Edited by Robert Ward

MS-dos®

SYSTEM PROGRAMMING

MS-DOS® System Programming

Edited By Robert Ward

R&D Publications, Inc.
2601 Iowa St.
Lawrence, KS 66046

Edited by:

Robert L. Ward

Published by:

R&D Publications, Inc.
2601 Iowa St.
Lawrence, KS 66046
March 1990

Printed by:

Clark Printing
North Kansas City, MO

Trademarks:

MS-DOS, MS-Windows, Microsoft C, Microsoft Macro Assembler, MASM, QuickC, QuickBASIC, OS/2, Microsoft Corporation.

Turbo Assembler, Turbo Pascal, Turbo Debugger, Borland International.

Above Board, Intel.

Rampage, AST.

Lotus 1-2-3, Lotus Development Corp.

UNIX, AT&T.

Barcom, Wenham.

Builder, Hyperkinetix.

Wordstar, MicroPro International Corp.

Macintosh, Apple Corp.

MS-DOS System Programming

Preface

This book is intended as a "how-to" guide for sophisticated PC developers. You won't find any tutorials on how to build a linked list or a binary tree. You will find complete standalone discussions of how to write TSRs, interrupt handlers, and exception handlers. You'll find detailed explanations of undocumented system calls and very thorough explanations of certain hardware interfaces. This is a book for programmers who have long since mastered the craft of programming and who are now looking for technical information to support advanced applications in a PC environment.

Each chapter gives complete details about some advanced technique or subject. Each chapter can be read independently. A keyword index will help you quickly reference details when you need them later.

Of course, this book isn't exhaustive — the subject is far too large. Rather than attempt to exhaust the subject, we've tried to create a book that will lead you to the information you need. Some chapters are excerpts from books with wider coverage. If these chapters are useful to your project, you may want to consider reading the book from which they are drawn. The final chapter is an annotated bibliography of books suitable for advanced PC developers.

We sincerely hope you find this book useful and welcome your comments and suggestions.

Locating The Master Environment In MS-DOS

Scott Robert Ladd

Developing software for MS-DOS can be frustrating. I often find what seem to be artificial restrictions on what can and can't be done. For example, some of the utilities provided with MS-DOS can accomplish tasks which a programmer can't duplicate by using the documented features of the operating system. This article attempts to lift the veil from one of MS-DOS's most useful hidden secrets by providing a function which locates the master copy of the MS-DOS environment.

The environment is a collection of text variables maintained by *COMMAND.COM*. These variables consist of a name and an associated text string. Environment variables are used for a wide variety of purposes by both the operating system and application programs. Common examples of environment variables include *COMSPEC* (which stores the path name of the MS-DOS shell), *PROMPT* (the prompt definition string), and *PATH* (a list of directories to be searched for executable files). Some environment variables are maintained by special commands; other environment variables are stored using the internal MS-DOS command *SET*. Programmers are well aware of environment variables; most assemblers and compilers use them for locating header files, libraries, and compiler components.

Every program in MS-DOS has its own environment. A program which executes another program is known as a "parent", while the program it invokes is called a "child". A child, in turn, can also be the parent of other programs. A child process inherits a copy of the environment associated with

its parent. Changes made by the child to its copy of the environment have no affect on the parent's copy — and vice versa.

The MS-DOS command shell, *COMMAND.COM*, is the ultimate parent of all resident programs, since it is the first program loaded. At boot time, *COMMAND.COM* allocates a block of memory into which it stores the master environment variables. Since most programs are executed from the *COMMAND.COM* prompt, it is the direct parent of most programs. However, many programs are capable of running other programs directly, and additional copies of *COMMAND.COM* can be resident simultaneously as well. This can muddy the waters when one is searching for the master copy of the environment, which is associated with the *COMMAND.COM* loaded at boot-up.

When a program is loaded MS-DOS creates a 256-byte header for it. This header contains important operating system data, and is called the Program Segment Prefix (PSP). A program can locate the copy of its local environment via a segment pointer stored at offset *2Ch* within the PSP.

Placing new information into a local copy of the environment is not particularly useful. When a local environment is created, it's size is only slightly larger than that required to hold all of the existing variables in the parent environment. A copy of the environment cannot be expanded, so there's almost no room to add new variables. Any changes to a local environment are transient; when a program terminates, its local environment vanishes too. In addition, changing the local copy of the environment is solely useful if child programs are to be executed, since they are the only ones which will see new or changed values.

On the other hand, it can be useful to make changes to the master environment. A program could pass along information to other programs through master environment variables. An application could store status information for future incarnations of itself. A TSR can use the variables in the global environment to ensure that it is aware of any changes since it was executed. Unfortunately, MS-DOS does not provide any documented way of accessing the master environment. In order to work with the master environment, we must enter the world of undocumented features.

MS-DOS Memory Management

MS-DOS organizes memory into blocks. Each block is prefaced by a 16-byte paragraph-aligned header called the Memory Control Block, or MCB for short. The MCB contains three pieces of information: a status indicator, the segment of the owning program's PSP, and the length of the block. The status indicator is a byte which contains either the character *M* (indicating that it is a member of the MCB chain) or the character *Z* (denoting this as the last MCB in the chain). The length of the block is stored as a number of 16- byte paragraphs. (For those who are curious, the characters *M* and *Z* are the initials of one of the original developers of MS-DOS.)

Many popular public domain programs use the chain of MCBs to display a map of the programs and data currently resident in memory. The first MCB can be located through an undocumented *INT 21h* service of MS-DOS, *52h*. Function *52h* returns a pointer (in *ES:BX*) to an internal table of MS-DOS values. Immediately preceding this table is the segment address of the first MCB. Starting with the first MCB, a program can follow the chain of memory blocks by a simple formula: add the size of a block plus one to the segment address of the current block to calculate the segment of the next MCB in the chain.

The initial copy of *COMMAND.COM* creates a memory segment which will contain the master environment. Usually, this is segment located in the memory block directly after the one which contains *COMMAND.COM*. At the same time, the environment pointer in *COMMAND.COM*'s PSP is set to *0*. Beginning with MS-DOS version 3.3, however, the location of the environment's memory block may be different. In later versions, the environment pointer in *COMMAND.COM*'s PSP contains the segment of the environment block.

Once the memory block containing the environment is located, the programmer can directly manipulate the variables stored there. Environment variables are stored in sequential order and are terminated by *NUL*s, exactly like C strings. The end of the valid data in the environment is indicated by a pair of consecutive *NUL*s. Each variable consists of a name (customarily in upper case), an equal sign, and a text value.

MSTR_ENV.ASM (Listing 1.1) is an 8086 assembly language module which directly locates the master environment. It contains the public function *findmenv* which returns a pointer to the master environment in *ES:BX*, and the byte size of the environment in *CX*. The function takes no parameters, and corrupts the *AX* and *DX* registers. With a small amount of work, the function could be rewritten to be callable from a high-level language such as C.

findmenv begins by invoking MS-DOS function *52h*. The returned values in the *ES* and *BX* registers are then used to construct a pointer to the segment address of the first MCB. The first MCB is the one for the MS-DOS kernel and device drivers; the second MCB is associated with *COMMAND.COM*. Using the formula mentioned above for following the chain of MCBs, *findmenv* finds the second MCB (for *COMMAND.COM*). That memory block contains the segment of *COMMAND.COM*'s PSP.

Once the PSP has been located, *findmenv* checks the environment pointer stored there. If the pointer is zero, the environment is stored in the next consecutive memory block above *COMMAND.COM*; otherwise, the value represents a segment address from which *findmenv* can build a direct pointer to the master environment block. *findmenv* calculates the size of the environment from the size of its memory block in bytes, and stores that value in *CX*. Finally, the pointer to the master environment is stored in *ES:BX*.

Once you have the address of the master environment, you can begin to work with the *NUL*-terminate strings stored therein. You must be careful not to delete or overwrite important MS-DOS environment variables such as *PATH* or *COMSPEC*. When deleting an environment variable, shift all of the environment variables "above" it "down", to overwrite the deleted variable's space. All of the strings in the environment must be contiguous. Make sure that all of your variables follow the correct format: an uppercase variable name, directly followed by an equal sign (=), directly followed by the variable's text value. The complete set of strings must be terminated by a pair of *NUL*s. Duplicating a variable will only waste space in the master environment; be sure to always delete the old variable, and add new variables at the end of the environment.

MSTR_ENV.ASM has been tested with Microsoft Macro Assembler v5.10 and Borland Turbo Assembler v2.0. It should compile and work under other

MS-DOS assemblers which support the *MASM* syntax. As with all functions which use "undocumented features" of MS-DOS, it is important to realize that these functions may not work with future versions of MS-DOS. I made sure that the functions in *MSTR_ENV.C* worked with MS-DOS versions 2.1, 3.0, 3.21, 3.30, and 4.01. In some variations of MS-DOS v4.00, the undocumented *52h* functions do not work.

Listing 1.1

```
;###########################################################################
;
;   Module Identification
;   ---------------------
;     Env  v1.00  09-Jan-1990
;
;   Purpose
;   -------
;     Allows access to the global MS-DOS environment.
;
;   Environment
;   -----------
;     Language:   Intel 80x86 Assembler
;     Assemblers: Microsoft Macro Assembler v5.10
;                 Borland Turbo Assembler v1.1
;
;   Requirements
;   ------------
;     Hardware:   IBM PC compatible
;     Software:   MS-DOS 2.11 thru 3.30, and 4.01
;                 Does not work with some buggy versions of MS-DOS 4.00
;
;   Author Information
;   ------------------
;     Written by: Scott Robert Ladd
;                 705 West Virginia
;                 Gunnison CO 81230
;
;   Legal Stuff
;   -----------
;     This module has been placed in the public domain.
;
;###########################################################################


TITLE   env.asm
NAME    env

.MODEL  SMALL

.CODE

PUBLIC  findmenv

findmenv PROC NEAR

        mov     ah, 52h                 ; undocumented MS-DOS function locates
        int     21h                     ; internal table. Returned in ES:BX

        mov     ax, word ptr es:[bx-2]  ; get segment ptr from internal table
        mov     es, ax                  ; move segment to ES
        mov     bx, 3                   ; set BX to point to size of first MCB

        add     ax, word ptr es:[bx]    ; add length of first MCB (in para)
        inc     ax                      ; increment 2 more bytes
        inc     ax                      ; to get segment of command shell
```

Listing 1.1 *(cont'd)*

```
        mov     es, ax                  ; move segment to ES
        mov     bx, 44                  ; set BX to point to env seg in shell PSP

        mov     ax, word ptr es:[bx]    ; load AX with env seg in shell PSP
        cmp     ax, 0                   ; check for 0

        jne     fge_1

        mov     ax, es                  ; mov ES to AX where it can be changed
        dec     ax                      ; dec AX to move back one paragraph
        mov     es, ax                  ; mov new segment to ES
        mov     bx, 3                   ; set BX to point to size of shell MCB
        add     ax, es:[bx]             ; add size of shell MCB to ax
        inc     ax                      ; dec AX so it points to env MCB

fge_1:  dec     ax                      ; move 1 para back to env seg MCB
        mov     es, ax                  ; mov ES to AX
        mov     bx, 3                   ; set BX to point to size of shell MCB

        mov     dx, es:[bx]             ; DX now contains size of environment
        mov     cl, 4                   ; set CL for 4 bit shift
        shl     dx, cl                  ; shift DX
        mov     cx, dx                  ; CX now contains size in bytes of env

        inc     ax                      ; set AX to point to env seg data
        mov     es, ax                  ; move that data to ES
        xor     bx, bx                  ; clear BX

        ret                             ; return

findmenv  ENDP

END
```

Converting a Microsoft C Program Into A TSR

Michael J. Young

A memory resident program, or TSR (terminate and stay resident application), is one that permanently establishes itself in memory and then returns control to MS-DOS so that the user can run other programs. The TSR remains dormant in the background until the user presses a designated hotkey, whereupon it springs to life, temporarily suspending the current application and providing instant access to the services it offers.

The module of functions presented in this chapter allows you to convert a normal Microsoft C program into a TSR through a single function call. You can specify the hotkey that is to activate your TSR, and the C function in your program that is to receive immediate control when the hotkey is pressed. You can also specify a unique code to prevent the same resident program from being installed more than once. All the details of installing and activating the program as a TSR are handled invisibly by the functions in this module. The module also includes a function for removing the TSR from memory.

The TSR functions written in C are listed at the end of this chapter in Listing 2.1, and the functions written in assembly language are listed in Listing 2.2. The header file that you must include in your C program to call these functions is given in Listing 2.3.

Writing a TSR under MS-DOS is a complex task. The primary challenge is to design a resident program that will peacefully coexist with the foreground program, with any other TSRs that are currently installed, and with the operating system. It is especially important that a TSR written in the C language be able to freely call MS-DOS functions, since many of the standard library routines make use of these services. This chapter presents one of the many approaches that can be used to achieve these design goals.

There are three functions in the TSR module that can be called directly by your program: *TsrInstall*, *TsrInDos*, and *TsrRemove*. The chapter begins by describing how to call these three functions, and presents an example program that demonstrates their use. The chapter then discusses how the functions work.

The TSR functions not only provide a useful addition to your collection of software tools, but they also furnish an excellent demonstration of the advanced features of Microsoft C. Normally it would be quite difficult to write memory resident interrupt handlers in the C language; however, using Microsoft C language extensions such as interrupt functions (provided with versions 5.0 and later), it was possible to write the entire module in C, with the exception of two short assembly language subroutines.

Using *TsrInstall*

The function *TsrInstall* has the prototype

```
int TsrInstall
    (void (*FPtr) (void),
    int HotKey,
    unsigned long Code)
```

and performs the following primary tasks:

- It checks to make sure that the program has not already been installed in memory. Each program that is installed by this routine should be identified with a unique value in the parameter *Code*.
- It stores all values that will be required by the interrupt handlers after the TSR is installed.
- It installs the appropriate interrupt handlers so that when theuser presses the shift-key combination specified by the *HotKey* parameter, the function given by the *FPtr* parameter receives immediate control.
- It terminates the C program, leaving the entire block of code and data resident in memory.

When you develop a memory resident program using the TSR module, your program should consist of two parts. The first part is the *main* function (plus any functions called by *main*). The function *main* receives control when the user types the name of the program on the command line, and serves to install the TSR. This function should perform any required initialization tasks, print any desired messages to the user, and finally call *TsrInstall* to install the program as a TSR. After the call to *TsrInstall*, control returns to the DOS command line just as if the program had exited normally; however, the program's code and data are not released, but rather remain resident in memory.

The second part of your memory resident program is the portion of the code that is subsequently activated whenever the user presses the designated hotkey. This part of the program consists of the TSR entry routine — the function that receives initial control when the hotkey is pressed — plus any subroutines called by this function.

The first parameter passed to *TsrInstall* (that is, *FPtr*) should be assigned the address of the TSR entry routine. The second parameter (*HotKey*) gives the shift-key combination that is used to activate the program. Note that this parameter does not specify either an ASCII code or an

extended keyboard code, but rather some combination of shift keys. The desired combination may be specified by joining two or more of the following constant identifiers with the C | operator (these constants are defined in the header file *TSR.H*, Listing 2.3):

Identifier	Shift Key
RIGHTSHIFT	Right shift key
LEFTSHIFT	Left Shift key
CONTROL	Ctrl key
ALT	Alt key
SCROLLLOCK	Scroll Lock key
NUMLOCK	Num Lock key
CAPSLOCK	Caps Lock key
INSERT	Ins key

For example, the combination of the *Alt* key and *Left-Shift* key could be specified by the following expression:

```
ALT | LEFTSHIFT
```

The parameter *Code* is of type *unsigned long*. It should range between the values *0x00000001* and *0xffffffff* and be unique for each separate program you convert to a TSR. Once a TSR is installed, no other program having the same ID number may subsequently be installed. By means of this parameter, *TsrInstall* avoids loading redundant copies of the same program.

Since *TsrInstall* terminates the current program, it should obviously never return. If the function does return, an error has occurred and the program has not been installed. In this case, one of the following error codes is passed back to the calling program:

- *INSTALLED*: This error code indicates that a TSR bearing the sameidentification code has already been installed.
- *NOINT*: This value means that there are no free "user" interrupts available. The interrupt vectors in the range from *60h* to *67h* are designated as available for user programs. The first of these vectors that is free (that is, containing zero) is used to store the identifying code (the parameter *Code*) assigned to the TSR. If all of these vectors are occupied, *TsrInstall* returns *NOINT*.
- *WRONGDOS*: The TSR functions require DOS v2.0 or later. If the version is prior to 2.0, this error code is returned. (Notethat the TSR can be successfully installed in the DOS-compatibility environment of OS/2, which returns a version number of 10.0 orhigher.)
- *ERROR*: This general code is returned if the installation procedure failed for some unknown reason.

In most respects, a program destined to become a TSR can be written like any other C program. However, there are five additional guidelines that you should follow when developing your memory resident program.

First, your program — and all the C modules that are linked with it — must be compiled using the small or medium memory model (both of these models place all data within a single segment and use near data pointers by default). You may not compile the program using either the compact, large, or huge model. As you will see, the code in the TSR module (specifically, TSR.C) uses techniques that require the small or medium memory model.

Second, if your program writes to the screen or alters the cursor, you must carefully save and restore the complete video state of the interrupted program. The functions in the TSR module automatically save and restore almost all of the machine state of the suspended program; the task of preserving the video state, however, is left to the application program. Note that you can use the routines *ScrSaveBlock* and *ScrRestoreBlock* (defined in Listing 2.4) to save and restore data from text mode screens.

Third, a resident program should not attempt to allocate memory from DOS. A transient application (that is, a program that does not remain resident in memory) can normally obtain additional memory blocks by dynamically allocating them from DOS. A TSR, however, should restrict its use of memory

to the block explicitly reserved for the program when it terminated and remained resident. Consequently, a TSR should not invoke the MS-DOS services for allocating, releasing, or changing the size of memory blocks (namely, services *48h*, *49h*, and *4Ah*, accessed through interrupt *21h*); likewise, it should not call any of the functions provided by Microsoft C that rely upon these DOS services (namely, *_dos_allocmem*, *_dos_freemem*, and *_dos_setblock*). Note, however, that a TSR can call the standard C memory allocation functions, such as *malloc* and *calloc*, since these functions obtain memory from within the block reserved at the time the program became resident. (However, as you will see in the next section, the TSR module places a fixed upper bound on the amount of memory that can be dynamically allocated through functions such as *malloc*.)

A corollary to the third guideline is that a TSR must not attempt to execute a child process, since doing so entails allocation of additional memory blocks. Consequently, a TSR must not call the DOS service for executing a child process (namely, function *4Bh* of interrupt *21h*); nor should it call any of the C library functions for starting new processes (namely, the *exec...* and *spawn...* families of functions).

Fourth, the TSR should not use floating point numbers. Specifically, it should not declare variables of type *float* or *double*, and it should not call functions in the C math library (which are those declared in the header file *MATH.H*). The inclusion of floating point code causes the C startup code to set certain interrupt vectors; floating point operations require that these interrupt vectors remain intact. As long as these vectors remain unaltered as various programs run in the foreground, the TSR can successfully execute floating point instructions; however, if a foreground application alters one or more of the vectors, any floating point calculations performed by the TSR will fail. Therefore, a robust TSR must avoid floating point operations performed by the math library.

The fifth and final guideline is that your TSR must always return from the function that was initially activated (the function specified by the FPtr parameter). You must not use the *exit* function to terminate the program. The *exit* function is designed to terminate a non-resident program and return to the parent process (usually the MS-DOS command interpreter).

Note that when your TSR entry function receives control in response to the user pressing the hotkey, the TSR module has temporarily disabled any control-break or critical error handlers that may have been installed by the suspended foreground application. (Specifically, during the execution of your TSR code, interrupt vectors *1Bh*, *23h*, and *24h* point to routines that simply return without performing any action; these vectors are restored to their former values before control returns to the suspended application.) If your TSR program needs to handle control-break keys or critical-error conditions, you can install appropriate handlers; however, you do not need to restore the former values for these vectors since this task is performed automatically by the TSR module code that receives control when your TSR issues its final return statement.

Note also that even if the TSR is properly installed, the program may not always receive immediate control when the hotkey is pressed. There are times when it is not safe to interrupt the current process (such as when DOS or the BIOS are engaged in disk activity). The TSR module senses these conditions and simply ignores the hotkey until it is safe to trigger the resident program.

A note to QuickC users: You should not attempt to load a TSR from within the QuickC integrated environment (the TSR should be loaded directly from the DOS command line so that it is given the lowest possible position in memory). Therefore, if you prepare the program within QuickC v1.0, you should select the *Exe* item in the *Output Options* column of the *Compile* menu; this option will produce a freestanding *.EXE* file that can be run from DOS. QuickC v2.0 will automatically produce such a file.

Using *TsrInDos*

The function *TsrInDos* has the prototype

```
int TsrInDos
    (void)
```

This function returns a nonzero value if an MS-DOS service was active when your TSR received control, and it returns 0 if MS-DOS was not active. With this function you may determine which DOS functions your program may safely call. Specifically, if TsrInDos returns a value of 0, you may directly call almost any MS-DOS service or almost any C library function that invokes MS-DOS (exceptions are noted later in this section). However, if TsrInDos returns a non-zero value, then you must not use DOS functions 01h through 0Ch. This set of forbidden functions manage input and output to the basic character devices (console, printer, and serial port). The following C library functions, which are defined in the CONIO.H header file, employ these services: cgets, cputs, cprintf, cscanf, getch, getche, kbhit, putch, and ungetch.

Fortunately, it is almost always possible to obtain the services offered by the forbidden set of procedures through the low-level C library functions provided by Microsoft (notably *_bios_keybrd* and display functions such as *_outtext*), or by using some of the video functions presented in Listing 2.4. Either you can call *TsrInDos* before using any of the restricted functions, or you can simply eliminate the restricted functions from your TSR program. Later in this chaper I'll explain why you shouldn't call this set of MS-DOS services.

Using *TsrRemove*

The function *TsrRemove* has the prototype

```
int TsrRemove
    (void)
```

The memory resident portion of your program can call *TsrRemove* to remove itself from memory. It should call this function immediately before returning control to the foreground program. *TsrRemove* releases the memory occupied by the TSR and restores all interrupt vectors to their former values.

If successful, *TsrRemove* returns the value *NOERROR* (which equals *0*; error constants are defined in *TSR.H*, Listing 2.3). At certain times, however, it is not safe for the TSR to remove itself from memory (specifically, when the foreground application or a subsequently loaded TSR has hooked one or more of the interrupt vectors used by the module). In this case, *TsrRemove*

returns the value *CANTREMOVE*. It may be possible to remove the TSR *after* terminating the foreground application, or after removing any other TSRs that were installed after your program.

An Example Program

The program in Listing 2.4 demonstrates the use of *TsrInstall*, *TsrInDos*, and *TsrRemove*. This program consists of two functions: *main* and *Test*. The function *main* calls *TsrInstall* to convert the program to a TSR, specifying *Test* as the function to receive control when the user presses the hotkey. The selected hotkey is *Ctrl-Left Shift*, and the identifying code is *0x11111111*. If another TSR is developed that might be installed at the same time as this program, a different hotkey combination and identifying code should be chosen. If *TsrInstall* is successful, it will never return; however, if an error occurs in the installation process, control returns to *main*, which then prints the appropriate error message and terminates with an *ERRORLEVEL* setting of *1*.

If *TsrInstall* is successful, control returns to DOS. Subsequently, each time the user presses *Ctrl-Left Shift* (and it is safe to interrupt the current process), the function *Test* receives control. This function saves and restores the current screen contents using *ScrSaveBlock* and *ScrRestoreBlock*, displays a window, and pauses for user input by calling the Microsoft C function *_bios_keybrd*. Note that this program, written to provide a simple example, will work properly only if the system is in a text mode. Since *Test* does not alter the cursor, it does not need to preserve the cursor parameters.

If the user presses *r* or *R*, *Test* calls *TsrRemove* to remove the TSR from memory. If the user presses any other key, the function simply continues.

Listing 2.5 provides a *MAKE* file for preparing both the TSR function module and the example program.

How The TSR Functions Work

The General Strategy

Before discussing the details of the individual functions in the TSR module, this section provides an overview of the basic tasks required to install and activate a memory resident program, and the general strategies that the TSR module uses to accomplish these tasks.

The most fundamental task required to install a TSR is to load the program into a safe area of memory that will not be overwritten by subsequent applications. Fortunately, MS-DOS provides a service that allows a program to terminate without relinquishing the memory it occupies. MS-DOS reserves this area of memory by removing it from its list of free memory blocks, and loads all subsequent applications at higher memory addresses. The TSR module accesses this DOS service through the Microsoft C library function *_dos_keep*. The only complication in using this function is that it is necessary to specify the size of the program that is to be kept resident. The details of using *_dos_keep* are discussed in the next section.

The next task required to create a TSR is to establish some means for activating the program through a hotkey. The basic strategy used by the TSR module is to replace the current keyboard interrupt (*09h*) handler with a Microsoft C interrupt function. Each time a key is pressed or released, the interrupt function receives control. This function first calls the former interrupt handler, and then simply tests the BIOS shift status flag to see whether the designated hotkey is pressed. If the hotkey is pressed, the routine performs several additional tests and possibly activates the TSR entry function.

Another important duty of the TSR code is to prevent the resident program from being activated at a time when it is not safe to interrupt the current process. The first such dangerous time is when the BIOS is engaged in disk activity. Therefore, the TSR module also installs a handler for interrupt *13h*, which is the entry point for the BIOS disk services used by MS-DOS and by some application programs. The interrupt *13h* function (in the assembly language file *TSRA.ASM*) first increments a global flag indicating that BIOS disk activity is in progress, calls the original interrupt *13h* routine, and then decrements the flag when the original routine returns. Therefore, before the keyboard interrupt handler activates the TSR entry function, it checks this flag and returns immediately if the flag is greater than zero, indicating that the disk services are currently in progress.

It is also unsafe to activate a TSR when most MS-DOS services are active. The danger of suspending an MS-DOS service is that the TSR itself may subsequently invoke an MS-DOS function, which would most likely corrupt the stack used by the original invocation of DOS. In general, the MS-DOS kernel cannot be interrupted at an arbitrary point and have its code reused

by another process (since the code cannot be recursively reentered, it is termed nonreentrant). MS-DOS, however, provides a partial solution to this problem. Whenever DOS is active, it sets an internal flag to a non-zero value; the address of this *INDOS* flag is returned by the undocumented, but often used, function *34h*. Therefore, in addition to checking the BIOS disk activity flag, the keyboard interrupt handler needs to check the *INDOS* flag, and return immediately if either of these flags is nonzero.

This technique for avoiding interrupting MS-DOS, however, has a serious problem. While the command interpreter is waiting for input at the DOS prompt, the *INDOS* flag is set to 1, since DOS uses one of its own services to read characters. While DOS is waiting at the command prompt, however, it is relatively safe to activate a TSR; in fact, all DOS function calls (accessed through interrupt *21h*) can be used at this time except those in the range *01h* to *0Ch*. Therefore, the TSR needs some way of knowing that although the *INDOS* flag is not *0*, DOS is simply waiting at the command line and the TSR function may be safely called. Again, MS-DOS provides a solution: while DOS waits for user input at the prompt, it continually invokes interrupt *28h* (which is used to activate background processes such as print spooling, and is known as the DOS idle interrupt). Therefore, the occurrence of interrupt *28h* is a sufficient indication (regardless of the state of the *INDOS* flag) that the TSR may be activated. Accordingly, the TSR module installs a fourth and final interrupt handler, which intercepts interrupt *28h* and provides a second door to the TSR.

