRINT or another TSR is installed that sets up a multiplex handler.

Therefore, a TSR running under version 2.*x* must first check the existing Interrupt 2Fh vector before installing a multiplex handler. The TSR locates the current Interrupt 2Fh handler through Function 35h (Get Interrupt Vector). If the function returns a null vector, the TSR's handler will be last in the chain of Interrupt 2Fh handlers. The handler must terminate with an **IRET** instruction rather than pass control to a nonexistent routine.

PRINT in MS-DOS version 2.*x* does not pass control to the previous handler. If you intend to run PRINT under version 2.*x*, the program must be installed before other TSRs that also handle Interrupt 2Fh. This places PRINT's multiplex handler last in the chain of handlers.

Deinstalling a TSR

A TSR should provide a means for the user to remove or "deinstall" it from memory. Deinstallation returns occupied memory to the system, offering these benefits:

- The freed memory becomes available to subsequent programs that may require additional memory space.
- Deinstallation restores the system to a normal state. Thus, sensitive programs that may be incompatible with TSRs can execute without the presence of installed routines.

A deinstallation program must first locate the TSR in memory, usually by requesting an address from the TSR's multiplex handler. When it has located the TSR, the deinstallation program should then compare addresses in the vector table with the addresses of the TSR's handlers. A mismatch indicates that another TSR has chained a handler to the interrupt routine. In this case, the deinstallation program should deny the request to deinstall. If the addresses of

the TSR's handlers match those in the vector table, deinstallation can safely continue.

You can deinstall the TSR with these three steps:

1. Restore to the vector table the original interrupt vectors replaced by the handler addresses.
2. Read the segment address stored at offset 2Ch of the resident TSR's program segment prefix (PSP). This address points to the TSR's "environment block," a list of environment variables that MS-DOS copies into memory when it loads a program. Place the block's address in the ES register and call MS-DOS Function 49h (Release Memory Block) to return the block's memory to the operating system.
3. Place the resident PSP segment address in ES and again call Function 49h. This call releases the block of memory occupied by the TSR's code and data.

The example program in the next section demonstrates how to locate a resident TSR through its multiplex handler, and deinstall it from memory.

Example of an Advanced TSR: SNAP

This section presents SNAP, a memory-resident program that demonstrates most of the techniques discussed in this chapter. SNAP takes a snapshot of the current screen and copies the text to a specified file. SNAP accommodates screens with various column and line counts, such as CGA's 40-column mode or VGA's 50-line mode. The program ignores graphics screens.

Once installed, SNAP occupies approximately 7.5K of memory. When it detects the ALT+LEFT SHIFT+S key combination, SNAP displays a prompt for a file specification. The user can type a new filename, accept the previous filename by pressing ENTER, or cancel the request by pressing ESC.

SNAP reads text directly from the video buffer and copies it to the specified file. The program sets the file pointer to the end of the file so that text is appended without overwriting previous data. SNAP copies each line only to the last character, ignoring trailing spaces. The program adds a carriage return–linefeed sequence (0D0Ah) to the end of each line. This makes the file accessible to any text editor that can read ASCII files.

To demonstrate how a program accesses resident data through the Multiplex interrupt, SNAP can reset the display attribute of its prompt box. After installing SNAP, run the main program with the /C option to change box colors:

```
SNAP /Cxx
```

The argument *xx* specifies the desired attribute as a two-digit hexadecimal number—for example, 7C for red on white, or 0F for monochrome high

intensity. For a list of color and monochrome display attributes, refer to the "Tables" section of the *Reference*.

SNAP can deinstall itself, provided another TSR has not been loaded after it. Deinstall SNAP by executing the main program with the /D option:

```
SNAP /D
```

If SNAP successfully deinstalls, it displays the following message:

```
TSR deinstalled
```

Building SNAP.EXE

SNAP combines four modules: SNAP.ASM, COMMON.ASM, HANDLERS.ASM, and INSTALL.ASM. Source files are located on one of your distribution disks. Each module stores temporary code and data in the segments INSTALLCODE and INSTALLDATA. These segments apply only to SNAP's installation phase; MS-DOS recovers the memory they occupy when the program exits through the terminate-and-stay-resident function. The following briefly describes each module:

- SNAP.ASM contains the TSR's main code and data.
- COMMON.ASM contains procedures used by other example programs.
- HANDLERS.ASM contains interrupt handler routines for Interrupts 08, 09, 10h, 13h, 15h, 28h, and 2Fh. It also provides simple error-trapping handlers for Interrupts 1Bh, 23h, and 24h. Additional routines set up and deinstall the handlers.
- INSTALL.ASM contains an exit routine that calls the terminate-and-stay-resident function and a deinstallation routine that removes the program from memory. The module includes error-checking services and a command-line parser.

