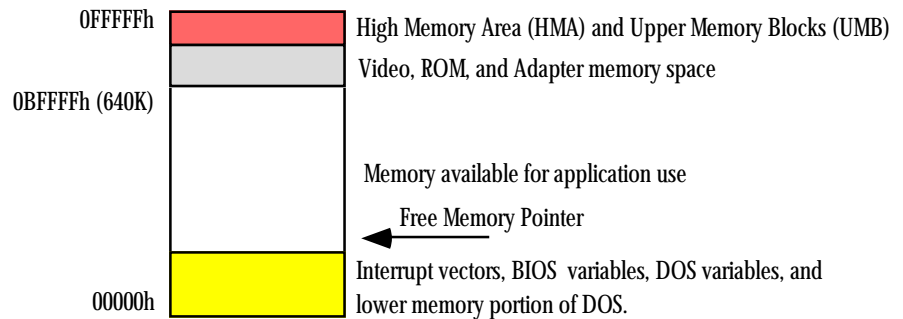


Most MS-DOS applications are *transient*. They load into memory, execute, terminate, and DOS uses the memory allocated to the application for the next program the user executes. Resident programs follow these same rules, except for the last. A resident program, upon termination, does not return all memory back to DOS. Instead, a portion of the program remains *resident*, ready to be reactivated by some other program at a future time.

Resident programs, also known as *terminate and stay resident programs* or *TSRs*, provide a tiny amount of *multitasking* to an otherwise single tasking operating system. Until Microsoft Windows became popular, resident programs were the most popular way to allow multiple applications to coexist in memory at one time. Although Windows has diminished the need for TSRs for background processing, TSRs are still valuable for writing *device drivers*, *antiviral tools*, and *program patches*. This chapter will discuss the issues you must deal with when writing resident programs.

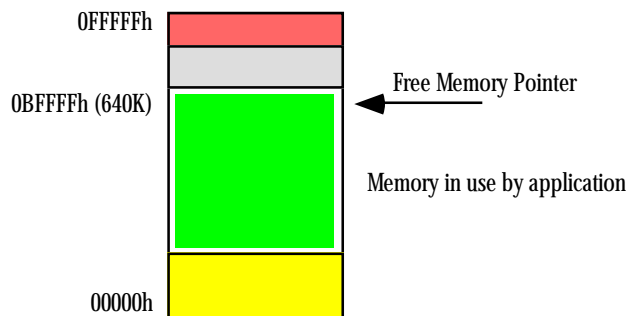
## 18.1 DOS Memory Usage and TSRs

When you first boot DOS, the memory layout will look something like the following:



DOS Memory Map (no active application)

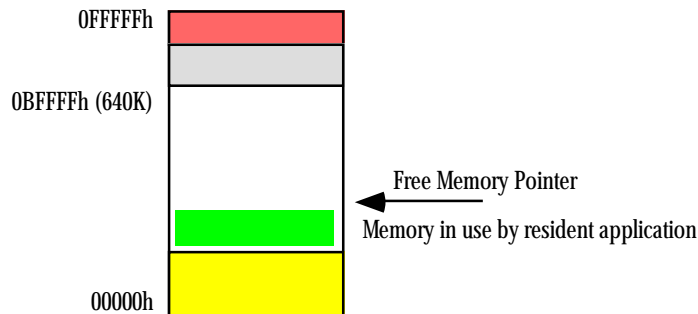
DOS maintains a *free memory pointer* that points to the beginning of the block of free memory. When the user runs an application program, DOS loads this application starting at the address the free memory pointer contains. Since DOS generally runs only a single application at a time, all the memory from the free memory pointer to the end of RAM (0BFFFFh) is available for the application's use:



DOS Memory Map (w/active application)

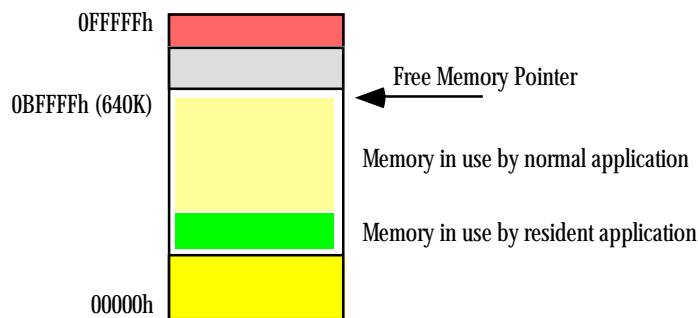
When the program terminates normally via DOS function 4Ch (the Standard Library `exitpgm` macro), MS-DOS reclaims the memory in use by the application and resets the free memory pointer to just above DOS in low memory.

MS-DOS provides a second termination call which is identical to the terminate call with one exception, it does not reset the free memory pointer to reclaim all the memory in use by the application. Instead, this *terminate and stay resident* call frees all but a specified block of memory. The TSR call (ah=31h) requires two parameters, a process termination code in the a1 register (usually zero) and dx must contain the size of the memory block to protect, in paragraphs. When DOS executes this code, it adjusts the free memory pointer so that it points to a location dx\*16 bytes above the program's PSP (see "MS-DOS, PC-BIOS, and File I/O" on page 699). This leaves memory looking like this:



### DOS Memory Map (w/resident application)

When the user executes a new application, DOS loads it into memory at the new free memory pointer address, protecting the resident program in memory:



### DOS Memory Map (w/resident and normal application)

When this new application terminates, DOS reclaims its memory and readjusts the free memory pointer to its location before running the application – just above the resident program. By using this free memory pointer scheme, DOS can protect the memory in use by the resident program<sup>1</sup>.

The trick to using the terminate and stay resident call is to figure out how many paragraphs should remain resident. Most TSRs contain two sections of code: a *resident* portion and a *transient* portion. The transient portion is the data, main program, and support routines that execute when you run the program from the command line. This code will probably never execute again. Therefore, you should not leave it in memory when your program terminates. After all, every byte consumed by the TSR program is one less byte available to other application programs.

The resident portion of the program is the code that remains in memory and provides whatever functions are necessary of the TSR. Since the PSP is usually right before the first byte of program code, to effectively use the DOS TSR call, your program must be organized as follows:

1. Of course, DOS could never protect the resident program from an errant application. If the application decides to write zeros all over memory, the resident program, DOS, and many other memory areas will be destroyed.



## Memory Organization for a Resident Program

To use TSRs effectively, you need to organize your code and data so that the resident portions of your program loads into lower memory addresses and the transient portions load into the higher memory addresses. MASM and the Microsoft Linker both provide facilities that let you control the loading order of segments within your code (see “MASM: Directives & Pseudo-Opcodes” on page 355). The simple solution, however, is to put all your resident code and data in a single segment and make sure that this segment appears *first* in every source module of your program. In particular, if you are using the UCR Standard Library SHELL.ASM file, you must make sure that you define your resident segments *before* the include directives for the standard library files. Otherwise MS-DOS will load all the standard library routines *before* your resident segment and that would waste considerable memory. Note that you only need to define your resident segment first, you do not have to place all the resident code and data before the includes. The following will work just fine:

```

ResidentSeg  segment  para public 'resident'
ResidentSeg  ends

EndResident  segment  para public 'EndRes'
EndResident  ends

                .xlist
                include  stdlib.a
                includelib stdlib.lib
                .list

ResidentSeg  segment  para public 'resident'
                assume   cs:ResidentSeg, ds:ResidentSeg

PSP           word    ?                ;This var must be here!

; Put resident code and data here

ResidentSeg  ends

dseg          segment  para public 'data'

; Put transient data here

dseg          ends

cseg          segment  para public 'code'
                assume   cs:cseg, ds:dseg

; Put Transient code here.

cseg          ends
                etc.

```

The purpose of the EndResident segment will become clear in a moment. For more information on DOS memory ordering, see Chapter Six.

Now the only problem is to figure out the size of the resident code, in paragraphs. With your code structured in the manner shown above, determining the size of the resident program is quite easy, just use the following statements to terminate the transient portion of your code (in cseg):

```

        mov     ax, ResidentSeg    ;Need access to ResidentSeg
        mov     es, ax
        mov     ah, 62h           ;DOS Get PSP call.
        int     21h
        mov     es:PSP, bx        ;Save PSP value in PSP variable.

; The following code computes the size of the resident portion of the code.
; The EndResident segment is the first segment in memory after resident code.
; The program's PSP value is the segment address of the start of the resident
; block. By computing EndResident-PSP we compute the size of the resident
; portion in paragraphs.

        mov     dx, EndResident    ;Get EndResident segment address.
        sub     dx, bx             ;Subtract PSP.

; Okay, execute the TSR call, preserving only the resident code.

        mov     ax, 3100h         ;AH=31h (TSR), AL=0 (return code).
        int     21h

```

Executing the code above returns control to MS-DOS, preserving your resident code in memory.

There is one final memory management detail to consider before moving on to other topics related to resident programs – accessing data within an resident program. Procedures within a resident program become active in response to a direct call from some other program or a hardware interrupt (see the next section). Upon entry, the resident routine *may* specify that certain registers contain various parameters, but one thing you cannot expect is for the calling code to properly set up the segment registers for you. Indeed, the only segment register that will contain a meaningful value (to the resident code) is the code segment register. Since many resident functions will want to access local data, this means that those functions may need to set up *ds* or some other segment register(s) upon initial entry. For example, suppose you have a function, *count*, that simply counts the number of times some other code calls it once it has gone resident. One would think that the body of this function would contain a single instruction: *inc counter*. Unfortunately, such an instruction would increment the variable at *counter*'s offset in the current data segment (that is, the segment pointed at by the *ds* register). It is unlikely that *ds* would be pointing at the data segment associated with the *count* procedure. Therefore, you would be incrementing some word in a different segment (probably the caller's data segment). This would produce disastrous results.

There are two solutions to this problem. The first is to put all variables in the code segment (a very common practice in resident sections of code) and use a *cs*: segment override prefix on all your variables. For example, to increment the *counter* variable you could use the instruction *inc cs:counter*. This technique works fine if there are only a few variable references in your procedures. However, it suffers from a few serious drawbacks. First, the segment override prefix makes your instructions larger and slower; this is a serious problem if you access many different variables throughout your resident code. Second, it is easy to forget to place the segment override prefix on a variable, thereby causing the TSR function to wipe out memory in the caller's data segment. Another solution to the segment problem is to change the value in the *ds* register upon entry to a resident procedure and restore it upon exit. The following code demonstrates how to do this:

```

        push    ds                ;Preserve original DS value.
        push    cs                ;Copy CS's value to DS.
        pop     ds
        inc     Counter           ;Bump the variable's value.
        pop     ds                ;Restore original DS value.

```

Of course, using the *cs*: segment override prefix is a much more reasonable solution here. However, had the code been extensive and had accessed many local variables, loading *ds* with *cs* (assuming you put your variables in the resident segment) would be more efficient.

## 18.2 Active vs. Passive TSRs

Microsoft identifies two types of TSR routines: active and passive. A passive TSR is one that activates in response to an explicit call from an executing application program. An active TSR is one that responds to a hardware interrupt or one that a hardware interrupt calls.

TSRs are almost always interrupt service routines (see “80x86 Interrupt Structure and Interrupt Service Routines (ISRs)” on page 996). Active TSRs are typically hardware interrupt service routines and passive TSRs are generally trap handlers (see “Traps” on page 999). Although, in theory, it is possible for a TSR to determine the address of a routine in a passive TSR and call that routine directly, the 80x86 trap mechanism is the perfect device for calling such routines, so most TSRs use it.