In summary, there are two parallel mechanisms for activating the TSR when the hotkey is pressed. First, the TSR may be activated through the keyboard interrupt handler, which must first check both the BIOS flags and the *INDOS* flag. Second, it may be activated through the interrupt *28h* handler, which needs to check only the BIOS disk activity flag.

Your TSR program can determine whether it was activated through the interrupt *28h* handler by calling the function *TsrInDos*, as described in the previous section. If this function returns *TRUE*, the *INDOS* flag is nonzero and the TSR must have been activated through the interrupt *28h* handler; in this case, your program must not call DOS functions *01h* through *0Ch*, since these functions use the same operating system stack that DOS was using when it

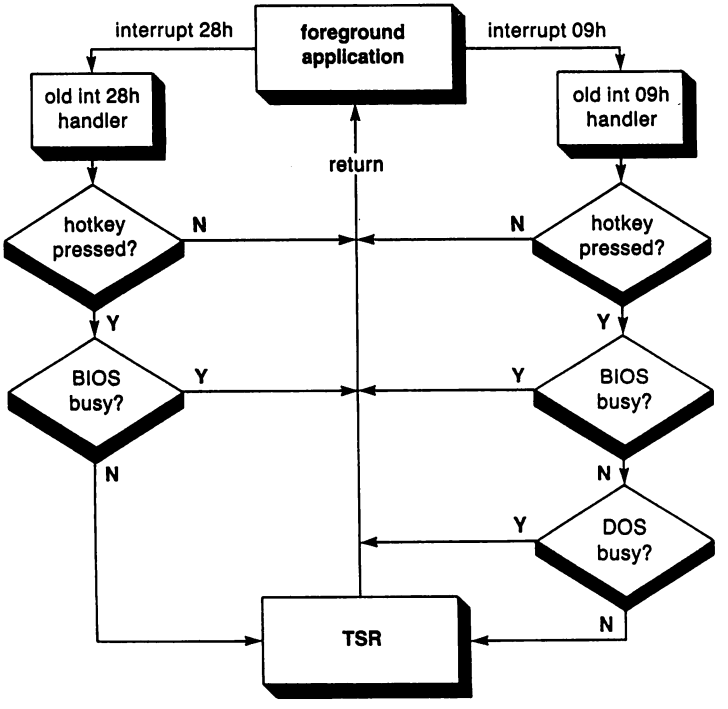
was interrupted. The parallel mechanism for activating a TSR is illustrated in Figure 2.1.

Once it has been determined that it is safe to activate the TSR, the final major task is to save the complete machine state of the interrupted program, set up the runtime environment for Microsoft C, and call the TSR entry function that was specified at installation time. When the entry function returns, the machine state must be restored before returning to the suspended application.

Details Of The Implementation

The installation function, *TsrInstall* (which is located in the C file of Listing 2.1), begins by testing to see if the TSR has already been installed in memory, by searching through the user interrupts (numbers *60h* to *67h*) for

Figure 2.1 *The parallel mechanism for activating a TSR.*



the identification number passed in the parameter *Code*. If the code is not found, indicating that the program has not already been installed, then the function proceeds to place the code value in the first unused (that is, equal to *0*) interrupt vector within this range. *TsrInstall* next checks that it is running under DOS v2.0 or later, since prior versions do not support some of the functions required to install a memory resident program.

Next, *TsrInstall* saves the following important data items that will be used by the interrupt handlers, which are installed later in the function:

- The hotkey shift mask is saved in the variable *HotKeyMask*. The handlers for interrupts *09h* and *28h* will use this flag to determine whether the hotkey is pressed.
- The address of the TSR entry routine, given by the parameter *FPtr*, is stored in the function pointer *UserRtn*. The handlers for interrupts *09h* and *28h* will use this pointer to invoke your TSR program.
- The segment and offset values of the current disk transfer area (DTA) are saved so that the C DTA can be assigned before the TSR receives control.
- The address of the *INDOS* flag, discussed in the previous section, is obtained through DOS service *34h*, and is stored in the *far* pointer *PtrInDos*. As explained, the interrupt *09h* handler will test this flag to determine whether it is safe to activate the TSR.
- Next, *TsrInstall* calls the function *InitPSP* to initialize the functions that obtain and set the value of the current program segment prefix (PSP) maintained by DOS. The PSP, and the functions that manage the system's record of the PSP, will be explained later in this section.
- Finally, *TsrInstall* saves the current contents of the three interrupt vectors that will subsequently be replaced.
- The values of the C stack segment and stack pointer must also be saved, so that the TSR can use the C stack rather than borrowing the stack belonging to the interrupted program. These values, however, are not stored until the necessary data are obtained in the course of calculating the size of the program, later in *TsrInstall*.

Once the required data have been stored, *TsrInstall* installs handlers for interrupts *28h* (the DOS "idle" interrupt), *13h* (the BIOS disk services), and *09h* (the hardware keyboard interrupt).

The last task performed by *TsrInstall* is to calculate the size of the memory block occupied by the program and then to call the Microsoft C function *_dos_keep* to terminate the program while leaving the specified block resident in memory. The size of the memory block passed to *_dos_keep* must be given as the number of 16-byte paragraphs, and is calculated through the following sequence of steps:

1. *TsrInstall* calls the Microsoft C function *sbrk*; when this function is passed a value of *0*, it simply returns the offset address (with respect to the *DS* register) of the first byte beyond the program's stack, which is the base of the area used for the program heap (the heap is the pool of memory that is dynamically allocated through C functions such as malloc). The return value is stored in the variable *OffHeapBase*.
2. If it encounters an error, the function *sbrk* returns a value of *0xffff*; in this case, *TsrInstall* immediately returns to the calling program, passing back the code for a general error, *ERROR*.
3. *TsrInstall* now calculates the paragraph address (that is the absolute number of 16-byte paragraphs from the beginning of memory) of the base of the heap. This value is calculated and assigned to the variable *ParaHeapBase* in two steps. First, the offset in bytes of the heap base with respect to the *DS* register is converted to paragraphs (rounding up to the nearest whole paragraph) using the expression

```
ParaHeapBase = (OffHeapBase + 15) >> 4;
```

Second, the value in the *DS* register (which is the paragraphaddress of the beginning of the default data segment) is added to *ParaHeapBase*, as follows:

```
segread (&SReg);
ParaHeapBase += SReg.ds;
```

Note that the Microsoft C function *segread* obtains the current contents of the segment registers. At this point *ParaHeapBase* contains the paragraph address of the base of the heap.

4. The final expression for the number of paragraphs to keep in memory that is passed to *_dos_keep* is as follows:

```
ParaHeapBase + HEAPSIZE - _psp
```

The constant *HEAPSIZE* is defined in *TSR.C*, and contains the number of paragraphs required for the program heap; accordingly, this value is added to *ParaHeapBase* to provide memory for the heap. Finally, the predefined variable *_psp* is subtracted; since this variable contains the paragraph address of the beginning of the program's memory allocation, the result is the total number of paragraphs to be kept resident in memory.

The first parameter passed to *_dos_keep* is the process return code, which is assigned to the DOS variable *ERRORLEVEL* when the program terminates.

Note that you can adjust the amount of memory reserved for the program's heap by assigning various values to the identifier *HEAPSIZE* (defined at the beginning of the C source file, *TSR.C*). Once you have developed a TSR, you can experiment to find the appropriate value for this constant. If the value is too large, the TSR will consume an undue amount of memory. If the value is too small, C memory allocation functions will overwrite memory that does not belong to the process, resulting in the DOS error message

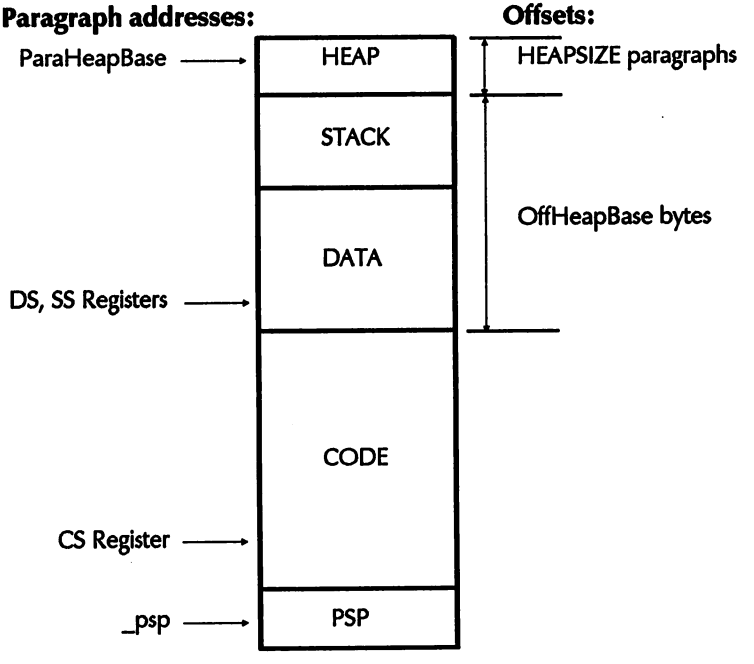
```
Memory Allocation Error. System Halted.
```

Note that even if your program does not explicitly allocate memory from the heap, C library functions that your program calls may allocate such memory.

Figure 2.2 provides a map of the block of code and data that remains resident in memory, and illustrates the values used to calculate the size of the program. Note that *TsrInstall* performed one additional step in the course of calculating the program size: assigning values to the variables CSS

and CSP, which store the values of the C stack segment and stack pointer. As you will see in the next section, these variables are used to switch to the C stack when the TSR is activated. The value of the stack segment (the paragraph address of the base of the stack) is obtained from the call to ***segread***. The value of the stack pointer is obtained from the call to ***sbrk***. As you saw, ***sbrk*** returns the offset address of the first byte beyond the program's stack. When a stack is first used, the stack pointer (which holds the offset of the top of the stack) should contain the address of the first byte beyond the end of the stack (the stack grows down in memory, and when a value is pushed on the stack, the processor first decrements the stack pointer by two, and then writes the value to memory). Therefore, the program saves the value returned from ***sbrk*** in the variable that stores the C stack pointer (CSP).

Figure 2.2 *A memory map of the block of code and data that remain resident in memory.*



The function *NewInt13*, located in the assembly language file of Listing 2.2, is the new handler for interrupt *13h*, which is used to access the BIOS disk services. *NewInt13* increments the value of *Int13Flag* (which is initialized to zero), calls the original handler (the address of which was stored by *TsrInstall*), and then decrements *Int13Flag* upon return from the original handler. The flag *Int13Flag* therefore has a nonzero value only when one of the BIOS disk services is active. The program uses a counter instead of a simple true or false flag because of the small possibility that the disk services may call themselves recursively. (Note that to access *Int13Flag*, which is contained in the C data segment, the *DS* register is temporarily assigned the address of the C data segment, *DGROUP*.)

The function *NewInt09* (in the C file of Listing 2.1) is the keyboard interrupt handler. This function first calls the former interrupt handler through the function pointer *OldInt09*. It then disables the interrupts so that it will not be interrupted in the middle of testing and setting the *Busy* flag. This flag serves to prevent the TSR from being recursively reentered while it is active. (Note that since the TSR switches to a fixed location in a local stack, and uses static and external variables to store data, the code is not reentrant.) *NewInt09* next calls *GetShift* to see if the hotkey is pressed. If the hotkey is currently pressed, the function tests whether the BIOS disk services are active through the *Int13Flag* variable, and whether DOS is busy through the *far* pointer to the *INDOS* flag, *PtrInDos*. If all tests are passed, the function *PreActivate* is called, which performs the initial steps necessary to activate the TSR entry function. Upon return from *PreActivate*, the *Busy* flag is switched off and the interrupt handler returns.

The function *NewInt28* (also in the C file of Listing 2.1) is the interrupt *28h* handler and performs the same tasks as *NewInt09* except that it does not check whether MS-DOS is active, for reasons described in the previous section.

The function *PreActivate* is located in the assembly language file of Listing 2.2, and is called by *NewInt09* or *NewInt28*, when either one of these interrupt handlers has detected that the hotkey is pressed and that it is safe to invoke the TSR. *PreActivate* performs the following tasks:

- It saves the existing values of the stack segment and stack pointer registers.

- It switches to the C stack by assigning *SS* and *SP* the values saved by *TsrInstall* in the variables *CSS* and *CSP* during installation.
- It calls the C function, *Activate*, which performs some additional preparations and invokes the TSR entry routine.
- Upon return from the TSR code, *PreActivate* restores the stack-belonging to the interrupted program.

Note that the interrupts are disabled while the program manipulates the stack registers, to prevent possible problems if a hardware interrupt were to occur before both registers have been assigned appropriate values.

There are two important reasons for switching to the C stack. First, the current stack may be quite small. Switching to the C stack provides the TSR with the full stack space that is normally allocated to a Microsoft C program.

Second, under the small data models, the C compiler assumes that the stack segment is equal to the data segment. Specifically, when an address is passed to a function, only the offset portion is used. The offset, however, may be relative either to the data segment (for external and static variables) or to the stack segment (for automatic variables). The function simply uses the offset in conjunction with the data segment register, assuming that this register is equal to the stack segment register. If the TSR did not switch to the C stack, this assumption would be false and C functions would be unable to access automatic data through addresses passed as parameters.

The function *Activate*, in the C file of Listing 2.1, completes the task of saving the machine state of the interrupted program and setting up the runtime environment for the TSR program, and then calls the TSR entry function.

Activate first saves the current values of the control-break interrupt vectors (numbers *23h* and *1Bh*) and the critical-error interrupt vector (number *24h*). It then sets the control-break vectors to point to routines (*NewInt23* and *NewInt1B*) that simply return without performing any action. This precaution is taken to prevent a break handler belonging to the interrupted foreground process from receiving control if the user presses *Ctrl-C* or *Ctrl-Break*. The critical-error vector is pointed to a routine (*NewInt24*) that simply assigns the value *0* to register *AX*, and then returns; this value signals DOS that the error is to be ignored. Installing this critical-error handler

prevents inadvertently activating the critical-error handler belonging to the interrupted foreground program, or activating the default DOS critical- error handler. (In the event of a critical error, the default DOS handler would overwrite the screen with the familiar "Abort, Retry, Ignore, Fail?" message, and then possibly attempt to abort the TSR.)

Installing these dummy control-break and critical-error handlers prevents the certain disaster of activating an inappropriate interrupt routine; however, as mentioned previously in the article, a robust TSR may need to install custom control-break and critical-error handlers. These handlers should be installed at the beginning of the TSR entry function (replacing the dummy routines installed by the TSR module).

Activate also saves the current contents of the keyboard interrupt vector (number *09h*) in the variable *OldRTInt9*. It then points this vector to the keyboard interrupt handler that was active at the time the TSR was installed (which was stored in *OldInt9* by *Install*). This step temporarily removes any keyboard handler that may have been installed by the foreground application; some programs (for example, the Quick C development environment) install keyboard handlers that are incompatible with a TSR (they prevent TSRs from properly reading the keyboard).

Activate next saves the old disk transfer address and switches to the C disk transfer address that was saved by the installation function.

Before calling the TSR entry routine, *Activate* must inform MS-DOS that a new process is active by setting the system's record of the current program segment prefix (PSP) to the correct value for the C program. The PSP is a 256 byte segment located immediately preceding a program in memory (see Figure 2.2); it contains important system information regarding the associated process. DOS maintains a record of the segment address of the PSP of the currently active process. (Among other uses for the PSP, DOS stores a file table within this area, which contains an entry for each open file handle belonging to the process.)

When a new program is started from the command line, or through the DOS service for executing a child process, DOS automatically updates its record of the current PSP. However, when a TSR is activated through a hotkey, DOS is unaware of the change in process, and therefore you must explicitly change the system's record of the current PSP. DOS provides two

undocumented functions for managing its record of the PSP. These functions are accessed through interrupt *21h*; function *51h* gets the current PSP, and function *50h* sets it. The problem with these two functions, however, is that under certain versions of DOS, they are unreliable if called when DOS is active (that is, when the *INDOS* flag is nonzero). Accordingly, the TSR module provides its own functions for getting and setting the PSP: *InitPSP*, *GetPSP*, and *SetPSP* (in the C file of Listing 2.1).

The function *InitPSP* is called by *TsrInstall* during installation of the TSR. *InitPSP* obtains a table (which has one or two entries) of the addresses within the operating system code and data segment where the values of the current PSP are stored. Once these addresses are known, *GetPSP* can obtain the current PSP by simply reading one of these addresses, and *SetPSP* can set the PSP by writing the specified value directly to each of the stored addresses.

InitPSP works by searching the MS-DOS code and data for bytes that match the current PSP (which is contained in the predefined C variable *_psp*). When it finds a byte that contains the value of the current PSP, it then calls function *50h* to set the PSP to some other value; if the DOS memory location changes to the new value after function *50h* is called, it must be one of the addresses where DOS stores the PSP, and therefore this address is placed in the table (*PtrPSPTable*). The routine then calls function *50h* once again to restore the former PSP value. (Note that *InitPSP* may safely call DOS function 50h, since InitPSP is called only from the installation routine, which is run from the DOS command line.)

InitPSP obtains the segment address of the beginning of the MS-DOS segment from the address of the *INDOS* pointer obtained during installation. It obtains the segment address of the end of DOS by calling another undocumented function, number *52h* (this function supplies the address of the first DOS memory control block, which immediately follows the DOS segment). Note that these two *far* addresses are converted to a common format in which the segment portion of the address is zero, so that the addresses may be numerically compared in the loop (this technique is possible since the DOS code is located entirely within the first 64 kilobytes of memory).

Accordingly, the function *Activate* calls *GetPSP* and saves the current value of the PSP in the variable *OldPSP*.

Finally, the TSR entry function is called through the function pointer *UserRtn* that was defined in the installation procedure. Upon return from the TSR routine, *Activate* restores all values that it saved before calling the routine: the PSP, the disk transfer address, the control- break and critical-error vectors, and the keyboard vector. Activate then returns control to *PreActivate*, which switches back to the previous stack and then returns to the interrupt handler that called it (*NewInt9* or *NewInt28*).

The function *TsrInDos* simply returns the current value of the *INDOS* flag, which is obtained through the pointer *InDosPtr*.

Finally, the function *TsrRemove* causes the TSR to be removed from memory by performing the following steps:

1. It verifies that the three interrupt vectors that were set by the installation program (numbers *09h*, *13h*, and *28h*) still point to your TSR code. If one or more of these vectors now points to some new address, either the foreground application or another TSR that was subsequently installed must have reset the vector(s) to point to its own code. In this

Listing 2.1 *C functions for converting a Microsoft C program into a TSR.*

```
/*
     File Name:  TSR.C

     This file contains the C functions belonging to the TSR module.  The
     assembly language functions are in the file TSRA.ASM (Figure 2).

     Copyright (C), 1989, Michael J. Young.  All rights reserved.
*/
#include <DOS.H>
#include <STDLIB.H>
#include <MALLOC.H>

#include "TSR.H"

#pragma check_stack (off)      /* Eliminate stack checks.                  */

#define HEAPSIZE 1024          /* Memory allowed for C heap, in 16-byte    */
                               /* paragraphs.                              */

typedef void (interrupt far *VIFP) ();  /* Void Interrupt Function Pointer. */

                               /* Variables used internally in file:       */
static int Busy = 0;           /* Flag indicates TSR is active.            */
static unsigned int CDtaOff;   /* C Disk Transfer Area offset.             */
static unsigned int CDtaSeg;   /* C Disk Transfer Area segment.            */
static int HotKeyMask;         /* Hot key shift mask.                      */
static unsigned int OldDtaOff; /* Old disk transfer address offset.        */
static unsigned int OldDtaSeg; /* Old disk transfer address segment.       */
static void (interrupt far *OldInt1B) (void);  /* Old int 1Bh vector.      */
static void (interrupt far *OldInt23) (void);  /* Old int 23h vector.      */
static void (interrupt far *OldInt24) (void);  /* Old int 24h vector.      */
static void (interrupt far *OldInt28) (void);  /* Old int 28h vector.      */
static void (interrupt far *OldInt9) (void);   /* Old int 09h vector.      */
static void (interrupt far *OldRTInt9) (void); /* Old int 09h vector at    */
                                               /* TSR runtime.             */
static unsigned far *PtrEnvSeg;    /* Pointer to environment segment address*/
static char far *PtrInDos;         /* Pointer to DOS 'indos' flag.          */
static int Unload = 0;             /* Flag to release TSR memory.           */
static unsigned int UserInt;       /* Number of user interrupt for storing  */
                                   /* the TSR's signature.                  */
static void (*UserRtn) (void);     /* Pointer to user's start routine.      */

                                   /* Functions defined within this file:   */
static unsigned GetPSP (void);     /* Gets current PSP address from DOS.    */
int GetShift (void);               /* Gets status of shift keys.            */
static void InitPSP (void);        /* Saves DOS PSP storage locations.      */
static void interrupt far NewInt1B (void);     /* New int 1Bh handler.      */
static void interrupt far NewInt23 (void);     /* New int 23h handler.      */
                                               /* New int 24h handler:      */
static void interrupt far NewInt24 (unsigned ES, unsigned DS, unsigned DI,
                                    unsigned SI, unsigned BP, unsigned SP,
                                    unsigned BX, unsigned DX, unsigned CX,
                                    unsigned AX);
```

Listing 2.1 *(cont'd)*

```
static void interrupt far NewInt28 (void);      /* New int 28h handler.      */
static void interrupt far NewInt9 (void);       /* New int 09h handler.      */
static void SetPSP (unsigned PSP); /* Sets DOS's record of PSP address.      */

                              /* Declarations shared with assembler module: */
unsigned int CSS;                  /* C stack segment.                       */
unsigned int CSP;                  /* C stack pointer.                       */
extern int Int13Flag;              /* Flag:  BIOS interrupt 13h active.      */
void (interrupt far *OldInt13) (void); /* Old int 13h vector.               */

void interrupt far NewInt13 (void);    /* New int 13h handler (in ASM mod.).*/
void far PreActivate (void);           /* Pre-activate routine (in ASM mod.)*/


void far Activate
     (void)
/*
     This function is called by the routine 'PreActivate' (defined in the
     assembly language module).
*/

     {
     union REGS Reg;                   /* Passes values to 'int86x'.       */
     struct SREGS SReg;                /* Passes values to 'int86x'.       */
     unsigned OldPSP;                  /* Stores PSP address of            */
                                       /* interrupted process.             */

                                       /* Set new interrupt vectors:       */
     OldInt23 = _dos_getvect (0x23);   /* Save old break-key vector.       */
     OldInt24 = _dos_getvect (0x24);   /* Save old critical-error vector.  */
     OldInt1B = _dos_getvect (0x1b);   /* Save old BIOS break vector.      */
     OldRTInt9 = _dos_getvect (0x09);  /* Save runtime int 9 handler.      */
     _dos_setvect (0x23,NewInt23);     /* Set new break-key handler.       */
     _dos_setvect (0x24,NewInt24);     /* Set new critical-error handler.  */
     _dos_setvect (0x1b,NewInt1B);     /* Set new BIOS break handler.      */
     _dos_setvect (0x09,OldInt9);      /* Set old int 9 handler.           */

     Reg.h.ah = 0x2f;                  /* Save old Disk Transfer Address.  */
     int86x (0x21,&Reg,&Reg,&SReg);
     OldDtaSeg = SReg.es;
     OldDtaOff = Reg.x.bx;

     Reg.x.dx = CDtaOff;               /* Set C Disk Transfer Address.     */
     SReg.ds = CDtaSeg;
     Reg.h.ah = 0x1a;                  /* DOS set DTA service.             */
     int86x (0x21,&Reg,&Reg,&SReg);

     OldPSP = GetPSP ();               /* Get PSP address of interrupted   */
                                       /* process.                         */
     SetPSP (_psp);                    /* Set PSP for C program.           */

     (*UserRtn) ();                    /* Call the user's C function.      */

     SetPSP (OldPSP);                  /* Restore PSP for the interrupted  */
                                       /* process.                         */
```

Listing 2.1 *(cont'd)*

```
    Reg.x.dx = OldDtaOff;              /* Restore Old Disk Transfer Address*/
    SReg.ds = OldDtaSeg;
    Reg.h.ah = 0x1a;
    int86x (0x21,&Reg,&Reg,&SReg);

                                       /* Restore old interrupt vectors:   */
    _dos_setvect (0x23,OldInt23);      /* Restore old break-key handler.   */
    _dos_setvect (0x24,OldInt24);      /* Restore old crit-error handler.  */
    _dos_setvect (0x1b,OldInt1B);      /* Restore old BIOS break handler.  */
    if (!Unload)
        _dos_setvect (0x09,OldRTInt9);/* Restore runtime int 9 handler.   */

    if (Unload)                        /* If TSR is being released, free   */
        {                              /* both blocks allocated to program.*/
        _dos_freemem (_psp);           /* Free the main program block.     */
        _dos_freemem (*PtrEnvSeg);     /* Free the environment block.      */
        }

    return;

    } /* end Activate */


int GetShift
    (void)
/*
    This function returns the BIOS shift status word.
*/
    {
    /* Far pointer to shift flag in BIOS data area:                        */
    unsigned far *PtrShiftFlag = (unsigned far *)0x00400017;

    return (*PtrShiftFlag);

    } /* end GetShift */


static void interrupt far NewInt1B
    (void)
/*
    This is the new handler for interrupt 1Bh, which is invoked by the BIOS
    keyboard routine when it detects the Ctrl-break keystroke.  While the
    TSR is active, this function replaces any BIOS-level break handler that
    might have been installed by the interrupted program.
*/
    {
    return;

    } /* end NewInt1B */
```

Listing 2.1 *(cont'd)*

```
static void interrupt far NewInt23
     (void)
/*
     This is the new handler for interrupt 23h, which DOS activates when it
     detects a control-break key.  The function replaces the default DOS
     control-break handler, which aborts the process;  by simply returning,
     this function prevents DOS from attempting to terminate the TSR.
*/
     {
     return;
     } /* end NewInt23 */


static void interrupt far NewInt24 (unsigned ES, unsigned DS, unsigned DI,
                                    unsigned SI, unsigned BP, unsigned SP,
                                    unsigned BX, unsigned DX, unsigned CX,
                                    unsigned AX)
/*
     This is the new handler for interrupt 24h, which DOS invokes when a
     critical error occurs.  It assigns a value of 0 to AX, which informs
     DOS that the error should be ignored.  The function prevents DOS from
     attempting to abort the TSR.
*/
     {
     AX = 0;
     } /* end NewInt24 */


static void interrupt far NewInt28
     (void)
/*
     This is the new interrupt 28h, the 'DOS idle interrupt', handler.
*/
     {
     (*OldInt28) ();          /* Chain to previous interrupt 28 handler.    */
     _disable ();             /* Disable interrupts to test and set         */
     if (Busy)                /* 'Busy' semaphore.                          */
          return;
     Busy = 1;
     _enable ();              /* Re-enable interrupts.                      */
     if ((GetShift () & HotKeyMask) != HotKeyMask)     /* Test if hot key   */
          {                                            /* is pressed.       */
          Busy = 0;
          return;
          }
     if (Int13Flag)           /* Test if BIOS service is active.            */
          {
          Busy = 0;
          return;
          }

     PreActivate ();          /* Conditions are safe, therefore activate TSR*/

     Busy = 0;                /* Reset the active semaphore.                */
     return;
     } /* end NewInt28 */
```