This building-block approach allows you to create other TSRs by replacing SNAP.ASM and linking with the HANDLERS and INSTALL object modules. The library of routines accommodates both keyboard-activated and time-activated TSRs. A time-activated TSR is a program that activates at a predetermined time of day, similar to the example program ALARM introduced earlier in this chapter. The header comments for the `Install` procedure in HANDLERS.ASM explain how to install a time-activated TSR.

You can write new TSRs in assembly language or any high-level language that conforms to the Microsoft conventions for ordering segments. Regardless of the language, the new code must not invoke an MS-DOS function that sets up the I/O stack (see "Interrupting MS-DOS Functions," earlier in this chapter). Code

in Microsoft C, for example, must not call **getche** or **kbhit**, since these functions in turn call MS-DOS Functions 01 and 0Bh.

Code written in a high-level language must not check for stack overflows. Compiler-generated stack probes do not recognize the new stack setup when the TSR executes, and therefore must be disabled. The example program BELL.C, included on disk with the TSR library routines, demonstrates how to disable stack checking in Microsoft C using the **check_stack** pragma.

Outline of SNAP

The following sections outline in detail how SNAP works. Each part of the outline covers a specific portion of SNAP's code. Headings refer to earlier sections of this chapter, providing cross-references to SNAP's key procedures. For example, the part of the outline that describes how SNAP searches for its startup signal refers to the section "Auditing Hardware Events for TSR Requests," earlier in this chapter.

Figures 11.2 through 11.4 are flowcharts of the SNAP program. Each chart illustrates a separate phase of SNAP's operation, from installation through memory-residency to deinstallation.