Passive TSRs generally provide a callable library of routines or extend some DOS or BIOS call. For example, you might want to reroute all characters an application sends to the printer to a file. By patching into the int 17h vector (see “The PC Parallel Ports” on page 1199) you can intercept all characters destined for the printer<sup>2</sup>. Or you could add additional functionality to a BIOS routine by chaining into its interrupt vector. For example, you could add new function calls to the int 10h BIOS video services routine (see “MS-DOS, PC-BIOS, and File I/O” on page 699) by looking for a special value in ah and passing all other int 10h calls on through to the original handler. Another use of a passive TSR is to provide a brand new set of services through a new interrupt vector that the BIOS does not already provide. The mouse services, provided by the mouse.com driver, is a good example of such a TSR.

Active TSRs generally serve one of two functions. They either service a hardware interrupt directly, or they piggyback off the hardware interrupt so they can activate themselves on a periodic basis without an explicit call from an application. *Pop-up* programs are a good example of active TSRs. A pop-up program chains itself into the PC’s keyboard interrupt (int 9). Pressing a key activates such a program. The program can read the PC’s keyboard port (see “The PC Keyboard” on page 1153) to see if the user is pressing a special key sequence. Should this keysequence appear, the application can save a portion of the screen memory and “pop-up” on the screen, perform some user-requested function, and then restore the screen when done. Borland’s Sidekick™ program is an example of an extremely popular TSR program, though many others exist.

Not all active TSRs are pop-ups, though. Certain viruses are good examples of active TSRs. They patch into various interrupt vectors that activate them automatically so they can go about their dastardly deeds. Fortunately, some anti-viral programs are also good examples of active TSRs, they patch into those same interrupt vectors and detect the activities of a virus and attempt to limit the damage the virus may cause.

Note that a TSR may contain both active and passive components. That is, there may be certain routines that a hardware interrupt invokes and others that an application calls explicitly. However, if any routine in a resident program is active, we’ll claim that the entire TSR is active.

The following program is a short example of a TSR that provides both active and passive routines. This program patches into the int 9 (keyboard interrupt) and int 16h (keyboard trap) interrupt vectors. Every time the system generates a keyboard interrupt, the active routine (int 9) increments a counter. Since the keyboard usually generates two keyboard interrupts per keystroke, dividing this value by two produces the approximate number of keys typed since starting the TSR<sup>3</sup>. A passive routine, tied into the int 16h vector, returns the number of keystrokes to the calling program. The following code provides two programs, the TSR and a short application to display the number of keystrokes since the TSR started running.

```
; This is an example of an active TSR that counts keyboard interrupts
; once activated.

; The resident segment definitions must come before everything else.
```

2. Assuming the application uses DOS or BIOS to print the characters and does not talk directly to the printer port itself.

3. It is not an exact count because some keys generate more than two keyboard interrupts.

```

ResidentSeg  segment    para public 'Resident'
ResidentSeg  ends

EndResident  segment    para public 'EndRes'
EndResident  ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

; Resident segment that holds the TSR code:

ResidentSeg  segment    para public 'Resident'
                assume    cs:ResidentSeg, ds:nothing

; The following variable counts the number of keyboard interrupts

KeyIntCnt    word        0

; These two variables contain the original INT 9 and INT 16h
; interrupt vector values:

OldInt9      dword      ?
OldInt16     dword      ?

; MyInt9-      The system calls this routine every time a keyboard
;              interrupt occurs. This routine increments the
;              KeyIntCnt variable and then passes control on to the
;              original Int9 handler.

MyInt9       proc        far
                inc        ResidentSeg:KeyIntCnt
                jmp        ResidentSeg:OldInt9
MyInt9       endp

; MyInt16-     This is the passive component of this TSR. An
;              application explicitly calls this routine with an
;              INT 16h instruction. If AH contains 0FFh, this
;              routine returns the number of keyboard interrupts
;              in the AX register. If AH contains any other value,
;              this routine passes control to the original INT 16h
;              (keyboard trap) handler.

MyInt16      proc        far
                cmp        ah, 0FFh
                je         ReturnCnt
                jmp        ResidentSeg:OldInt16;Call original handler.

; If AH=0FFh, return the keyboard interrupt count

ReturnCnt:   mov        ax, ResidentSeg:KeyIntCnt
                iredt
MyInt16      endp

ResidentSeg  ends

cseg         segment    para public 'code'
                assume    cs:cseg, ds:ResidentSeg

Main         proc
                meminit

                mov        ax, ResidentSeg
                mov        ds, ax

```

```

        mov     ax, 0
        mov     es, ax

        print
        byte   "Keyboard interrupt counter TSR program",cr,lf
        byte   "Installing...",cr,lf,0

; Patch into the INT 9 and INT 16 interrupt vectors. Note that the
; statements above have made ResidentSeg the current data segment,
; so we can store the old INT 9 and INT 16 values directly into
; the OldInt9 and OldInt16 variables.

        cli                               ;Turn off interrupts!
        mov     ax, es:[9*4]
        mov     word ptr OldInt9, ax
        mov     ax, es:[9*4 + 2]
        mov     word ptr OldInt9+2, ax
        mov     es:[9*4], offset MyInt9
        mov     es:[9*4+2], seg ResidentSeg

        mov     ax, es:[16h*4]
        mov     word ptr OldInt16, ax
        mov     ax, es:[16h*4 + 2]
        mov     word ptr OldInt16+2, ax
        mov     es:[16h*4], offset MyInt16
        mov     es:[16h*4+2], seg ResidentSeg
        sti                               ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

        print
        byte   "Installed.",cr,lf,0

        mov     ah, 62h                    ;Get this program's PSP
        int     21h                        ; value.

        mov     dx, EndResident            ;Compute size of program.
        sub     dx, bx
        mov     ax, 3100h                  ;DOS TSR command.
        int     21h

Main
cseg    endp
        ends

sseg    segment    para stack 'stack'
stk     db         1024 dup ("stack ")
sseg    ends

zzzzzzseg    segment    para public 'zzzzzz'
LastBytes   db         16 dup (?)
zzzzzzseg    ends
end        Main

```

### Here's the application that calls MyInt16 to print the number of keystrokes:

```

; This is the companion program to the keycnt TSR.
; This program calls the "MyInt16" routine in the TSR to
; determine the number of keyboard interrupts. It displays
; the approximate number of keystrokes (keyboard ints/2)
; and quits.

```

```

        .xlist
        include    stdlib.a
        includelib stdlib.lib
        .list

cseg    segment    para public 'code'
        assume     cs:cseg, ds:nothing

Main    proc
        meminit

        print

```

```

        byte    "Approximate number of keys pressed: ",0
        mov     ah, 0FFh
        int    16h
        shr    ax, 1           ;Must divide by two.
        putu
        putcr
        ExitPgm

Main    endp
cseg   ends

sseg   segment    para stack 'stack'
stk    db         1024 dup ("stack ")
sseg   ends

zzzzzseg segment    para public 'zzzzzz'
LastBytes db      16 dup (?)
zzzzzseg ends
end    Main

```

---

## 18.3 Reentrancy

One big problem with active TSRs is that their invocation is asynchronous. They can activate at the touch of a keystroke, timer interrupt, or via an incoming character on the serial port, just to name a few. Since they activate on a hardware interrupt, the PC could have been executing just about any code when the interrupt came along. This isn't a problem unless the TSR itself decides to call some foreign code, such as DOS, a BIOS routine, or some other TSR. For example, the main application may be making a DOS call when a timer interrupt activates a TSR, interrupting the call to DOS while the CPU is still executing code inside DOS. If the TSR attempts to make a call to DOS at this point, then this will *reenter* DOS. Of course, DOS is not reentrant, so this creates all kinds of problems (usually, it hangs the system). When writing active TSRs that call other routines besides those provided directly in the TSR, you must be aware of possible reentrancy problems.

Note that passive TSRs never suffer from this problem. Indeed, any TSR routine you call passively will execute in the caller's environment. Unless some other hardware ISR or active TSR makes the call to your routine, you do not need to worry about reentrancy with passive routines. However, reentrancy is an issue for active TSR routines and passive routines that active TSRs call.

---

### 18.3.1 Reentrancy Problems with DOS

DOS is probably the biggest sore point to TSR developers. DOS is not reentrant yet DOS contains many services a TSR might use. Realizing this, Microsoft has added some support to DOS to allow TSRs to see if DOS is currently active. After all, reentrancy is only a problem if you call DOS while it is already active. If it isn't already active, you can certainly call it from a TSR with no ill effects.

MS-DOS provides a special one-byte flag (InDOS) that contains a zero if DOS is currently active and a non-zero value if DOS is already processing an application request. By testing the InDOS flag your TSR can determine if it can safely make a DOS call. If this flag is zero, you can always make the DOS call. If this flag contains one, you may not be able to make the DOS call. MS-DOS provides a function call, *Get InDOS Flag Address*, that returns the address of the InDOS flag. To use this function, load ah with 34h and call DOS. DOS will return the address of the InDOS flag in es:bx. If you save this address, your resident programs will be able to test the InDOS flag to see if DOS is active.

Actually, there are two flags you should test, the InDOS flag and the *critical error flag* (criterr). Both of these flags should contain zero before you call DOS from a TSR. In DOS version 3.1 and later, the critical error flag appears in the byte just before the InDOS flag.



So what should you do if these flags aren't both zero? It's easy enough to say "hey, come back and do this stuff later when MS-DOS returns back to the user program." But how do you do this? For example, if a keyboard interrupt activates your TSR and you pass control on to the real keyboard handler because DOS is busy, you can't expect your TSR to be magically restarted later on when DOS is no longer active.

The trick is to patch your TSR into the timer interrupt as well as the keyboard interrupt. When the key-stroke interrupt wakes your TSR and you discover that DOS is busy, the keyboard ISR can simply set a flag to tell itself to try again later; then it passes control to the original keyboard handler. In the meantime, a timer ISR you've written is constantly checking this flag you've created. If the flag is clear, it simply passes control on to the original timer interrupt handler, if the flag is set, then the code checks the InDOS and CritErr flags. If these guys say that DOS is busy, the timer ISR passes control on to the original timer handler. Shortly after DOS finishes whatever it was doing, a timer interrupt will come along and detect that DOS is no longer active. Now your ISR can take over and make any necessary calls to DOS that it wants. Of course, once your timer code determines that DOS is not busy, it should clear the "I want service" flag so that future timer interrupts don't inadvertently restart the TSR.

There is only one problem with this approach. There are certain DOS calls that can take an indefinite amount of time to execute. For example, if you call DOS to read a key from the keyboard (or call the Standard Library's `getc` routine that calls DOS to read a key), it could be *hours, days*, or even longer before somebody actually bothers to press a key. Inside DOS there is a loop that waits until the user actually presses a key. And until the user presses some key, the InDOS flag is going to remain non-zero. If you've written a timer-based TSR that is buffering data every few seconds and needs to write the results to disk every now and then, you will overflow your buffer with new data if you wait for the user, who just went to lunch, to press a key in DOS' `command.com` program.

Luckily, MS-DOS provides a solution to this problem as well – the idle interrupt. While MS-DOS is in an indefinite loop wait for an I/O device, it continually executes an `int 28h` instruction. By patching into the `int 28h` vector, your TSR can determine when DOS is sitting in such a loop. When DOS executes the `int 28h` instruction, it is safe to make any DOS call whose function number (the value in `ah`) is greater than `0Ch`.

So if DOS is busy when your TSR wants to make a DOS call, you must use either a timer interrupt or the idle interrupt (`int 28h`) to activate the portion of your TSR that must make DOS calls. One final thing to keep in mind is that *whenever you test or modify any of the above mentioned flags, you are in a critical section*. Make sure the interrupts are off. If not, your TSR make activate two copies of itself or you may wind up entering DOS at the same time some other TSR enters DOS.

An example of a TSR using these techniques will appear a little later, but there are some additional reentrancy problems we need to discuss first.