Listing 2.1 *(cont'd)*

```
static void interrupt far NewInt9
     (void)
/*
     This is the new handler for interrupt 09, the hardware interrupt
     activated by the keyboard.
*/
     {
     (*OldInt9) ();       /* Chain to prior interrupt 09 handler.          */
     _disable ();         /* Disable interrupts to test and set 'Busy'     */
     if (Busy)            /* Test if TSR already active.                   */
          return;
     Busy = 1;            /* Set busy flag.                                */
     _enable ();          /* Enable interrupts.                            */
     if ((GetShift () & HotKeyMask) != HotKeyMask)     /* Test if hot key  */
          {                                            /* is pressed.      */
          Busy = 0;
          return;
          }
                          /* Test if EITHER a BIOS service OR DOS is active: */
     if (Int13Flag || *PtrInDos)
          {
          Busy = 0;
          return;
          }

     PreActivate ();          /* Conditions are safe, therefore activate TSR*/

     Busy = 0;                /* Reset the active semaphore.                */

     return;

     } /* end NewInt9 */


int TsrInDos
     (void)
/*
     This function returns zero if DOS is not currently active, and a
     non-zero value otherwise.
*/
     {
     return (*PtrInDos);      /* Uses the 'indos' flag maintained by DOS.   */

     } /* end TsrInDos */


int TsrInstall
     (void (*FPtr) (void),
     int HotKey,
     unsigned long Code)
/*
     This routine terminates the C program, but leaves the code resident in
     memory.  It installs interrupt handlers so that when the shift key
     combination specified by 'HotKey' is pressed, the function pointed to
     by 'FPtr' receives control, provided that conditions are safe.
```

Listing 2.1 *(cont'd)*

```
     If successful, the function never returns.  If an error occurs, it
     returns one of the following error codes:

     INSTALLED      A TSR bearing the same 'Code' value has already been
                    installed.
     NOINT          No free "user" interrupts are available.
     WRONGDOS       DOS version is prior to 2.0.
     ERROR          An unidentified error has occurred.
*/
     {
     unsigned int i;                    /* Loop index.                    */
     struct SREGS SReg;                 /* Gets segment registers.        */
     union REGS Reg;                    /* Holds registers for 'int86'.   */
     unsigned int OffHeapBase;          /* Offset of heap base from DS.   */
     unsigned int ParaHeapBase;         /* Paragraph address of heap base.*/

     UserInt = 0;
     for (i = 0x60; i <= 0x67; ++i)     /* Search "user" interrupt vectors.*/
          {
          if (_dos_getvect (i) == (VIFP)Code)  /* Test for TSR signature.  */
               return (INSTALLED);
                                        /* Look for free vector:          */
          if (UserInt == 0 && _dos_getvect (i) == (VIFP)0)
               UserInt = i;
          }
     if (UserInt == 0)                  /* No free "user" interrupts.     */
          return (NOINT);

     _dos_setvect (UserInt, (VIFP)Code);  /* Write TSR signature to vector.*/

     if (_osmajor < 2)                  /* Test that OS version is >= 2.0.*/
          return (WRONGDOS);            /* Wrong DOS version.             */
     FP_SEG (PtrEnvSeg) = _psp;         /* Initialize pointer to segment  */
     FP_OFF (PtrEnvSeg) = 0x2c;         /* address of TSR's environment.  */

     HotKeyMask = HotKey;               /* Save hotkey shift mask.        */

     UserRtn = FPtr;      /* Save address of the routine that is activated */
                          /* when the hotkey is pressed.                   */

     Reg.h.ah = 0x2f;                   /* Invoke MS-DOS function to get  */
     int86x (0x21,&Reg,&Reg,&SReg);     /* current Disk Transfer Address. */
     CDtaSeg = SReg.es;
     CDtaOff = Reg.x.bx;

     Reg.h.ah = 0x34;                   /* Get pointer to INDOS flag      */
     int86x (0x21,&Reg,&Reg,&SReg);     /* using the undocumented DOS     */
                                        /* function 34h.                  */
     FP_SEG (PtrInDos) = SReg.es;
     FP_OFF (PtrInDos) = Reg.x.bx;
     InitPSP ();                        /* Find and save table of PSP     */
                                        /* addresses from DOS.            */
     OldInt28 = _dos_getvect (0x28);    /* Save old interrupt vectors.    */
     OldInt13 = _dos_getvect (0x13);
     OldInt9 = _dos_getvect (0x9);
```

Listing 2.1 *(cont'd)*

```
    _dos_setvect (0x28,NewInt28);      /* Initialize new interrupt vectors.*/
    _dos_setvect (0x13,NewInt13);
    _dos_setvect (0x9,NewInt9);

                                       /* Calculate memory to keep:        */
    OffHeapBase = (unsigned int)sbrk (0);   /* Get offset of heap base     */
                                            /* with respect to DS register.*/

    if (OffHeapBase == 0xffff)         /* Return value 0xffff indicates    */
        return (ERROR);                /* that an error has occurred.      */

                              /* Calculate paragraph address of heap base:  */

                              /* First, convert offset to paragraphs,       */
                              /* rounding up:                               */
    ParaHeapBase = (OffHeapBase + 15) >> 4;

    segread (&SReg);          /* Then, add DS register.                     */
    ParaHeapBase += SReg.ds;

    CSS = SReg.ss;            /* Save the C stack segment.                  */
    CSP = OffHeapBase;        /* Save the heap base offset as the value to  */
                              /* be assigned to the stack pointer when the  */
                              /* TSR is activated.                          */

                              /* Terminate and stay resident:               */
    _dos_keep (0,ParaHeapBase + HEAPSIZE - _psp);

    return (ERROR);           /* '_dos_keep' function should not return;    */
                              /* therefore, something has gone terribly     */
                              /* wrong.  Return the general error code.     */
    } /* end TsrInstall */


int TsrRemove
    (void)
/*
    Removes the TSR from memory.  If successful, returns NOERROR (0);  if
    the TSR cannot be removed, it returns CANTREMOVE.
*/
    {
    if (OldRTInt9          != NewInt9  ||  /* First, must make sure that  */
        _dos_getvect (0x13) != NewInt13 ||  /* none of the TSR interrupts  */
        _dos_getvect (0x28) != NewInt28)    /* have been hooked by a       */
        return (CANTREMOVE);                /* subsequently loaded program.*/

    _dos_setvect (0x9,OldInt9);             /* Now restore the TSR         */
    _dos_setvect (0x13,OldInt13);           /* interrupts to their values  */
    _dos_setvect (0x28,OldInt28);           /* before TSR installation.    */

    _dos_setvect (UserInt,0);               /* Remove TSR signature from   */
                                            /* user interrupt.             */
    Unload = 1;                             /* Set flag so that 'Activate' */
                                            /* will free memory.           */
    return (NOERROR);
    } /* end TsrRemove */
```

Listing 2.1 *(cont'd)*

```
/*** PSP Functions. ********************************************************/


static unsigned far *PtrPSPTable [2]; /* Table of pointers to DOS PSP values*/
static int PSPCount;                  /* Number of pointers in PtrPSPTable. */


static void InitPSP (void)
/*
     This function saves the addresses of the locations in the MS-DOS segment
     where DOS stores the current PSP value.
*/
     {
     union REGS Reg;
     struct SREGS SReg;
     unsigned char far *PtrDos;    /* Points to locations in DOS segment.   */
     unsigned char far *PtrEndDos; /* Points to the end of DOS segment.     */
     unsigned far *PtrMCB;         /* Points to the location where the      */
                                   /* segment address of end of DOS is kept.*/

     PSPCount = 0;                 /* Initialize counter of valid entries   */
                                   /* in PtrPSPTable.                       */

                                   /* Assign PtrDos the address of the      */
                                   /* BEGINNING of the DOS segment.         */
     FP_SEG (PtrDos) = 0;
     FP_OFF (PtrDos) = FP_SEG (PtrInDos) << 4;
                                   /* Assign PtrEndDos the address of the   */
     Reg.h.ah = 0x52;              /* END of the DOS segment.               */
     int86x (0x21,&Reg,&Reg,&SReg);
     FP_SEG (PtrMCB) = SReg.es;
     FP_OFF (PtrMCB) = Reg.x.bx - 2;
     FP_SEG (PtrEndDos) = 0;
     FP_OFF (PtrEndDos) = *PtrMCB << 4;
                                   /* Obtain addresses where DOS stores the */
                                   /* PSP of the current process.           */
     while (PtrDos < PtrEndDos && PSPCount < 2)
          {
          if (*(unsigned far *)PtrDos ==  _psp)
               {
               Reg.h.ah = 0x50;
               Reg.x.bx = _psp + 1;
               int86 (0x21,&Reg,&Reg);
               if (*(unsigned far *)PtrDos == _psp + 1)
                    PtrPSPTable [PSPCount++] = (unsigned far *)PtrDos;
               Reg.h.ah = 0x50;
               Reg.x.bx = _psp;
               int86 (0x21,&Reg,&Reg);
               }
          ++PtrDos;
          }

     } /* end InitPSP */
```

Listing 2.1 *(cont'd)*

```
static unsigned GetPSP (void)

/*
     This function gets current PSP address from DOS.
*/
     {
     return *PtrPSPTable [0];

     } /* end GetPSP */


static void SetPSP (unsigned PSP)
/*
     This function sets DOS's record of PSP address.
*/
     {
     int i;

     for (i = 0; i < PSPCount; ++i)
          *PtrPSPTable [i] = PSP;

     } /* end SetPSP */
```

Listing 2.2 *Assembly language functions for converting a Microsoft C program into a TSR.*

```
;
;    File Name:  TSRA.ASM
;
;    This file contains the assembly language functions belonging to the TSR
;    module.  The C functions are defined in the file TSR.C (Figure 1).
;
;    Copyright (C), 1989, Michael J. Young.  All rights reserved.
;

.MODEL MEDIUM

.DATA

PUBLIC _Int13Flag                   ;Flag indicating int 13h active.
       _Int13Flag  DW  0
EXTRN  _OldInt13 : DWORD            ;Defined in RES.C.
EXTRN  _CSS      : WORD             ;Defined in RES.C.
EXTRN  _CSP      : WORD             ;Defined in RES.C.
       _OldSS      DW  ?            ;Stores old stack segment.
       _OldSP      DW  ?            ;Stores old stack pointer.


.CODE

EXTRN _Activate : FAR

PUBLIC _NewInt13
;
;    Prototype:
;         void interrupt far NewInt13
;              (void)
;    This is the new handler for interrupt 13h, the BIOS disk services.
;    This function maintains a flag of interrupt 13h invocations, Int13Flag.
;    Each time interrupt 13h is invoked, the flag is incremented, and each
;    time control returns from this interrupt, the flag is decremented.  A
;    zero value of Int13Flag indicates that interrupt 13h is not active.
;
_NewInt13 PROC

     push ds                        ;Temporarily load the C data segment
     push ax                        ;address into DS.
     mov  ax, DGROUP
     mov  ds, ax
     pop  ax

     inc  _Int13Flag                ;Increment the interrupt 10h active flag.

     pushf                          ;Simulate an interrupt to the old
     call _OldInt13                 ;handler.

     dec  _Int13Flag                ;Decrement the interrupt 10h active flag.

     pop  ds                        ;Restore the DS register.
     ret  2                         ;Return from interrupt, preserving the
                                    ;flags.
_NewInt13 ENDP
```

Listing 2.2 *(cont'd)*

```
PUBLIC _PreActivate
;
;    Prototype:
;         void far PreActivate
;              (void)
;    This is the first routine called to activate the TSR.  It performs
;    initial activation tasks and then calls the C routine Activate to
;    complete activation of the TSR.
;
_PreActivate PROC

     cli                              ;Disable interrupts.
     mov  _OldSS, ss                  ;Save the old SS and SP.
     mov  _OldSP, sp
     mov  ss, _CSS                    ;Switch to the C stack.
     mov  sp, _CSP
     sti                              ;Reenable interrupts.

     call _Activate                   ;Call C function to activate TSR.

     cli                              ;Disable interrupts.
     mov  ss, _OldSS                  ;Restore the old stack.
     mov  sp, _OldSP
     sti                              ;Reenable interrupts.

     ret                              ;Return control.

_PreActivate ENDP

END
```

Listing 2.3 **TSR.H:** *Header file you must include in your program to call the functions in the TSR module.*

```
/*
     File Name:  TSR.H

     This file contains the function prototypes and constant definitions for
     the TSR module.  You should include this file in any program that calls
     one or more of the TSR functions.

     Copyright (C), 1989, Michael J. Young.  All rights reserved.
*/

int TsrInDos (void);
int TsrInstall (void (*FPtr) (void),int HotKey, unsigned long Code);
int TsrRemove (void);

                              /* BIT MASKS FOR TSR HOTKEY:                  */
#define RIGHTSHIFT       0x0001
#define LEFTSHIFT        0x0002
#define CONTROL          0x0004
#define ALT              0x0008
#define SCROLLLOCK       0x1000
#define NUMLOCK          0x2000
#define CAPSLOCK         0x4000
#define INSERT           0x8000

                              /* ERROR RETURN CODES:                        */
#define NOERROR     0         /* No Error.                                  */
#define ERROR       1         /* General, undefined error.                  */
#define WRONGDOS    2         /* Version of DOS prior to 2.0.               */
#define INSTALLED   3         /* TSR has already been installed in memory.  */
#define NOINT       4         /* No free user interrupts are available.     */
#define CANTREMOVE  5         /* Can't remove TSR from memory.              */
```

case, the foreground application or other TSR probably calls your TSR (generally, a newly installed interrupt handler should call the previous handler — a process known as chaining). If your TSR code is being called by some other process, it is obviously not safe to remove it from memory. Also, restoring the original vector values would effectively uninstall any TSR that was installed after yours. Accordingly, if one of these three vectors has changed, *TsrRemove* returns the value *CANTREMOVE*, without removing the TSR.

2. *TsrRemove* restores the original handlers for interrupts *09h*, *13h*, and *28h* from the values that were saved by the installation routine (*Install*).

Listing 2.4 TSRDEMO.C: *A simple memory resident program.*

```
/*


     File Name:  TSRDEMO.C

     This program demonstrates the following TSR functions:
          TsrInstall
          TsrInDos
          TsrRemove

     Copyright (C), 1989, Michael J. Young.  All rights reserved.
*/
#include <BIOS.H>
#include <DOS.H>
#include <MALLOC.H>
#include <MEMORY.H>
#include <PROCESS.H>
#include <STDIO.H>

#include "TSR.H"

#define ULR 6                           /* Position of centered window.     */
#define ULC 20

static void Test (void);                /* The TSR demo routine.            */

                                        /* Supporting video routines:       */
void ScrClear (int StartRow,int StartCol,int StopRow,int StopCol,int Attr);
void ScrPutBox (int UR,int LC,int LR,int RC,int Attr,int Style);
void ScrPutS (char *String,unsigned char Attr,unsigned char Row,
              unsigned char Col);
void ScrRestoreBlock (char *BlockAddr, int Free);
char *ScrSaveBlock (int UR, int LC, int LR, int RC);

                                        /* Constants for video colors:      */
#define FG_I        0x08                /* Lightens foreground color.       */
#define FG_R        0x04                /* Red foreground.                  */
#define FG_G        0x02                /* Green foreground.                */


void main
     (void)
     {
     /* Install 'Test' as TSR function;  Ctrl-Left Shift is hotkey.         */
     switch (TsrInstall (Test,CONTROL | LEFTSHIFT,0x11111111))
          {
          case WRONGDOS:
               printf ("Cannot install TSR;  must have DOS version 2.0 "
                       "or higher.\n");
               exit (1);

          case INSTALLED:
               printf ("TSR already installed.\n");
               exit (1);

          case NOINT:
               printf ("Cannot install;  no free user interrupts.\n");
               exit (1);
```

Listing 2.4 *(cont'd)*

```
        case ERROR:                    /* General error case.          */
        default:
             printf ("Error installing TSR function.\n");
             exit (1);
        }

    } /* end main */


static void Test (void)
/*
    This routine is called when the hotkey is pressed.  It displays a
    window and indicates whether DOS is currently active.
*/
    {
    char *ScreenHandle;
    int Key;

    ScreenHandle = ScrSaveBlock (ULR,ULC,ULR+11,ULC+39);
    ScrClear (ULR,ULC,ULR+11,ULC+39,0x0f);
    ScrPutBox (ULR,ULC,ULR+11,ULC+39,FG_I | FG_R | FG_G,2);
    ScrPutS ("T S R   D E M O",FG_I | FG_R | FG_G,ULR+2,ULC+12);
    if (TsrInDos ())
         ScrPutS ("Now in DOS",FG_I | FG_R | FG_G,ULR+5,ULC+14);
    else
         ScrPutS ("NOT in DOS",FG_I | FG_R | FG_G,ULR+5,ULC+14);
    ScrPutS ("Press R to remove TSR from memory.",FG_I | FG_R | FG_G,
            ULR+9,ULC+3);
    ScrPutS ("Press any other key to continue.",FG_I | FG_R | FG_G,
            ULR+10,ULC+4);

    Key = (_bios_keybrd (_KEYBRD_READ) & 0x00ff);

    if (Key == 'r' || Key == 'R')
         if (TsrRemove ())
              {
              ScrClear (ULR+5,ULC+1,ULR+10,ULC+38,0x0f);
              ScrPutS ("Sorry, cannot remove TSR now.",FG_I | FG_R | FG_G,
                      ULR+5,ULC+6);
              ScrPutS ("Press any key to continue.",FG_I | FG_R | FG_G,
                      ULR+8,ULC+7);
              _bios_keybrd (_KEYBRD_READ);
              }

    ScrRestoreBlock (ScreenHandle,1);

    } /* end Test */
```

Listing 2.4 *(cont'd)*

```
/*** Supporting video functions. ***********************************************/

void ScrClear
     (int StartRow,
     int StartCol,
     int StopRow,
     int StopCol,
     int Attr)
/*
     This function clears the rectangular section of the screen specified
     by the four parameters.  It fills the blank area with the video display
     attribute given by 'Attr'.
*/
     {
     union REGS Reg;

     Reg.h.bh = (unsigned char) Attr;     /* BH specified video attribute.*/
     Reg.h.ch = (unsigned char) StartRow; /* CH specifies start row.      */
     Reg.h.cl = (unsigned char) StartCol; /* CL specifies start column.   */
     Reg.h.dh = (unsigned char) StopRow;  /* DH specifies stop row.       */
     Reg.h.dl = (unsigned char) StopCol;  /* DL specifies stop column.    */
     Reg.x.ax = 0x0600;       /* BIOS scroll page up function.            */
     int86 (0x10,&Reg,&Reg);  /* Invoke BIOS video services.              */

     } /* end ScrClear */


void ScrPutBox
     (int UR,
     int LC,
     int LR,
     int RC,
     int Attr,
     int Style)
/*
     This function displays a box on the screen starting at row and column
     'UR' and 'LC' and ending at row and column 'LR' and 'RC'.  It uses
     the video attribute 'Attr'.  'Style' selects the line style, and may
     be a value between 0 and 3.
*/
     {
     register int i;          /* Loop counter.                            */
     int Delta1, Delta2;      /* Bytes between lines.                     */
                              /* Store the box characters for each style. */
     static unsigned char ulc [] = {218,201,213,214};
     static unsigned char urc [] = {191,187,184,183};
     static unsigned char llc [] = {192,200,212,211};
     static unsigned char lrc [] = {217,188,190,189};
     static unsigned char hl  [] = {196,205,205,196};
     static unsigned char vl  [] = {179,186,179,186};
     unsigned far *Video;         /* Far pointer to video memory.         */
     unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;

     Delta1 = (RC - LC);          /* Characters between 2 vertical lines.*/
     Delta2 = 80 - Delta1;        /* Characters between right vertical   */
                                  /* line and left vertical line of next */
                                  /* row.                                */
```

Listing 2.4 *(cont'd)*

```
     /* Initialize far pointer to video memory, upper left corner of box: */
     FP_SEG (Video) = (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
     FP_OFF (Video) = UR * 160 + LC * 2;
     *Video++ = (Attr << 8) | ulc [Style];   /* Draw upper left corner.   */
     for (i = 1; i <= RC - LC - 1; ++i)      /* Draw top horizontal line. */
          *Video++ = (Attr << 8) | hl [Style];
     *Video = (Attr << 8) | urc [Style];     /* Draw upper right corner.  */
     Video += Delta2;
     for (i = 1; i <= LR - UR - 1; ++i)      /* Draw both vertical lines. */
          {
          *Video = (Attr << 8) | vl [Style]; /* Left vertical line.       */
          Video += Delta1;
          *Video = (Attr << 8) | vl [Style]; /* Right vertical line.      */
          Video += Delta2;
          }
     *Video++ = (Attr << 8) | llc [Style];   /* Draw lower left corner.   */
     for (i = 1; i <= RC - LC - 1; ++i)      /* Draw bottom horiz. line.  */
          *Video++ = (Attr << 8) | hl [Style];
     *Video = (Attr << 8) | lrc [Style];     /* Draw lower right corner.  */

     } /* end ScrPutBox */


void ScrPutS
     (char *String,
     unsigned char Attr,
     unsigned char Row,
     unsigned char Col)
/*
     This function displays null terminated 'String' with video attribute
     'Attr', beginning at the position given by 'Row' and 'Col'.
*/
     {
     register unsigned char A;          /* Fast register storage for Attr.*/
     unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;
     unsigned char far *Video;          /* Far pointer to video memory.   */

     A = Attr;                          /* Store attribute in register.   */

     /* Calculate far pointer to video memory:                            */
     FP_SEG (Video) =  (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
     FP_OFF (Video) =  Row * 160 + Col * 2;
     while (*String)          /* Write characters from string until null. */
          {
          *Video++ = *String++;
          *Video++ = A;
          }

     } /* end ScrPutS */


typedef struct
     {
     int UR;
     int LR;
     int LC;
     int BytesPerRow;
     }
HEADER;
```

Listing 2.4 *(cont'd)*

```
void ScrRestoreBlock
     (char *BlockAddr,
     int Free)
/*
     This function restores to the screen a block of data saved previously
     through the function 'ScrSaveBlock'.  'BlockAddr' is the address of
     the block returned by 'ScrSaveBlock'.  If you assign the parameter
     'Free' a nonzero value, the heap memory used to store the screen data
     will be released, and you may NOT call 'ScrRestoreBlock' to restore the
     data again.  If you assign 'Free' 0, the heap memory is not released
     and you may restore the data to the screen again.
*/
     {
     register int Row;
     HEADER *PtrHeader;
     unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;
     int VideoSeg, VideoOff;
     char far *PtrBlock;

     /* Set a pointer to the header at the beginning of the block in heap:*/
     PtrHeader = (HEADER *)BlockAddr;

     /* Calculate target and source addresses:                          */
     VideoSeg = (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
     VideoOff = PtrHeader->UR * 160 + PtrHeader->LC * 2;
     PtrBlock = (char far *)(BlockAddr + sizeof (HEADER));

     /* Write each row of data to the Screen:                           */
     for (Row = PtrHeader->UR; Row <= PtrHeader->LR; ++Row)
          {
          movedata                 /* Copy a row.                       */
               (FP_SEG (PtrBlock), /* Source segment.                   */
               FP_OFF (PtrBlock),  /* Source offset.                    */
               VideoSeg,           /* Target segment.                   */
               VideoOff,           /* Target offset.                    */
               PtrHeader->BytesPerRow);      /* Bytes to copy.          */

          /* Increment source and target addresses to next row:         */
          PtrBlock += PtrHeader->BytesPerRow;
          VideoOff += 160;
          }

     /* Free the block if the flag is set:                              */
     if (Free)
          free (BlockAddr);

     return;

     } /* end ScrRestoreBlock */


char *ScrSaveBlock
     (int UR,
     int LC,
     int LR,
     int RC)
/*
     This function saves the rectangular block of screen text data specified
     through the first four parameters:
```

Listing 2.4 *(cont'd)*

```
        UR   Upper row (rows are numbered beginning with 0 at the top of
             the screen).
        LC   Left column (columns are numbered beginning with 0 at the
             left of the screen).
        LR   Lower row.
        RC   Right column

    If successful, it returns the address of the block of data.  If an error
    occurs, it returns the value NULL.  The function works only in a text
    mode.
*/
    {
    register int Row;
    int BytesPerRow;
    size_t BlockSize;
    char *BlockAddr;
    HEADER *PtrHeader;
    unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;
    int VideoSeg, VideoOff;
    char far *PtrBlock;

    /* Calculate the size of the block and allocate memory from the heap:*/
    BytesPerRow = (RC - LC + 1)  * 2;
    BlockSize = sizeof (HEADER) + BytesPerRow * (LR - UR + 1);
    BlockAddr = (char *)malloc (BlockSize);
    if (BlockAddr == NULL)
        return (NULL);

    /* Save information on the saved block in the header at the beginning*/
    /* of the memory allocation.                                         */
    PtrHeader = (HEADER *)BlockAddr;
    PtrHeader->UR = UR;
    PtrHeader->LR = LR;
    PtrHeader->LC = LC;
    PtrHeader->BytesPerRow = BytesPerRow;

    /* Calculate initial source and target addresses:                    */
    VideoSeg = (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
    VideoOff = UR * 160 + LC * 2;
    PtrBlock = (char far *)(BlockAddr + sizeof (HEADER));

    /* Write each row of screen data to the block in the heap:           */
    for (Row = UR; Row <= LR; ++Row)
        {
        movedata                /* Copy a row.                           */
            (VideoSeg,          /* Source segment.                       */
            VideoOff,           /* Source offset.                        */
            FP_SEG (PtrBlock),  /* Target segment.                       */
            FP_OFF (PtrBlock),  /* Target offset.                        */
            BytesPerRow);       /* Bytes to copy.                        */

        /* Increment source and target addresses for next row:           */
        PtrBlock += BytesPerRow;
        VideoOff += 160;
        }

    return (BlockAddr);      /* Return block address as a handle to the  */
                             /* saved data.                               */

    } /* end ScrSaveBlock */
```

Listing 2.5 **TSRDEMO.MAK:** *A MAKE file for preparing both TSR.C and TSRDEMO.C.*

```
TSRDEMO.OBJ : TSRDEMO.C TSR.H
     cl /c /FPa /W2 /Zp TSRDEMO.C

TSR.OBJ : TSR.C TSR.H
     cl /c /W2 /Zp TSR.C

TSRA.OBJ : TSRA.ASM
     masm /MX TSRA.ASM;

TSRDEMO.EXE : TSRDEMO.OBJ TSR.OBJ TSRA.OBJ
     link /NOI /NOD TSRDEMO+TSR+TSRA,,NUL,SLIBCER;
```

3. It removes the TSR signature from the user interrupt vector (by writing the value *0* to this vector).
4. It sets the flag *Unload* to *1*. When this flag is set, the function *Activate* later frees the memory blocks occupied by the TSR. *TsrRemove* does not perform this task itself, since the memory should be released immediately before exiting from the TSR. Note that a program normally occupies two memory blocks: the main block beginning at the start of the program segment prefix (*_psp*), and the block containing the program's copy of the DOS environment.

Accordingly, *Activate* must free both of these blocks (it uses the C library function *_dos_freemem*).

Michael J. Young is the author of six advanced programming books on MS-DOS and OS/2. He is also the developer of several programmer toolkits for these operating systems. If you have any questions or comments, or would like to receive a catalog of his products, you can contact him at the following address:

Michael J. Young, Young Software Engineering, 20 Sunnyside Avenue, Suite A Mill Valley, CA 94941 (415) 383-5354.

