Experiment N^o 9

Interrupts

Introduction:

On the 80x86, there are three types of events commonly known as interrupts: *traps*, *exceptions, and interrupts* (hardware interrupts). In this experiment we will describe each of these forms and discuss their support on the 80x86 CPU's and PC compatible machines. We will also describe Interrupt Service Routines (ISR) and Terminate and Stay Resident (TSR) programs. A TSR is a program that remains in memory after execution.

The purpose of a TSR is to install an interrupt hook. This experiment illustrates the installation of TSR software using interrupt hooks and hot-key sequences. In most cases, a TSR is activated by either an INT 8 clock tick, or a hot-key sequence.

Objectives:

1- Install, use and uninstall an Interrupt Service Routine (ISR).
2- Install interrupt service procedures (hooks) as TSR software.
3- Install a TSR interrupt hook that uses the clock tick interrupt (INT 8).
4- Install a TSR interrupt hook that intercepts the keyboard and responds to a particular key code or hot-key.

References:

1. Barry B. Brey, "Programming the 80286, 80386, 80486, and Pentium- Based Personal Computer", Prentice Hall, (1996).

2. Randall Hyde, "The Art of Assembly Language Programming", http://webster.cs.ucr.edu/Page_asm/ArtofAssembly/ArtofAsm.html

Interrupts, Traps, and Exceptions:

On the 80x86, there are three types of interrupts: *traps*, *exceptions, and interrupts* (hardware interrupts). In the following we will describe each of these forms and discuss their implementation. Although the terms trap and exception are often used synonymously, we will use the term trap to denote a programmer initiated and expected transfer of control to a special handler routine.

Traps:

Traps, or software interrupts, are specialized subroutine calls invoked by a software-interrupt. The 80x86 ***`int`*** instruction is the main instruction for executing a trap. Since traps execute via an explicit instruction, it is easy to determine exactly which instructions in a program will invoke a trap handling routine. There are two main differences between a trap and an arbitrary far procedure call: the instruction used to call the routine (CALL vs. INT) and the fact that a trap pushes the flags on the stack so you the IRET instruction must be used to return from it.

The main purpose of a trap is to provide a fixed subroutine that various programs can call without having to actually know the run-time address. The `INT 21h` instruction is an example of a trap invocation. Programs do not have to know the actual memory address of DOS entry point to call DOS. Instead, DOS patches the interrupt 21h vector when it loads into memory. When `an INT 21h` is executed, the 80x86 automatically transfer control to DOS entry point.

Traps are used to call a resident program function. By patching an interrupt vector to point at a subroutine within the resident code, other programs that run after the resident program terminates can call the resident subroutines by executing the appropriate `INT` instruction.

Most resident programs do not use a separate interrupt vector entry for each function they provide. Instead, they usually use a single interrupt vector and transfer control to an appropriate routine through a function number that the caller passes in a register, usually and conventionally the `AH` register. A typical trap handler would execute a case statement on the value in the AH register and transfer control to the appropriate handler function.

Exceptions:

An exception is an automatically generated trap, forced rather than requested, that occurs in response to some exceptional condition. Generally, there isn't a specific instruction associated with an exception, instead, an exception occurs in response to some erroneous behavior of normal program execution. Examples of conditions that may cause an exception include executing a division instruction with a zero divisor, executing an illegal opcode, and a memory protection fault. Whenever such a condition occurs, the CPU immediately suspends execution of the current instruction and transfers control to an exception handler routine. This routine can decide how to handle the exceptional condition; it can attempt to rectify the problem or abort the program and print an appropriate error message.

Exceptions occur when an abnormal condition occurs during execution. There are fewer than eight possible exceptions on machines running in real mode. Protected mode execution provides many others.

Although exception handlers are user defined, the 80x86 hardware defines the exceptions that can occur. The 80x86 also assigns a fixed interrupt number to each of the exceptions. **Table 10. 1** describes each of these exceptions in detail.