### 18.3.2 Reentrancy Problems with BIOS

DOS isn't the only non-reentrant code a TSR might want to call. The PC's BIOS routines also fall into this category. Unfortunately, BIOS doesn't provide an "InBIOS" flag or a multiplex interrupt. You will have to supply such functionality yourself.

The key to preventing reentering a BIOS routine you want to call is to use a *wrapper*. A wrapper is a short ISR that patches into an existing BIOS interrupt specifically to manipulate an InUse flag. For example, suppose you need to make an `int 10h` (video services) call from within your TSR. You could use the following code to provide an "Int10InUse" flag that your TSR could test:

```
MyInt10    proc    far
            inc    cs: Int10InUse
            pushf
            call   cs: OldInt10
            dec    cs: Int10InUse
            iret
MyInt10    endp
```

Assuming you've initialized the `Int10InUse` variable to zero, the in use flag will contain zero when it is safe to execute an `int 10h` instruction in your TSR. It will contain a non-zero value when the interrupt 10h handler is busy. You can use this flag like the `inDOS` flag to defer the execution of your TSR code.

Like DOS, there are certain BIOS routines that may take an indefinite amount of time to complete. Reading a key from the keyboard buffer, reading or writing characters on the serial port, or printing characters to the printer are some examples. While, in some cases, it is possible to create a wrapper that lets your TSR activate itself while a BIOS routine is executing one of these polling loops, there is probably no benefit to doing so. For example, if an application program is waiting for the printer to take a character before it sends another to printer, having your TSR preempt this and attempt to send a character to the printer won't accomplish much (other than scramble the data sent to the print). Therefore, BIOS wrappers generally don't worry about *indefinite postponement* in a BIOS routine.

5, 8, 9, D, E, 10, 13, 16, 17, 21, 28

If you run into problems with your TSR code and certain application programs, you may want to place wrappers around the following interrupts to see if this solves your problem: `int 5`, `int 8`, `int 9`, `int B`, `int C`, `int D`, `int E`, `int 10`, `int 13`, `int 14`, `int 16`, or `int 17`. These are common culprits when TSR problems develop.

### 18.3.3 Reentrancy Problems with Other Code

Reentrancy problems occur in other code you might call as well. For example, consider the UCR Standard Library. The UCR Standard Library is not reentrant. This usually isn't much of a problem for a couple of reasons. First, most TSRs do *not* call Standard Library subroutines. Instead, they provide results that normal applications can use; those applications use the Standard Library routines to manipulate such results. A second reason is that were you to include some Standard Library routines in a TSR, the application would have a *separate* copy of the library routines. The TSR might execute an `strcmp` instruction while the application is in the middle of an `strcmp` routine, *but these are not the same routines!* The TSR is not reentering the application's code, it is executing a separate routine.

However, many of the Standard Library functions make DOS or BIOS calls. Such calls do not check to see if DOS or BIOS is already active. Therefore, calling many Standard Library routines from within a TSR may cause you to reenter DOS or BIOS.

One situation does exist where a TSR could reenter a Standard Library routine. Suppose your TSR has both passive and active components. If the main application makes a call to a passive routine in your TSR and that routine call a Standard Library routine, there is the possibility that a system interrupt could interrupt the Standard Library routine and the active portion of the TSR reenter that same code. Although such a situation would be extremely rare, you should be aware of this possibility.

Of course, the best solution is to avoid using the Standard Library within your TSRs. If for no other reason, the Standard Library routines are quite large and TSRs should be as small as possible.

## 18.4 The Multiplex Interrupt (INT 2Fh)

When installing a passive TSR, or an active TSR with passive components, you will need to choose some interrupt vector to patch so other programs can communicate with your passive routines. You could pick an interrupt vector almost at random, say `int 84h`, but this could lead to some compatibility problems. What happens if someone else is already using that interrupt vector? Sometimes, the choice of interrupt vector is clear. For example, if your passive TSR is extended the `int 16h` keyboard services, it makes sense to patch in to the `int 16h` vector and add additional functions above and beyond those already provided by the BIOS. On the other hand, if you are creating a driver for some brand new device for the PC, you probably would not want to piggyback the support functions for this device on some other interrupt. Yet arbitrarily picking an unused interrupt vector is risky; how many other programs out there decided to do the

same thing? Fortunately, MS-DOS provides a solution: the multiplex interrupt. Int 2Fh provides a general mechanism for installing, testing the presence of, and communicating with a TSR.

To use the multiplex interrupt, an application places an identification value in ah and a function number in al and then executes an int 2Fh instruction. Each TSR in the int 2Fh chain compares the value in ah against its own unique identifier value. If the values match, the TSR process the command specified by the value in the al register. If the identification values do not match, the TSR passes control to the next int 2Fh handler in the chain.

Of course, this only reduces the problem somewhat, it doesn't eliminate it. Sure, we don't have to guess an interrupt vector number at random, but we still have to choose a random identification number. After all, it seems reasonable that we must choose this number before designing the TSR and any applications that call it, after all, how will the applications know what value to load into ah if we dynamically assign this value when the TSR goes resident?

Well, there is a little trick we can play to dynamically assign TSR identifiers *and* let any interested applications determine the TSR's ID. By convention, function zero is the "Are you there?" call. An application should always execute this function to determine if the TSR is actually present in memory before making any service requests. Normally, function zero returns a zero in al if the TSR is *not* present, it returns 0FFh if it is present. However, when this function returns 0FFh it only tells you that *some* TSR has responded to your query; it does not guarantee that the TSR you are interested in is actually present in memory. However, by extending the convention somewhat, it is very easy to verify the presence of the desired TSR. Suppose the function zero call also returns a pointer to a unique identification string in the es:di registers. Then the code testing for the presence of a specific TSR could test this string when the int 2Fh call detects the presence of a TSR. the following code segment demonstrates how a TSR could determine if a TSR identified as "Randy's INT 10h Extension" is present in memory; this code will also determine the unique identification code for that TSR, for future reference:

```

; Scan through all the possible TSR IDs. If one is installed, see if
; it's the TSR we're interested in.

IDLoop:      mov     cx, 0FFh           ;This will be the ID number.
             mov     ah, cl         ;ID -> AH.
             push    cx           ;Preserve CX across call
             mov     al, 0         ;Test presence function code.
             int     2Fh          ;Call multiplex interrupt.
             pop     cx           ;Restore CX.
             cmp     al, 0         ;Installed TSR?
             je      TryNext       ;Returns zero if none there.
             strcpl  ;See if it's the one we want.
             byte   "Randy's INT "
             byte   "10h Extension",0
             je      Success       ;Branch off if it is ours.
TryNext:     loop   IDLoop         ;Otherwise, try the next one.
             jmp    NotInstalled   ;Failure if we get to this point.

Success:     mov     FuncID, cl    ;Save function result.
             .
             .
             .

```

If this code succeeds, the variable FuncID contains the identification value for resident TSR. If it fails, the application program probably needs to abort, or otherwise ensure that it never calls the missing TSR.

The code above lets an application easily detect the presence of and determine the ID number for a specific TSR. The next question is "How do we pick the ID number for the TSR in the first place?" The next section will address that issue, as well as how the TSR must respond to the multiplex interrupt.

---

## 18.5 Installing a TSR

Although we've already discussed how to make a program go resident (see "DOS Memory Usage and TSRs" on page 1025), there are a few aspects to installing a TSR that we need to address. First, what hap-

pens if a user installs a TSR and then tries to install it a second time without first removing the one that is already resident? Second, how can we assign a TSR identification number that won't conflict with a TSR that is already installed? This section will address these issues.

The first problem to address is an attempt to reinstall a TSR program. Although one could imagine a type of TSR that allows multiple copies of itself in memory at one time, such TSRs are few and far in-between. In most cases, having multiple copies of a TSR in memory will, at best, waste memory and, at worst, crash the system. Therefore, unless you are specifically written a TSR that allows multiple copies of itself in memory at one time, you should check to see if the TSR is installed before actually installing it. This code is identical to the code an application would use to see if the TSR is installed, the only difference is that the TSR should print a nasty message and refuse to go TSR if it finds a copy of itself already installed in memory. The following code does this:

```

SearchLoop:  mov     cx, 0FFh
             mov     ah, cl
             push    cx
             mov     al, 0
             int     2Fh
             pop     cx
             cmp     al, 0
             je      TryNext
             strcmpl
             byte    "Randy's INT "
             byte    "10h Extension",0
             je      AlreadyThere
TryNext:     loop    SearchLoop
             jmp     NotInstalled

AlreadyThere: print
             byte    "A copy of this TSR already exists in memory",cr,lf
             byte    "Aborting installation process.",cr,lf,0
             .
             .
             .

```

In the previous section, you saw how to write some code that would allow an application to determine the TSR ID of a specific resident program. Now we need to look at how to dynamically choose an identification number for the TSR, one that does not conflict with any other TSRs. This is yet another modification to the scanning loop. In fact, we can modify the code above to do this for us. All we need to do is save away some ID value that does not have an installed TSR. We need only add a few lines to the above code to accomplish this:

```

             mov     FuncID, 0           ;Initialize FuncID to zero.
             mov     cx, 0FFh
SearchLoop:  mov     ah, cl
             push    cx
             mov     al, 0
             int     2Fh
             pop     cx
             cmp     al, 0
             je      TryNext
             strcmpl
             byte    "Randy's INT "
             byte    "10h Extension",0
             je      AlreadyThere
             loop    SearchLoop
             jmp     NotInstalled

; Note:  presumably DS points at the resident data segment that contains
;        the FuncID variable.  Otherwise you must modify the following to
;        point some segment register at the segment containing FuncID and
;        use the appropriate segment override on FuncID.

TryNext:     mov     FuncID, cl           ;Save possible function ID if this
             loop    SearchLoop         ; identifier is not in use.
             jmp     NotInstalled

AlreadyThere: print

```

```

byte      "A copy of this TSR already exists in memory",cr,lf
byte      "Aborting installation process.",cr,lf,0
ExitPgm

NotInstalled: cmp      FuncID, 0          ;If there are no available IDs, this
               jne      GoodID           ; will still contain zero.
               print
               byte      "There are too many TSRs already installed.",cr,lf
               byte      "Sorry, aborting installation process.",cr,lf,0
               ExitPgm

```

GoodID:

If this code gets to label "GoodID" then a previous copy of the TSR is not present in memory and the FuncID variable contains an unused function identifier.

Of course, when you install your TSR in this manner, you must not forget to patch your interrupt 2Fh handler into the int 2Fh chain. Also, you have to write an interrupt 2Fh handler to process int 2Fh calls. The following is a very simple multiplex interrupt handler for the code we've been developing:

```

FuncID      byte      0          ;Should be in resident segment.
OldInt2F    dword     ?          ; Ditto.

MyInt2F     proc      far
             cmp      ah, cs:FuncID ;Is this call for us?
             je       ItsUs
             jmp      cs:OldInt2F   ;Chain to previous guy, if not.

; Now decode the function value in AL:

ItsUs:      cmp      al, 0          ;Verify presence call?
             jne      TryOtherFunc
             mov      al, 0FFh      ;Return "present" value in AL.
             lesi     IDString      ;Return pointer to string in es:di.
             iret                    ;Return to caller.
IDString    byte      "Randy's INT "
             byte      "10h Extension",0

; Down here, handle other multiplex requests.
; This code doesn't offer any, but here's where they would go.
; Just test the value in AL to determine which function to execute.

TryOtherFunc:
             .
             .
             .
             iret
MyInt2F     endp

```

---

## 18.6 Removing a TSR

Removing a TSR is quite a bit more difficult than installing one. There are three things the removal code must do in order to properly remove a TSR from memory: first, it needs to stop any pending activities (e.g., the TSR may have some flags set to start some activity at a future time); second it needs to restore all interrupt vectors to their former values; third, it needs to return all reserved memory back to DOS so other applications can make use of it. The primary difficulty with these three activities is that it is not always possible to properly restore the interrupt vectors.