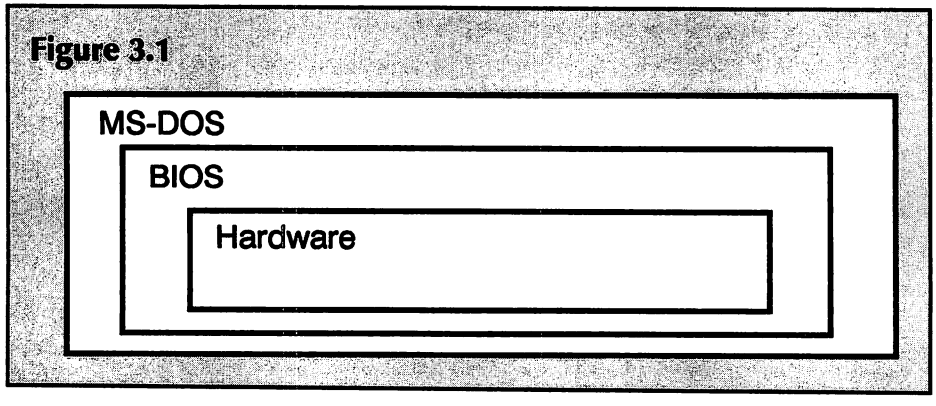
Event Timing On MS-DOS PCs

Phyllis K. Lang

An MS-DOS PC system may be viewed as a layered system (Figure 3.1), consisting of MS-DOS at the top, the BIOS underneath and the PC hardware on the bottom. Each of these three layers harbors its own solution to event timing. The precision required by your application determines which of the three timing alternatives should be used.

The time of day service provided by MS-DOS may be used for timing events with duration on the order of seconds or more. This service is available through MS-DOS interrupt *21H*, function *2CH*. The precision obtained is approximately 55 ms, but the time returned by MS-DOS is formatted in hours, minutes, seconds and hundredths of seconds. This format requires special consideration at midnight.

The same 55 ms accuracy is available to applications via a BIOS interrupt whose service may be customized by manipulating the MS-DOS interrupt



vector table and writing a corresponding interrupt service routine. It is imperative that interrupt service routines be written efficiently so that system performance is not degraded. Furthermore, the system environment must be maintained by the interrupt service.

The MS-DOS timing chain includes a free BIOS interrupt (*1CH*) for applications. This interrupt is issued every 55 ms by the system time tick interrupt (*int 8*). The standard MS-DOS interrupt *1C* handler does essentially nothing (an *iret* instruction). The standard handler may be replaced by a custom handler, which might, for example, decrement a globally defined counter that may be interrogated by other procedures. The custom handler should be installed by using MS-DOS interrupt *21* functions *25* and *35* to change the interrupt vector. This is the safe method of changing interrupt vectors and should be used whenever possible.

Since interrupt *1C* is a free vector for applications, beware of resident programs that use it. At a minimum, you should restore the original vector at program exit. To minimize conflicts with other applications, the new interrupt *1C* handler should chain to any existing handler as in Listing 3.1. This listing illustrates how to write the interrupt handler, how to install it, and how to use the new timing service. Both Listing 3.1 and Listing 3.2 are written in Turbo Pascal 5.0 which provides the necessary interrupt service support. The stack is adjusted, the system flags are saved and the necessary *iret* instruction is generated by the *interrupt* keyword in the procedure header. Inline assembly language is used in the procedure *sti* to enable interrupts and procedure *intchain* to daisy chain interrupts.

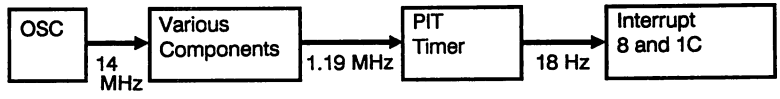
Listing 3.1

```
{ ------------------------------------------------------------------------
  I1CEX inserts a user defined timer tick interrupt service procedure
  into the MS-DOS timekeeping chain.

  Author: pkl
  ------------------------------------------------------------------------ }

{$F+}

  program i1cexample;

  uses
    dos, crt;

  var
    countdowntimer : integer;
    int1cvec : pointer;

{ ------------------------------------------------------------------------
  STI enables interrupts.
  ------------------------------------------------------------------------ }

  procedure sti; inline($fa);

{ ------------------------------------------------------------------------
  INTCHAIN chains to an interrupt.  The stack is prepared so that the DOS
  interrupt handler will receive it in proper condition.
  ------------------------------------------------------------------------ }

  procedure intchain(ivec : pointer);

  begin
    inline($9c/                          { pushf }
           $ff/$5e/$06);                 { call dword ptr [bp+6] }
  end;

{ ------------------------------------------------------------------------
  I1CHNDLR is an interrupt handler for the user timer tick (1C) interrupt.

  The BIOS time tick interrupt (int 8) service routine (TIMER_INT) updates
  the time of day clock, checks the diskette motor counter for timeout, and
  issues a user time tick interrupt (int 1C). This routine decrements a count-
  down timer maintained in external memory where it may be set and polled by
  functions within the program.
  ------------------------------------------------------------------------ }

  procedure i1chndlr(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

  begin

    sti;                { enable interrupts }

    countdowntimer := countdowntimer - 1;
    intchain(int1cvec);                 { chain to the preexisting handler }
  end;
```

Listing 3.1 *(cont'd)*

```
{ ------------------------------------------------------------------------
  Insert a 54.9 ms countdown timer into the MS-DOS timing chain that
  is accessible to the application.
  ------------------------------------------------------------------------ }


  begin

    GetIntVec($1c, int1cvec);       { Save present vector }
    SetIntVec($1c, @i1chndlr);      { Install new one }

{ Example: delay for 200 ms. }

    countdowntimer := 200 div 55;
    while countdowntimer > 0 do;

{ Example: wait for a keyboard stroke for 2 seconds. }

    countdowntimer := 2000 div 55;
    write('Press any key>');
    while (countdowntimer > 0) and (not KeyPressed) do;

    if (countdowntimer <= 0)    then    { Time out on keyboard }
      writeln(' ** Keyboard timed out **')
    else
      writeln(' ** Ok **');

    SetIntVec($1c, int1cvec);       { Restore the interrupt 1C vector. }

  end.
```

Figure 3.2



Some applications may demand more accuracy than *int 1C*'s 55 ms resolution. The MS-DOS PC time chain is designed to provide 55 ms accuracy; therefore, to increase this precision you must change the operation of both MS-DOS and the PC hardware.

Figure 3.2 shows the PC timing chain. A system clock oscillates at 14.318 MHz. This pulse rate is divided by various components to 1.19318 MHz, which becomes the input frequency to the i8253 programmable interval timer (PIT). The PIT contains three independently programmable timers, each of which may be loaded with a different 16-bit down count value. Each timer also may be programmed to operate in one of six modes. The mode selection dictates how the timer behaves upon decrementing to zero. MS-DOS programs *timer 0* to output a square wave at the terminal down count of 65536. The output of *timer 0* is tied directly to an i8259 programmable interrupt controller (PIC). Thus *timer 0* produces a hardware interrupt every 54.9 ms. (1.19318 MHz / 65536 = 18.17874 Hz; 1 / 18.17874 Hz = 54.9 ms). The PIC is programmed to map the time interrupt to *int 8* in the interrupt vector table. This is the "hook" into the timer interrupt system.

Timing resolution may be improved by increasing the frequency of timer interrupts by loading *timer 0* with a smaller down count value. However, the MS-DOS *int 8* handler controls many system resources which rely upon the 54.9 ms heartbeat. If the *int 8* interrupt frequency is increased, other system functions are serviced more frequently as well. The challenge is to ensure that MS-DOS functions are serviced at the normal rate of 54.9 ms while the timer interrupt occurs more frequently.

The program in Listing 3.2 accomplishes this by saving the MS-DOS *int 8* vector, replacing it with a vector to the new handler and chaining to the normal MS-DOS *int 8* handler at the proper time. The interrupt rate is increased by an integral factor so that the MS-DOS service may be called at a regular frequency. *Timer 0* is programmed in *i8swapin* to operate at a

frequency of sixteen times the normal rate, or one interrupt every 3.43 ms. The new handler, *i08hndlr* chains to the MS-DOS *int 8* handler every sixteenth call.

A final issue involves installing the new timing scheme. PIT *timer 0* must be reprogrammed to interrupt at the new rate on an even boundary of the 54.9 ms interrupt; otherwise, some time in the system is lost and the system clock will show it. Likewise, the old rate must be restored in a similar fashion. Two additional *int 8* interrupt handlers synchronize the changes. *i8swapin()* in Listing 3.2 is called on the 54.9 ms interrupt. It programs PIT *timer 0* to operate at the increased frequency and installs the new interrupt handler, *i08hndlr()*. Before exiting the program, *i8swapout()* must be called to restore the timer interrupt vector. MS-DOS restores some interrupt vectors itself at program exit; the timer interrupt vector is *not* among them, so it must be restored by the application. *i8swapout()* is called on the 3.43 ms interrupt. At the boundary of the normal 54.9 ms interrupt, it programs PIT *timer 0* to operate at the normal frequency and restores the MS-DOS *int 8* vector. Restoring on the interrupt boundary assures that no system time is lost.

The interrupt frequency can be altered by changing *int8_passcount_reset* in Listing 3.2. Realize, though, that there is an upper limit to the interrupt frequency. The service of an interrupt occupies some of the execution time available before the next interrupt occurs. If the interrupt is programmed to occur every 100 μs and the service requires 100 μs to execute, no time remains for the application: the system locks up! Of course the processor clock speed also must be considered in determining the upper limit of the interrupt frequency. A 4.77 MHz i8086 operates at about 210 ns per cycle. Assuming the average i8086 instruction requires seven cycles, one instruction executes in 1.47 μs. Thus, the 3.43 ms interrupt frequency described above allows about 2300 instructions to execute between interrupts — not enough time for a significant amount of overhead. In the same interval a 16 MHz i80386 executes about 7800 instructions.

Summary

Event timing may be accomplished with MS-DOS time of day service, BIOS interrupt *1C*, or reprogramming PC hardware. A problem may arise in using the BIOS interrupt if a terminate and stay resident program tries to use that vector after it has been changed. Reprogramming the hardware may be dangerous if the system hardware differs from that described or if the application does not restore interrupt vectors properly. In addition, there may be several exit routes in a program: an application defined route, *Ctrl-Break* and the critical error handler's abort option. Each of these possibilities should be addressed. If these problems are properly addressed, the timing resolution attainable with an MS-DOS PC can accommodate many demanding real-time applications.

Listing 3.2

```
{ ------------------------------------------------------------------------
  I8EX reprograms the i8253 PIT to interrupt at 16 times it normal rate
  of 54.9 ms.  The interrupt decrements a user accessible timer every
  3.43 ms.

  Author: pkl.
  ------------------------------------------------------------------------ }

{$F+}

  program i8ex;

  uses
    dos, crt;

   type
    itab = array [0..$ff] of pointer;

  var
    int08vec : pointer;          { Interrupt 08 save vector }
    int8_passcount : integer;       { Current interrupt downcounter }
    timer_count : integer;       { Application countdown clock timer }
    vectab : itab absolute $0:$0;   { 80x86 interrupt table }

  const
    isr_8259 : word = $0020;        { i8259 in-service reg addr }
    eoi_8259 : byte = $20;       { i8259 end-of-intrpt instr }

    cwr_8253 : byte = $43;       { control word reg addr }
    ctr0_8253 : byte = $40;      { timer 0 counter addr }
    set_ctr0_8253 : byte = $36;     { Binary mode 3 ctr 0 }

{ Define speedup factor and the corresponding i8253 counter values.
  The normal timer 0 interrupt occurs every 54.9 ms. The frequency of that
  interrupt is increased by a factor of 16 for this application so
  interrupts occur every 3.43 ms.                                        }

    int8_passcount_reset : integer = 16;

{ The input clock frequency is 1.19318 MHz on the PC; the corresponding
  downcount values for 54.9ms and 3.43ms follow.                         }

    int8_newct_msb : byte = $10;
    int8_newct_lsb : byte = $00;
    int8_dosct_msb : byte = $00;
    int8_dosct_lsb : byte = $00;

{ ------------------------------------------------------------------------
  STI enables interrupts.
  ------------------------------------------------------------------------ }

  procedure sti; inline($fa);
```

Listing 3.2 *(cont'd)*

```
{ ------------------------------------------------------------------------
  INTCHAIN chains to an interrupt.  The stack is prepared so that the DOS
  interrupt handler will receive it in proper condition.
  ------------------------------------------------------------------------ }

  procedure intchain(ivec : pointer);

  begin
    inline($9c/                          { pushf }
           $ff/$5e/$06);                 { call dword ptr [bp+6] }
  end;

{ ------------------------------------------------------------------------
  I8HNDLR is an interrupt handler for the DOS time tick interrupt (int 8).
  This handler processes more frequent hardware clock interrupts while
  maintaining the normal 54.9 ms system clock updating.
  ------------------------------------------------------------------------ }

  procedure i08hndlr(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

  begin

    sti;                  { Reenable interupts }
    timer_count := timer_count - 1; { Decrement users timer }

{ If it is time to do the system time chores, reset the pass counter and
  chain to the normal DOS interrupt vector.
  The normal DOS handler does an sti and reenables the i8259.            }

    int8_passcount := int8_passcount - 1;
    if (int8_passcount = 0) then
    begin
      int8_passcount := int8_passcount_reset;
      intchain(int08vec);
    end
    else
      port[isr_8259] := eoi_8259;
  end;
```

Listing 3.2 *(cont'd)*

```
{ ------------------------------------------------------------------------
  I8SWAPIN is an interrupt handler for the DOS time tick interrupt (int 8).
  This handler swaps in a new handler for more frequent hardware interrupts
  while maintaining the normal 54.9 ms system clock updating. The swap
  occurs at interrupt time so that no system clock ticks are lost.
  ------------------------------------------------------------------------ }

  procedure i8swapin(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

{ i8259 interrupts are disabled upon entry; re-enable interrupts.
  Configure the i8253 to run in mode 3 (square wave) with a new 16 bit
  binary countdown value.                                              }

  begin

    sti;                { Enable interrupts }

    port[cwr_8253] := set_ctr0_8253;    { Timer output is square wave }
    port[ctr0_8253] := int8_newct_lsb;  { New down count value }
    port[ctr0_8253] := int8_newct_msb;

{ Initialize pass counter for int 8 handler.                           }

    int8_passcount := int8_passcount_reset;

{ Swap in the new time tick interrupt handler vector. DOS interrupts
  shouldn't be used within an interrupt handler.                       }

    vectab[8] := @i08hndlr;

{ Go perform DOS time services. DOS does the eoi for the i8259.        }

    intchain(int08vec);

  end;

{ ------------------------------------------------------------------------
  I8SWAPOUT is an interrupt handler for the DOS time tick interrupt
  (int 8).  This handler swaps out the new handler and restores the DOS
  handler.  The swap occurs at interrupt time so that no system clock
  ticks are lost.
  ------------------------------------------------------------------------ }

  procedure i8swapout(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

{ i8259 interrupts are disabled at entry; re-enable.  Restore the normal
  DOS operation of i8253 timer 0: mode 3 with a binary downcounter.    }

  begin

    sti;                { Enable interrupts }

    timer_count := timer_count - 1; { Decrement user timer }
    int8_passcount := int8_passcount - 1;
```

Listing 3.2 *(cont'd)*

```
    if (int8_passcount = 0) then
    begin                { Time to change vectors }

      port[cwr_8253] := set_ctr0_8253;  { Binary downcount mode }
      port[ctr0_8253] := int8_dosct_lsb;    { Timer expires in 54.9 ms }
      port[ctr0_8253] := int8_dosct_msb;
      vectab[8] := int08vec;          { Restore vector for interrupt 8 }
      intchain(int08vec);        { Do the DOS time service. }
    end
    else
      port[isr_8259] := eoi_8259;   { End of interrupt for i8259. }

  end;

  var
    reply : byte;

  begin
    GetIntVec($08, int08vec);        { Save present int 8 vector }
    SetIntVec($08, @i8swapin);       { Install vector to swap in new
                      timing scheme }

{  Example: wait 300 ms for pin 2 on LPT1: to go high (1).                }

    timer_count := 300 * int8_passcount_reset div 55;
    reply := $00;
    while (timer_count > 0) and (reply = $00) do
    begin
      reply := port[$03BC];      { Read the port }
      reply := reply and $01;        { Mask the desired bit }
    end;

    if (timer_count <= 0) then
       writeln('Pin 2 on LPT1: did not go high.');

    timer_count := 2000 * int8_passcount_reset div 55;
    write('Press any key>');
    while (timer_count > 0) and (not KeyPressed) do;

    if (timer_count <= 0)   then    { Time out on keyboard }
      writeln(' ** Keyboard timed out **')
    else
      writeln(' ** Ok **');

{ Return the system to its normal timekeeping.  Install a vector to the
  handler that reprograms the PIT and restores the DOS handler at the next
  interrupt.                                                              }

    SetIntVec($08, @i8swapout);
    writeln('Give vector time to swap>');
    while (not KeyPressed) do;
  end.
```

Writing MS-DOS Exception Handlers

Robert B. Stout

"Exceptions" are catastrophic failures which usually cause an executing program to abort to the operating system. "Errors", on the other hand, usually include less disastrous unexpected events. A key difference between errors and exceptions is that errors are defined by the executing program whereas exceptions are defined at the system level. For example, telling an application to open a file in a directory which doesn't exist creates an *error*, since only the application presupposed the directory to exist in the first place. On the other hand, when a disk is unreadable for some reason, it triggers an *exception*, since the failure violates the operating system's expectations.

Exceptions in MS-DOS programs come in basically three flavors; those initiated by the user, those resulting from a coprocessor error, and those resulting from some other sort of system error. Since most modern high-level language (HLL) compilers include some mechanism for trapping coprocessor errors, this chapter will deal with how to handle the other two types of exceptions.

The code presented here is not necessarily simple "plug and play" code to be dropped into your next project. This was a conscious decision. To be universally useful, the code presented here makes no assumptions whether it is to be embedded in an assembly or HLL application. Since different HLLs use different conventions for passing arguments, these portions of the code may change from one language to another. You must refer to MASM or to your HLL compiler's reference section on interfacing assembly modules to customize this code for your own use. All code presented here uses Microsoft

MASM v5.1 which allows for simplified memory model and language conventions. In many cases, the code may be used as-is by simply defining the macros *memodel* and *lang* on the command line — but be safe and double-check everything!

User-Generated Exceptions

MS-DOS users can generate exceptions by means of two separate mechanisms: *Ctrl-C* and *Ctrl-Break*.

When a program is executed, the *Ctrl-C* interrupt *23h* is set up to point to a default error handler, which is called whenever a *Ctrl-C* character is detected in the keyboard input buffer. Unfortunately, the operating system checks the keyboard buffer only occasionally, most usually when actually processing input or output calls. Another problem with interrupt *23h* is that when a child process is spawned, it will usually inherit the parent program's *Ctrl-C* handler, which is almost certainly not what is desired. When a program terminates in any way, MS-DOS resets the interrupt *23h* vector to its default state.

The *Ctrl-Break* interrupt *1Bh* works somewhat differently, though usually in concert with interrupt *23h*. Whenever the ROM BIOS detects the *Ctrl-Break* key combination, the keyboard buffer is flushed and a *Ctrl-C* key combination is stuffed in place of the previous contents. This *Ctrl-C* will later be detected and processed by interrupt *23h*. *Ctrl-Break* processing therefore may offer more immediate response than *Ctrl-C* processing if the default action is overridden.

Two major caveats are in order, however:

Unlike interrupt *23h*, MS-DOS does not restore the default state of interrupt *1Bh* upon program termination.

While *Ctrl-C* processing is standardized among the various machines utilizing both MS-DOS and PC-DOS, *Ctrl-Break* processing is much less standardized.

In the following discussion of *Ctrl-C* exception handling, be aware that the same techniques could be used for *Ctrl-Break* processing provided that the application supplies some totally reliable way to assure that interrupt *1Bh*

is restored prior to program termination. In ANSI C, the *atexit()* function could be used for this as could similar facilities in other HLLs.

Handling Ctrl-C Exceptions

DOS's default *Ctrl-C* handler is triggered whenever the *Ctrl-C* character is detected in the input buffer. DOS responds by closing all files which were opened using handle functions and terminating the program. There are therefore several things DOS doesn't do that your application may require:

The default *Ctrl-C* handler has no knowledge of any HLL's buffered file I/O. Where a HLL may provide buffered file I/O (e.g., C program files opened with calls to *fopen()*) portions of files which your application had assumed written to disk may in fact be left orphaned in a buffer.

The default *Ctrl-C* handler doesn't attempt to close files opened using any of the FCB functions. Even though it has declared FCB functions obsolete, Microsoft continues to use them in critical elements of DOS such as *COMMAND.COM*. These calls remain the only mechanism for manipulating volume lables and deleting files using wildcard specifications.

DOS's normal program termination sequence only restores the interrupt *23h* and *24h* handlers. Any other intercepted interrupts which your application uses will be left pointing to free RAM once the application has been terminated using the default *Ctrl-C* handler.

Obviously, purely procedural issues such as updating database index files are also beyond the scope of DOS's default *Ctrl-C* processing.

The simplest *Ctrl-C* handler is an IRET. Simply stuff a pointer to an IRET into the interrupt *23h* vector and the operating system will ignore *Ctrl-C* — handy in programs which only want to check for *Ctrl-C* or *Ctrl-Break* at certain times. If your application explicitly checks for Ctrl-C inputs (which are, after all, simply another keystroke), you don't need DOS doing it for you.

Listing 4.1 demonstrates simple interrupt *23h* interception installation and de-installation routines used to disable DOS *Ctrl-C* handling. In this handler, whenever a *Ctrl-C* interrupt is received, a global variable *ccrecvd* is set to *-1*. In this way, your application has the option of checking the keyboard buffer for the *Ctrl-C* character or just testing the value of this variable at regular intervals. Note that this variable may change asynchronously with regard to the rest of your application. (In C, *ccrecvd* should be declared *volatile*.)

The next option is to explicitly perform your own Ctrl-C exception processing. Listing 4.2 demonstrates the code to install and de-install your own customized exception handler. Note that the code is written to accept the address of a function specified with an explicit segment and offset. This code is virtually identical to that in Listing 4.1 and is actually slightly simpler since the *ccrecvd* variable isn't needed and the actual interrupt service routine (ISR) is external to the code.

Listing 4.1

```
;  Install a do-nothing Interrupt 23 (Ctrl-C exception) handler

%       .MODEL  memodel,lang            ;Add model and language support via
                                        ;command line macros, e.g.
                                        ;MASM /Dmemodel=LARGE /Dlang=C

        .DATA?
        PUBLIC  ccrecvd
_origvec        dd      ?
ccrecvd         dw      ?

        .CODE

;
;  This is our actual ISR
;
myint23:
        mov     ccrecvd,-1
        iret

;
;  Call this to install  our ISR
;
dis23   PROC    USES AX BX DS ES
        mov     ax,3523h                ;get old vector...
        int     21h
        mov     word PTR _origvec,bx
        mov     word PTR _origvec+2,es  ;...and save it
        push    cs                      ;get myint23 segment in DS
        pop     ds
        mov     dx, OFFSET myint23      ;install myint23 in int 23h
        mov     ax,2523h
        int     21h
        mov     ccrecvd,0               ;clear the interrupt received flag
        ret
dis23   ENDP

;
;  Call this to uninstall our ISR
;
redo23  PROC    USES AX BX DS
        mov     dx, word PTR _origvec   ;restore original vector
        mov     ds, word PTR _origvec+2
        mov     ax,2523h
        int     21h
        ret
redo23  ENDP

        end
```

Listing 4.2

```
;  Install a custom Interrupt 23 (Ctrl-C exception) handler


%       .MODEL  memodel,lang            ;Add model and language support via
                                        ;command line macros, e.g.
                                        ;MASM /Dmemodel=LARGE /Dlang=C

        .DATA?
_origvec        dd      ?
_newvec         dd      ?

        .CODE

;
;  This is our actual ISR
;
myint23:
        call    far PTR _newvec         ;call our handler
        iret

;
;  Call this to install  our ISR
;
ins23   PROC    USES AX BX DS ES, segm:WORD, offs:WORD
        mov     ax,3523h                ;get old vector...
        int     21h
        mov     word PTR _origvec,bx
        mov     word PTR _origvec+2,es  ;...and save it
        mov     ax,offs                 ;load handler offset...
        mov     word PTR _newvec,ax
        mov     ax,segm                 ; & segment into _newvec
        mov     word PTR _newvec+2,ax
        push    cs                      ;get myint23 segment in DS
        pop     ds
        mov     dx, OFFSET myint23      ;install myint23 in int 23h
        mov     ax,2523h
        int     21h
        ret
ins23   ENDP

;
;  Call this to uninstall our ISR
;
redo23  PROC    USES AX BX DS
        mov     dx, word PTR _origvec   ;restore original vector
        mov     ds, word PTR _origvec+2
        mov     ax,2523h
        int     21h
        ret
redo23  ENDP

        end
```

System-Generated Exceptions

DOS's infamous "Abort, Retry..." message is the signal to the user that it was unable to successfully complete an I/O operation and has called its own internal exception handler for system failures, usually referred to as the "Critical Error Handler". Your code can intercept DOS critical error exceptions just as simply as it does *Ctrl-C* or *Ctrl-Break* exceptions (see Listing 4.3). The real challenge in writing critical error handlers is to correctly interpret the exception.

Listing 4.3

```
;  Install an Interrupt 24 (DOS critical error exception) handler


%       .MODEL  memodel,lang            ;Add model and language support via
                                        ;command line macros, e.g.
                                        ;MASM /Dmemodel=LARGE /Dlang=C

        .DATA?
        PUBLIC cedevdvr, cetype, ceerror, cereturn
_origvec        dd      ?
_newvec         dd      ?
cedevdvr        dd      ?
cetype          dw      ?
ceerror         dw      ?
cereturn        db      ?

        .CODE

;
;  This is our actual ISR
;
myint24:
        push    ds                      ;save registers which may be
        push    es                      ;required in case "Retry" is
        push    bx                      ;selected
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        mov     word PTR cedevdvr,si    ;save device driver header address
        mov     word PTR cedevdvr+2,bp
        mov     cetype,ax               ;save error type information
        mov     ceerror,di              ;save error code information
        call    far PTR _newvec         ;call our handler
        mov     al,cereturn             ;set up return code (abort, retry...)
```

The critical error handler requires more information in order to decide what action to take than does a *Ctrl-C* handler. All of this information is passed in the CPU's registers as shown in Table 4.1. To facilitate individual bit operations, Table 4.1 lists the mask you would logically and against the register data.

Listing 4.3 *(cont'd)*

```
        pop bp                      ;restore necessary registers
        pop di
        pop si
        pop     dx
        pop     cx
        pop     bx
        pop     es
        pop     ds
        iret

;
;  Call this to install  our ISR
;
ins24   PROC    USES AX BX DS ES, segm:WORD, offs:WORD
        mov     ax,3524h                ;get old vector...
        int     21h
        mov     word PTR _origvec,bx
        mov     word PTR _origvec+2,es  ;...and save it
        mov     ax,offs                 ;load handler offset...
        mov     word PTR _newvec,ax
        mov     ax,segm                 ; & segment into _newvec
        mov     word PTR _newvec+2,ax
        push    cs                      ;get myint24 segment in DS
        pop     ds
        mov     dx, OFFSET myint24      ;install myint24 in int 24h
        mov     ax,2524h                ;  into Interrupt 24h
        int     21h
ins24   ENDP

;
;  Call this to uninstall our ISR
;
redo24  PROC    USES AX BX DS
        mov     dx, word PTR _origvec   ;restore original vector
        mov     ds, word PTR _origvec+2
        mov     ax,2524h
        int     21h
        ret
redo24  ENDP

        end
```

Table 4.1 Critical error register data

BP:SI Under DOS 2.x or later, contains the segment:offset of the device driver which reported the exception. This is a pointer to the device driver's header. The most important entry in this header is the word located at BP:SI+4, the device driver attribute word. Important masks used to test this word are 8000h which indicates a character, rather than a block, device and 0800h which indicates support of removable media.