In general, an exception handler preserves all registers. However, there are several special cases where you may want to tweak a register value before returning. Nevertheless, you should not arbitrarily modify registers in an exception handling routine unless you intend to immediately abort the execution of your program.

INT #	Exception Function	Description
INT 0	Divide Error	Occurs whenever an attempt to divide a value by zero or the quotient does not fit in the destination register when using the `div` or `IDIV` instructions. The FPU's FDIV and FDIVR instructions do not raise this exception.
INT 1	Single Step (Trace)	Occurs after every instruction if the trace bit in the flag register is set. Debuggers will often set this flag to trace the execution of a program.
INT 3	Breakpoint	This exception is actually a trap, not an exception. It occurs when the CPU executes an `INT3` instruction. However, it is considered as an exception since programmers rarely put `INT 3` instructions directly into their programs. Instead, a debugger like CodeView often manages the placement and removal of `INT 3` instructions.
INT 4 **INTO**	Overflow	Like `INT 3`, this exception is technically a trap. It is raised when an `INTO` instruction is executed and the overflow flag is set. If the overflow flag is clear, the `INTO` instruction is a `NOP`. If the overflow flag is set, `INTO` behaves like an `INT 4` instruction. An `INTO` instruction can be inserted after an integer computation to check for an arithmetic overflow.
INT 6	Invalid Opcode	The 80286 and later processors raise this exception if an attempt to execute an opcode that does not correspond to a legal 80x86 instruction is made. These processors also raise this exception if you attempt to execute a bound, LDS, LES, LIDT, or other instruction that requires a memory operand but you specify a register operand in the mod/rm field of the mod/reg/rm byte.
INT 7	Coprocessor Not Available	The 80286 and later processors raise this exception if an FPU (or other coprocessor) instruction is attempted to execute without having the coprocessor installed. This exception can be used to simulate the coprocessor in software.

Table 10. 1: Exceptions

Hardware interrupts:

Hardware interrupts, or simply interrupts, are program control interruptions based on an external hardware event (external to the CPU). These interrupts generally have nothing at all to do with the instructions currently executing; instead, some event, such as pressing a key on the keyboard or a time out on a timer chip, informs the CPU that a device needs some attention. The CPU interrupts the currently executing program, services the device, and then returns control back to the program.

On the PC, interrupts come from many different sources. The primary sources of interrupts, however, are the timer chip, keyboard, serial ports, parallel ports, disk drives, CMOS real-time clock, mouse, sound cards, and other peripheral devices. These devices connect to an Intel 8259A programmable interrupt controller (PIC) that prioritizes the interrupts and interfaces with the 80x86 CPU. The 8259A chip adds considerable complexity to the software that processes interrupts, so it makes perfect sense to discuss the PIC first, before trying to describe how the interrupt service routines have to deal with it. Afterwards, this section will briefly describe each device

and the conditions under which it interrupts the CPU. This text will fully describe many of these devices in later chapters, so this chapter will not go into a lot of detail except when discussing the timer interrupt.

80x86 Interrupt Structure and Interrupt Service Routines:

An interrupt service routine is a procedure written specifically to handle a trap, exception, or interrupt. Although different phenomenon cause traps, exceptions, and interrupts, the structure of an interrupt service routine, or ISR, is approximately the same for each of these.

The 80x86 allow up to 256 vectored interrupts. This means that up to 256 different sources can exist for an interrupt and the 80x86 will directly call the service routine for that interrupt without any software processing. Non-vectored interrupts transfer control directly to a single interrupt service routine, regardless of the interrupt source.

The 80x86 provides a 256 entry interrupt vector table beginning at address 0000:0000 in memory. This is a 1Kbyte table containing 256 4-byte entries. Each entry in this table contains a segmented address that points at the interrupt service routine in memory. Generally, we will refer to interrupts by their index into this table, so the address (vector) of interrupt n is at memory location 0000:n*4. Interrupt zero's vector is at address 0000:0000, interrupt one's vector is at address 0000:0004, etc.