If your TSR removal code simply restores the old interrupt vector values, you may create a really big problem. What happens if the user runs some other TSRs after running yours and they patch into the same interrupt vectors as your TSR? This would produce interrupt chains that look something like the following:



If you restore the interrupt vector with your original value, you will create the following:



This effectively disables the TSRs that chain into your code. Worse yet, this only disables the interrupts that those TSRs have in common with your TSR. The other interrupts those TSRs patch into are still active. Who knows how those interrupts will behave under such circumstances?

One solution is to simply print an error message informing the user that they cannot remove this TSR until they remove all TSRs installed prior to this one. This is a common problem with TSRs and most DOS users who install and remove TSRs should be comfortable with the fact that they must remove TSRs in the reverse order that they install them.

It would be tempting to suggest a new convention that TSRs should obey; perhaps if the function number is 0FFh, a TSR should store the value in `es:bx` away in the interrupt vector specified in `c1`. This would allow a TSR that would like to remove itself to pass the address of its original interrupt handler to the previous TSR in the chain. There are only three problems with this approach: first, almost no TSRs in existence currently support this feature, so it would be of little value; second, some TSRs might use function 0FFh for something else, calling them with this value, *even if you knew their ID number*, could create a problem; finally, just because you've removed the TSR from the interrupt chain doesn't mean you can (truly) free up the memory the TSR uses. DOS' memory management scheme (the free pointer business) works like a stack. If there are other TSRs installed above yours in memory, most applications wouldn't be able to use the memory freed up by removing your TSR anyway.

Therefore, we'll also adopt the strategy of simply informing the user that they cannot remove a TSR if there are others installed in shared interrupt chains. Of course, that does bring up a good question, how can we determine if there are other TSRs chained in to our interrupts? Well, this isn't so hard. We know that the 80x86's interrupt vectors should still be pointing at our routines if we're the last TSR run. So all we've got to do is compare the patched interrupt vectors against the addresses of our interrupt service routines. If they *all* match, then we can safely remove our TSR from memory. If only one of them does not match, then we cannot remove the TSR from memory. The following code sequence tests to see if it is okay to detach a TSR containing ISRs for `int 2fH` and `int 9`:

```

; OkayToRmv- This routine returns the carry flag set if it is okay to
;            remove the current TSR from memory. It checks the interrupt
;            vectors for int 2F and int 9 to make sure they
;            are still pointing at our local routines.
;            This code assumes DS is pointing at the resident code's
;            data segment.

OkayToRmv  proc      near
            push     es
            mov     ax, 0             ;Point ES at interrupt vector
            mov     es, ax          ; table.
            mov     ax, word ptr OldInt2F
            cmp     ax, es:[2fh*4]
            jne     CantRemove
            mov     ax, word ptr OldInt2F+2
            cmp     ax, es:[2Fh*4 + 2]
            jne     CantRemove

            mov     ax, word ptr OldInt9
            cmp     ax, es:[9*4]
            jne     CantRemove
            mov     ax, word ptr OldInt9+2
            cmp     ax, es:[9*4 + 2]
            jne     CantRemove

; We can safely remove this TSR from memory.

            stc
            pop     es
            ret
  
```

` Someone else is in the way, we cannot remove this TSR.

```
CantRemove:  clc
              pop     es
              ret
OkayToRmv  endp
```

Before the TSR attempts to remove itself, it should call a routine like this one to see if removal is possible.

Of course, the fact that no other TSR has chained into the same interrupts does *not* guarantee that there are not TSRs above yours in memory. However, removing the TSR in that case will not crash the system. True, you may not be able to reclaim the memory the TSR is using (at least until you remove the other TSRs), but at least the removal will not create complications.

To remove the TSR from memory requires two DOS calls, one to free the memory in use by the TSR and one to free the memory in use by the environment area assigned to the TSR. To do this, you need to make the DOS deallocation call (see “MS-DOS, PC-BIOS, and File I/O” on page 699). This call requires that you pass the segment address of the block to release in the es register. For the TSR program itself, you need to pass the address of the TSR’s PSP. This is one of the reasons a TSR needs to save its PSP when it first installs itself. The other free call you must make frees the space associated with the TSR’s *environment block*. The address of this block is at offset 2Ch in the PSP. So we should probably free it first. The following calls handle the job of free the memory associated with a TSR:

```
; Presumably, the PSP variable was initialized with the address of this
; program’s PSP before the terminate and stay resident call.

      mov     es, PSP
      mov     es, es:[2Ch]      ;Get address of environment block.
      mov     ah, 49h          ;DOS deallocate block call.
      int     21h

      mov     es, PSP          ;Now free the program’s memory
      mov     ah, 49h          ; space.
      int     21h
```

Some poorly-written TSRs provide no facilities to allow you to remove them from memory. If someone wants remove such a TSR, they will have to reboot the PC. Obviously, this is a poor design. Any TSR you design for anything other than a quick test should be capable of removing itself from memory. The multiplex interrupt with function number one is often used for this purpose. To remove a TSR from memory, some application program passes the TSR ID and a function number of one to the TSR. If the TSR can remove itself from memory, it does so and returns a value denoting success. If the TSR cannot remove itself from memory, it returns some sort of error condition.

Generally, the removal program is the TSR itself with a special parameter that tells it to remove the TSR currently loaded into memory. A little later this chapter presents an example of a TSR that works precisely in this fashion (see “A Keyboard Monitor TSR” on page 1041).

## 18.7 Other DOS Related Issues

In addition to reentrancy problems with DOS, there are a few other issues your TSRs must deal with if they are going to make DOS calls. Although your calls might not cause DOS to reenter itself, it is quite possible for your TSR’s DOS calls to disturb data structures in use by an executing application. These data structures include the application’s stack, PSP, disk transfer area (DTA), and the DOS extended error information record.

When an active or passive TSR gains control of the CPU, it is operating in the environment of the main (foreground) application. For example, the TSR’s return address and any values it saves on the stack are pushed onto the application’s stack. If the TSR does not use much stack space, this is fine, it need not switch stacks. However, if the TSR consumes considerable amounts of stack space because of recursive

calls or the allocation of local variables, the TSR should save the application's `ss` and `sp` values and switch to a local stack. Before returning, of course, the TSR should switch back to the foreground application's stack.

Likewise, if the TSR execute's DOS' *get psp address* call, DOS returns the address of the foreground application's PSP, not the TSR's PSP<sup>4</sup>. The PSP contains several important address that DOS uses in the event of an error. For example, the PSP contains the address of the termination handler, `ctrl-break` handler, and critical error handler. If you do not switch the PSP from the foreground application to the TSR's and one of the exceptions occurs (e.g., someone hits control-break or a disk error occurs), the handler associated with the application may take over. Therefore, when making DOS calls that can result in one of these conditions, you need to switch PSPs. Likewise, when your TSR returns control to the foreground application, it must restore the PSP value. MS-DOS provides two functions that get and set the current PSP address. The DOS *Set PSP* call (`ah=51h`) sets the current program's PSP address to the value in the `bx` register. The DOS *Get PSP* call (`ah=50h`) returns the current program's PSP address in the `bx` register. Assuming the transient portion of your TSR has saved it's PSP address in the variable `PSP`, you switch between the TSR's PSP and the foreground application's PSP as follows:

```

; Assume we've just entered the TSR code, determined that it's okay to
; call DOS, and we've switch DS so that it points at our local variables.

        mov     ah, 51h           ;Get application's PSP address
        int     21h
        mov     AppPSP, bx       ;Save application's PSP locally.
        mov     bx, PSP          ;Change system PSP to TSR's PSP.
        mov     ah, 50h         ;Set PSP call
        int     21h
        .
        .                       ;TSR code
        .
        mov     bx, AppPSP       ;Restore system PSP address to
        mov     ah, 50h         ; point at application's PSP.
        int     21h

« clean up and return from TSR »

```

Another global data structure that DOS uses is the *disk transfer area*. This buffer area was used extensively for disk I/O in DOS version 1.0. Since then, the main use for the DTA has been the `find first file` and `find next file` functions (see "MS-DOS, PC-BIOS, and File I/O" on page 699). Obviously, if the application is in the middle of using data in the DTA and your TSR makes a DOS call that changes the data in the DTA, you will affect the operation of the foreground process. MS-DOS provides two calls that let you get and set the address of the DTA. The *Get DTA Address* call, with `ah=2Fh`, returns the address of the DTA in the `es:bx` registers. The *Set DTA* call (`ah=1Ah`) sets the DTA to the value found in the `ds:dx` register pair. With these two calls you can save and restore the DTA as we did for the PSP address above. The DTA is usually at offset 80h in the PSP, the following code preserve's the foreground application's DTA and sets the current DTA to the TSR's at offset `PSP:80`.

```

; This code makes the same assumptions as the previous example.

        mov     ah, 2Fh           ;Get application DTA
        int     21h
        mov     word ptr AppDTA, bx
        mov     word ptr AppDTA+2, es

        push    ds
        mov     ds, PSP          ;DTA is in PSP
        mov     dx, 80h         ; at offset 80h
        mov     ah, 1Ah         ;Set DTA call.
        int     21h
        pop     ds
        .
        .                       ;TSR code.
        .

```

---

4. This is another reason the transient portion of the TSR must save the PSP address in a resident variable for the TSR.



```

push    ds
mov     dx, word ptr AppDTA
mov     ds, word ptr AppDTA+2
mov     ax, 1ah           ;Set DTA call.
int     21h

```

The last issue a TSR must deal with is the extended error information in DOS. If a TSR interrupts a program immediately after DOS returns to that program, there may be some error information the foreground application needs to check in the DOS extended error information. If the TSR makes any DOS calls, DOS may replace this information with the status of the TSR DOS call. When control returns to the foreground application, it may read the extended error status and get the information generated by the TSR DOS call, not the application's DOS call. DOS provides two asymmetrical calls, *Get Extended Error* and *Set Extended Error* that read and write these values, respectively. The call to *Get Extended Error* returns the error status in the ax, bx, cx, dx, si, di, es, and ds registers. You need to save the registers in a data structure that takes the following form:

```

ExtError    struct
eeAX        word    ?
eeBX        word    ?
eeCX        word    ?
eeDX        word    ?
eeSI        word    ?
eeDI        word    ?
eeDS        word    ?
eeES        word    ?
            word    3 dup (0)           ;Reserved.
ExtError    ends

```

The *Set Extended Error* call requires that you pass an address to this structure in the ds:si register pair (which is why these two calls are asymmetrical). To preserve the extended error information, you would use code similar to the following:

```

; Save assumptions as the above routines here. Also, assume the error
; data structure is named ERR and is in the same segment as this code.

            push    ds                ;Save ptr to our DS.
            mov     ah, 59h           ;Get extended error call
            mov     bx, 0             ;Required by this call
            int     21h

            mov     cs:ERR.eeDS, ds
            pop     ds                ;Retrieve ptr to our data.
            mov     ERR.eeAX, ax
            mov     ERR.eeBX, bx
            mov     ERR.eeCX, cx
            mov     ERR.eeDX, dx
            mov     ERR.eeSI, si
            mov     ERR.eeDI, di
            mov     ERR.eeES, es
            .
            .
            .
            mov     si, offset ERR    ;DS already points at correct seg.
            mov     ax, 5D0Ah         ;5D0Ah is Set Extended Error code.
            int     21h

« clean up and quit »

```

---

## 18.8 A Keyboard Monitor TSR

The following program extends the keystroke counter program presented a little earlier in this chapter. This particular program monitors keystrokes and each minute writes out data to a file listing the date, time, and approximate number of keystrokes in the last minute.