AH If the high bit of AH is zero, a disk error has occurred. When this happens, the remaining bits of AH are set as follows:

Bit	Mask	Meaning
0	01h	00h indicates a read operation
		01h indicates a write operation
1-2	06h	Indicates the affected disk area:
		00h if DOS
		02h if File Allocation Table (FAT)
		04h if root directory
		06h if files area
----- Bits 3-5 only used by DOS 3.1 and later -----		
3	08h	00h Fail response not allowed
		08h Fail response allowed
4	10h	00h Retry response not allowed
		10h Retry response allowed
5	20h	00h Ignore response not allowed
		20h Ignore response allowed

AL If AH masked with 80h equals zero indicating a disk error, AL contains the drive which failed. AL = 00h indicates drive A, AL = 01h indicates drive B, etc.

DI The DI register contains the following error codes:

Code	Meaning
0000h	Write-protected disk
0001h *	Unknown unit
0002h	Drive not ready
0003h *	Invalid command
0004h	Data (CRC) error
0005h *	Length of request structure invalid
0006h	Seek error
0007h *	Non-DOS disk
0008h	Sector not found
0009h *	Printer out of paper
000Ah	Write fault
000Bh *	Read fault
000Ch	General failure
000Dh **	Invalid disk change

The asterisks next to the odd-numbered codes indicate these codes are not available under DOS 1.x. In addition, code 000Dh is not available prior to DOS 3.x.

Only you, the programmer, can determine what your program requires of a critical error handler. Listing 4.4 demonstrates the skeleton of a critical error function which might be called from the handler in Listing 4.3. (Note that in this example as well as in Listing 4.3, your choice of a HLL could

Listing 4.4

```
; A sample Interrupt 24 (DOS critical error exception) handler

%       .MODEL  memodel,lang            ;Add model and language support via
                                        ;command line macros, e.g.
                                        ;MASM /Dmemodel=LARGE /Dlang=C

        EXTRN cedevdvr:dword, cetype:word, ceerror:word, cereturn:byte
        EXTRN read_err:far, write_err:far, bad_FAT:far
        EXTRN no_paper:far, fixup_ret:far, FAT_err:far

        ;NOTE:  All the above routines MUST set cereturn to:
        ;       0 - Ignore
        ;       1 - Retry
        ;       2 - Abort
        ;       3 - Fail (DOS 3.3 and later)

        .DATA?

        PUBLIC  osver, rmvbl, exerr, locus, class, suggest
osver   db ?
rmvbl   db ?
exerr   dw ?
locus   db ?
class   db ?
suggest db ?

        .CODE

;
;  This is called by myint24
;
mynew24 PROC    USES BX
        mov     ah,030h                 ;get DOS version number
        int     21
        or      al,al                   ;zero means DOS 1.x
        jnz     NotDOS1
        mov     al,1
NotDOS1:
        mov     osver,al                ;save DOS version
        mov     ax,cetype               ;get type of exception...
        mov     bx,ax                   ; & save it for later
        and     ax,80h                  ;disk error?
        jnz     NotDiskErr              ;no, continue
        cmp     al,1                    ;yes, DOS 1.x?
        jz      wrong_DOS               ;yes, can't check for removable media
```

Listing 4.4 *(cont'd)*

```
        mov     ah,-1                   ;no, assume removable media
        test    word PTR cedevdvr,0800h ;is the media removable?
        jz      removable
        xor     ah,ah                   ;no, flag fixed media
removable:
        mov     rmvbl,ah                ;save media type
        cmp     al,3                    ;DOS 3.0 or greater?
        jb      wrong_DOS               ;no, skip it
        push    ds                      ;yes, save regs
        push    es
        push    dx
        push    si
        push    di
        push    bp
        mov     ah,59h                  ;get extended error info
        int     21
        pop     bp                      ;restore regs
        pop     di
        pop     si
        pop     dx
        pop     es
        pop     ds
        mov     exerr,ax                ;save extended error code...
        mov     locus,ch                ; locus...
        mov     class,bh                ;  class...
        mov     suggest,bl              ;   & suggested action
wrong_DOS:
        mov     ax,bx                   ;get exception type
        and     ax,06h                  ;FAT problems?
        cmp     ax,02h
        jnz     ok_fat                  ;no, continue
        jmp     far PTR FAT_err         ;yes, handle it
ok_fat:
        mov     ax,bx                   ;get exception type
        and     ax,01h                  ;handle read and write separately
        jz      rd_err
        jmp     far PTR write_err
rd_err:
        jmp     far PTR read_err
NotDiskErr:
        test    word PTR cedevdvr,8000h ;non-disk block device?
        jnz     good_fat                ;no, continue
        jmp     far PTR bad_FAT         ;yes, assume bad FAT
good_fat:
        test    ceerror,0009h           ;printer out of paper?
        jnz     not_eop                 ;no, continue
        jmp     far PTR no_paper        ;yes, handle it
not_eop:
        call    far PTR fixup_ret       ;unknown error - handle loose ends...
        mov     al,2                    ; & Abort!
        mov     cereturn,al
        ret
mynew24 ENDP

        end
```

obviate the use of the global variables used to pass the register contents to your critical error handling function(s).) In this example, specific external routines are called to separately handle read and write disk errors, printer out-of-paper errors, and corrupted FAT errors reported for a disk (reported in *AH*) or character device (as indicated in the driver attribute word). In the case of an unrecognized error, a routine is called to perform necessary maintenance before the program is aborted. To assist in interpreting the information available to the critical error handler, this function also sets two global variables to indicate the DOS version number, whether the error occurred on a device with removable media, and the DOS extended error information, if any (see below).

In writing your specific critical error handler functions, there are several important restrictions to keep in mind. First, no DOS system services may be requested other than interrupt *21h* functions *01h-0Ch* (character I/O), *30h* (get DOS version number), and *59h* (get extended error information). For example, a function which mimics DOS's "Abort, Retry..." message must output text using interrupt *21h*, functions *02h*, *06h*, or *09h* and get the user's response using interrupt *21h*, functions *01h*, *06h*, *07h*, *08h*, *0Ah*, or *0Ch*. As noted in Table 4.1, all registers except *AL* must be preserved since DOS sets them up for processing retry returns prior to invoking the critical error interrupt.

Table 4.2

Critical error return codes

[AL]	Meaning	DOS 3.1+ default *
00h	Ignore	Fail ***
01h	Retry	Fail ***
02h	Abort	
03h **	Fail	Abort

* Response if corresponding control bit (AH bits 3-5, see Table 1) disallows this option.

** The Fail option is only available in DOS 3.3 or later.

*** Under DOS 3.1 and 3.2, this defaults to Abort since Fail is not supported.

Finally, the handler must return with an *IRET* instruction, passing a return code in *AL* to tell DOS what to do next. The available codes and their actions under various DOS versions are detailed in Table 4.2.

DOS Extended Error Information

Beginning with version 3.0, DOS provides an extremely powerful facility to aid those writing critical error handlers. Interrupt *21h*, function *59h* not only provides extremely detailed error reporting, but also suggests strategies for handling errors. One caution is in order, though — calling the DOS extended error function destroys all registers except *CS:IP* and *SS:SP*. In a critical error handler this usually isn't a problem since all registers must be saved anyway.

Function *59h* returns the *Locus*, or location of the error, in register CH. Table 4.3 gives the interpretation associated with the various locus values.

Table 4.3 Extended error locus

[CH]	Meaning
01h	Unknown location
02h	Block device, usually a disk error
03h	Network error
04h	Serial device, often a timeout from a character device
05h	Memory error, usually RAM

Table 4.4 Extended error classes

[BH]	Meaning
01h	Out of resource, e.g. storage space or I/O channels
02h	Temporary error, usually a network file or record lock
03h	Permission to access device not authorized
04h	Internal software error - system bugs
05h	Hardware failure - very serious
06h	System software failure, usually bad configuration files
07h	Application software error - something looks wrong to DOS
08h	File not found
09h	File or item of invalid or inappropriate type detected
0Ah	File interlocked by system
0Bh	Media failure, usually bad disks
0Ch	Already exists
0Dh	Unknown error

Table 4.5

Suggested action codes

[BL]	Meaning
01h	Retry a few times, then suggest Ignore or Abort
02h	Pause a few seconds then do as above
03h	Ask the user to re-enter input data
04h	Clean up as best possible then Abort in a hurry!
05h	Abort as soon as possible - don't try to clean up!
06h	Ignore the error as it's mostly informational
07h	Prompt the user to take action, e.g. change a floppy.

Function *59h* also returns the error *Class* or category in register *BH*. Table 4.4 interprets *Class* values.

DOS also returns a suggested action code (Table 4.5) in register BL. These values are extracted from the relevant device driver.

Finally, the actual extended error codes are returned in register AX. Extended error codes are divided into three general classes depending on whether they are compatible with DOS 2.x errors, DOS 3.x+ errors, or related to critical error handling (see Table 4.6).

The best way to plan your exception handling strategy is to begin with your design, making notes to yourself as you go and structuring your code to allow graceful exits along the way. A well-designed program that makes full use of DOS's exception handling features will exhibit the kind of reliability that distinguishes the professional product from an obviously amateur effort.

Table 4.6 Extended error codes

[AX]	Meaning
-----	Errors compatible with DOS 2.x -----
01h	Invalid function number
02h	File not found
03h	Path not found
04h	No file handles available
05h	Access denied
06h	Invalid file handle
07h	Memory allocation error - memory control blocks destroyed
08h	Insufficient memory
09h	Memory allocation error - invalid memory control block
0Ah	Invalid environment
0Bh	Invalid format
0Ch	Invalid access code
0Dh	Invalid data
0Eh	* Reserved *
0Fh	Invalid disk drive specification
10h	Attempt to remove the current directory
11h	Not the same device
12h	No more files
-----	Critical errors -----
13h	Attempt to write on write-protected disk
14h	Unknown unit
15h	Drive not ready
16h	Invalid command
17h	Data (CRC) error
18h	Length of request structure invalid
19h	Seek error
1Ah	Unknown media type (non-DOS disk)
1Bh	Sector not found
1Ch	Printer out of paper
1Dh	Write error
1Eh	Read error
1Fh	General failure
-----	DOS 3.x+ extended errors -----
20h	File sharing violation
21h	File lock violation
22h	Invalid disk change
23h	FCB unavailable
24h	Sharing buffer exceeded
25h	* Reserved *
26h	* Reserved *
27h	* Reserved *
28h	* Reserved *
29h	* Reserved *
2Ah	* Reserved *
2Bh	* Reserved *
2Ch	* Reserved *
2Dh	* Reserved *
2Eh	* Reserved *

Table 4.6 *(cont'd)*

2Fh	* Reserved *
30h	* Reserved *
31h	* Reserved *
32h	Unsupported network request
33h	Remote computer not listening
34h	Duplicate network name
35h	Network name not found
36h	Network busy
37h	Device no longer on network
38h	NetBIOS command limit exceeded
39h	Network hardware error
3Ah	Incorrect network response
3Bh	Unexpected network error
3Ch	Incompatible remote adapter
3Dh	Print queue full
3Eh	Print queue not full
3Fh	No room for print file
40h	Network name deleted
41h	Access denied
42h	Incorrect network device type
43h	Network name not found
44h	Network name limit exceeded
45h	NetBIOS session limit exceeded
46h	Temporary pause
47h	Network request not accepted
48h	Print or disk redirection paused
49h	* Reserved *
4Ah	* Reserved *
4Bh	* Reserved *
4Ch	* Reserved *
4Dh	* Reserved *
4Eh	* Reserved *
4Fh	* Reserved *
50h	File already exists
51h	* Reserved *
52h	Cannot create directory
53h	Failure on Interrupt 24h
54h	Out of structures
55h	Already assigned
56h	Invalid password
57h	Invalid parameter
58h	Network write fault

The EXEC Function

Ray Duncan

The MS-DOS *EXEC* function (Int *21H* Function *4BH*) allows a program (called the *parent*) to load any other program (called the *child*) from a storage device, execute it, and then regain control when the child program is finished.

A parent program can pass information to the child in a command line, in default file control blocks, and by means of a set of strings called the environment block (discussed later in this chapter). All files or devices that the parent opened using the handle file-management functions are duplicated in the newly created child task; that is, the child inherits all the active handles of the parent task. Any file operations on those handles by the child, such as seeks or file I/O, also affect the file pointers associated with the parent's handles.

MS-DOS suspends execution of the parent program until the child program terminates. When the child program finishes its work, it can pass an exit code back to the parent, indicating whether it encountered any errors. It can also, in turn, load other programs, and so on through many levels of control, until the system runs out of memory.

The MS-DOS command interpreter, *COMMAND.COM*, uses the *EXEC* function to run its external commands and other application programs. Many popular commercial programs, such as database managers and word proces-

sors, use *EXEC* to run other programs (spelling checkers, for example) or to load a second copy of *COMMAND.COM*, thereby allowing the user to list directories or copy and rename files without closing all the application files and stopping the main work in progress. *EXEC* can also be used to load program overlay segments, although this use is uncommon.

Making Memory Available

In order for a parent program to use the *EXEC* function to load a child program, sufficient unallocated memory must be available in the transient program area.

When the parent itself was loaded, MS-DOS allocated it a variable amount of memory, depending upon its original file type — *.COM* or *.EXE* — and any other information that was available to the loader. Because the operating system has no foolproof way of predicting how much memory any given program will require, it generally allocates far more memory to a program than is really necessary.

Therefore, a prospective parent program's first action should be to use Int *21H* Function *4AH* (Resize Memory Block) to release any excess memory allocation of its own to MS-DOS. In this case, the program should call Int *21H* Function *4AH* with the *ES* register pointing to the program segment prefix of the program releasing memory and the *BX* register containing the number of paragraphs of memory to retain for that program.

■ **WARNING** *A .COM program must move its stack to a safe area if it is reducing its memory allocation to less than 64 KB.*

Requesting The *EXEC* Function

To load and execute a child program, the parent must execute an Int *21H* with the registers set up as follows:

```
AH = 4BH
AL = 00H (subfunction to load child program)
DS:DX = segment:offset of pathname for child program
ES:BX = segment:offset of parameter block
```

The parameter block, in turn, contains addresses of other information needed by the *EXEC* function.

The Program Name

The name of the program to be run, which the calling program provides to the *EXEC* function, must be an unambiguous file specification (no wildcard characters) and must include an explicit *.COM* or *.EXE* extension. If the path and disk drive are not supplied in the program name, MS-DOS uses the current directory and default disk drive. (The sequential search for *.COM*, *.EXE*, and *.BAT* files in all the locations listed in the *PATH* variable is not a function of *EXEC*, but rather of the internal logic of *COMMAND.COM*.)

You cannot *EXEC* a batch file directly; instead, you must *EXEC* a copy of *COMMAND.COM* and pass the name of the batch file in the command tail, along with the */C* switch.

The Parameter Block

The parameter block contains the addresses of four data objects:

- The environment block
- The command tail
- Two default file control blocks

The space reserved in the parameter block for the address of the environment block is only two bytes and holds a segment address. The remaining three addresses are all double-word addresses; that is, they are four bytes, with the offset in the first two bytes and the segment address in the last two bytes.

The Environment Block

Each program that the *EXEC* function loads inherits a data structure called an *environment block* from its parent. The pointer to the segment of the block is at offset *002CH* in the PSP. The environment block holds certain information used by the system's command interpreter (usually *COMMAND.COM*) and may also hold information to be used by transient programs. It has no effect on the operation of the operating system proper.

If the environment-block pointer in the *EXEC* parameter block contains zero, the child program acquires a copy of the parent program's environment

block. Alternatively, the parent program can provide a segment pointer to a different or expanded environment. The maximum size of the environment block is 32K, so very large chunks of information can be passed between programs by this mechanism.

The environment block for any given program is static, implying that if more than one generation of child programs is resident in RAM, each one will have a distinct and separate copy of the environment block. Furthermore, the environment block for a program that terminates and stays resident is not updated by subsequent *PATH* and *SET* commands.

You will find more details about the environment block later in this chapter.

The Command Tail

MS-DOS copies the command tail into the child program's PSP at offset *0080H*. The information takes the form of a count byte, followed by a string of ASCII characters, terminated by a carriage return; the carriage return is not included in the count.

The command tail can include filenames, switches, or other parameters. From the child program's point of view, the command tail should provide the same information that would be present if the program had been run by a direct user command at the MS-DOS prompt. *EXEC* ignores any I/O-redirection parameters placed in the command tail; the parent program must provide for redirection of the standard devices *before* the *EXEC* call is made.

The Default File Control Blocks

MS-DOS copies the two default file control blocks pointed to by the *EXEC* parameter block into the child program's PSP at offsets *005CH* and *006CH*. To emulate the function of *COMMAND.COM* from the child program's point of view, the parent program should use Int *21H* Function *29H* (the system parse-filename service) to parse the first two parameters of the command tail into the default file control blocks before invoking the *EXEC* function.

File control blocks are not much use under MS-DOS v2 and v3, because they do not support the hierarchical file structure, but some application

programs do inspect them as a quick way to get at the first two switches or other parameters in the command tail.

Returning From The *EXEC* Function

In MS-DOS v2, the *EXEC* function destroys the contents of all registers except the code segment (*CS*) and instruction pointer (*IP*). Therefore, *before* making the *EXEC* call, the parent program must push the contents of any other registers that are important onto the stack and then save the stack segment (*SS*) and stack pointer (*SP*) registers in variables. Upon return from a successful *EXEC* call (that is, the child program has finished executing), the parent program should reload *SS* and *SP* from the variables where they were saved and then pop the other saved registers off the stack. In MS-DOS v3.0 and later, the stack and other registers are preserved across the *EXEC* call in the usual fashion.

Finally, the parent can use Int *21H* Function *4DH* to obtain the termination type and return code of the child program.

The *EXEC* function will fail under the following conditions:

- Not enough unallocated memory is available to load and execute the requested program file.
- The requested program can't be found on the disk.
- The transient portion of *COMMAND.COM* in highest RAM (which contains the actual loader) has been destroyed and not enough free memory is available to reload it (PC-DOS v2 only).

Figure 5.1 summarizes the calling convention for function *4BH*. Listing 5.1 shows a skeleton of a typical *EXEC* call. This particular example uses the *EXEC* function to load and run the MS-DOS utility *CHKDSK.COM*. The *SHELL.ASM* program listing later in this chapter (Listing 5.3) presents a more complete example that includes the use of Int *21H* Function *4AH* to free unneeded memory.

Figure 5.1 *Calling convention for the EXEC function (Int 21H Function 4BH). Called with:*

AH	=	4BH
AL	=	function type
		00 = load and execute program
		03 = load overlay
ES:BX	=	segment:offset of parameter block
DS:DX	=	segment:offset of program specification

Returns:

If call succeeded

Carry flag clear. In MS-DOS version 2, all registers except for *CS:IP* may be destroyed. In MS-DOS versions 3.0 and later, registers are preserved in the usual fashion.

If call failed

Carry flag set and AX = error code.

Parameter block format:

If AL = 0 (load and execute program)

Bytes 0-1	=	segment pointer, environment block
Bytes 2-3	=	offset of command-line tail
Bytes 4-5	=	segment of command-line tail
Bytes 6-7	=	offset of first file control block to be copied into new PSP + 5CH
Bytes 8-9	=	segment of first file control block
Bytes 10-11	=	offset of second file control block to be copied into new PSP + 6CH
Bytes 12-13	=	segment of second file control block

If AL = 3 (load overlay)

Bytes 0-1	=	segment address where file will be loaded
Bytes 2-3	=	relocation factor to apply to loaded image

Listing 5.1 *A brief example of the use of the MS-DOS EXEC call, with all necessary variables and command blocks. Note the protection of the registers for MS-DOS version 2 and the masking of interrupts during loading of SS:SP to circumvent a bug in some early 8088 CPUs.*

```
cr      equ     0dh             ; ASCII carriage return
        .
        .
        .
        mov     stkseg,ss       ; save stack pointer
        mov     stkptr,sp

        mov     dx,offset pname ; DS:DX = program name
        mov     bx,offset pars  ; ES:BX = param block
        mov     ax,4b00h        ; function 4bh, subfunction 00h
        int     21h             ; transfer to MS-DOS

        mov     ax,_DATA        ; make our data segment
        mov     ds,ax           ; addressable again
        mov     es,ax

        cli                     ; (for bug in some 8088s)
        mov     ss,stkseg       ; restore stack pointer
        mov     sp,stkptr
        sti                     ; (for bug in some 8088s)
        jc      error           ; jump if EXEC failed
        .
        .
        .

stkseg  dw      0               ; original SS contents
stkptr  dw      0               ; original SP contents

pname   db      '\CHKDSK.COM',0 ; pathname of child program

pars    dw      envir           ; environment segment
        dd      cmdline         ; command line for child
        dd      fcb1            ; file control block #1
        dd      fcb2            ; file control block #2
```

Listing 5.1 *(cont'd)*

```
cmdlinedb     4,' *.*',cr          ; command line for child

fcb1    db    0                    ; file control block #1
        db    11 dup ('?')
        db    25 dup (0)

fcb2    db    0                    ; file control block #2
        db    11 dup (' ')
        db    25 dup (0)

envir   segment para 'ENVIR'       ; environment segment

        db    'PATH=',0            ; empty search path
                                   ; location of COMMAND.COM

        db    'COMSPEC=A:\COMMAND.COM',0
        db    0                    ; end of environment

envir   ends
```

More About the Environment Block

The environment block is always paragraph aligned (starts at an address that is a multiple of 16 bytes) and contains a series of ASCIIZ strings. Each of the strings takes the following form:

```
NAME=PARAMETER
```

An additional zero byte (Listing 5.2) indicates the end of the entire set of strings. Under MS-DOS v3, the block of environment strings and the extra zero byte are followed by a word count and the complete drive, path, filename, and extension used by *EXEC* to load the program.

Figure 5.2 *Dump of a typical environment block under MS-DOS v3. This particular example contains the default COMSPEC parameter and two relatively complex PATH and PROMPT control strings that were set up by entries in the user's AUTOEXEC file. Note the path and file specification of the executing program following the double zeros at offset 0073H that denote the end of the environment block.*

```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F 0123456789ABCDEF
0000 43 4F 4D 53 50 45 43 3D 43 3A 5C 43 4F 4D 4D 41 COMSPEC=C:\COMMA
0010 4E 44 2E 43 4F 4D 00 50 52 4F 4D 50 54 3D 24 70 NDcom.PROMPT=$p
0020 24 5F 24 64 20 20 20 24 74 24 68 24 68 24 68 24 $_$d   $t$h$h$h$
0030 68 24 68 24 68 20 24 71 24 71 24 67 00 50 41 54 h$h$h $q$q$g.PAT
0040 48 3D 43 3A 5C 53 59 53 54 45 4D 3B 43 3A 5C 41 H=C:\SYSTEM;C:\A
0050 53 4D 3B 43 3A 5C 57 53 3B 43 3A 5C 45 54 48 45 SM;C:\WS;C:\ETHE
0060 52 4E 45 54 3B 43 3A 5C 46 4F 52 54 48 5C 50 43 RNET;C:\FORTH\PC
0070 33 31 3B 00 00 01 00 43 3A 5C 46 4F 52 54 48 5C 31;....C:\FORTH\
0080 50 43 33 31 5C 46 4F 52 54 48 2E 43 4F 4D 00 20 PC31\FORTH.COM.
```

Under normal conditions, the environment block inherited by a program will contain at least three strings:

```
COMSPEC=variable
PATH=variable
PROMPT=variable
```

MS-DOS places these three strings into the environment block at system initialization, during the interpretation of *SHELL*, *PATH*, and *PROMPT* directives in the *CONFIG.SYS* and *AUTOEXEC.BAT* files. The strings tell the MS-DOS command interpreter, *COMMAND.COM*, the location of its executable file (to enable it to reload the transient portion), where to search for executable external commands or program files, and the format of the user prompt.

You can add other strings to the environment block, either interactively or in batch files, with the *SET* command. Transient programs can use these strings for informational purposes. For example, the Microsoft C Compiler looks in the environment block for *INCLUDE*, *LIB*, and *TMP* strings to tell it where to find its *#include* files and library files and where to build its temporary working files.

Example Programs: *SHELL.C* And *SHELL.ASM*

As a practical example of use of the MS-DOS *EXEC* function, I have included a small command interpreter called *SHELL,* with equivalent Microsoft C (Listing 5.2) and Microsoft Macro Assembler (Listing 5.3) source code. The source code for the assembly-language version is considerably more complex than the code for the C version, but the names and functionality of the various procedures are quite parallel.