When an interrupt occurs, regardless of its source, the 80x86 does the following:

1. The CPU pushes the flags register onto the stack.
2. The CPU pushes a far return address (segment:offset) onto the stack, segment value first.
3. The CPU determines the cause of the interrupt, i.e., the interrupt number, and fetches the four byte interrupt vector from address 0000:vector*4.
4. The CPU transfers control to the routine specified by the interrupt vector table entry.

After completion of these steps, the ISR takes control. When the ISR wants to return control, it must execute an IRET (interrupt return) instruction. The interrupt return pops the far return address and the flags off the stack. Note that executing a far return is insufficient since that would leave the flags on the stack.

Hardware interrupts are processed differently than other types of interrupts. Upon entry into the hardware ISR, the 80x86 disables further hardware interrupts by clearing the interrupt flag. Traps and exceptions do not do this. If further hardware interrupts are to be disabled within a trap or exception handler, one must explicitly clear the interrupt flag with a clear interrupt flag instruction (CLI). Conversely, if interrupts are to be enabled within a hardware ISR, one must explicitly turn them back on with a Set Interrupt instruction (STI). Note that the 80x86's interrupt disable flag only affects hardware interrupts. Clearing the interrupt flag will not prevent the execution of a trap or an exception.

ISRs are written like almost any other assembly language procedure except that they return with an `IRET` instruction rather than `ret`. Although the distance of the ISR procedure (near vs. far) is usually of no significance, you should make all ISRs far procedures. This will make programming easier if you decide to call an ISR directly rather than using the normal interrupt handling mechanism.

Exceptions and hardware interrupts ISRs have a very special restriction: they must preserve all registers they modify.

DOS Memory Usage and TSRs

When DOS is first booted, the memory layout will look something like **Figure 10. 1**. DOS maintains a free memory pointer that points to the beginning of the block of free memory.

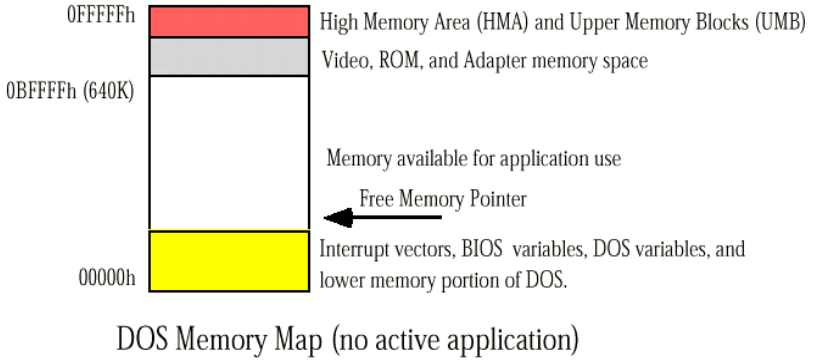


Figure 10. 1: DOS Memory with no active application

When an application program is run, DOS loads this application starting at the address the free memory pointer contains. Since DOS runs only a single application at a time, all the memory, starting from the free memory pointer to the end of RAM (0BFFFFh), is available for the application's use. When the program terminates normally, via DOS function 4CH, MS-DOS reclaims the memory in use by the application and resets the free memory pointer to just above DOS in low memory.

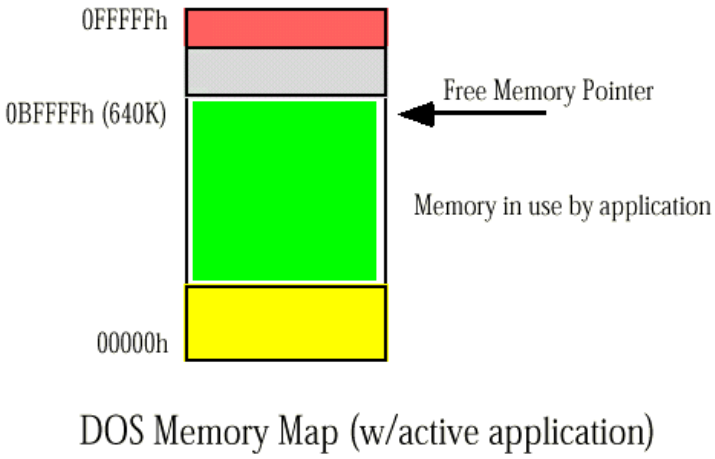
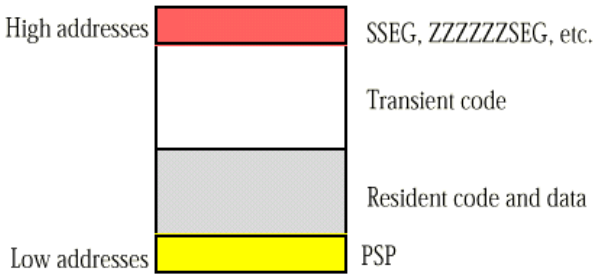