This program can help you discover how much time you spend typing versus thinking at a display screen<sup>5</sup>.

```

; This is an example of an active TSR that counts keyboard interrupts
; once activated. Every minute it writes the number of keyboard
; interrupts that occurred in the previous minute to an output file.
; This continues until the user removes the program from memory.
;
;
; Usage:
;   KEYEVAL filename      -      Begins logging keystroke data to
;                               this file.
;
;   KEYEVAL REMOVE       -      Removes the resident program from
;                               memory.
;
;
; This TSR checks to make sure there isn't a copy already active in
; memory. When doing disk I/O from the interrupts, it checks to make
; sure DOS isn't busy and it preserves application globals (PSP, DTA,
; and extended error info). When removing itself from memory, it
; makes sure there are no other interrupts chained into any of its
; interrupts before doing the remove.
;
; The resident segment definitions must come before everything else.

ResidentSeg  segment    para public 'Resident'
ResidentSeg  ends

EndResident  segment    para public 'EndRes'
EndResident  ends

               .xlist
               .286
               include   stdlib.a
               includelib stdlib.lib
               .list

; Resident segment that holds the TSR code:

ResidentSeg  segment    para public 'Resident'
               assume    cs:ResidentSeg, ds:nothing

; Int 2Fh ID number for this TSR:

MyTSRID      byte       0

; The following variable counts the number of keyboard interrupts

KeyIntCnt    word       0

; Counter counts off the number of milliseconds that pass, SecCounter
; counts off the number of seconds (up to 60).

Counter      word       0
SecCounter   word       0

; FileHandle is the handle for the log file:

FileHandle   word       0

; NeedIO determines if we have a pending I/O operation.

NeedIO       word       0

; PSP is the psp address for this program.

PSP          word       0

```

---

5. This program is intended for your personal enjoyment only, it is not intended to be used for unethical purposes such as monitoring employees for evaluation purposes.

```

; Variables to tell us if DOS, INT 13h, or INT 16h are busy:

InInt13      byte      0
InInt16      byte      0
InDOSFlag    dword     ?

; These variables contain the original values in the interrupt vectors
; we've patched.

OldInt9      dword     ?
OldInt13     dword     ?
OldInt16     dword     ?
OldInt1C     dword     ?
OldInt28     dword     ?
OldInt2F     dword     ?

; DOS data structures:

ExtErr       struct
eeAX         word      ?
eeBX         word      ?
eeCX         word      ?
eeDX         word      ?
eeSI         word      ?
eeDI         word      ?
eeDS         word      ?
eeES         word      ?
             word      3 dup (0)
ExtErr       ends

XErr         ExtErr    {}           ;Extended Error Status.
AppPSP       word      ?           ;Application PSP value.
AppDTA       dword     ?           ;Application DTA address.

; The following data is the output record. After storing this data
; to these variables, the TSR writes this data to disk.

month        byte      0
day          byte      0
year         word      0
hour         byte      0
minute       byte      0
second       byte      0
Keystrokes   word      0
RecSize      =         $-month

; MyInt9-      The system calls this routine every time a keyboard
;              interrupt occurs. This routine increments the
;              KeyIntCnt variable and then passes control on to the
;              original Int9 handler.

MyInt9       proc      far
             inc       ResidentSeg:KeyIntCnt
             jmp       ResidentSeg:OldInt9
MyInt9       endp

; MyInt1C-     Timer interrupt. This guy counts off 60 seconds and then
;              attempts to write a record to the output file. Of course,
;              this call has to jump through all sorts of hoops to keep
;              from reentering DOS and other problematic code.

```

```

MyInt1C      proc      far
             assume   ds:ResidentSeg

             push     ds
             push     es
             pusha                    ;Save all the registers.
             mov      ax, ResidentSeg
             mov      ds, ax

             pushf
             call     OldInt1C

; First things first, let's bump our interrupt counter so we can count
; off a minute. Since we're getting interrupted about every 54.92549
; milliseconds, let's shoot for a little more accuracy than 18 times
; per second so the timings don't drift too much.

             add      Counter, 549      ;54.9 msec per int 1C.
             cmp      Counter, 10000   ;1 second.
             jb       NotSecYet
             sub      Counter, 10000
             inc      SecCounter

NotSecYet:

; If NEEDIO is not zero, then there is an I/O operation in progress.
; Do not disturb the output values if this is the case.

             cli                                ;This is a critical region.
             cmp      NeedIO, 0
             jne      SkipSetNIO

; Okay, no I/O in progress, see if a minute has passed since the last
; time we logged the keystrokes to the file. If so, it's time to start
; another I/O operation.

             cmp      SecCounter, 60      ;One minute passed yet?
             jb       IntlCDone
             mov      NeedIO, 1          ;Flag need for I/O.
             mov      ax, KeyIntCnt      ;Copy this to the output
             shr      ax, 1              ; buffer after computing
             mov      KeyStrokes, ax     ; # of keystrokes.
             mov      KeyIntCnt, 0       ;Reset for next minute.
             mov      SecCounter, 0

SkipSetNIO:  cmp      NeedIO, 1          ;Is the I/O already in
             jne      IntlCDone         ; progress? Or done?

             call     ChkDOSStatus      ;See if DOS/BIOS are free.
             jnc      IntlCDone         ;Branch if busy.

             call     DoIO              ;Do I/O if DOS is free.

IntlCDone:  popa                                ;Restore registers and quit.
             pop      es
             pop      ds
             iret

MyInt1C     endp
             assume   ds:nothing

; MyInt28- Idle interrupt. If DOS is in a busy-wait loop waiting for
; I/O to complete, it executes an int 28h instruction each
; time through the loop. We can ignore the InDOS and CritErr
; flags at that time, and do the I/O if the other interrupts
; are free.

MyInt28     proc      far
             assume   ds:ResidentSeg

             push     ds
             push     es
             pusha                    ;Save all the registers.

```

```

mov     ax, ResidentSeg
mov     ds, ax

pushf
call    OldInt28           ;Call the next INT 28h
                           ; ISR in the chain.

cmp     NeedIO, 1         ;Do we have a pending I/O?
jne     Int28Done

mov     al, InInt13       ;See if BIOS is busy.
or      al, InInt16
jne     Int28Done

call    DoIO              ;Go do I/O if BIOS is free.

Int28Done:  popa
             pop     es
             pop     ds
             iret

MyInt28    endp
             assume  ds:nothing

; MyInt16- This is just a wrapper for the INT 16h (keyboard trap)
; handler.

MyInt16    proc     far
             inc     ResidentSeg:InInt16

; Call original handler:

             pushf
             call    ResidentSeg:OldInt16

; For INT 16h we need to return the flags that come from the previous call.

             pushf
             dec     ResidentSeg:InInt16
             popf
             retf   2           ;Fake IRET to keep flags.

MyInt16    endp

; MyInt13- This is just a wrapper for the INT 13h (disk I/O trap)
; handler.

MyInt13    proc     far
             inc     ResidentSeg:InInt13
             pushf
             call    ResidentSeg:OldInt13
             pushf
             dec     ResidentSeg:InInt13
             popf
             retf   2           ;Fake iret to keep flags.

MyInt13    endp

; ChkDOSStatus- Returns with the carry clear if DOS or a BIOS routine
; is busy and we can't interrupt them.

ChkDOSStatus  proc     near
             assume  ds:ResidentSeg
             les     bx, InDOSFlag
             mov     al, es:[bx]       ;Get InDOS flag.
             or      al, es:[bx-1]    ;OR with CritErr flag.
             or      al, InInt16     ;OR with our wrapper
             or      al, InInt13     ; values.
             je      Okay2Call
             clc
             ret

Okay2Call:  clc
             ret

ChkDOSStatus  endp

```

```

                assume    ds:nothing

; PreserveDOS- Gets a copy's of DOS' current PSP, DTA, and extended
; error information and saves this stuff. Then it sets
; the PSP to our local PSP and the DTA to PSP:80h.

PreserveDOS    proc        near
                assume    ds:ResidentSeg

                mov     ah, 51h           ;Get app's PSP.
                int     21h
                mov     AppPSP, bx       ;Save for later

                mov     ah, 2Fh         ;Get app's DTA.
                int     21h
                mov     word ptr AppDTA, bx
                mov     word ptr AppDTA+2, es

                push    ds
                mov     ah, 59h         ;Get extended err info.
                xor     bx, bx
                int     21h

                mov     cs:XErr.eeDS, ds
                pop     ds
                mov     XErr.eeAX, ax
                mov     XErr.eeBX, bx
                mov     XErr.eeCX, cx
                mov     XErr.eeDX, dx
                mov     XErr.eeSI, si
                mov     XErr.eeDI, di
                mov     XErr.eeES, es

; Okay, point DOS's pointers at us:

                mov     bx, PSP
                mov     ah, 50h         ;Set PSP.
                int     21h

                push    ds
                mov     ds, PSP         ;Set the DTA to
                mov     dx, 80h         ; address PSP:80h
                mov     ah, 1Ah         ;Set DTA call.
                int     21h
                pop     ds

                ret
PreserveDOS    endp
                assume    ds:nothing

; RestoreDOS- Restores DOS' important global data values back to the
; application's values.

RestoreDOS     proc        near
                assume    ds:ResidentSeg

                mov     bx, AppPSP
                mov     ah, 50h         ;Set PSP
                int     21h

                push    ds
                lds     dx, AppDTA
                mov     ah, 1Ah         ;Set DTA
                int     21h
                pop     ds
                push    ds

                mov     si, offset XErr ;Saved extended error stuff.
                mov     ax, 5D0Ah      ;Restore XErr call.
                int     21h
                pop     ds

```

```

RestoreDOS    ret
              endp
              assume     ds:nothing

; DoIO-      This routine processes each of the I/O operations
;            required to write data to the file.

DoIO          proc      near
              assume     ds:ResidentSeg

              mov       NeedIO, 0FFh      ;A busy flag for us.

; The following Get Date DOS call may take a while, so turn the
; interrupts back on (we're clear of the critical section once we
; write 0FFh to NeedIO).

              sti
              call      PreserveDOS      ;Save DOS data.

              mov       ah, 2Ah          ;Get Date DOS call
              int       21h
              mov       month, dh
              mov       day, dl
              mov       year, cx

              mov       ah, 2Ch          ;Get Time DOS call
              int       21h
              mov       hour, ch
              mov       minute, cl
              mov       second, dh

              mov       ah, 40h          ;DOS Write call
              mov       bx, FileHandle    ;Write data to this file.
              mov       cx, RecSize      ;This many bytes.
              mov       dx, offset month ;Starting at this address.
              int       21h              ;Ignore return errors (!).
              mov       ah, 68h          ;DOS Commit call
              mov       bx, FileHandle    ;Write data to this file.
              int       21h              ;Ignore return errors (!).

              mov       NeedIO, 0        ;Ready to start over.
              call      RestoreDOS

PhasesDone:   ret
DoIO          endp
              assume     ds:nothing

; MyInt2F-   Provides int 2Fh (multiplex interrupt) support for this
;            TSR. The multiplex interrupt recognizes the following
;            subfunctions (passed in AL):
;
;            00- Verify presence.      Returns 0FFh in AL and a pointer
;                                       to an ID string in es:di if the
;                                       TSR ID (in AH) matches this
;                                       particular TSR.
;
;            01- Remove.               Removes the TSR from memory.
;                                       Returns 0 in AL if successful,
;                                       1 in AL if failure.

MyInt2F       proc      far
              assume     ds:nothing

              cmp       ah, MyTSRID      ;Match our TSR identifier?
              je        YepItsOurs
              jmp       OldInt2F

; Okay, we know this is our ID, now check for a verify vs. remove call.

YepItsOurs:   cmp       al, 0            ;Verify Call
              jne      TryRmv