Listing 5.2 **SHELL.C:** *A table-driven command interpreter written in Microsoft C.*

```
/*

   SHELL.C    Simple extendable command interpreter for MS-DOS
              versions 2.0 and later

   Copyright 1988 Ray Duncan

   Compile:   C>CL SHELL.C

   Usage:     C>SHELL
*/

#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <signal.h>

                                    /* macro to return number of
                                       elements in a structure */
#define dim(x) (sizeof(x) / sizeof(x[0]))

unsigned intrinsic(char *);         /* function prototypes */
void extrinsic(char *);
void get_cmd(char *);
void get_comspec(char *);
void break_handler(void);
void cls_cmd(void);
void dos_cmd(void);
```

Listing 5.2 *(cont'd)*

```
void exit_cmd(void);

struct cmd_table {                  /* intrinsic commands table */
                    char *cmd_name;
                    int  (*cmd_fxn)();
                 }  commands[] =

                 {  "CLS", cls_cmd,
                    "DOS", dos_cmd,
                    "EXIT",exit_cmd, };

static char com_spec[64];           /* COMMAND.COM filespec */

main(int argc, char *argv[])
{
    char inp_buf[80];               /* keyboard input buffer */

    get_comspec(com_spec);          /* get COMMAND.COM filespec */

                                    /* register new handler
                                       for Ctrl-C interrupts */

    if(signal(SIGINT, break_handler) == (int(*)()) -1)
    {
        fputs("Can't capture Control-C Interrupt", stderr);
        exit(1);
    }

    while(1)                        /* main interpreter loop */
    {
        get_cmd(inp_buf);           /* get a command */
        if (! intrinsic(inp_buf) )  /* if it's intrinsic,
                                       run its subroutine */
            extrinsic(inp_buf);     /* else pass to COMMAND.COM */
        }
}
```

Listing 5.2 *(cont'd)*

```
/*
   Try to match user's command with intrinsic command
   table. If a match is found, run the associated routine
   and return true; else return false.
*/

unsigned intrinsic(char *input_string)
{
   int i, j;                          /* some scratch variables */

                                      /* scan off leading blanks */
   while(*input_string == '\x20') input_string++ ;

                                      /* search command table     */
   for(i=0; i < dim(commands); i++)
   {
      j = strcmp(commands[i].cmd_name, input_string);
      if(j == 0)                      /* if match, run routine */
      {
         (*commands[i].cmd_fxn)();
         return(1);                   /* and return true */
      }
   }
   return(0);                         /* no match, return false */
}

/*
   Process an extrinsic command by passing it
   to an EXEC'd copy of COMMAND.COM.
*/

void extrinsic(char *input_string)
{
   int status;
   status = system(input_string);  /* call EXEC function */
   if(status)                      /* if failed, display
                                      error message */
      fputs("\nEXEC of COMMAND.COM failed\n", stderr);
}
```

Listing 5.2 *(cont'd)*

```
/*
    Issue prompt, get user's command from standard input,
    fold it to uppercase.
*/

void get_cmd(char *buffer)
{
    printf("\nsh: ");                /* display prompt */
    gets(buffer);                    /* get keyboard entry */
    strupr(buffer);                  /* fold to uppercase */
}

/*
    Get the full path and file specification for COMMAND.COM
    from the COMSPEC variable in the environment.
*/

void get_comspec(char *buffer)
{
    strcpy(buffer, getenv("COMSPEC"));

    if(buffer[0] == NULL)
    {
        fputs("\nNo COMSPEC in environment\n", stderr);
        exit(1);
    }
}

/*
    This Ctrl-C handler keeps SHELL from losing control.
    It just reissues the prompt and returns.
*/

void break_handler(void)
{
    signal(SIGINT, break_handler);  /* reset handler */
    printf("\nsh: ");                /* display prompt */
}
```

Listing 5.2 *(cont'd)*

```
/*
    These are the subroutines for the intrinsic commands.
*/

void cls_cmd(void)                  /* CLS command */
{
    printf("\033[2J");              /* ANSI escape sequence */
}                                   /* to clear screen */

void dos_cmd(void)                  /* DOS command */
{
    int status;
                                    /* run COMMAND.COM */
    status = spawnlp(P_WAIT, com_spec, com_spec, NULL);

    if (status)
        fputs("\nEXEC of COMMAND.COM failed\n",stderr);
}

void exit_cmd(void)                 /* EXIT command */

{
    exit(0);                        /* terminate SHELL      */
}
```

Listing 5.3 **SHELL.ASM:** *A simple table-driven command interpreter written in Microsoft Macro Assembler.*

```
        name    shell
        page    55,132
        title   SHELL.ASM-simple MS-DOS shell
;
; SHELL.ASM    Simple extendable command interpreter
;              for MS-DOS versions 2.0 and later
;
; Copyright 1988 by Ray Duncan

;
; Build:       C>MASM SHELL;
;              C>LINK SHELL;
;
; Usage:       C>SHELL;
;

stdin   equ     0                       ; standard input handle
stdout  equ     1                       ; standard output handle
stderr  equ     2                       ; standard error handle

cr      equ     0dh                     ; ASCII carriage return
lf      equ     0ah                     ; ASCII linefeed
blank   equ     20h                     ; ASCII blank code
escape  equ     01bh                    ; ASCII escape code

_TEXT   segment word public 'CODE'

        assume cs:_TEXT,ds:_DATA,ss:STACK

shell   proc    far                     ; at entry DS = ES = PSP

        mov     ax,_DATA                ; make our data segment
        mov     ds,ax                   ; addressable

        mov     ax,es:[002ch]           ; get environment segment
```

Listing 5.3 *(cont'd)*

```
        mov     env_seg,ax              ; from PSP and save it

                                        ; release unneeded memory...
                                        ; ES already = PSP segment
        mov     bx,100h                 ; BX = paragraphs needed
        mov     ah,4ah                  ; function 4ah = resize block
        int     21h                     ; transfer to MS-DOS
        jnc     shell1                  ; jump if resize OK

        mov     dx,offset msg1          ; resize failed, display
        mov     cx,msg1_length          ; error message and exit
        jmp     shell4

shell1:call     get_comspec             ; get COMMAND.COM filespec
        jnc     shell2                  ; jump if it was found

        mov     dx,offset msg3          ; COMSPEC not found in
        mov     cx,msg3_length          ; environment, display error
        jmp     shell4                  ; message and exit

shell2:mov      dx,offset shell3        ; set Ctrl-C vector (int 23h)
        mov     ax,cs                   ; for this program's handler
        mov     ds,ax                   ; DS:DX = handler address
        mov     ax,2523h                ; function 25h = set vector
        int     21h                     ; transfer to MS-DOS

        mov     ax,_DATA                ; make our data segment
        mov     ds,ax                   ; addressable again
        mov     es,ax

shell3:                                 ; main interpreter loop

        call    get_cmd                 ; get a command from user

        call    intrinsic               ; check if intrinsic function
        jnc     shell3                  ; yes, it was processed

        call    extrinsic               ; no, pass it to COMMAND.COM
        jmp     shell3                  ; then get another command
```

Listing 5.3 *(cont'd)*

```
shell4:                             ; come here if error detected
                                    ; DS:DX = message address
                                    ; CX = message length
        mov     bx,stderr           ; BX = standard error handle
        mov     ah,40h              ; function 40h = write
        int     21h                 ; transfer to MS-DOS

        mov     ax,4c01h            ; function 4ch = terminate with
                                    ; return code = 1
        int     21h                 ; transfer to MS-DOS

shell   endp


intrinsic proc near                 ; decode user entry against
                                    ; the table "COMMANDS"
                                    ; if match, run the routine,
                                    ; and return carry = false
                                    ; if no match, carry = true
                                    ; return carry = true

        mov     si,offset commands  ; DS:SI = command table

intr1:  cmp     byte ptr [si],0     ; end of table?
        je      intr7               ; jump, end of table found
        mov     di,offset inp_buf   ; no, let DI = addr of user input

intr2:  cmp     byte ptr [di],blank ; scan off any leading blanks
        jne     intr3

        inc     di                  ; found blank, go past it
        jmp     intr2

intr3:  mov     al,[si]             ; next character from table

        or      al,al               ; end of string?
        jz      intr4               ; jump, entire string matched

        cmp     al,[di]             ; compare to input character
```

Listing 5.3 *(cont'd)*

```
        jnz     intr6                   ; jump, found mismatch

        inc     si                      ; advance string pointers
        inc     di
        jmp     intr3

intr4:  cmp     byte ptr [di],cr        ; be sure user's entry
        je      intr5                   ; is the same length...
        cmp     byte ptr [di],blank     ; next character in entry
        jne     intr6                   ; must be blank or return

intr5:  call    word ptr [si+1]         ; run the command routine

        clc                             ; return carry flag = false
        ret                             ; as success flag

intr6:  lodsb                           ; look for end of this
        or      al,al                   ; command string (null byte)
        jnz     intr6                   ; not end yet, loop

        add     si,2                    ; skip over routine address
        jmp     intr1                   ; try to match next command

intr7:  stc                             ; command not matched, exit
        ret                             ; with carry = true

intrinsic endp
extrinsic procnear                      ; process extrinsic command
                                        ; by passing it to
                                        ; COMMAND.COM with a
                                        ; " /C " command tail

    mov     al,cr                       ; find length of command
    mov     cx,cmd_tail_length          ; by scanning for carriage
    mov     di,offset cmd_tail+1        ; return
    cld
    repnz scasb

    mov     ax,di                       ; calculate command-tail
```

Listing 5.3 *(cont'd)*

```
    sub     ax,offset cmd_tail+2      ; length without carriage
    mov     cmd_tail,al               ; return, and store it

                                      ; set command-tail address
    mov     word ptr par_cmd,offset cmd_tail
    call    exec                      ; and run COMMAND.COM
    ret

extrinsic endp


get_cmd proc  near                    ; prompt user, get command

                                      ; display the shell prompt
    mov     dx,offset prompt          ; DS:DX = message address
    mov     cx,prompt_length          ; CX = message length
    movbx,stdout                      ; BX = standard output handle
    mov     ah,40h                    ; function 40h = write
    int     21h                       ; transfer to MS-DOS
                                      ; get entry from user
    mov     dx,offset inp_buf         ; DS:DX = input buffer
    mov     cx,inp_buf_length         ; CX = max length to read
    mov     bx,stdin                  ; BX = standard input handle
    mov     ah,3fh                    ; function 3fh = read
    int     21h                       ; transfer to MS-DOS

        mov     si,offset inp_buf     ; fold lowercase characters
        mov     cx,inp_buf_length     ; in entry to uppercase

gcmd1:  cmp     byte ptr [si],'a'     ; check if 'a-z'
        jb      gcmd2                 ; jump, not in range
        cmp     byte ptr [si],'z'     ; check if 'a-z'
        ja      gcmd2                 ; jump, not in range
        sub     byte ptr [si],'a'-'A' ; convert to uppercase

gcmd2:  inc     si                    ; advance through entry
        loop    gcmd1
        ret                           ; back to caller
get_cmd endp
```

Listing 5.3 *(cont'd)*

```
get_comspec proc near                    ; get location of COMMAND.COM
                                         ; from environment "COMSPEC="
                                         ; returns carry = false
                                         ; if COMSPEC found
                                         ; returns carry = true
                                         ; if no COMSPEC

        mov     si,offset com_var        ; DS:SI = string to match...
        call    get_env                  ; search environment block
        jc      gcsp2                    ; jump if COMSPEC not found
                                         ; ES:DI points past "="
        mov     si,offset com_spec       ; DS:SI = local buffer

gcsp1:  mov     al,es:[di]               ; copy COMSPEC variable
        mov     [si],al                  ; to local buffer
        inc     si
        inc     di
        or      al,al                    ; null char? (turns off carry)
        jnz     gcsp1                    ; no, get next character

gcsp2:  ret                              ; back to caller

get_comspec endp


get_env proc  near                       ; search environment
                                         ; call DS:SI = "NAME="
                                         ; uses contents of "ENV_SEG"
                                         ; returns carry = false and ES:DI
                                         ; pointing to parameter if found,
                                         ; returns carry = true if no match
        mov     es,env_seg               ; get environment segment
        xor     di,di                    ; initialize env offset

genv1:  mov     bx,si                    ; initialize pointer to name
        cmp     byte ptr es:[di],0       ; end of environment?
        jne     genv2                    ; jump, end not found
        stc                              ; no match, return carry set
    ret
```

Listing 5.3 *(cont'd)*

```
genv2:  mov     al,[bx]             ; get character from name
        or      al,al               ; end of name? (turns off carry)
        jz      genv3               ; yes, name matched

        cmp     al,es:[di]          ; compare to environment
        jne     genv4               ; jump if match failed

        inc     bx                  ; advance environment
        inc     di                  ; and name pointers
        jmp     genv2

genv3:                              ; match found, carry = clear,
        ret                         ; ES:DI = variable

genv4:  xor     al,al               ; scan forward in environment
        mov     cx,-1               ; for zero byte
        cld
        repnz   scasb
        jmp     genv1               ; go compare next string

get_env endp


exec    proc    near                ; call MS-DOS EXEC function
                                    ; to run COMMAND.COM

        mov     stkseg,ss           ; save stack pointer
        mov     stkptr,sp

                                    ; now run COMMAND.COM
        mov     dx,offset com_spec  ; DS:DX = filename
        mov     bx,offset par_blk   ; ES:BX = parameter block
        mov     ax,4b00h            ; function 4bh = EXEC
                                    ; subfunction 0 =
                                    ; load and execute
        int     21h                 ; transfer to MS-DOS
```

Listing 5.3 *(cont'd)*

```
        mov     ax,_DATA            ; make data segment
        mov     ds,ax               ; addressable again
        mov     es,ax

        cli                         ; (for bug in some 8088s)
        mov     ss,stkseg           ; restore stack pointer
        mov     sp,stkptr
        sti                         ; (for bug in some 8088s)

        jnc     exec1               ; jump if no errors

                                    ; display error message
        mov     dx,offset msg2      ; DS:DX = message address
        mov     cx,msg2_length      ; CX = message length
        mov     bx,stderr           ; BX = standard error handle
        mov     ah,40h              ; function 40h = write
        int     21h                 ; transfer to MS-DOS

exec1:  ret                         ; back to caller

exec    endp


cls_cmdproc     near                ; intrinsic CLS command

        mov     dx,offset cls_str   ; send the ANSI escape
        mov     cx,cls_str_length   ; sequence to clear
        mov     bx,stdout           ; the screen
        mov     ah,40h
        int 21h
        ret

cls_cmd endp
```

Listing 5.3 *(cont'd)*

```
dos_cmd proc  near                  ; intrinsic DOS command

                                    ; set null command tail
        mov   word ptr par_cmd,offset nultail

        call  exec                  ; and run COMMAND.COM
        ret

dos_cmd endp
exit_cmd proc near                  ; intrinsic EXIT command

        mov   ax,4c00h              ; call MS-DOS terminate
        int   21h                   ; function with
                                    ; return code of zero

exit_cmd endp

_TEXT  ends


STACK  segment para stack 'STACK'   ; declare stack segment

        dw    64 dup (?)

STACK  ends


_DATA      segment word public 'DATA'

commands   equ $                    ; "intrinsic" commands table
                                    ; each entry is ASCIIZ string
                                    ; followed by the offset
                                    ; of the procedure to be
                                    ; executed for that command
        db    'CLS',0
        dw    cls_cmd
        db    'DOS',0
        dw    dos_cmd
```

Listing 5.3 *(cont'd)*

```
        db      'EXIT',0
        dw          exit_cmd

        db      0                       ; end of table

com_var db      'COMSPEC=',0            ; environment variable

                                        ; COMMAND.COM filespec
com_spec db     80 dup (0)              ; from environment COMSPEC=

nultail db      0,cr                    ; null command tail for
                                        ; invoking COMMAND.COM
                                        ; as another shell

cmd_tail db     0,' /C '                ; command tail for invoking
                                        ; COMMAND.COM as a transient

inp_buf db      80 dup (0)              ; command line from standard input

inp_buf_length equ $-inp_buf
cmd_tail_length equ $-cmd_tail-1

prompt  db      cr,lf,'sh: '            ; SHELL's user prompt
prompt_length equ $-prompt

env_seg     dw      0                   ; segment of environment block

msg1        db      cr,lf
            db      'Unable to release memory.'
            db      cr,lf
msg1_length equ $-msg1

msg2        db      cr,lf
            db      'EXEC of COMMAND.COM failed.'
            db      cr,lf
msg2_length equ $-msg2

msg3        db      cr,lf
            db      'No COMSPEC variable in environment.'
```

Listing 5.3 *(cont'd)*

```
          db     cr,lf
msg3_length equ $-msg3

cls_str db       escape,'[2J'      ; ANSI escape sequence
 cls_str_length equ $-cls_str      ; to clear the screen

                                   ; EXEC parameter block
par_blk   dw         0             ; environment segment
par_cmd   dd         cmd_tail      ; command line
          dd     fcb1              ; file control block #1
          dd     fcb2              ; file control block #2

fcb1      db     0                 ; file control block #1
          db     11 dup (' ')
          db     25 dup (0)

fcb2      db     0                 ; file control block #2
          db     11 dup (' ')
          db     25 dup (0)

stkseg    dw     0; original SS contents
stkptr    dw     0                 ; original SP contents

_DATA   ends


       end    shell
```

The *SHELL* program is table driven and can easily be extended to provide a powerful customized user interface for almost any application. When *SHELL* takes control of the system, it displays the prompt

```
sh:
```

and waits for input from the user. After the user types a line terminated by a carriage return, *SHELL* tries to match the first token in the line against its table of internal (intrinsic) commands. If it finds a match, it calls the appropriate subroutine. If it does not find a match, it calls the MS-DOS *EXEC* function and passes the user's input to *COMMAND.COM* with the */C* switch, essentially using *COMMAND.COM* as a transient command processor under its own control.

As supplied in these listings, *SHELL* "knows" exactly three internal commands:

Command	*Action*
CLS	Uses the ANSI standard control sequence to clear the display screen and home the cursor.
DOS	Runs a copy of COMMAND.COM.
EXIT	Exits SHELL, returning control of the system to the next lower command interpreter.

You can quickly add new intrinsic commands to either the C version or the assembly-language version of *SHELL*. Simply code a procedure with the appropriate action and insert the name of that procedure, along with the text string that defines the command, into the table *COMMANDS*. In addition, you can easily prevent *SHELL* from passing certain "dangerous" commands (such as *MKDIR* or *ERASE*) to *COMMAND.COM* simply by putting the names of the commands to be screened out into the intrinsic command table with the address of a subroutine that prints an error message.

To summarize, the basic flow of both versions of the SHELL program is as follows:

1. The program calls MS-DOS Int *21H* Function *4AH* (Resize Memory Block) to shrink its memory allocation, so that the maximum possible space will be available for *COMMAND.COM* if it is run as an overlay. (This is explicit in the assembly-language version only. To keep the example code simple, the number of paragraphs to be reserved is coded as a generous literal value, rather than being figured out at runtime from the size and location of the various program segments.)
2. The program searches the environment for the *COMSPEC* variable, which defines the location of an executable copy of *COMMAND.COM*. If it can't find the *COMSPEC* variable, it prints an error message and exits.
3. The program puts the address of its own handler in the Ctrl-C vector (Int *23H*) so that it won't lose control if the user enters a *Ctrl-C* or a *Ctrl-Break*.
4. The program issues a prompt to the standard output device.
5. The program reads a buffered line from the standard input device to get the user's command.
6. The program matches the first blank-delimited token in the line against its table of intrinsic commands. If it finds a match, it executes the associated procedure.
7. If the program does not find a match in the table of intrinsic commands, it synthesizes a command-line tail by appending the user's input to the */C* switch and then *EXEC*s a copy of *COMMAND.COM*, passing the address of the synthesized command tail in the *EXEC* parameter block.
8. The program repeats steps 4 through 7 until the user enters the command EXIT, which is one of the intrinsic commands, and which causes *SHELL* to terminate execution.

In its present form, *SHELL* allows *COMMAND.COM* to inherit a full copy of the current environment. However, in some applications it may be helpful, or safer, to pass a modified copy of the environment block so that the secondary copy of *COMMAND.COM* will not have access to certain information.

Using *EXEC* To Load Overlays

Loading overlays with the *EXEC* function is much less complex than using *EXEC* to run another program. The overlay can be constructed as either a memory image (*.COM*) or relocatable (*.EXE*) file and need not be the same type as the program that loads it. The main program, called the root segment, must carry out the following steps to load and execute an overlay:

1. Make a memory block available to receive the overlay. The program that calls *EXEC* must own the memory block for the overlay.

2. Set up the overlay parameter block to be passed to the *EXEC* function. This block contains the segment address of the block that will receive the overlay, plus a segment relocation value to be applied to the contents of the overlay file (if it is a *.EXE* file). These are normally the same value.

3. Call the MS-DOS *EXEC* function to load the overlay by issuing an Int *21H* with the registers set up as follows:

```
AH = 4BH
AL = 03H (EXEC subfunction to load overlay)
DS:DX = segment:offset of overlay file pathname
ES:BX = segment:offset of overlay parameter block
```

Upon return from the *EXEC* function, the carry flag is clear if the overlay was found and loaded. The carry flag is set if the file could not be found or if some other error occurred.

4. Execute the code within the overlay by transferring to it with a far call. The overlay should be designed so that either the entry point or a pointer to the entry point is at the beginning of the module after it is loaded. This technique allows you to maintain the root and overlay modules separately, because the root module does not contain any "magical" knowledge of addresses within the overlay segment.

To prevent users from inadvertently running an overlay directly from the command line, you should assign overlay files an extension other than *.COM* or *.EXE*. It is most convenient to relate overlays to their root segment by assigning them the same filename but a different extension, such as *.OVL* or *.OV1*, *.OV2*, and so on.

Listing 5.4 shows the use of *EXEC* to load and execute an overlay.

Listing 5.4 *A code skeleton for loading and executing an overlay with the EXEC function. The overlay file may be in either .COM or .EXE format.*

```
        .
        .
        .
                                ; allocate memory for overlay
mov     bx,1000h                ; get 64 KB (4096 paragraphs)
mov     ah,48h                  ; function 48h = allocate block
int     21h                     ; transfer to MS-DOS
jc      error                   ; jump if allocation failed

mov     pars,ax                 ; set load address for overlay
mov     pars+2,ax               ; set relocation segment for overlay

                                ; set segment of entry point
mov     word ptr entry+2,ax

mov     stkseg,ss               ; save root's stack pointer
mov     stkptr,sp

mov     ax,ds                   ; set ES = DS
mov     es,ax

mov     dx,offset oname         ; DS:DX = overlay pathname
mov     bx,offset pars          ; ES:BX = parameter block
mov     ax,4b03h                ; function 4bh, subfunction 03h
int     21h                     ; transfer to MS-DOS

mov     ax,_DATA                ; make our data segment
mov     ds,ax                   ; addressable again
mov     es,ax

cli                             ; (for bug in some early 8088s)
mov       ss,stkseg             ; restore stack pointer
mov       sp,stkptr
```

Listing 5.4 *(cont'd)*

```
        sti                         ; (for bug in some early 8088s)

        jc          error           ; jump if EXEC failed

                                    ; otherwise EXEC succeeded...
        push        ds              ; save our data segment
        call        dword ptr entry ; now call the overlay
        pop         ds              ; restore our data segment
        .
        .
        .

oname   db          'OVERLAY.OVL',0 ; pathname of overlay file

pars    dw          0               ; load address (segment) for file
        dw          0               ; relocation (segment) for file

entry   dd          0               ; entry point for overlay

stkseg  dw          0               ; save SS register
stkptr  dw          0               ; save SP register
```

PC Interrupt-Driven Serial I/O

Philip Erdelsky

Serial Ports

Most IBM PCs and compatibles have at least one serial port; many have two or more. A serial port also goes under several other names: it may be called an asynchronous communications port, a UART (Universal Asynchronous Receiver and Transmitter) or an RS-232 port.

Some serial ports are used to drive printers, but on PC clones serial ports are more commonly used for communication, usually through a modem. An internal modem is actually a serial port and modem on a single card, and is accessed by the computer just like a serial port. Some kind of communication software (at the very least a terminal emulation program) is required to use a serial port for communication.

Writing a simple terminal emulation program seems easy. Just take keyboard input and pass it to the serial port output, take serial port input and pass it to the screen. Just for good measure, save all serial port input to a disk file so it can be reviewed and edited later. Finally, if the program is not to remain permanently memory-resident, it must look for some special key combination (such as *Alt-X*) and terminate when it appears. This is basically what the accompanying *TERMEMUL* program (Listing 6.1) does, although a sophisticated programming technique called interrupt-driven I/O is required to make it work.

Why Interrupt-Driven I/O Is Needed

Unfortunately, the BIOS interface to the serial ports (*INT 14H*) and the DOS interface (*COM1*, *COM2*, *COM3*, and *COM4*) have long been regarded as inadequate for communications. These routines were apparently designed for printers. They use polling, not interrupts, to determine when input has been received and when the serial port is ready to accept output. This reliance on polling is a serious shortcoming, especially for input, which may be missed if it arrives while the program is occupied with other tasks, such as disk or screen operations. Since most programs have other things to do besides waiting for input from the serial port, they must either:

- use interrupts, allowing them to wait for and accept input while doing other things, or
- use hardware or software handshaking stop the input while they attend to other things, or
- rely on the device at the other end to keep its output rate safely below the program's input capacity.

Even a simple terminal emulation program, if written without interrupts or handshaking, will often lose characters that arrive while the screen display is scrolling. (Some dial-up services send a few *NUL* characters at the end of each line to allow time for scrolling .)

Hardware handshaking involves the use of the RTS, CTS, DSR and DTR signals. All serial port connectors have pins for these signals, but not all serial cables have wires for them. When communication is by modem, these signals are not available. A typical telephone line has only two wires.

Software handshaking (also called *XON/XOFF* protocol) requires nothing beyond the bare minimum of connectors and cables, but it does require the cooperation of the device at the other end of the communication channel. Software handshaking works just like a standard screen display; type (or send) a *Ctrl-S* to stop; type (or send) a *Ctrl-Q* to resume. Actually, for a communication program it is a little more complicated: send a *Ctrl-S*, wait for the input to stop (there may have been a few characters in the pipeline), wait a little longer to make sure the input has stopped, and then do something else. A terminal emulation program that relies on only *XON-XOFF* control

would have to invoke handshaking at the end of almost every line, to allow time for scrolling.

In theory, it is always possible to replace interrupt-driven I/O by polled I/O, provided the *complete* program is written this way. However, PC application programs are not complete; they share some of the computing load with DOS and the BIOS, which were written without regard for the special needs of applications. Interrupt-driven I/O allows an application to reach into the DOS and BIOS and do some I/O processing even if an external event occurs while DOS or BIOS code is running.

How Interrupts Work

When it has been installed and configured to do so, an external device can interrupt the CPU by sending a signal on one of eight interrupt request (IRQ) lines. The signal is routed through a programmable interrupt controller (PIC), which passes the interrupt to the CPU only if the corresponding bit in the PIC's Interrupt Disable Register (at I/O address *21H*) has been cleared to zero. If the interrupt-enable bit in the CPU flag register is set, the interrupt is honored immediately; otherwise, the interrupt is stored (but not lost) until the bit is set. When the interrupt is honored, the CPU acts as though the next instruction were an *INT* instruction; that is, it pushes the flag register and the address of the next instruction onto the stack, clears the interrupt-enable bit to prevent further interrupts, and then starts executing an interrupt handler whose starting address is taken from an interrupt vector. IRQ lines 0 to 7 are assigned to interrupt vectors 8 through 15, respectively.

An interrupt vector may be set with this DOS system call:

```
AH = 25H
AL = interrupt vector number
DS:DX = starting address of Interrupt Handler
INT 21H
```

Since an application should restore an interrupt vector upon termination, there is also a DOS system call to read an interrupt vector:

```
AH = 35H
AL = interrupt vector number
INT 21H
returns with ES:BX = starting address of Interrupt Handler
```

A general assembly language interrupt handler looks something like the one in Figure 6.1. Since the interrupt may occur at any time, the interrupt handler must preserve all register values. The handler must service the interrupt according to the requirements of the device that generated it, and then send an end-of-interrupt signal (*20H*) to the PIC (at address *20H*). Finally, the handler must restore all the registers and return to the interrupted code with an *IRET* instruction. It is not necessary to save and restore the flag register because that is done automatically.

An important consideration in interrupt programming is the interrupt latency time, which is the delay from the time the interrupt signal is received by the CPU until the time it is serviced. If several interrupt handlers run with interrupts disabled, as most do, the worst-case interrupt latency will be the

Figure 6.1 *Form of an interrupt handler for an external device interrupt*

```
IHAND   PROC    FAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        <device-specific operations>
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        MOV     AL,20H
        OUT     20H,AL
        POP     AX
        IRET
IHAND   ENDP
```

sum of their individual latencies. If some interrupt handlers allow themselves to be interrupted (by setting the interrupt-enable bit with an *STI* instruction), the worst-case latency is reduced, but then their combined stack requirements may lead to stack overflow, unless each interrupt handler switches to its own stack.

On an AT system, there are eight additional IRQ lines, numbered from 8 to 15, that are assigned to interrupt vectors 112 to 119, respectively. These lines feed to a second PIC, whose interrupt mask register is at I/O address *A1H* (the interrupts are actually passed through IRQ number 2 in the first PIC, but the BIOS takes care of those details).

Interrupt Handling In The Terminal Emulator

The *TERMEMUL* program, like more sophisticated communication programs, bypasses both DOS and the BIOS and accesses the serial port directly so it can use interrupts. The program maintains two FIFO buffers, one for input and one for output. These are "ring" buffers, because the modular arithmetic used for their indices makes them effectively circular. The program was written in Turbo Pascal, v5.0, a high-level language with good access to low-level features such as interrupts, hardware I/O operations, and bitwise operations on integers.

Input is fairly straightforward. When a character arrives through the serial port, an interrupt is generated and the interrupt handler puts it into the input buffer. The main program takes characters from the buffer, displays them on the screen and saves them in an optional disk file (whose specifications were typed on the command line). Some characters take longer to display than others. A line feed, for example, takes a relatively long time if the screen needs to be scrolled. During a scroll, a few characters after the line feed will accumulate in the buffer. If the program is also saving characters in a disk file, a fairly large buffer will be needed to hold all the characters that arrive while the program is writing to the disk file.