Figure 11.2 Flowchart for SNAP.EXE: Installation Phase

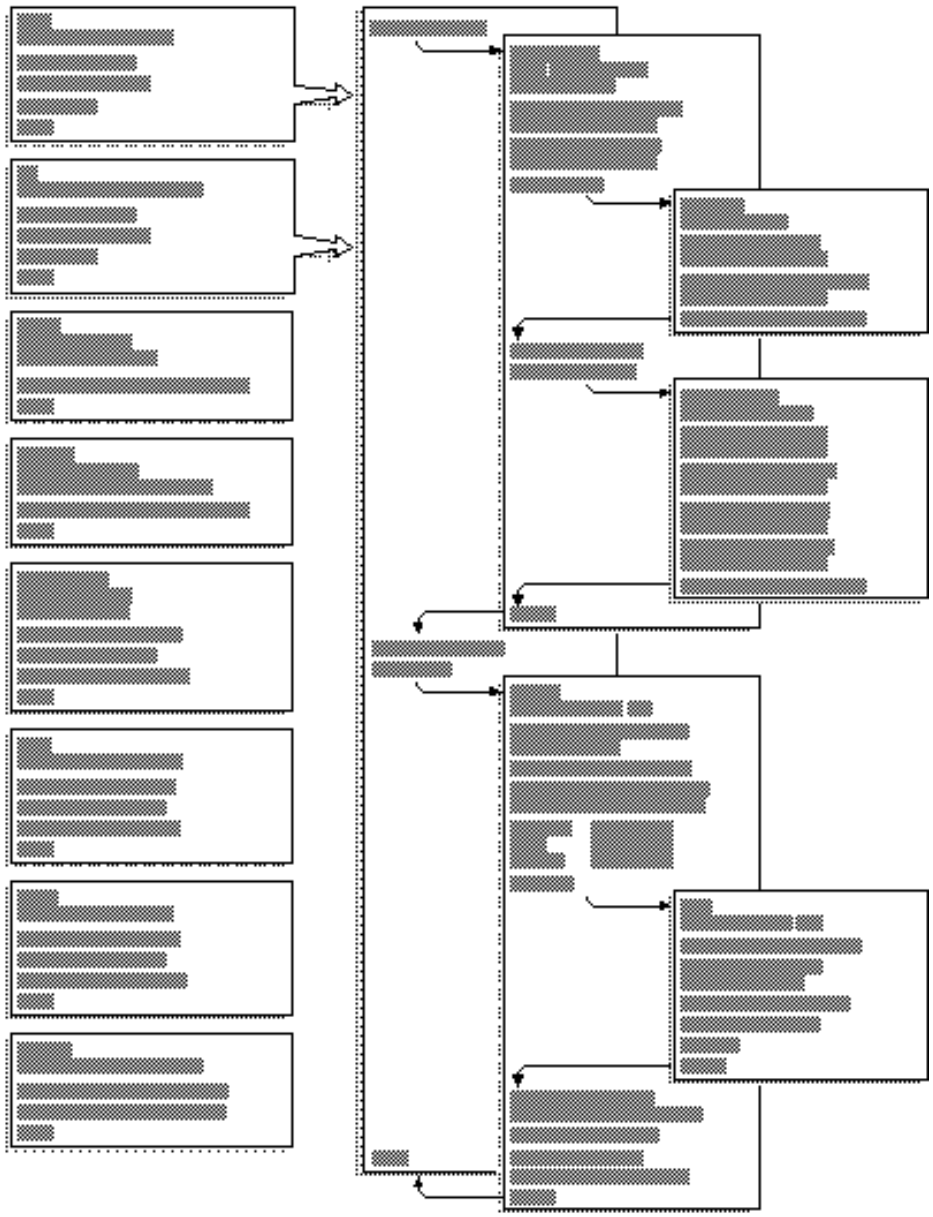

Figure 11.3 Flowchart for SNAP.EXE: Resident Phase

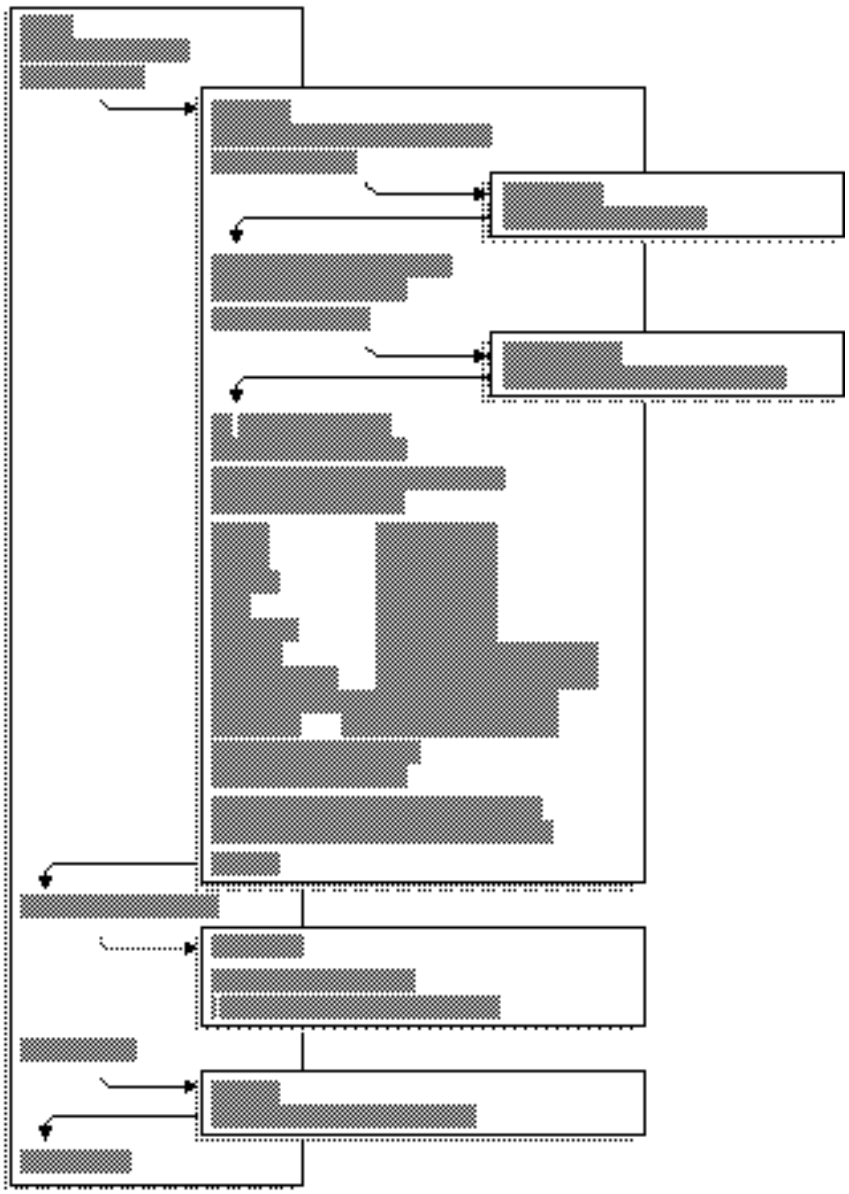

Figure 11.4 Flowchart for SNAP.EXE: Deinstallation Phase

Refer to the flowcharts as you read the following outline. They will help you maintain perspective while exploring the details of SNAP's operation. Text in the outline cross-references the charts.

Note that information in both the outline and the flowcharts is generic. Except for references to the SNAP procedure, all descriptions in the outline and the

flowcharts apply to any TSR created with the HANDLERS and INSTALL modules.

Auditing Hardware Events for TSR Requests

To search for its startup signal, SNAP audits the keyboard with an interrupt handler for either Interrupt 09 (keyboard) or Interrupt 15h (Miscellaneous System Services). The `Install` procedure determines which of the two interrupts to handle based on the following code:

```
        .IF     HotScan == 0    ; If valid scan code given:
        mov     ah, HotShift    ; AH = hour to activate
        mov     al, HotMask     ; AL = minute to activate
        call    GetTimeToElapse ; Get number of 5-second intervals
        mov     CountDown, ax   ;   to elapse before activation

        .ELSE                   ; Force use of KeybrdMonitor as
                                ;   keyboard handler
        cmp     Version, 031Eh  ; DOS Version 3.3 or higher?
        jb      setup           ; No?  Skip next step

; Test for IBM PS/2 series. If not PS/2, use Keybrd and
; SkipMiscServ as handlers for Interrupts 09 and 15h
; respectively. If PS/2 system, set up KeybrdMonitor as the