```

```

        mov     al, 0ffh           ;Return success.
        lesi   IDString
        ired   ;Return back to caller.

IDString    byte    "Keypress Logger TSR",0

TryRmv:    cmp     al, 1           ;Remove call.
           jne     IllegalOp

           call    TstRmvable     ;See if we can remove this guy.
           je     CanRemove       ;Branch if we can.
           mov     ax, 1           ;Return failure for now.
           ired

; Okay, they want to remove this guy *and* we can remove it from memory.
; Take care of all that here.

           assume  ds:ResidentSeg

CanRemove:  push   ds
           push   es
           pusha
           cli     ;Turn off the interrupts while
           mov     ax, 0           ; we mess with the interrupt
           mov     es, ax         ; vectors.
           mov     ax, cs
           mov     ds, ax

           mov     ax, word ptr OldInt9
           mov     es:[9*4], ax
           mov     ax, word ptr OldInt9+2
           mov     es:[9*4 + 2], ax

           mov     ax, word ptr OldInt13
           mov     es:[13h*4], ax
           mov     ax, word ptr OldInt13+2
           mov     es:[13h*4 + 2], ax

           mov     ax, word ptr OldInt16
           mov     es:[16h*4], ax
           mov     ax, word ptr OldInt16+2
           mov     es:[16h*4 + 2], ax

           mov     ax, word ptr OldInt1C
           mov     es:[1Ch*4], ax
           mov     ax, word ptr OldInt1C+2
           mov     es:[1Ch*4 + 2], ax

           mov     ax, word ptr OldInt28
           mov     es:[28h*4], ax
           mov     ax, word ptr OldInt28+2
           mov     es:[28h*4 + 2], ax

           mov     ax, word ptr OldInt2F
           mov     es:[2Fh*4], ax
           mov     ax, word ptr OldInt2F+2
           mov     es:[2Fh*4 + 2], ax

; Okay, with that out of the way, let's close the file.
; Note: INT 2F shouldn't have to deal with DOS busy because it's
; a passive TSR call.

           mov     ah, 3Eh         ;Close file command
           mov     bx, FileHandle
           int     21h

; Okay, one last thing before we quit- Let's give the memory allocated
; to this TSR back to DOS.

           mov     ds, PSP
           mov     es, ds:[2Ch]    ;Ptr to environment block.
           mov     ah, 49h         ;DOS release memory call.
           int     21h

```



```

        mov     ax, ds             ;Release program code space.
        mov     es, ax
        mov     ah, 49h
        int     21h

        popa
        pop     es
        pop     ds
        mov     ax, 0             ;Return Success.
        iret

; They called us with an illegal subfunction value. Try to do as little
; damage as possible.

IllegalOp:  mov     ax, 0             ;Who knows what they were thinking?
            iret
MyInt2F    endp
            assume  ds:nothing

; TstRmvable- Checks to see if we can remove this TSR from memory.
; Returns the zero flag set if we can remove it, clear
; otherwise.

TstRmvable proc     near
            cli
            push  ds
            mov   ax, 0
            mov   ds, ax

            cmp   word ptr ds:[9*4], offset MyInt9
            jne   TRDone
            cmp   word ptr ds:[9*4 + 2], seg MyInt9
            jne   TRDone

            cmp   word ptr ds:[13h*4], offset MyInt13
            jne   TRDone
            cmp   word ptr ds:[13h*4 + 2], seg MyInt13
            jne   TRDone

            cmp   word ptr ds:[16h*4], offset MyInt16
            jne   TRDone
            cmp   word ptr ds:[16h*4 + 2], seg MyInt16
            jne   TRDone

            cmp   word ptr ds:[1Ch*4], offset MyInt1C
            jne   TRDone
            cmp   word ptr ds:[1Ch*4 + 2], seg MyInt1C
            jne   TRDone

            cmp   word ptr ds:[28h*4], offset MyInt28
            jne   TRDone
            cmp   word ptr ds:[28h*4 + 2], seg MyInt28
            jne   TRDone

            cmp   word ptr ds:[2Fh*4], offset MyInt2F
            jne   TRDone
            cmp   word ptr ds:[2Fh*4 + 2], seg MyInt2F
TRDone:    pop     ds
            sti
            ret
TstRmvable endp
ResidentSeg ends

cseg      segment para public 'code'
            assume cs:cseg, ds:ResidentSeg

```

```

; SeeIfPresent-      Checks to see if our TSR is already present in memory.
;                   Sets the zero flag if it is, clears the zero flag if
;                   it is not.

SeeIfPresent  proc      near
              push     es
              push     ds
              push     di
              mov     cx, 0ffh      ;Start with ID 0FFh.
IDLoop:      mov     ah, cl
              push     cx
              mov     al, 0         ;Verify presence call.
              int     2Fh
              pop      cx
              cmp     al, 0         ;Present in memory?
              je      TryNext
              strcml  byte    "Keypress Logger TSR",0
              je      Success

TryNext:     dec     cl             ;Test USER IDs of 80h..FFh
              js     IDLoop
              cmp     cx, 0         ;Clear zero flag.
Success:     pop     di
              pop     ds
              pop     es
              ret

SeeIfPresent endp

; FindID-           Determines the first (well, last actually) TSR ID available
;                   in the multiplex interrupt chain. Returns this value in
;                   the CL register.
;
;                   Returns the zero flag set if it locates an empty slot.
;                   Returns the zero flag clear if failure.

FindID       proc      near
              push     es
              push     ds
              push     di

IDLoop:      mov     cx, 0ffh      ;Start with ID 0FFh.
              mov     ah, cl
              push     cx
              mov     al, 0         ;Verify presence call.
              int     2Fh
              pop      cx
              cmp     al, 0         ;Present in memory?
              je      Success
              dec     cl             ;Test USER IDs of 80h..FFh
              js     IDLoop
              xor     cx, cx
              cmp     cx, 1         ;Clear zero flag
Success:     pop     di
              pop     ds
              pop     es
              ret

FindID       endp

Main         proc      meminit

              mov     ax, ResidentSeg
              mov     ds, ax

              mov     ah, 62h      ;Get this program's PSP
              int     21h          ; value.
              mov     PSP, bx

; Before we do anything else, we need to check the command line

```

```

; parameters. We must have either a valid filename or the
; command "remove". If remove appears on the command line, then remove
; the resident copy from memory using the multiplex (2Fh) interrupt.
; If remove is not on the command line, we'd better have a filename and
; there had better not be a copy already loaded into memory.

```

```

        argc
        cmp     cx, 1           ;Must have exactly 1 parm.
        je     GoodParmCnt
        print
        byte   "Usage:",cr,lf
        byte   " KeyEval filename",cr,lf
        byte   "or KeyEval REMOVE",cr,lf,0
        ExitPgm

```

```

; Check for the REMOVE command.

```

```

GoodParmCnt:  mov     ax, 1
              argv
              stricmp  "REMOVE",0
              byte   "REMOVE",0
              jne    TstPresent

              call   SeeIfPresent
              je     RemoveIt
              print
              byte   "TSR is not present in memory, cannot remove"
              byte   cr,lf,0
              ExitPgm

```

```

RemoveIt:    mov     MyTSRID, cl
              printf
              byte   "Removing TSR (ID #%d) from memory...",0
              dword  MyTSRID

              mov     ah, cl
              mov     al, 1       ;Remove cmd, ah contains ID
              int     2Fh
              cmp     al, 1       ;Succeed?
              je     RmvFailure
              print
              byte   "removed.",cr,lf,0
              ExitPgm

```

```

RmvFailure:  print
              byte   cr,lf
              byte   "Could not remove TSR from memory.",cr,lf
              byte   "Try removing other TSRs in the reverse order "
              byte   "you installed them.",cr,lf,0
              ExitPgm

```

```

; Okay, see if the TSR is already in memory. If so, abort the
; installation process.

```

```

TstPresent:  call   SeeIfPresent
              jne    GetTSRID
              print
              byte   "TSR is already present in memory.",cr,lf
              byte   "Aborting installation process",cr,lf,0
              ExitPgm

```

```

; Get an ID for our TSR and save it away.

```

```

GetTSRID:    call   FindID
              je     GetFileName
              print
              byte   "Too many resident TSRs, cannot install",cr,lf,0
              ExitPgm

```

```
; Things look cool so far, check the filename and open the file.
```

```
GetFileName:  mov     MyTSRID, c1
              printf
              byte   "Keypress logger TSR program",cr,lf
              byte   "TSR ID = %d",cr,lf
              byte   "Processing file:",0
              dword  MyTSRID

              puts
              putcr

              mov     ah, 3Ch           ;Create file command.
              mov     cx, 0           ;Normal file.
              push   ds
              push   es               ;Point ds:dx at name
              pop    ds
              mov     dx, di
              int     21h             ;Open the file
              jnc    GoodOpen
              print
              byte   "DOS error #",0
              puti
              print
              byte   " opening file.",cr,lf,0
              ExitPgm

GoodOpen:    pop     ds
              mov     FileHandle, ax  ;Save file handle.

InstallInts: print
              byte   "Installing interrupts...",0
```

```
; Patch into the INT 9, 13h, 16h, 1Ch, 28h, and 2Fh interrupt vectors.
; Note that the statements above have made ResidentSeg the current data
; segment, so we can store the old values directly into
; the OldIntxx variables.
```

```
cli                               ;Turn off interrupts!
mov     ax, 0
mov     es, ax
mov     ax, es:[9*4]
mov     word ptr OldInt9, ax
mov     ax, es:[9*4 + 2]
mov     word ptr OldInt9+2, ax
mov     es:[9*4], offset MyInt9
mov     es:[9*4+2], seg ResidentSeg

mov     ax, es:[13h*4]
mov     word ptr OldInt13, ax
mov     ax, es:[13h*4 + 2]
mov     word ptr OldInt13+2, ax
mov     es:[13h*4], offset MyInt13
mov     es:[13h*4+2], seg ResidentSeg

mov     ax, es:[16h*4]
mov     word ptr OldInt16, ax
mov     ax, es:[16h*4 + 2]
mov     word ptr OldInt16+2, ax
mov     es:[16h*4], offset MyInt16
mov     es:[16h*4+2], seg ResidentSeg

mov     ax, es:[1Ch*4]
mov     word ptr OldInt1C, ax
mov     ax, es:[1Ch*4 + 2]
mov     word ptr OldInt1C+2, ax
mov     es:[1Ch*4], offset MyInt1C
mov     es:[1Ch*4+2], seg ResidentSeg

mov     ax, es:[28h*4]
mov     word ptr OldInt28, ax
mov     ax, es:[28h*4 + 2]
```

```

mov     word ptr OldInt28+2, ax
mov     es:[28h*4], offset MyInt28
mov     es:[28h*4+2], seg ResidentSeg

mov     ax, es:[2Fh*4]
mov     word ptr OldInt2F, ax
mov     ax, es:[2Fh*4 + 2]
mov     word ptr OldInt2F+2, ax
mov     es:[2Fh*4], offset MyInt2F
mov     es:[2Fh*4+2], seg ResidentSeg
sti                                     ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

        print
        byte      "Installed.",cr,lf,0

        mov     dx, EndResident      ;Compute size of program.
        sub     dx, PSP
        mov     ax, 3100h           ;DOS TSR command.
        int     21h

Main
cseg    ends

sseg    segment    para stack 'stack'
stk     db         1024 dup ("stack ")
sseg    ends

zzzzzzseg segment    para public 'zzzzzz'
LastBytes db         16 dup (?)
zzzzzzseg ends
end     Main

```

The following is a short little application that reads the data file produced by the above program and produces a simple report of the date, time, and keystrokes:

```

; This program reads the file created by the KEYEVAL.EXE TSR program.
; It displays the log containing dates, times, and number of keystrokes.