Figure 10. 2: DOS Memory with no active application

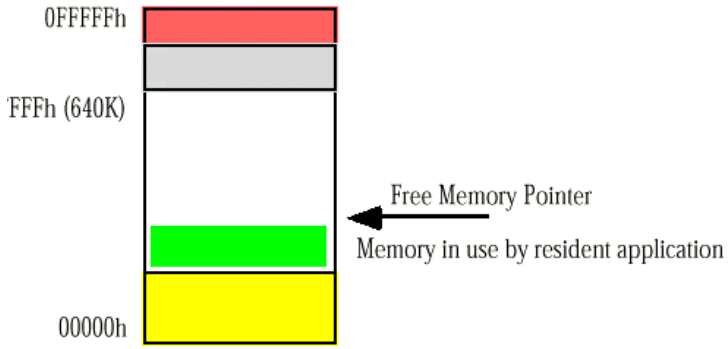
MS-DOS provides a second termination call which is identical to the terminate call e except that it does not reset the free memory pointer to reclaim all the memory in use by the application. Instead, this terminate-and-stay-resident call frees all but a specified block of memory. The TSR call (AH = 31H) requires two parameters, a process termination code in the AL register (usually zero) and DX must contain the size of the memory block in paragraphs to protect. When DOS executes this code, it adjusts the free memory pointer so that it points at a location DX*16 bytes above the program's PSP. This leaves memory looking like this:



Memory Organization for a Resident Program

Figure 10. 3: DOS Memory Organization for a Resident Program

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



DOS Memory Map (w/resident application)

Figure 10. 4: DOS Memory with a Resident Application

Writing and Installing a TSR program:

A TSR program consists of two parts an installation section and a service routine. The service routine may consist of more than one ISR. The installation section is executed only at load time. The ISR is executed each time the interrupt is invoked. An ISR must end with a IRET instruction or a FAR JMP to another ISR.

1 / Installing an ISR:

The installation section prepares the TSR program's service routine to be used by other programs or to service a hardware interrupt. The installation uses INT 21H Function 25H.

The following code is a typical installation section:

```
MOV AX, CS
MOV DS, AX
MOV DX, Offset Pgm_ISR
MOV AH, 25H
MOV AL, Int_Number
INT 21H
```

2 / Make an ISR a TSR:

This procedure uses INT 21H Function 31H. DX contains the number of paragraphs to be kept in memory. The following code makes the installed ISR a TSR routine.

```
MOV DX, Number_Of_Paragraphs
MOV AL, Return_Code
MOV AH, 31H
INT 21H
```

Function	Effect	Input	Output
25H	Set Interrupt Vector	DS:DX = Segment Offset Address	
31H	Make ISR as TSR	DX= number of paragraphs to be kept in memory	
35H	Get Interrupt Vector for a Specified Interrupt.	AL = Interrupt number	ES:BX = Segment Offset Address

Table 10. 2: ISR Function Manipulation

Notes:

1 - The code for an installation routine of a TSR should come after the service section, to ensure that it is freed by INT 21H Function 31H, which return control to DOS. Therefore, such a procedure does not need to have a RET instruction.