; Interrupt 09 handler. Audit keystrokes with MiscServ
; handler, which searches for the hot key by handling calls
; to Interrupt 15h (Miscellaneous System Services). Refer to
; Section 11.2.1 for more information about keyboard handlers.

        mov     ax, 0C00h       ; Function 0Ch (Get System
        int     15h             ;   Configuration Parameters)
        sti                     ; Compaq ROM may leave disabled

        jc      setup           ; If carry set,
        or      ah, ah          ;   or if AH not 0,
        jnz     setup           ;   services are not supported

; Test bit 4 to see if Intercept is implemented
        test    BYTE PTR es:[bx+5], 00010000y
        jz      setup

; If so, set up MiscServ as Interrupt 15h handler
        mov     ax, OFFSET MiscServ
        mov     WORD PTR intMisc.NewHand, ax
        .ENDIF

; Set up KeybrdMonitor as Interrupt 09 handler
        mov     ax, OFFSET KeybrdMonitor
        mov     WORD PTR intKeybrd.NewHand, ax
```

The following describes the code's logic:

- If the program is running under MS-DOS version 3.3 or higher and if Interrupt 15h supports Function 4Fh, set up handler `MiscServ` to search for the hot key. Handle Interrupt 09 with `KeybrdMonitor` only to maintain the keyboard active flag.
- Otherwise, set up a handler for Interrupt 09 to search for the hot key. Handle calls to Interrupt 15h with the routine `SkipMiscServ`, which contains this single instruction:

  ```
  jmp     cs:intMisc.OldHand
  ```

  The jump immediately passes control to the original Interrupt 15h routine; thus, `SkipMiscServ` has no effect. It serves only to simplify coding in other parts of the program.