```

```

        .xlist
        .286
        include  stdlib.a
        includelib stdlib.lib
        .list

dseg    segment    para public 'data'

FileHandle word      ?

month   byte      0
day     byte      0
year    word      0
hour    byte      0
minute  byte      0
second  byte      0
KeyStrokes word    0
RecSize =        $-month

dseg    ends

cseg    segment    para public 'code'
        assume    cs:cseg, ds:dseg

```

```

; SeeIfPresent-      Checks to see if our TSR is present in memory.
;                   Sets the zero flag if it is, clears the zero flag if
;                   it is not.

SeeIfPresent  proc      near
               push     es
               push     ds
               pusha
IDLoop:       mov      cx, 0ffh          ;Start with ID 0FFh.
               mov      ah, cl
               push     cx
               mov      al, 0          ;Verify presence call.
               int     2Fh
               pop      cx
               cmp      al, 0          ;Present in memory?
               je       TryNext
               strcml  "Keypress Logger TSR",0
               byte    Success
               je
TryNext:      dec      cl              ;Test USER IDs of 80h..FFh
               js      IDLoop
               cmp      cx, 0          ;Clear zero flag.
Success:      popa
               pop      ds
               pop      es
               ret
SeeIfPresent  endp

Main          proc
               meminit

               mov      ax, dseg
               mov      ds, ax

               argc
               cmp      cx, 1          ;Must have exactly 1 parm.
               je       GoodParmCnt
               print
               byte    "Usage:",cr,lf
               byte    " KEYRPT filename",cr,lf,0
               ExitPgm

GoodParmCnt:  mov      ax, 1
               argv

               print
               byte    "Keypress logger report program",cr,lf
               byte    "Processing file:",0
               puts
               putcr

               mov      ah, 3Dh        ;Open file command.
               mov      al, 0          ;Open for reading.
               push     ds
               push     es            ;Point ds:dx at name
               pop      ds
               mov      dx, di
               int     21h            ;Open the file
               jnc     GoodOpen
               print
               byte    "DOS error #",0
               puti
               print
               byte    " opening file.",cr,lf,0
               ExitPgm

```

```

GoodOpen:    pop        ds
             mov        FileHandle, ax    ;Save file handle.

; Okay, read the data and display it:

ReadLoop:   mov        ah, 3Fh            ;Read file command
             mov        bx, FileHandle
             mov        cx, RecSize      ;Number of bytes.
             mov        dx, offset month ;Place to put data.
             int        21h
             jc        ReadError
             test       ax, ax           ;EOF?
             je        Quit

             mov        cx, year
             mov        dl, day
             mov        dh, month
             dtoam
             puts
             free
             print
             byte      ", ", 0

             mov        ch, hour
             mov        cl, minute
             mov        dh, second
             mov        dl, 0
             ttoam
             puts
             free
             printf
             byte      ", keystrokes = %d\n", 0
             dword    KeyStrokes
             jmp       ReadLoop

ReadError:  print
             byte      "Error reading file", cr, lf, 0

Quit:       mov        bx, FileHandle
             mov        ah, 3Eh        ;Close file
             int        21h
             ExitPgm

Main        endp
cseg        ends

sseg        segment    para stack 'stack'
stk         db         1024 dup ("stack ")
sseg        ends

zzzzzzsseg segment    para public 'zzzzzz'
LastBytes  db         16 dup (?)
zzzzzzsseg ends
end         Main

```

---

## 18.9 Semiresident Programs

A *semiresident* program is one that temporarily loads itself into memory, executes another program (a child process), and then removes itself from memory after the child process terminates. Semiresident programs behave like resident programs while the child executes, but they do not stay in memory once the child terminates.

The main use for semiresident programs is to extend an existing application or *patch* an application<sup>6</sup> (the child process). The nice thing about a semiresident program patch is that it does not have to modify

---

6. *Patching* a program means to replace certain opcode bytes in the object file. Programmers apply patches to correct bugs or extend a product whose sources are not available.

the application's ".EXE" file directly on the disk. If for some reason the patch fails, you haven't destroyed the ".EXE" file, you've only wiped out the object code in memory.

A semiresident application, like a TSR, has a transient and a resident part. The resident part remains in memory while the child process executes. The transient part initializes the program and then transfers control to the resident part that loads the child application over the resident portion. The transient code patches the interrupt vectors and does all the things a TSR does *except it doesn't issue the TSR command*. Instead, the resident program loads the application into memory and transfers control to that program. When the application returns control to the resident program, it exits to DOS using the standard ExitPgm call (ah=4Ch).

While the application is running, the resident code behaves like any other TSR. Unless the child process is aware of the semiresident program, or the semiresident program patches interrupt vectors the application normally uses, the semiresident program will probably be an active resident program, patching into one or more of the hardware interrupts. Of course, all the rules that apply to active TSRs also apply to active semiresident programs.

The following is a very generic example of a semiresident program. This program, "RUN.ASM", runs the application whose name and command line parameters appear as command line parameters to run. In other words:

```
c:> run pgm.exe parm1 parm2 etc.
```

is equivalent to

```
pgm parm1 parm2 etc.
```

Note that you must supply the ".EXE" or ".COM" extension to the program's filename. This code begins by extracting the program's filename and command line parameters from run's command line. Run builds an exec structure (see "MS-DOS, PC-BIOS, and File I/O" on page 699) and then calls DOS to execute the program. On return, run fixes up the stack and returns to DOS.

```
; RUN.ASM - The barebones semiresident program.
;
;      Usage:
;      RUN <program.exe> <program's command line>
;      or  RUN <program.com> <program's command line>
;
; RUN executes the specified program with the supplied command line parameters.
; At first, this may seem like a stupid program. After all, why not just run
; the program directly from DOS and skip the RUN altogether? Actually, there
; is a good reason for RUN-- It lets you (by modifying the RUN source file)
; set up some environment prior to running the program and clean up that
; environment after the program terminates ("environment" in this sense does
; not necessarily refer to the MS-DOS ENVIRONMENT area).
;
; For example, I have used this program to switch the mode of a TSR prior to
; executing an EXE file and then I restored the operating mode of that TSR
; after the program terminated.
;
; In general, you should create a new version of RUN.EXE (and, presumably,
; give it a unique name) for each application you want to use this program
; with.
;
;-----
;
;
; Put these segment definitions 1st because we want the Standard Library
; routines to load last in memory, so they wind up in the transient portion.

CSEG      segment      para public 'CODE'
CSEG      ends
SSEG      segment      para stack 'stack'
SSEG      ends
ZZZZZZSEG segment      para public 'zzzzzzseg'
ZZZZZZSEG ends
```



```

; Includes for UCR Standard Library macros.

        include  consts.a
        include  stdin.a
        include  stdout.a
        include  misc.a
        include  memory.a
        include  strings.a

        includelib  stdlib.lib

CSEG          segment    para public 'CODE'
              assume    cs:cseg, ds:cseg

; Variables used by this program.

; MS-DOS EXEC structure.

ExecStruct    dw          0                ;Use parent's Environment blk.
              dd          CmdLine          ;For the cmd ln parms.
              dd          DfltFCB
              dd          DfltFCB

DfltFCB      db          3," ",0,0,0,0
CmdLine      db          0, 0dh, 126 dup (" ") ;Cmd line for program.
PgmName      dd          ?                ;Points at pgm name.

Main          proc
              mov         ax, cseg        ;Get ptr to vars segment
              mov         ds, ax

              MemInit          ;Start the memory mgr.

; If you want to do something before the execution of the command-line
; specified program, here is a good place to do it:

; -----

; Now let's fetch the program name, etc., from the command line and execute
; it.

              argc          ;See how many cmd ln parms
              or           cx, cx        ; we have.
              jz           Quit         ;Just quit if no parameters.

              mov         ax, 1         ;Get the first parm (pgm name)
              mov         argv
              mov         word ptr PgmName, di ;Save ptr to name
              mov         word ptr PgmName+2, es

; Okay, for each word on the command line after the filename, copy
; that word to CmdLine buffer and separate each word with a space,
; just like COMMAND.COM does with command line parameters it processes.

ParmLoop:    lea         si, CmdLine+1 ;Index into cmdline.
              dec         cx
              jz           ExecutePgm

              inc         ax            ;Point at next parm.
              mov         argv         ;Get the next parm.

```

```

                push    ax
                mov     byte ptr [si], ' ' ;1st item and separator on ln.
                inc     CmdLine
                inc     si
CpyLp:         mov     al, es:[di]
                cmp     al, 0
                je      StrDone
                inc     CmdLine           ;Increment byte cnt
                mov     ds:[si], al
                inc     si
                inc     di
                jmp     CpyLp

StrDone:       mov     byte ptr ds:[si], cr ;In case this is the end.
                pop     ax                ;Get current parm #
                jmp     ParmLoop

```

```

; Okay, we've built the MS-DOS execute structure and the necessary
; command line, now let's see about running the program.
; The first step is to free up all the memory that this program
; isn't using. That would be everything from zzzzzzseg on.

```

```

ExecutePgm:   mov     ah, 62h           ;Get our PSP value
                int     21h
                mov     es, bx
                mov     ax, zzzzzzseg   ;Compute size of
                sub     ax, bx           ; resident run code.
                mov     bx, ax
                mov     ah, 4ah         ;Release unused memory.
                int     21h

```

```

; Warning! No Standard Library calls after this point. We've just
; released the memory that they're sitting in. So the program load
; we're about to do will wipe out the Standard Library code.

```

```

                mov     bx, seg ExecStruct
                mov     es, bx
                mov     bx, offset ExecStruct ;Ptr to program record.
                lds     dx, PgmName
                mov     ax, 4b00h         ;Exec pgm
                int     21h

```

```

; When we get back, we can't count on *anything* being correct. First, fix
; the stack pointer and then we can finish up anything else that needs to
; be done.

```

```

                mov     ax, sseg
                mov     ss, ax
                mov     sp, offset EndStk
                mov     ax, seg cseg
                mov     ds, ax

```

```

; Okay, if you have any great deeds to do after the program, this is a
; good place to put such stuff.

```

```

; -----

```

```

; Return control to MS-DOS

```

```

Quit:         ExitPgm
Main         endp
cseg        ends

```

```

sseg        segment    para stack 'stack'
                dw      128 dup (0)
endstk      dw          ?
sseg        ends

```

```

; Set aside some room for the heap.

```

```

zzzzzzseg   segment    para public 'zzzzzzseg'
Heap        db          200h dup (?)

```

```

zzzzzzseg      ends
                end        Main

```

Since RUN.ASM is rather simple perhaps a more complex example is in order. The following is a fully functional patch for the Lucasart's game XWING™. The motivation for this patch can be about because of the annoyance of having to look up a password everytime you play the game. This little patch searches for the code that calls the password routine and stores NOPs over that code in memory.

The operation of this code is a little different than that of RUN.ASM. The RUN program sends an execute command to DOS that runs the desired program. All system changes RUN needs to make must be made before or after the application executes. XWPATCH operates a little differently. It loads the XWING.EXE program into memory and searches for some specific code (the call to the password routine). Once it finds this code, it stores NOP instructions over the top of the call.