If the input buffer is full when another character is received, the interrupt handler sets the *Overflow* flag and discards the character. A more sophisticated version of *TERMEMUL* would probably take some additional action in this case, such as invoking either hardware or software handshaking or notifying the operator in some manner.

Output is very similar. The main program accepts characters from the PC keyboard and puts them into the output buffer. The interrupt handler sends them out when the serial port generates an interrupt to tell the CPU that it is ready to send a character. However, there is one little complication. The serial port generates an interrupt only *after* it has finished sending the previous character. Hence the first character in a series has to be sent by the main program. Subsequent characters are sent by the interrupt handler, as long as the main program puts characters into the output buffer fast enough to keep the buffer from emptying. When an interrupt occurs and the buffer is empty, the situation reverts to to its initial state.

Advantages Of Output Buffering

As long as characters are accepted only from the keyboard, buffering serial output is actually gilding the lily. The DOS and BIOS keyboard input interface is interrupt-driven, and it uses a FIFO buffer. If the operator somehow manages to type a few characters faster than the serial port can transmit them, the keyboard buffer could hold them even if there were no serial output buffer. Nevertheless, output buffering is included for purposes of illustration.

If the program is modified to allow the contents of a disk file to be sent directly to the serial port, output buffering will greatly improve the program's efficiency, provided the output buffer is at least as large as the file system's input buffer. Since disk files can be read faster than they can be sent through a serial port, the program will sometimes find the output buffer full and will have to wait until some room appears in it. However, when it returns to the file system for more data, it will generally leave a full output buffer that the interrupt handler can transmit while DOS is reading more information from the disk file. It is this parallel operation that produces the efficiency enhancement.

Timing Problems

The interaction between the main program and the interrupt handler illustrates a basic principle of multitasking called mutual exclusion. Two tasks that run in parallel, such as the interrupt handler and the main program in *TERMEMUL*, must be prevented from accessing a common data structure such as *OutBuffer* at the same time. Even simple operations like incrementing

and decrementing a counter may not work if the two operations are interleaved. For example, suppose *Count* contains *2*. After the main program increments it and the interrupt handler decrements it, the result should be *2*; but look what happens when temporary registers are used and the operations are interleaved:

main program	Interrupt Handler	Count
`Count -> Reg1`		2
	`Count -> Reg2`	2
	`Reg2-1 -> Reg2`	2
	`Reg2 -> Count`	1
`Reg1+1 -> Reg1`		1
`Reg1 -> Count`		3

Of course, this cannot happen if the operations are implemented in single, indivisible instructions; but compilers seldom generate such instructions for these operations. In *TERMEMUL* the variable *Index* and the internal state of the serial port are also involved. These "synchronization" errors are likely to be very intermittent, because they occur only when an interrupt occurs during the few microseconds spanned by a critical period.

Therefore, the main program must disable interrupts while it is putting a character into the output buffer or taking a character out of the input buffer. Otherwise, the program would generate some rare and mysterious errors that would probably be blamed incorrectly on the hardware.

There is one rare type of hardware failure for which the *TERMEMUL* program makes no allowance. Glitches happen, and interrupts that are supposed to occur are sometimes missed. In the case of input interrupts, the result is a single missed character, which is not especially serious. In the case of output interrupts, however, the interrupt handler would never send another character to the serial port. The output buffer would eventually fill up. In the *TERMEMUL* program the only escape is to terminate and then restart the program, which is probably an adequate remedy in that situation. However, designers of large systems do (or should) implement timouts to deal with missed interrupts.

Figure 6.2 *I/O addresses of serial port registers relative to the base address.*

```
address         input register

Base            received data
Base+2          interrupt identification
Base+5          line status
Base+6          modem status

address         output register

Base            transmitted data
Base+1          interrupt enable
Base            LS byte of baud rate count
Base+1          MS byte of baud rate count
Base+3          line control (and baud rate count selection)
Base+4          modem control
```

Details Of The Serial Port Interface

Now for a bit-by-bit description of the interface between the CPU and the serial port. This interface has remained unchanged for years, probably to maintain compatibility with communication software that accesses serial ports directly.

The Serial Port Registers

Each serial port has six registers that can be written by CPU output instructions and four registers that can be read by CPU input instructions. When configured to do so, a serial port can generate a single interrupt on one of the interrupt request (IRQ) lines. The interface is determined by two quantities, which can usually be selected by jumpers or switches:

- the base I/O address, which is *3F8H* for *COM1* and *2F8H* for *COM2*, and
- the interrupt request (IRQ) line number, which is 4 for *COM1* and 3 for *COM2*.

The accompanying program can easily be modified to accommodate other choices by changing the constants *Base* and *IRQ*. The serial port uses a range of seven consecutive I/O addresses, starting with the base address. Obviously these addresses must not overlap the range of any other installed I/O device.

The registers are as shown in Figure 6.2.

Figure 6.3 *The Line Control Register (address = Base+3).*

```
bit 7 (MS) = 1: select baud rate count registers

bit 6 = 1: send break signal

bits 5,4,3 = 000  no parity
bits 5,4,3 = 001  odd parity
bits 5,4,3 = 010  no parity
bits 5,4,3 = 011  even parity
bits 5,4,3 = 100  no parity
bits 5,4,3 = 101  parity bit always 1
bits 5,4,3 = 110  no parity
bits 5,4,3 = 111  parity bit always 0

bit 2 = 0: 1 stop bit
bit 2 = 1: 1.5 stop bits if 5 data bits; 2 stop bits otherwise

bit 1,0 = 00: 5 data bits
bit 1,0 = 01: 6 data bits
bit 1,0 = 10: 7 data bits
bit 1,0 = 11: 8 data bits
```

Setting The Interrupt Vector

The program's first task is to set the interrupt vector. If the program is to terminate and return to DOS, the interrupt vector must be saved and restored upon termination. Otherwise a serial port interrupt that occurs after termination will be vectored to an interrupt handler that may no longer be in memory. The result is almost always a system crash. In the standard PC system, IRQ numbers 0 to 7 are assigned to interrupt vectors 8 to 15, respectively. An AT system also possesses IRQ numbers 8 to 15, which are assigned to interrupt vectors 112 to 119, respectively, by the BIOS.

Initializing The Serial Port

Next, the program must initialize the serial port. Interrupts should be disabled during this process to prevent serial port interrupts while the serial port is partially initialized.

First, the program writes *80H* to the Line Control Register to select the baud rate count registers. Then the program forms the baud rate count, which is a 16-bit quotient *115200/BaudRate*, and writes it to the two Baud Rate Count Registers.

The program then writes an appropriate value to the Line Control Register to set the parity, stop bits and data bits, and also to deselect the Baud Rate Count Registers. The format of the Line Control Register is shown in Figure 6.3.

Next the program initializes the Modem Control Register (Figure 6.4). The two signals RTS and DTS are used for hardware handshaking. Set both to *1* when hardware handshaking is not used.

By setting bit 3 of the Modem Control Register, the program has not actually enabled interrupts. It has merely opened the first of several doors through which the interrupt signal must pass. The serial port is capable of interrupting on any of four conditions. To specify which are to cause interrupts, the program must write to the Interrupt Enable Register, whose format is shown in Figure 6.5.

All four conditions can also be detected by polling the appropriate bits in the Line Status Register or the Modem Status Register (Figure 6.6 and Figure 6.7). Finally, the program should read the Received Character Register

Figure 6.4 *The Modem Control Register (address = Base+4).*

```
bits 7 (MS), 6, 5: unused

bit 4 = 1: enable loopback test (output is echoed back to input)

bit 3 = 1: enable interrupts

bit 2: unused

bit 1: RTS (request to send) signal

bit 0: DTR (data terminal ready) signal
```

Figure 6.5 *The Interrupt Enable Register (address = Base+1).*

```
bits 7 (MS), 6, 5, 4: unused

bit 3 = 1: interrupt when the modem status changes

bit 2 = 1: interrupt upon error or break condition in input

bit 1 = 1: interrupt when ready to send a character

bit 0 = 1: interrupt when a character is received
```

to remove any nonsense left by the previous user. (Always rinse the cup out before drinking from it.)

The serial port is now fully configured, but any interrupts it generates must pass through the PIC (programmable interrupt controller), which has its own Interrupt Disable Register at I/O address *21H*. The bit corresponding to the IRQ being used by the serial port must be *cleared* to enable the interrupt. Other bits must be left unchanged to avoid interfering with other interrupt-driven devices. Fortunately, the register can be read or written, so this is easy.

If the serial port is using one of IRQ numbers 8 to 15 on an AT system, the interrupt must pass through a second PIC with Interrupt Disable Register at *A1H*.

Figure 6.6 *The Line Status Register (address = Base+5).*

```
bit 6 = 1: transmitter shift register empty

bit 5 = 1: ready to send a character (TXRDY)

bit 4 = 1: break signal detected

bit 3 = 1: framing error

bit 2 = 1: parity error

bit 1 = 1: overrun error

bit 0 = 1: a character has been received (RXRDY)
```

Figure 6.7 *The Modem Status Register (address = Base+6).*

```
bit 7 (MS) = 1: carrier tone detected by attached modem

bit 6 = 1: (telephone) ringing detected by attached modem

bit 5 = 1: DSR (data set ready) - used for hardware handshaking

bit 4 = 1: CTS (clear to send)  - used for hardware handshaking

bit 3 = 1: there has been a change in bit 7

bit 2 = 1: there has been a change in bit 6

bit 1 = 1: there has been a change in bit 5

bit 0 = 1: there has been a change in bit 4
```

Special Interrupt Techniques For Serial Ports

When an interrupt occurs, the interrupt handler must read the Interrupt Identification Register to see which condition produced the interrupt. The format of this register is shown in Figure 6.8.

Even if only one interrupt condition has been specified, the interrupt handler must read the Interrupt Identification Register to reset the interrupt logic and make the next interrupt possible.

If two or more interrupt conditions have been enabled, as in the accompanying program, the interrupt handler must read the Interrupt Identification Register repeatedly and service all interrupt conditions that appear in it. This is necessary because if two or more interrupt conditions occur nearly simultaneously, they may generate only one interrupt. Interrupt conditions are supposed to be presented in the order indicated in Figure 6.8.

Also, to ensure continued interrupts when the same condition occurs again, and to prevent further interrupts until the same condition occurs again, the interrupt handler or the main program must reset the interrupt by reading or writing the appropriate register as follows:

interrupt condition	**action to reset interrupt**
change in modem status	read Modem Status Register
ready to send next character	write Transmitted Data Register
character has been received	read Received Data Register
error or break condition	read Line Status Register

Figure 6.8 *The Interrupt Identification Register (address = Base+2).*

```
bit 7 (MS), 6, 5, 4, 3: not used

bits 2,1 = 00: change in modem status (presented last)
bits 2,1 = 01: ready to send next character
bits 2,1 = 10: character has been received
bits 2,1 = 11: error or break condition detected (presented first)

bit 0 = 1: no interrupts pending
```

The interrupt handler in *TERMEMUL* has been defensively written. It does not assume that both interrupt conditions, if present, will be presented in the specified order. Even though reading the Interrupt Identification Register should be sufficient to identify the interrupt, in some cases it apparently is not, so the interrupt handler also reads the Line Status Register. Finally, to prevent a system lockup if the serial port fails to set bit 0 when all interrupt conditions have been serviced, it reads the Interrupt Identification Register at most three times.

Most of these difficulties could be eliminated if only a single interrupt condition is enabled at a time. Indeed, this is usually the recommended practice. The *TERMEMUL* program works quite well with interrupt-driven buffering only on input. Output buffering is recommended only when uploading a file, and in that case input buffering is unnecessary. Input buffering can be eliminated by clearing the flag *OutputBuffering-Implemented*.

Keyboard And Screen Access

The *TERMEMUL* program gets its keyboard input through DOS, which splits the codes for function keys and other special keys into two-character sequences, the first of which is zero. The *Key* function in the *TERMEMUL* program puts them back together, and the main program ignores all of them except *Alt-X*, but it could be modified to make good use of other special keys. This process is rather slow and inefficient, but entirely adequate for input at finger speed.

The *TERMEMUL* program uses the standard *Write* function for screen output, modifying it only by screening out ASCII *nulls*, which communication programs should ignore but which the *Write* function displays as spaces. *Write*'s speed is adequate in many cases, but may be inadequate when communicating at high baud rates on slow machines. Also, *Write* terminates the program when a *Break* signal (*Ctrl-NumLock* on most keyboards) has been typed, and leaves the interrupt vector pointing to an interrupt handler that does not stay resident. To avoid these problems, more sophisticated communications programs bypass the BIOS and access the screen directly.

Listing 6.1

```
program TERMEMUL;

uses Crt, Dos;

const

  InputBufferSize = 1024; { must be power of 2 }
  OutputBufferSize = 256; { must be power of 2 }
  BaudRate = 1200;
  DataBits = 8;
  Parity = 0; { 0=none, 1=odd, 3=even }
  StopBits = 1;
  Base = $2F8; { $3F8=COM1, $2F8=COM2 }
  IRQ = 3; { 4=COM1, 3=COM2 }
  OutputBufferingImplemented = true;

type

  InputBufferType = record
    Chars : array[0..InputBufferSize-1] of char;
    Count, Index : integer;
    Overflow : boolean;
  end;

  OutputBufferType = record
    Chars : array[0..OutputBufferSize-1] of char;
    Count, Index : integer;
    Overflow : boolean;
  end;

var

  InputBuffer : InputBufferType;
  OutputBuffer : OutputBufferType;
  SavedInterruptVector : pointer;
  InterruptVectorNumber : integer;
  InChar : char;
  OutChar : integer;
  LogFile : text;
  Logging : boolean;

procedure InterruptHandler;
interrupt;
var
  InterruptIdentification : Byte;
  LineStatus : Byte;
  Ch : Byte;
  LoopCount : integer;
begin
  LoopCount := 0;
  repeat
    InterruptIdentification := Port[Base+2] and 7;
    LineStatus := Port[Base+5];
```

Listing 6.1 *(cont'd)*

```
    if (InterruptIdentification = 4) or ((LineStatus and 1) <> 0) then
    with InputBuffer do
    begin
        Ch := Port[Base];
        if Count < InputBufferSize then
      begin
        Chars[Index] := Chr(Ch);
        Index := (Index+1) and (InputBufferSize-1);
        Count := Count+1;
      end
      else
        Overflow := true;
    end;
    if (InterruptIdentification = 2) or ((LineStatus and $20) <> 0) then
    with OutputBuffer do
    begin
      if Count > 0 then
      begin
        Port[Base] := Ord(Chars[Index]);
        Index := (Index+1) and (OutputBufferSize-1);
        Count := Count-1;
      end
      else Count := -1;
    end;
    LoopCount := LoopCount+1;
  until ((InterruptIdentification and 1) = 1) or (LoopCount = 3);
  if IRQ<8 then Port[$20] := $20; { reset PIC }
end;

function Key : integer;
var
  k : char;
begin
  Key := 0;
  if KeyPressed then
  begin
    k := ReadKey;
    if k = #0 then
      Key := 256+ord(ReadKey)
    else
      Key := ord(k);
  end;
end;

procedure DisableInterrupts; inline($FA);
procedure EnableInterrupts; inline($FB);

begin

  if ParamCount >0 then
  begin
    Assign(LogFile, ParamStr(1));
    Rewrite(LogFile);
    Logging := true;
  end
```

Listing 6.1 *(cont'd)*

```
else Logging := false;
InputBuffer.Count := 0;
InputBuffer.Index := 0;
InputBuffer.Overflow := false;
OutputBuffer.Count := -1;
OutputBuffer.Index := 0;
OutputBuffer.Overflow := false;
if IRQ<8 then InterruptVectorNumber := IRQ+8
else InterruptVectorNumber := IRQ+112;
GetIntVec(InterruptVectorNumber, SavedInterruptVector);
SetIntVec(InterruptVectorNumber, @InterruptHandler);
DisableInterrupts;
Port[Base+3] := $80; {select baud rate count registers }
Port[Base] := (115200 div BaudRate) and $FF;
Port[Base+1] := (115200 div BaudRate) div 256;
Port[Base+3] := Parity*8 + (StopBits-1)*4 + (DataBits-5);
Port[Base+4] := 8+3; { enable interrupts, set RTS and DTR }
Port[Base+1] := 3; { define interrupt conditions }
OutChar := Port[Base]; { clean out Received Data Register }
if IRQ<8 then  { enable interrupt via PIC }
  Port[$21] := Port[$21] and ($FF-(1 shl IRQ))
else
  Port[$A1] := Port[$A1] and ($FF-(1 shl (IRQ-8)));
EnableInterrupts;
OutChar := 0;

repeat

  if OutChar = 0 then OutChar := Key;

  if (0<OutChar) and (OutChar<127) then
  begin
    if OutputBufferingImplemented then
    with OutputBuffer do
    begin
      DisableInterrupts;
      if Count = OutputBufferSize then Overflow := true
      else
      begin
        if Count = -1 then Port[Base] := OutChar
        else Chars[(Index+Count) and (OutputBufferSize-1)] := Chr(OutChar);
        Count := Count+1;
      end;
      EnableInterrupts;
      OutChar := 0;
    end
    else if (Port[Base+5] and $20) <> 0 then
    begin
      Port[Base] := OutChar;
      OutChar := 0;
    end;
  end;
```

Listing 6.1 *(cont'd)*

```
    with InputBuffer do if Count > 0 then
    begin
      DisableInterrupts;
      InChar := Chars[(Index-Count) and (InputBufferSize-1)];
      Count := Count-1;
      EnableInterrupts;
      if InChar <> #0 then
      begin
        Write(InChar);
        if Logging then Write(LogFile, InChar);
      end;
    end;

  until OutChar = 301 {Alt-X};

  Port[Base+4] := 0; { disable interrupts, clear RTS and DTR }
  SetIntVec(InterruptVectorNumber, SavedInterruptVector);
    { restore interrupt vector }
  if IRQ<8 then                              { restore PIC }
    Port[$21] := Port[$21] or (1 shl IRQ)
  else
    Port[$A1] := Port[$A1] or (1 shl (IRQ-8));
  if Logging then Close(LogFile);
end.
```

A Programmer's Bibliography

Harold C. Ogg

In attempts to coax the last ounce of power from a C language or assembly language routine, PC programmers often overlook the hidden powers of the foundation of their labors — the Disk Operating System. Whether it be MS-DOS or PC-DOS, DOS has inherent powers and talents that require only the knowledge of their existences to unlock a potential that is relatively free for the taking.

Perhaps it is best to think of MS-DOS as a language. Indeed, its features can be unleashed with batch files (simple or extensive), and its system calls are dormant only to those who would not take advantage of the various interrupts and service routines via coded subroutines. The knowledge is available to those who would read and study; this bibliography purports to provide a library foundation for those who actively pursue the offerings of this best-selling operating system.

Even for those who would not use MS-DOS as a tool in itself, a thorough knowledge of its anatomy is essential for tapping the full potential from applications and utility programs. Many software packages insist on "version 2.1 or higher," and more and more of the newer programs require DOS v3.0 or better. This is particularly true for networkable programs, which demand version 3.1. It is said that "the person who knows not of foreign languages knows nothing of his own." The same could be said of MS-DOS, and this reading list should render MS-DOS a little less "foreign."

This bibliography is not all-inclusive, and it reviews no journals. Furthermore, the listing of a particular book does not imply endorsement of its

contents. However, all these books *are* aimed at power users and developers. Consumer-oriented publications and beginner level manuals are deliberately omitted, hopefully, to leave the serious programmer with a point of departure for developing a personal library of professional MS-DOS literature. The degree of value of any individual work will be left with the reader. There is a trend in computer book publishing to issue everything in paperback (noted as "paper" in the following list). This keeps the price down, but makes a book vulnerable to wear in cases of repeated use. If you are going to use a particular book as a reference tool, spend an extra five dollars and have it hardbound.

Take a moment also to examine the title page. The computer book publishing industry, like the PC technology itself, is volatile. Many books are issued as new editions each year, and most of them logically follow the issuance of new versions of DOS. You'll pay a dollar or two more for the latest edition, but you're buying insurance that the book is in sync with the latest DOS on the market. Even so, don't ignore the older editions if you're bargain shopping; there are many good values to be found on bookstores' remainder tables.

Some of the titles include a program or utility disk as part of the cover price. Others offer a supplementary disk, which generally must be mail ordered from the publisher. The latter are noted in the bibliography by "companion disk available."

■ **Alonso, Robert.** *QuickC DOS Utilities*. New York: Wiley, 1988. Paper, 258 pages, $19.95.

While the more advanced developmental programmer may prefer the power of one of the standard compilers, *QuickC DOS Utilities* melds nicely with persons wishing to hone their skills in a more user-friendly setting. The manual contains material on directory, file, and printer utilities to be used directly in the QuickC programming environment and a considerable discussion of routines applicable to computer security and performance. The book is a worthy springboard to more involved coding.

■ **Alperson, Burton L.** *The Fully Powered PC*. New York: Brady, 1988. Revised edition. Paper, 641 pages, $39.95 (includes two 5 1/4" disks).

With the necessary emphasis on DOS, the text presents utilities and instructional methodologies for customizing a personal computer to run faster and to automate some program tasks. Features batch file operations, memory management, and macro processing. Required to run utilities: 640K RAM, dual 360K floppy drives or one 360K floppy/one hard drive, MS-DOS or PC-DOS version 3.0 or greater, Prokey or Smartkey (optional), and a PC/XT/AT/PS/2 or compatible machine.

■ **Angermeyer, John... [et al.].** *The Waite Group's MS-DOS Developer's Guide*. Indianapolis: Howard Sams & Company, 1988. Second edition. Paper, 783 pages, $24.95.

The previous edition was published as *The MS-DOS Developer's Guide*. The focus in this edition is on structured, modular DOS programming. There are included many of the usual topics found in advanced MS-DOS books, written with a more sophisticated slant which should be appreciated by the serious programmer. Discussed are: TSR programs, serial port interfacing, the writing of installable device drivers, graphics programming (emphasizing EGA and VGA screens), and expanded (EMS) memory management. The not-so-usual is here, too: real time programming, programming the Intel Numeric Processing Extension, accessing undocumented interrupts and DOS functions, and recovering data lost in memory. Includes a generous amount of material on the difference between MS-DOS versions.

■ **Angermeyer, John, Fahringer... [et al.]** (a.k.a. The Waite Group). *Tricks of the MS-DOS Masters*. Indianapolis: Howard Sams & Company, 1987. Paper, 542 pages, $24.95.

Presents out-of-the-ordinary methodologies to coax ultimate performance from MS-DOS. Topics include: tree-structured wizardry for quicker directory accesses, screen manipulations, and the entering of control characters into text files. Also includes use of *DEBUG*, add-on software, and accessory boards for maximum benefit of the operating system.

■ **Bursch, David D.** *DOS Customized: Create Your Own DOS Commands for the IBM PC, XT, AT and Compatibles*. New York: Brady, 1987. Revised edition. Paper, 326 pages, $19.95.

The previous edition was published as *PC-DOS Customized*. This text allows the user to invoke a specific powerup sequence of screen colors and

user-defined function keys. It also instructs how to enable one DOS command to call a second without utilizing version 3.3's CALL structure. There is also material for building your own menu system and a DOS shell that bypasses the DOS prompt. Considerable emphasis is on parameter passing. Special features incude the creation of interactive commands that can accept user input, the writing of new directory commands, and use of the environment to create a PC security system. The latter allows building of user lists and passwords, and allows the supervisor to manipulate users' privileges. A generous amount of code examples are included, many in the form of DOS batch files.

■ **Campbell, Joe.** *C Programmer's Guide to Serial Communications*. Indianapolis: Howard Sams & Company, 1987. Paper, 655 pages, $24.95.

At first glance, this *Guide* might seem to offer "all things to all programmers" on the subject of PC data communications. Given its depth of material in RS-232 interfacing, modem protocols, the 8250/Z80 SIO UART's, timing functions and error detection — it may do just that! The DOS programmer must first consider the "PC DOS Assembly Language Interface" appendix to be able to access the many C library code examples. From there, the C programmer is shown the intricacies of Hayes/Smartmodem functions, the details of XMODEM/CRC, and the fundamentals of modems in general. A unique feature is the discussion of interfacing DOS machines to CP/M boxes. Because of the gathering of material from many sources, the *C Programmer's Guide to Serial Communications* is destined to become a classic.

■ **Chesley, Harry R. and Waite, Mitchell** (a.k.a. The Waite Group). *Supercharging C With Assembly Language*. Reading, MA: Addison-Wesley Publishing Company, 1987. Paper, 402 pages, $22.95.

It is often easier and more efficient to address MS-DOS interrupt calls and ROM BIOS services by direct invocation of assembly code than by C language subroutines. "How to do it" is a common request made of technical support personnel, and Supercharging C with Assembly Language has spendidly reported many of the concerns. Chesley and Waite assume more than a little reader sophistication; this isn't an assembly language guide for beginners. Much initial emphasis is on calling conventions — each compiler has its own "quirks." From there, discussion and examples are lent to very technical subjects such as fractal geometry, asynchronous communication,

direct screen addressing, and file encryption. The example code (Intel 8086-based) is fundamental and pragmatic, leaving the programmer to add the bells and whistles. The illustrations are clear and informative. Any C language programmer who uses MASM should keep this book within reach at all times.

■ **Dettmann, Terry R.** *DOS Programmer's Reference*. Carmel, IN: Que Corp., 1989. Second edition (revised by Jim Kyle). Paper, 892 pages, $27.95.

Includes both PC-DOS and MS-DOS. An encyclopedic reference which details the DOS functions and utilities in a manner which assumes some background knowledge on the part of the reader. In short, the book is a compendium of technical information which would not necessarily be understood by beginning or applications users, but which is invaluable to the developmental programmer for its degree of detail. Uses a style and format similar to Que Corporation's books on C language programming, and is worthwhile for the degree of accuracy in its examples.

■ **DeVoney, Chris with Hale, Norman.** *DOS Tips, Tricks, and Traps*. Carmel, IN: Que Corp., 1989. Paper, 522 pages. $22.95.

The emphasis is on versions 3.3 and 4.0, both MS-DOS and PC-DOS. As it is with many of Que's publications, this text has a definite hardware slant. As such, it provides a unique approach to disk drive and memory manipulation via DOS. This is the Year of the Advanced Batch File, and DeVoney can't resist the temptation occasionally to use the medium. Discussed are expanded and extended memory, in the context of LIMulators, expansion busses, and linear vs. real memory. Disk buffering routines, cachers, and defragmenters are here, too. Also, data transfer rates, interleaving, shockproofing, and sophisticated disk backups are written up. Disk maintenance is outlined as "troubleshooting physical damage." You'll also find some useful end-of-day processes as well as some variations on the CONFIG.SYS file, along with some classy device drivers.

■ **Duncan, Ray.** *Advanced MS-DOS Programming: The Microsoft Guide for Assembly Language and C Programmers*. Redmond, WA: Microsoft Press, 1988. Second edition. Paper, 669 pages, $24.95 (companion disk available for $15.95).

Topics include: disk file and record operations, memory management, the *EXEC* function, and the MS-DOS environment. Specializes in outlining the Lotus-Intel-Microsoft (LIM) expanded memory specification, device drivers, the ROM BIOS, and directory/file matters globally described as "disk internals." More than a third of the book is a reference on MS-DOS interrupts, including the *INT67h* EMS series, with code (assembler) examples on the use of each. Utilities include a terminal emulation program, a DOS shell, and a customized critical error interrupt handler.