At each keystroke, the keyboard interrupt handler (either `Keybrd` or `MiscServ`) calls the procedure `CheckHotKey` with the scan code of the current key. `CheckHotKey` compares the scan code and shift status with the bytes `HotScan` and `HotShift`. If the current key matches, `CheckHotKey` returns the carry flag clear to indicate that the user has pressed the hot key.

If the keyboard handler finds the carry flag clear, it sets the flag `TsrRequestFlag` and exits. Otherwise, the handler transfers control to the original interrupt routine to service the interrupt.

The timer handler `Clock` reads the request flag at every occurrence of the timer interrupt. `Clock` takes no action if it finds a zero value in `TsrRequestFlag`. Figures 11.1 and 11.3 depict the relationship between the keyboard and timer handlers.

Monitoring System Status

Because SNAP produces output to both video and disk, it avoids interrupting either video or disk operations. The program uses interrupt handlers `Video` and `DiskIO` to monitor Interrupts 10h (video) and 13h (disk). SNAP also avoids interrupting keyboard use. The instructions at the far label `KeybrdMonitor` serve as the monitor handler for Interrupt 09 (keyboard).

The three handlers perform similar functions. Each sets an active flag and then calls the original routine to service the interrupt. When the service routine returns, the handler clears the active flag to indicate that the device is no longer in use.

The BIOS Interrupt 13h routine clears or sets the carry flag to indicate the operation's success or failure. `DiskIO` therefore preserves the flags register when returning, as shown here:

```
DiskIO  PROC    FAR
        mov     cs:intDiskIO.Flag, TRUE ; Set active flag
; Simulate interrupt by pushing flags and far-calling old
; Int 13h routine
        pushf
        call    cs:intDiskIO.OldHand
; Clear active flag without disturbing flags register
        mov     cs:intDiskIO.Flag, FALSE
        sti                             ; Enable interrupts
; Simulate IRET without popping flags (since services use
; carry flag)
        ret     2
DiskIO  ENDP
```

The terminating **RET 2** instruction discards the original flags from the stack when the handler returns.

Determining Whether to Invoke the TSR

The procedure `CheckRequest` determines whether the TSR:

- Has been requested.
- Can safely interrupt the system.

Each time it executes, the timer handler `Clock` calls `CheckRequest` to read the flag `TsrRequestFlag`. If `CheckRequest` finds the flag set, it scans other flags maintained by the TSR's interrupt handlers and by MS-DOS. These flags indicate the current system status. As the flowchart in Figure 11.3 shows, `CheckRequest` calls `CheckDos` (described following) to determine the status of the operating system. `CheckRequest` then calls `CheckHardware` to check hardware status.

`CheckHardware` queries the interrupt controller to determine if any device is currently being serviced. It also reads the active flags maintained by the `KeybrdMonitor`, `Video`, and `DiskIO` handlers. If the controller, keyboard, video, and disk are all inactive, `CheckHardware` clears the carry flag and returns.

`CheckRequest` indicates system status with the carry flag. If the procedure returns the carry flag set, the caller exits without invoking the TSR. A clear carry signals that the caller can safely execute the TSR.

Determining MS-DOS Activity

As Figure 11.2 shows, the procedure `GetDosFlags` locates the InDos flag during SNAP's installation phase. `GetDosFlags` calls Function 34h (Get Address of InDos Flag) and then stores the flag's address in the far pointer `InDosAddr`.

When called from the `CheckRequest` procedure, `CheckDos` reads InDos to determine whether the operating system is active. Note that `CheckDos` reads the flag directly from the address in `InDosAddr`. It does not call Function 34h to locate the flag, since it has not yet established whether MS-DOS is active. This follows from the general rule that interrupt handlers must not call any MS-DOS function.

The next two sections more fully describe the procedure `CheckDos`.

Interrupting MS-DOS Functions

Figure 11.3 shows that the call to `CheckDos` can initiate either from `Clock` (timer handler) or `Idle` (Interrupt 28h handler). If `CheckDos` finds the InDos flag set, it reacts in different ways, depending on the caller:

- If called from `Clock`, `CheckDos` cannot know which MS-DOS function is active. In this case, it returns the carry flag set, indicating that `Clock` must deny the request for the TSR.
- If called from `Idle`, `CheckDos` assumes that one of the low-order polling functions is active. It therefore clears the carry flag to let the caller know the TSR can safely interrupt the function.