Unfortunately, life isn't quite that simple. When XWING.EXE loads, the password code isn't yet present in memory. XWING loads that code as an overlay later on. So the XWPATCH program finds something that XWING.EXE does load into memory right away – the joystick code. XWPATCH patches the joystick code so that any call to the joystick routine (when detecting or calibrating the joystick) produces a call to XWPATCH's code that searches for the password code. Once XWPATCH locates and NOPs out the call to the password routine, it restores the code in the joystick routine. From that point forward, XWPATCH is simply taking up memory space; XWING will never call it again until XWING terminates.

```

; XWPATCH.ASM
;
;      Usage:
;          XWPATCH      - must be in same directory as XWING.EXE
;
; This program executes the XWING.EXE program and patches it to avoid
; having to enter the password every time you run it.
;
; This program is intended for educational purposes only.
; It is a demonstration of how to write a semiresident program.
; It is not intended as a device to allow the piracy of commercial software.
; Such use is illegal and is punishable by law.
;
; This software is offered without warranty or any expectation of
; correctness. Due to the dynamic nature of software design, programs
; that patch other programs may not work with slight changes in the
; patched program (XWING.EXE). USE THIS CODE AT YOUR OWN RISK.
;
;-----

byp          textequ    <byte ptr>
wp           textequ    <word ptr>

; Put these segment definitions here so the UCR Standard Library will
; load after zzzzzzseg (in the transient section).

cseg         segment para public 'CODE'
cseg         ends

sseg         segment    para stack 'STACK'
sseg         ends

zzzzzzseg    segment    para public 'zzzzzzseg'
zzzzzzseg    ends

                .286
                include      stdlib.a
                includelib  stdlib.lib

CSEG         segment    para public 'CODE'

```

```

                assume     cs:cseg, ds:nothing

; CountJSCalls-Number of times xwing calls the Joystick code before
; we patch out the password call.

CountJSCalls   dw         250

; PSP-   Program Segment Prefix. Needed to free up memory before running
;        the real application program.

PSP            dw         0

; Program Loading data structures (for DOS).

ExecStruct     dw         0                ;Use parent's Environment blk.
               dd         CmdLine         ;For the cmd ln parms.
               dd         DfltFCB
               dd         DfltFCB
LoadSSSP       dd         ?
LoadCSIP       dd         ?
PgmName        dd         Pgm

DfltFCB        db         3," ",0,0,0,0,0
CmdLine        db         2, " ", 0dh, 16 dup (" ");Cmd line for program
Pgm            db         "XWING.EXE",0

;*****
; XWPATCH begins here. This is the memory resident part. Only put code
; which has to be present at run-time or needs to be resident after
; freeing up memory.
;*****

Main           proc
               mov         cs:PSP, ds
               mov         ax, cseg        ;Get ptr to vars segment
               mov         ds, ax

               mov         ax, zzzzzzseg
               mov         es, ax
               mov         cx, 1024/16
               meminit2

; Now, free up memory from ZZZZZZSEG on to make room for XWING.
; Note: Absolutely no calls to UCR Standard Library routines from
; this point forward! (ExitPgm is okay, it's just a macro which calls DOS.)
; Note that after the execution of this code, none of the code & data
; from zzzzzzseg on is valid.

               mov         bx, zzzzzzseg
               sub         bx, PSP
               inc         bx
               mov         es, PSP
               mov         ah, 4ah
               int         21h
               jnc         GoodRealloc

; Okay, I lied. Here's a StdLib call, but it's okay because we failed
; to load the application over the top of the standard library code.
; But from this point on, absolutely no more calls!

               print
               byte        "Memory allocation error."
               byte        cr,lf,0
               jmp         Quit

GoodRealloc:

; Now load the XWING program into memory:

```

```

        mov     bx, seg ExecStruct
        mov     es, bx
        mov     bx, offset ExecStruct ;Ptr to program record.
        lds    dx, PgmName
        mov     ax, 4b01h           ;Load, do not exec, pgm
        int     21h
        jc     Quit                 ;If error loading file.

; Unfortunately, the password code gets loaded dynamically later on.
; So it's not anywhere in memory where we can search for it. But we
; do know that the joystick code is in memory, so we'll search for
; that code. Once we find it, we'll patch it so it calls our SearchPW
; routine. Note that you must use a joystick (and have one installed)
; for this patch to work properly.

        mov     si, zzzzzzseg
        mov     ds, si
        xor     si, si

        mov     di, cs
        mov     es, di
        mov     di, offset JoyStickCode
        mov     cx, JoyLength
        call    FindCode
        jc     Quit                 ;If didn't find joystick code.

; Patch the XWING joystick code here

        mov     byp ds:[si], 09ah;Far call
        mov     wp ds:[si+1], offset SearchPW
        mov     wp ds:[si+3], cs

; Okay, start the XWING.EXE program running

        mov     ah, 62h           ;Get PSP
        int     21h
        mov     ds, bx
        mov     es, bx
        mov     wp ds:[10], offset Quit
        mov     wp ds:[12], cs
        mov     ss, wp cseg:LoadSSSP+2
        mov     sp, wp cseg:LoadSSSP
        jmp     dword ptr cseg:LoadCSIP

Quit:    ExitPgm
Main:    endp

; SearchPW gets call from XWING when it attempts to calibrate the joystick.
; We'll let XWING call the joystick several hundred times before we
; actually search for the password code. The reason we do this is because
; XWING calls the joystick code early on to test for the presence of a
; joystick. Once we get into the calibration code, however, it calls
; the joystick code repetitively, so a few hundred calls doesn't take
; very long to expire. Once we're in the calibration code, the password
; code has been loaded into memory, so we can search for it then.

SearchPW  proc     far
        cmp     cs:CountJSCalls, 0
        je     DoSearch
        dec     cs:CountJSCalls
        sti     ;Code we stole from xwing for
        neg     bx           ; the patch.
        neg     di
        ret

; Okay, search for the password code.

DoSearch:  push    bp
        mov     bp, sp
        push   ds

```

```

        push    es
        pusha

; Search for the password code in memory:

        mov     si, zzzzzzseg
        mov     ds, si
        xor     si, si

        mov     di, cs
        mov     es, di
        mov     di, offset PasswordCode
        mov     cx, PWLength
        call    FindCode
        jc     NotThere          ;If didn't find pw code.

; Patch the XWING password code here. Just store NOPs over the five
; bytes of the far call to the password routine.

        mov     byp ds:[si+11], 090h          ;NOP out a far call
        mov     byp ds:[si+12], 090h
        mov     byp ds:[si+13], 090h
        mov     byp ds:[si+14], 090h
        mov     byp ds:[si+15], 090h

; Adjust the return address and restore the patched joystick code so
; that it doesn't bother jumping to us anymore.

NotThere:  sub     word ptr [bp+2], 5 ;Back up return address.
          les     bx, [bp+2]      ;Fetch return address.

; Store the original joystick code over the call we patched to this
; routine.

          mov     ax, word ptr JoyStickCode
          mov     es:[bx], ax
          mov     ax, word ptr JoyStickCode+2
          mov     es:[bx+2], ax
          mov     al, byte ptr JoyStickCode+4
          mov     es:[bx+4], al

          popa
          pop     es
          pop     ds
          pop     bp
          ret

SearchPW   endp

;*****
;
; FindCode: On entry, ES:DI points at some code in *this* program which
;           appears in the XWING game. DS:SI points at a block of memory
;           in the XWING game. FindCode searches through memory to find the
;           suspect piece of code and returns DS:SI pointing at the start of
;           that code. This code assumes that it *will* find the code!
;           It returns the carry clear if it finds it, set if it doesn't.

FindCode   proc     near
          push    ax
          push    bx
          push    dx

DoCmp:     mov     dx, 1000h          ;Search in 4K blocks.
CmpLoop:   push    di                ;Save ptr to compare code.
          push    si                ;Save ptr to start of string.
          push    cx                ;Save count.
          repe   cmpsb
          pop     cx
          pop     si
          pop     di
          je     FoundCode
          inc     si
          dec     dx

```

```

        jne      CmpLoop
        sub     si, 1000h
        mov     ax, ds
        inc     ah
        mov     ds, ax
        cmp     ax, 9000h      ;Stop at address 9000:0
        jb     DoCmp          ; and fail if not found.

        pop     dx
        pop     bx
        pop     ax
        stc
        ret

FoundCode:  pop     dx
            pop     bx
            pop     ax
            clc
            ret

FindCode   endp

```

```

;*****
;

```

```

; Call to password code that appears in the XWING game. This is actually
; data that we're going to search for in the XWING object code.

```

```

PasswordCode  proc      near
              call     $+47h
              mov     [bp-4], ax
              mov     [bp-2], dx
              push    dx
              push    ax
              byte    9ah, 04h, 00
PasswordCode  endp
EndPW:

PWLenth      =          EndPW-PasswordCode

```

```

; The following is the joystick code we're going to search for.

```

```

JoyStickCode  proc      near
              sti
              neg     bx
              neg     di
              pop     bp
              pop     dx
              pop     cx
              ret
              mov     bp, bx
              in     al, dx
              mov     bl, al
              not     al
              and     al, ah
              jnz    $+11h
              in     al, dx
JoyStickCode  endp
EndJSC:

JoyLenth      =          EndJSC-JoyStickCode
cseg          ends

sseg          segment   para stack 'STACK'
              dw       256 dup (0)
endstk        dw       ?
sseg          ends

zzzzzzseg     segment   para public 'zzzzzzseg'
Heap          db       1024 dup (0)
zzzzzzseg     ends
end           Main

```

---

## 18.10 Summary

Resident programs provide a small amount of multitasking to DOS' single tasking world. DOS provides support for resident programs through a rudimentary memory management system. When an application issues the terminate and stay resident call, DOS adjusts its memory pointers so the memory space reserved by the TSR code is protected from future program loading operations. For more information on how this process works, see

- “DOS Memory Usage and TSRs” on page 1025

TSRs come in two basic forms: active and passive. Passive TSRs are not self-activating. A foreground application must call a routine in a passive TSR to activate it. Generally, an application interfaces to a passive TSR using the 80x86 trap mechanism (software interrupts). Active TSRs, on the other hand, do not rely on the foreground application for activation. Instead, they attach themselves to a hardware interrupt that activates them independently of the foreground process. For more information, see

- “Active vs. Passive TSRs” on page 1029

The nature of an active TSR introduces many compatibility problems. The primary problem is that an active TSR might want to call a DOS or BIOS routine after having just interrupted either of these systems. This creates problems because DOS and BIOS are not *reentrant*. Fortunately, MS-DOS provides some hooks that give active TSRs the ability to schedule DOS calls with DOS is inactive. Although the BIOS routines do not provide this same facility, it is easy to add a *wrapper* around a BIOS call to let you schedule calls appropriately. One additional problem with DOS is that an active TSR might disturb some global variable in use by the foreground process. Fortunately, DOS lets the TSR save and restore these values, preventing some nasty compatibility problems. For details, see

- “Reentrancy” on page 1032
- “Reentrancy Problems with DOS” on page 1032
- “Reentrancy Problems with BIOS” on page 1033
- “Reentrancy Problems with Other Code” on page 1034
- “Other DOS Related Issues” on page 1039

MS-DOS provides a special interrupt to coordinate communication between TSRs and other applications. The *multiplex* interrupt lets you easily check for the presence of a TSR in memory, remove a TSR from memory, or pass various information between the TSR and an active application. For more information, see

- “The Multiplex Interrupt (INT 2Fh)” on page 1034

Well written TSRs follow stringent rules. In particular, a good TSR follows certain conventions during installation and always provide the user with a safe removal mechanism that frees all memory in use by the TSR. In those rare cases where a TSR cannot remove itself, it always reports an appropriate error and instructs the user how to solve the problem. For more information on load and removing TSRs, see

- “Installing a TSR” on page 1035
- “Removing a TSR” on page 1037
- “A Keyboard Monitor TSR” on page 1041

A semiresident routine is one that is resident during the execution of some specific program. It automatically unloads itself when that application terminates. Semiresident applications find application as program patchers and “time-release TSRs.” For more information on semiresident programs, see

- “Semiresident Programs” on page 1055