■ **Duncan, Ray.** *The MS-DOS Encyclopedia*. Redmond, WA: Microsoft Press, 1988. 1,529 pages, $134.95; paper, $69.95.

A comprehensive and authoritative reference on MS-DOS commands, directives, utilities, and systems calls. An annotated volume of technical specifications, featuring C-callable, assembly language routines. All the user commands are included, as are the function calls, with version specific descriptions and usage information on each. Features system management, interrupt handling, keyboard and ANSI.SYS control, and insurance of compatibility of MS-DOS programs liable to be run under OS/2's real mode. Exhaustive and authoritative.

■ **Duncan, Ray.** *MS-DOS Functions*. Redmond, WA: Microsoft Press, 1988. Paper, 122 pages, $5.95.

A ready reference (handbook) tool. Lists the DOS interrupt functions and error codes, with appropriate (register) calling conventions. No sample code segments are given, but supplemental information is footnoted where warranted. Essential for system calls from assembly language.

■ **Forney, James.** *MS-DOS Beyond 640K: Working With Extended and Expanded Memory*. Blue Ridge Summit, PA: Windcrest, 1989. Paper, 235 pages, $19.95.

Exhaustive treatment of memory management, with emphasis on device drivers and hardware descriptions. Surprisingly, the author did not include much code; this is a book to be studied for its richness of narratives and diagrams. Details the LIM 3.2 and 4.0 standards, and describes the architecture of memory cards such as Intel's AboveBoard and AST's RAMpage. Some of the book is tangential to EMS per se, but the departures are valuable: the virtual control program interface (VCPI) is outlined, as are other DOS

extenders, hard disk interleaving, multitasking, and accelerator cards. DOS 4.X is considered in context of memory management, as are PC-MOS/386, Concurrent DOS, and UNIX. The author's preference for extended memory is subtle in this one-of-a-kind text.

■ **Gliedman, John.** *Tips and Techniques for Using Low-Cost and Public Domain Software*. New York: McGraw-Hill, 1989. Paper, 387 pages, $24.95.

The title disguises the book's main contents. Herein lies an exhaustive compendium of utility programs for MS-DOS enhancement. The focus is on versions 2.XX and 3.XX, and there are descriptions of programs for keyboard control, hard disk management, hardware speedups, file control/archiving, mouse manipulation, multitasking, data communications, and protection from Trojan horses and viruses. There is even some discussion on interfacing CP/M and Z80 and V20 microprocessors. Not much sample code is presented, however, and the programs described are not always free or inexpensive. A valuable reference for unusual and often undetected programs.

■ **Goodell, Thomas.** *DOS 4.0: Customizing the Shell*. Portland, OR: MIS: Press, 1989. Paper, 371 pages, $22.95.

A handbook for taking advantage of the new DOS version 4.0 menu-driven shell. Expands on the *SELECT* command and outlines its context sensitivity, and details the *MEM* command which maps the entire contents of the PC's memory. Explains the *SWITCHES* command and its techniques for allowing an extended keyboard to emulate a conventional keyboard. Details how version 4.0 breaks the old 32 megabyte hard disk volume limitation, as well as how 4.0 relates to the LIM expanded memory standard. A must for DOS users who wish to personalize the 4.0 work environment.

■ **Gookin, Dan.** *Advanced MS-DOS Batch File Programming*. Blue Ridge Summit, PA: Windcrest, 1989. Paper, 385 pages, $24.95 (companion disk available for $24.95).

An intensive treatment of the use of batch commands as elements of a programming language. Covers structured programming, use of *ERRORLEVEL*, environment manipulation, hard disk strategies, and keyboard enhancements. Shows how to tweak *ANSI.SYS*, *AUTOEXEC.BAT*, and *CONFIG.SYS* to

maximum advantage. OS/2 is included, and MS-DOS through version 4.0 is discussed. The programs in this book would be ideal candidates for use with Wenham Software's Batcom or Hyperkinetix's Builder batch file compilers.

■ Harriman, Cynthia W. with Hodgson, Jack. *The MS-DOS—Mac Connection: Data Sharing, Networking, and Support in the Mixed Office*. New York: Brady, 1988. Paper, 348 pages, $21.95.

Not a book on local area networks—LAN's actually take up a small part of this work on data translation and resource sharing. The text discusses external DOS drives for the Mac, coprocessing, and disk formats/swapping. Considered also are conversions between both text/program and graphics files. Two chapters compare and contrast the MS-DOS/Macintosh operating systems, noting both command and keyboard equivalences. The appendix features a lengthy product list of resources to effect data transfer solutions. The book is sufficiently authoritative to convince you that a Mac/DOS marriage could just work.

■ **Held, Gilbert.** *DOS Productivity Tips and Tricks*. New York: Wiley, 1989. Paper, 286 pages, $22.95.

This is more of a utility manual than a developer's book, but the information is useful and comprehensive. The emphasis is on productivity routines — some twenty in all, mostly based on DOS batch files. Included are instructions for creating your own VDISK or RAM disk, master menus, a usage log, and a rudimentary data base. You can also effect time-dependent program execution and write your own DOS commands. The material further contains screen controls, color setting routines, display switching commands, and screen dumps. Additionally, you can change printer modes, swap ports, and make mailing labels. There is a program to remap the function keys and one to create a calculator. Also detailed are routines for exit code processing and the elimination of disk fragmentation.

■ **Holub, Allen.** *On Command: Writing a UNIX-Like Shell for MS-DOS*. Redwood City, CA: M&T Publishing, 1987. Second edition. Paper, 319 pages, $39.95 (includes 5 1/4" disk).

The book includes a complete listing of C code for working under a UNIX emulating shell. The feature program, *SH*, is presented as an interpreter. More for comparison study than developmental programming, but

an excellent tutorial for those who must occasionally switch from DOS to UNIX or vice versa. Disk utilities require an IBM compatible PC, 256K RAM, and MS-DOS version 2.1 or greater.

■ **Hyman, Michael.** *Advanced DOS*. Portland, OR: MIS: Press, 1989. Paper, 363 pages, $22.95 ($44.95 if purchased with the 5 1/4" utilities disk).

Covers DOS through v4.0. Deals with some topics not found in many advanced DOS manuals, including programming to manipulate the boot sector, the file allocation table, and the program segment prefix, as well as parameter passing and the exploration of the root/subdirectories. The writing of popup utilities and keyboard triggered routines are discussed in the context of "program cooperation" and multi-tasking. The book also discusses memory resident programs (TSR's), disk sector management, BIOS calls and DOS interrupts. Examples are given in assembly language and Turbo Pascal v5.0.

■ **Jamsa, Kris.** *DOS Power User's Guide*. Berkeley, CA: Osborne/McGraw-Hill, 1988. Paper, 921 pages, $22.95.

Written to allow maximum performance from DOS. The emphasis is on pipes and filters, with a thorough excursion through the PC's memory map. Includes both MS-DOS and PC-DOS. Code samples include a routine for erasing deleted files and for mapping disk layouts. Speaks of the DOS pretender commands, and delves into programming for OS/2, using the latter's Application Program Interface (API). Many of the book's examples are written in Turbo Pascal.

■ **Jamsa, Kris.** *DOS: The Complete Reference*. Berkeley, CA: Osborne/McGraw-Hill, 1987. Second edition. Paper, 1,046 pages, $27.95.

Not the most advanced work on DOS, but a comprehensive foundation for later departure. Covers advanced commands, customizing the shell, redirection, and ANSI drivers. Lengthy discussion of the *LINK* and *DEBUG* utilities. Valuable for its extensive treatment on the use of DOS with Microsoft Windows. Not much on the anatomy of DOS. Both PC-DOS and MS-DOS through v3.X are included. A comprehensive desktop resource.

■ **Jamsa, Kris.** *MS-DOS Batch Files*. Redmond, WA: Microsoft Press, 1989. Paper, 166 pages, $6.95.

Also a ready reference (handbook) tool. Contains the essentials of batch file programming with MS-DOS. This is a barebones guide, but it presents considerably more material than found in the DOS user's manual. There are enough examples to create meaningful batch files and to take advantage of batch file utilities in applications program environments.

■ **Kamin, Jonathan.** *MS-DOS Power User's Guide, Volume I.* Berkeley, CA: Sybex, 1987. Second edition. Paper, 182 pages, $21.95.

Covers MS-DOS versions through 3.3. Discusses undocumented MS-DOS commands, pipes and filters, as well as extensions to what the book calls the DOS "programming language" found in each successive DOS version. The book also considers the *ANSI.SYS* driver for customizing screens and keyboards, nationalization of keyboards for foreign languages, alternate date/time and currency formats, recovery methods for damaged disks, and batch files with emphasis on the use of replaceable parameters. Much attention is given to *DEBUG* as a programming tool, and many examples in assembly language are provided to be coded/altered with the *DEBUG* utility.

■ **King, Richard Allen.** *The MS-DOS Handbook.* Berkeley, CA: Sybex, 1988. Third edition. Paper, 362 pages, $19.95.

An appropriate label for this work might be "the beginner's technical manual." Introductory material on serial/parallel port manipulation, disk maps, nonstandard disk drives, the keyboard, video I/O and ROM BIOS calls. Considers file recovery, RS-232 communications, memory management, error handling, and interrupts. The text contains user-level discussions of DOS only for the advanced functions. An excellent transitional book for programmers no longer requiring the basics, but in need of preambulatory material to work up to power MS-DOS programming.

■ **King, Richard Allen.** *The IBM PC-DOS Handbook.* Berkeley, CA: Sybex, 1988. Third edition. Paper, 359 pages, $19.95.

Includes material on the expanded command summary; fills in the pieces for PC-DOS not covered in King's other book (above).

■ **Krumm, Robert.** *Getting the Most From Utilities on the IBM PC: Perfecting the System Environment.* New York: Brady, 1987. Paper, 520 pages, $22.95.

The emphasis in this work is on a few commercially available programs, but the book contributes some tips and hints not found in user manuals. Features the Norton Utilities, ProKey, SuperKey, Smartkey, Turbo Lightning, DeskSet, others. Discusses file handling, macro invocation, environment manipulation, and keymapping. Not a first purchase, but valuable for the insights the author has gained from experience using DOS utilities.

■ **Lai, Robert S.** (a.k.a. The Waite Group). *Writing MS-DOS Device Drivers*. Reading, MA: Addison-Wesley Publishing Company, 1987. Paper, 466 pages, $24.95.

Begins with a lengthy discussion of device drivers in general, providing a motivation for the programming of one's own prototype drivers. The lectures focus on the console and printers, as well as on the system drivers for these pieces of hardware. Much emphasis is on creating an IOCTL driver (code included), expanding on the standard printer driver that DOS provides. Of particular interest is a chapter on creating a RAM disk device driver. The book concludes with *"Building a Complete Full- Function Device Driver"* to allow the reader to experiment with custom design.

■ **Mikes, Stephen.** *UNIX for MS-DOS Programmers*. Reading, MA: Addison-Wesley Publishing Company, 1989. Paper, 500 pages, $24.95.

Compares and contrasts the MS-DOS and UNIX operating systems, detailing their similarities and differences by functions. Focuses on the UNIX file system and I/O subsystem, shell programming, processes, multitasking, the UNIX keyboard, and screen techniques. For systems programmers, an essential cross reference for noting the subtle elements common to the two systems.

■ **Miller, Alan R.** *DOS Assembly Language Programming*. Berkeley, CA: Sybex, 1988. Paper, 365 pages, $24.95.

While it covers DOS only through v3.3, *DOS Assembly Language Programming* nicely marries assembler macro programming to DOS system functions and interrupt handling. Its value lies in the assembler code segments that teach system calls one function at a time, as examples that can be incorporated into more detailed programs. The emphasis is on disk I/O, keyboard control, screen and port addressing, and memory handling. Several projects reinforce the learning: Wordstar to ASCII file conversion,

printer typeface manipulation, increasing memory to 704K bytes, and changing the default colors of Lotus 1-2-3. Read this book if you're tired of wading through the MASM user's manual.

■ **Norton, Peter.** *Peter Norton's DOS Guide: Revised and Expanded.* New York: Brady, 1989. Third edition. Paper, 408 pages, $19.95.

More of an intermediate level user's manual, but written in light of Peter Norton's usual experiences with PC software. Features batch file programming and insights on advanced commands. A no-nonsense manual, probably of best value to mainframe programmers seeking a switchover to DOS and as a textbook.

■ **Richardson, Ronny.** *MS-DOS Batch File Programming... Including OS/2.* Blue Ridge Summit, PA: Windcrest, 1988. 300 pages, $25.95; paper, $17.60.

The basics, but considerably more on batch files than found in the DOS user's manual. Covers menuing, environment handling, looping, and passing of replaceable parameters. Explains each batch command in detail. Use this text as an introduction to Gookin's book (above).

■ **Richardson, Ronny.** *MS-DOS Utility Programs: Add-On Software Resources.* Blue Ridge Summit, PA: Windcrest, 1989. Paper, 665 pages, $24.95.

Richardson has performed some exhaustive research in compiling this annotated compendium of MS-DOS specific tools. There isn't much consumer-oriented material here; this is a reference book for developers and systems managers. Features menu systems, alternative DOS shells, disk optimizers and console accelerator software, document indexers and data translation utilities. Data protectors, copy unprotectors, and anti-viral programs are here, too. TSR managers, file compressors, DOS command enhancers, and expanded memory emulators round out the collection. Each program is adequately described, and many are illustrated with screen snapshots to emphasize their features. Some of the programs are shareware, but most are for sale—you'll pay for the privileges. Browse through this book as you would with the Sears & Roebuck catalog, and be pleasantly enlightened with what you find.

■ **Rochkind, Marc J.** *Advanced C Programming for Displays, Character Displays, Windows, and Keyboards for the UNIX and MS-DOS Operating*

Systems. Englewood Cliffs, NJ: Prentice-Hall, 1988. Paper, 331 pages, $34.95.

Unless married to a specific graphics environment, developers with a need for screen service routines will want to give this book serious consideration. A good starting point for the basics of graphics displays, particularly where the programmer must be loyal to a dual DOS/UNIX environment. Presents the techniques and algorithms to support code examples, with the intent of combining portability with efficiency.

■ **Ross, Steven S.** *Data Exchange in PC/MS-DOS*. New York: McGraw- Hill, 1990. Paper, 411 pages, $27.95.

This book collects data regarding the data coding schemes of popular applications programs, and much of the information isn't found anywhere else. Although there is not much sample code, the text provides enough material on the coding schemes of word processors, spreadsheets, and database managers that the professional programmer will have little difficulty writing proprietary conversion software to switch back and forth between several dozen applications formats. EMACS and ASCII systems are considered, and many of the examples are given as sample applications screen illustrations. There is also material on conversion by hardware, conversion by media (disk format) exchange, and conversion between DOS-based and Macintosh systems. For those who would write software conversion programs, some examples are given in BASIC and C language, as well as techniques for performing file conversions using ***EDLIN***, ***DEBUG***, and the Norton Utilities.

■ **Scanlon, Leo J.** *Assembly Language Subroutines for MS-DOS Computers*. Blue Ridge Summit, PA: TAB Books, 1986. Paper, 332 pages, $19.95 (companion disk available for $24.95).

Although much of the book emphasizes PC architecture, there is a relatively significant association with MS-DOS to warrant inclusion in this bibliography. The presentation is an abbreviated group of assembly language instructions, with considerable detail of each one, accompanied by code examples. Particular to DOS, the date/time functions are considered, along with disk drive operations, IBM PC-specific I/O, subdirectory operations, and disk file operations. Some non-DOS specific material includes: 16- and

32-bit arithmetic, ordered and unordered list manipulation, code conversions (binary, BCD, and hex), sorting, and string manipulations. Most of the assembler instructions discussed are incorporated into useful macro routines. The utility disk requires 256K RAM, an IBM PC/XT/AT or compatible, and an IBM or Microsoft macro assembler to run.

■ **Simrin, Steven.** *The Waite Group's MS-DOS Bible.* Indianapolis: Howard Sams & Company, 1989. Third edition. Paper, 630 pages, $22.95.

Covers MS-DOS through v4.0. An encyclopedia of syntax, examples, and descriptions of each of the MS-DOS resident and transient commands. Includes discussions of memory management (EMS), file management, installable device drivers, DOS interrupts and functions. Examples are written in assembly language.

■ **Somerson, Paul.** *PC Magazine DOS Power Tools.* New York: Bantam, 1988. Paper, 1,275 pages, $44.95 (includes a 5 1/4" disk).

Tons of Assembler and BASIC code, collected from past issues of PC Magazine. The book is a virtual flea market of screen utilities, memory managers, drivers, and DOS enhancers. Somerson likes *DEBUG* and *EDLIN* and waxes eloquent on their uses and power. "Many hours of enjoyment," as the saying goes; if only two or three of the hundreds of the book's routines prove worthwhile for your particular environment, the capital outlay for *Power Tools* will have been money well spent.

■ **Tischer, Michael.** *PC System Programming for Developers.* Grand Rapids, MI: Abacus, 1989. Paper, 928 pages, $59.95 (includes two 5 1/4" disks).

Written for the professional programmer, with no extraneous lecture or fluff. The examples comprise straightforward, working code segments with few adjectives in the explanations. Topics include: memory organization, use of extended/expanded memory, interrupt handling, *.COM* and *.EXE* programs, character I/O, TSR writing, and creation of device drivers. Details DOS structures and thoroughly discusses BIOS fundamentals. Gives a considerable amount of space to graphics programming, including the intricacies of EGA and VGA cards. The author expects that the DOS systems programmer will interface to Turbo Pascal, assembly language, BASIC, or to one of several brands of C compilers.

■ **Townsend, Carl.** *Advanced MS-DOS Expert Techniques for Programmers*. Indianapolis: Howard Sams & Company, 1989. Paper, 597 pages, $24.95.

Includes DOS v4.0. The value of this advanced work is the variety of examples of DOS-targeted programming in C (Microsoft version 5.1, Turbo C and QuickC), as well as in Microsoft's macro assembler (MASM). Discusses methods for calling a child program from a parent, interfacing applications programs with the operating system, the use of hardware resources, the recognition of and protection against viruses, and the writing of device drivers and TSR's. Also discusses the pros and cons of developers' use of Microsoft Windows with their MS-DOS based programs. An unusual feature is material on the use of disassemblers, and several different debugging tools—the Turbo debugger, *SYMDEB*, and, of course, *DEBUG* — are considered.

■ **The Waite Group.** *The Waite Group's MS-DOS Papers For MS-DOS Developers and Power Users*. Indianapolis: Howard Sams & Company, 1988. Paper, 400 pages, $26.95.

Thirteen experts lecture on heretofore undocumented features of the MS-DOS operating system. Included are techniques and details of the MS-DOS user interface, along with some obscure programming tools and methodologies. A recurring theme is the addressing and invocation of hardware via MS-DOS and the interaction with various types of devices.

■ **Waterhouse, Martin, and Kamin, Jonathan.** *MS-DOS Power User's Guide, Volume II*. Berkeley, CA: Sybex, 1988. Paper, 424 pages, $19.95.

The companion to Kamin, Volume I (above), extended to cover MS-DOS v4.0.

■ **Wolverton, Van.** *Running MS-DOS*. Redmond, WA: Microsoft Press, 1988. Third edition. Paper, 478 pages, $22.95.

Relatively elementary, but this work is included for the teachers in the audience. It is so well organized that it is invaluable as a textbook. Includes the usual tutorial on the basic transient and resident commands (including the *EDLIN* utility), but concludes with three chapters on "smart" commands. As such, provides a better discussion of batch files than the DOS user's manual, and offers a good transition into *Supercharging MS-DOS* (below).

■ **Wolverton, Van.** *Supercharging MS-DOS.* Redmond, WA: Microsoft Press, 1989. Second edition. Paper, 336 pages, $34.95 (includes a 5 1/4" disk).

Includes versions of MS-DOS through 4.0. The utility disk includes all the batch files, script files, and programs outlined in the book. Discusses automatic definition of RAM disks, the modification of *ANSI.SYS*, file/program contents manipulations, and customization of *CONFIG.SYS*. Also includes system environment modification, interactive menuing, and a detailing of *DEBUG*.

■ **Young, Michael J.** *MS-DOS Advanced Programming.* Berkeley, CA: Sybex, 1988. Paper, 490 pages, $22.95.

Extends the performance of DOS by allowing maximum use of system resources. Covers functions, interrupts, devices, multitasking, and memory resident programs. Features DOS 3.3 and 4.0, giving examples in both assembly language and C language. Includes discussion of real mode OS/2 (the compatibility box).

■ **Young, Michael J.** *Systems Programming in Microsoft C.* Berkeley, CA: Sybex, 1989. Paper, 604 pages, $24.95 (companion diskettes available for $34.95, or $39.95 for Turbo C or Turbo Pascal versions with documentation).

A companion to Young's other book (above), gives the code and details for several dozen callable C language functions. Although primarily a book on C, the chapters' library actually comprises a complementary set of low level DOS routines. Included are subprograms to handle expanded memory functions, interrupts, keyboard and mouse functions, TSRs, graphics and video functions, and printer functions. The set is curiously similar to that found in the OS/2 API. The library allows some insight into interfacing with assembly language, and includes some miscellaneous utility functions for string and time/date manipulations that Microsoft C doesn't provide. Compatible with MSC v5.0 and v5.1 as well as QuickC v1.0 and v2.0.

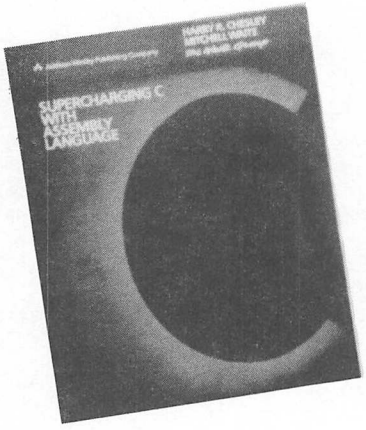
Subsequent to printing we discovered a pagination error that causes many page references in the index to be incorrect. In each case the reference may be corrected by adding an offset:

52-58	add	1 page
59-68	add	2 pages
69-88	add	3 pages
89-92	add	13 pages
93-100	add	14 pages

We suggest that you adhere this label to the top of page 141. We apologize for the inconvenience this causes.

Index

Tech Bookshelf



Supercharging C With Assembly Language

By Harry R. Chesley, Mitchell Waite, and The Waite Group

Written for programmers who want to create faster, more powerful C programs on PC clones, this book offers techniques for combining C and assembly language to reach their goal. The authors show the impact of alternate optimization strategies by successive refinements to each of several non-trivial applications including a terminal emulator, a fractal program, and encryption utilities. In the process they give detailed information about interrupt handlers, the ROM BIOS, MS-DOS disk I/O, video adapters, and the asynchronous communications adapter.

Addison-Wesley, 402 pp., ISBN 0-201-18349-8.

Y35 . . . . . .$22.95

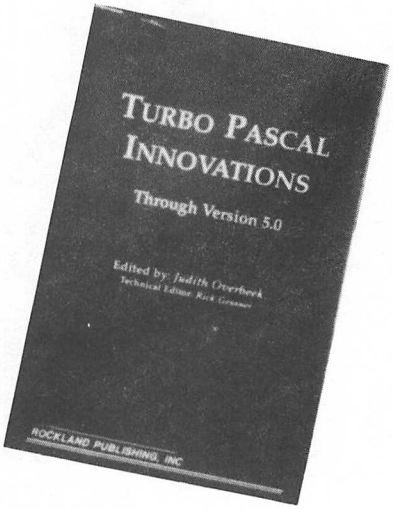
Turbo Pascal Innovations (Through Version 5.0)

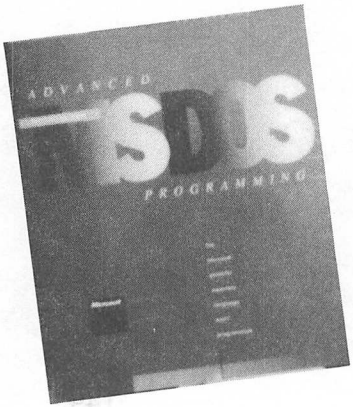
Edited by Judith Overbeek

Turbo Pascal Innovations provides practical information for intermediate to advanced Pascal programmers. Chapters by seven different authors cover object-oriented programming, TSRs, user interfaces, graphics, and DOS subdirectory routines. Also included is an MS-DOS disk with all code presented in the book.

Rockland Publishing, Inc., 346 pp., ISBN 0-939621-01-0.

Z30 . . . . . $32.95





Advanced MS-DOS (2nd Edition)

By Ray Duncan

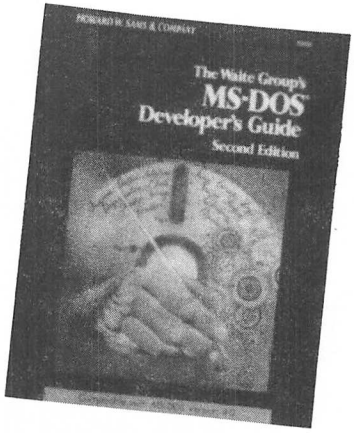
This reference for advanced assembly language and C programmers covers the essentials of MS-DOS. Duncan discusses disk file and record operations, disk directories and volume labels, MS-DOS disk internals, installable device drivers, memory management, and more. Also included is an extensive reference section on all MS-DOS functions and interrupts.

Microsoft Press, 468 pp., ISBN 1-55615-1578.

Z28 . . . . . $24.95

MS-DOS Developer's Guide (2nd Edition)

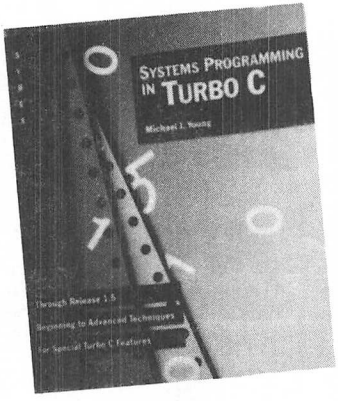
By The Waite Group



Compatible with MS-DOS v4.0, MASM 5.1 and Microsoft C 5.1, this edition represents an expanded revision of the original MS-DOS Developer's Guide. The latest edition contains information about undocumented functions and interrupts for MS-DOS v2.0 and 4.0, file I/O handles, TSR programs, Enhanced Graphics Adapter and Virtual Graphics Array display standards, programming the serial port, EMS, and more.

Howard W. Sams and Company, 783 pp., ISBN 0-672-22630-8.

Z29 . . . . . $24.95



Systems Programming In Turbo C

By Michael J. Young

Systems Programming In Turbo C offers tools to optimize the use of Turbo C for applications and systems-level programming. For the applications programmer it contains an extensive collection of software tools that can be immediately incorporated into a C program. Systems programmers can make use of the complete source code listing and detailed explanations of Turbo C's advanced features. All functions are designed to operate on IBM PCs, ATs and compatibles, and PS/2 series computers. The book covers Turbo v1.0 and 1.5.

Sybex, 503 pp., ISBN 0-89588-467-4.

Z22 . . . . . .$24.95

Quantity	Code #	Title	Unit Price	Total

All payments *must* be in US dollars (MC/VISA accepted).
Note: Call (913) 841-1631 for special shipping.

Shipping (North America)	$3.50
Overseas shipping (add 45%)	
TOTAL	

Name _____

Company _____

Address _____

City _____

State _____ Zip/Mail code _____

Country _____

Daytime phone _____

☐ MC ☐ VISA Exp. date _____

Card Number _____

Signature _____

Tech Bookshelf

TSP

Mail to: R&D Publications
2601 Iowa St.
Lawrence, KS 66046
Or call: (913) 841-1631