For more information on this topic, see the section "Interrupting MS-DOS Functions," earlier in this chapter.

Monitoring the Critical Error Flag

The procedure `GetDosFlags` (Figure 11.2) determines the address of the Critical Error flag. The procedure stores the flag's address in the far pointer `CritErrAddr`.

When called from either the `Clock` or `Idle` handlers, `CheckDos` reads the Critical Error flag. A nonzero value in the flag indicates that the Critical Error Handler (Interrupt 24h) is processing a critical error and the TSR must not interrupt. In this case, `CheckDos` sets the carry flag and returns, causing the caller to exit without executing the TSR.

Trapping Errors

As Figure 11.3 shows, `Clock` and `Idle` invoke the TSR by calling the procedure `Activate`. Before calling the main body of the TSR, `Activate` sets up the following handlers:

Handler Name	For Interrupt	Receives Control When
CtrlBreak	1Bh (CTRL+BREAK Handler)	CTRL+BREAK sequence entered at keyboard
CtrlC	23h (CTRL+C Handler)	MS-DOS detects a CTRL+C sequence from the keyboard or input stream
CritError	24h (Critical Error Handler)	MS-DOS encounters a critical error

These handlers trap keyboard break signals and critical errors that would otherwise trigger the original handler routines. The `CtrlBreak` and `CtrlC` handlers contain a single **IRET** instruction, thus rendering a keyboard break ineffective. The `CritError` handler contains the following instructions:

```
CritError PROC  FAR
        sti
        sub     al, al          ; Assume DOS 2.x
                                ; Set AL = 0 for ignore error
        .IF     cs:major != 2   ; If DOS 3.x, set AL = 3
        mov     al, 3           ; DOS call fails
        .ENDIF
        iret
CritError ENDP
```

The return code in AL stops MS-DOS from taking further action when it encounters a critical error.

As an added precaution, `Activate` also calls Function 33h (Get or Set CTRL+BREAK Flag) to determine the current setting of the checking flag. `Activate` stores the setting, then calls Function 33h again to turn off break checking.

When the TSR's main procedure finishes its work, it returns to `Activate`, which restores the original setting for the checking flag. It also replaces the original vectors for Interrupts 1Bh, 23h, and 24h.

SNAP's error-trapping safeguards enable the TSR to retain control in the event of an error. Pressing CTRL+BREAK or CTRL+C at SNAP's prompt has no effect. If the user specifies a nonexistent drive—a critical error—SNAP merely beeps the speaker and returns normally.

Preserving an Existing Condition

`Activate` records the stack pointer SS:SP in the doubleword `OldStackAddr`. The procedure then resets the pointer to the address of a new stack before calling the TSR. Switching stacks ensures that SNAP has adequate stack depth while it executes.

The label `NewStack` points to the top of the new stack buffer, located in the code segment of the HANDLERS.ASM module. The equate constant

`STACK_SIZ` determines the size of the stack. The include file TSR.INC contains the declaration for `STACK_SIZ`.

`Activate` preserves the values in all registers by pushing them onto the new stack. It does not push DS, since that register is already preserved in the `Clock` or `Idle` handler.

SNAP does not alter the application's video configuration other than by moving the cursor. Figure 11.3 shows that `Activate` calls the procedure `Snap`, which executes Interrupt 10h to determine the current cursor position. `Snap` stores the row and column in the word `OldPos`. The procedure restores the cursor to its original location before returning to `Activate`.

Preserving Existing Data

Because SNAP does not call an MS-DOS function that writes to the DTA, it does not need to preserve the DTA belonging to the interrupted process. However, the code for switching and restoring the DTA is included within **IFDEF** blocks in the procedure `Activate`. The equate constant `DTA_SIZ`, declared in the TSR.INC file, governs the assembly of the blocks as well as the size of the new DTA.

It is possible for SNAP to overwrite existing extended error information by committing a file error. The program does not attempt to preserve the original information by calling Functions 59h and 5Dh. In certain rare instances, this may confuse the interrupted process after SNAP returns.

Communicating Through the Multiplex Interrupt

The program uses the Multiplex interrupt (Interrupt 2Fh) to

- Verify that SNAP is installed.
- Select a unique multiplex identity number.
- Locate resident data.

For more information about Interrupt 2Fh, see the section "Communicating through the Multiplex Interrupt," earlier in this chapter.

SNAP accesses Interrupt 2Fh through the procedure `CallMultiplex`, as shown in Figures 11.2 and 11.4. By searching for a prior installation, `CallMultiplex` ensures that SNAP is not installed more than once. During deinstallation, `CallMultiplex` locates data required to deinstall the resident TSR.

The procedure `Multiplex` serves as SNAP's multiplex handler. When it recognizes its identity number in AH, `Multiplex` determines its tasks from the function number in the AL register. The handler responds to Function 0 by

returning AL equalling 0FFh and ES:DI pointing to an identifier string unique to SNAP.

`CallMultiplex` searches for the handler by invoking Interrupt 2Fh in a loop, beginning with a trial identity number of 192 in AH. At the start of each iteration of the loop, the procedure sets AL to zero to request presence verification from the multiplex handler. If the handler returns 0FFh in AL, `CallMultiplex` compares its copy of SNAP's identifier string with the text at memory location ES:DI. A failed match indicates that the multiplex handler servicing the call is not SNAP's handler. In this case, `CallMultiplex` increments AH and cycles back to the beginning of the loop.

The process repeats until the call to Interrupt 2Fh returns a matching identifier string at ES:DI, or until AL returns as zero. A matching string verifies that SNAP is installed, since its multiplex handler has serviced the call. A return value of zero indicates that SNAP is not installed and that no multiplex handler claims the trial identity number in AH. In this case, SNAP assigns the number to its own handler.

Deinstalling a TSR

During deinstallation, `CallMultiplex` locates SNAP's multiplex handler as described previously. The handler `Multiplex` receives the verification request and returns in ES the code segment of the resident program.

`Deinstall` reads the addresses of the following interrupt handlers from the data structure in the resident code segment:

Handler Name	Description
Clock	Timer handler
Keybrd	Keyboard handler (non-PS/2)
KeybrdMonitor	Keyboard monitor handler (PS/2)
Video	Video monitor handler
DiskIO	Disk monitor handler
SkipMiscServ	Miscellaneous Systems Services handler (non-PS/2)
MiscServ	Miscellaneous Systems Services handler (PS/2)
Idle	MS-DOS Idle handler
Multiplex	Multiplex handler

`Deinstall` calls MS-DOS Function 35h (Get Interrupt Vector) to retrieve the current vectors for each of the listed interrupts. By comparing each handler address with the corresponding vector, `Deinstall` ensures that SNAP can be safely deinstalled. Failure in any of the comparisons indicates that another TSR has been installed after SNAP and has set up a handler for the same interrupt. In

this case, `Deinstall` returns an error code, stopping the program with the following message:

```
Can't deinstall TSR
```

If all addresses match, `Deinstall` calls Interrupt 2Fh with SNAP's identity number in AH and AL set to 1. The handler `Multiplex` responds by returning in ES the address of the resident code's PSP. `Deinstall` then calls MS-DOS Function 25h (Set Interrupt Vector) to restore the vectors for the original service routines. This is called "unhooking" or "unchaining" the interrupt handlers.

After unhooking all of SNAP's interrupt handlers, `Deinstall` returns with AX pointing to the resident code's PSP. The procedure `FreeTsr` then calls MS-DOS Function 49h (Release Memory) to return SNAP's memory to the operating system. The program ends with the message

```
TSR deinstalled
```

to indicate a successful deinstallation.

Deinstalling SNAP does not guarantee more available memory space for the next program. If another TSR loads after SNAP but handles interrupts other than 08, 09, 10h, 13h, 15h, 28h, or 2Fh, SNAP still deinstalls properly. The result is a harmless gap of deallocated memory formerly occupied by SNAP. MS-DOS can use the free memory to store the next program's environment block. However, MS-DOS loads the program itself above the still-resident TSR.