2 - The number of paragraphs to be kept resident in memory is found by dividing the number of bytes to be kept resident by 16. The following code performs such an operation:

```
MOV DX, Offset Installation_Routine
MOV CL, 04
SHR DX, CL                    ; Divide by 16
INC DX                        ; to take care of truncation if any
```

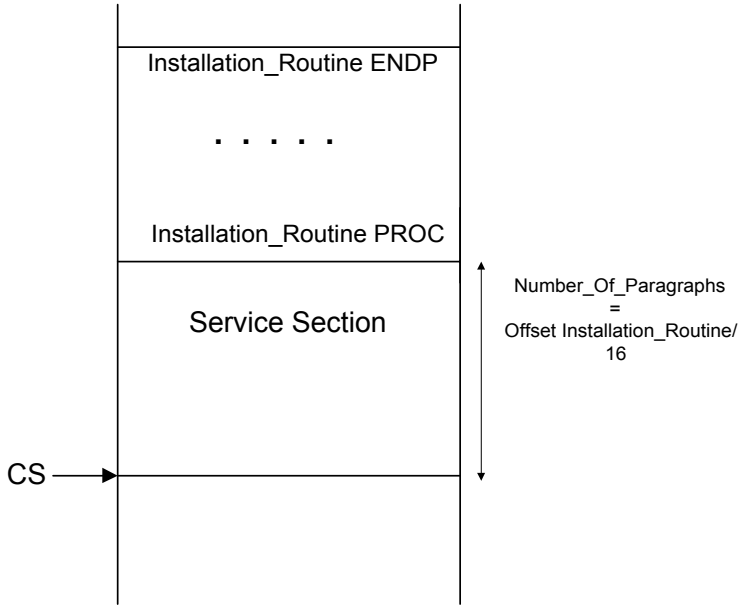


Figure 10. 5: Memory Structure after a TSR program is installed

User Defined TSRs:

The software vectors from 60H through 67H and those from 0F1H through 0FFH in the Interrupt Vector Table are undefined and available for user defined ISRs. Program 10.1 installs an ISR at INT 60H. From the time the ISR is installed until either the system is powered down or INT60H is re-vectored to address some other routine, any program that contains the sequence:

```
MOV AX, Operand
INT 60H
```

will display the message “Welcome To The Interesting World Of TSR Routines'.

If we change the body of program 10.1, in the following way:

```
DECIMAL_DISPLAY     PROC FAR
                    MOV SI, 0000
                    MOV BX, 10
```

```
DIV_LOOP:           MOV AX,operand
                    DIV BX
                    MOV CS:ARRAY[SI], DL
                    INC SI
                    CMP AX, 0000
                    JNZ DIV_LOOP

    DISPLAY_LOOP:   DEC SI
                    MOV AH, 02H
                    MOV DL, CS:ARRAY[SI]
                    OR DL, 30H
                    INT 21H
                    CMP SI, 00
                    JNZ DISPLAY_LOOP
                    POP SI
                    POP BX
                    POP AX
                    IRET
DECIMAL_DISPLAY     ENDP
```

In this case a call to INT 60H will display the contents of AX register (operand) on the console screen in decimal notation.
Another program, which displays a number passed through DX in different numbering systems will be given in the lab. The numbering system is decided by the value given in the AH register.

Redifinition of an existing ISR:

An existing system ISR can be redefined by replacing its vector by the vector of a user defined ISR. If execution is to resume at the redefined ISR, DOS function 35H is used to retrieve the segment:Offset address of the interrupt to be redefined. This address is then stored in a double-word variable (VAR). The address is then used by the new ISR to return control to the redefined ISR by executing the FAR jump:

JMP CS:VAR

If control is not to resume at the redefined ISR, the new ISR ends with an IRET. **INT 21H Function 25H** returns the 32-bit vector address of a specified
interrupt vector in ES:BX (Table 10. 2).
The following code retrieves the vector address of INT 09H:

```
MOV AH, 35H                 ;Get Int. Vector
MOV AL, 09H                 ;For INT 09H
INT 21H
MOV WORD PTR VAR,BX         ;Store Offset Address
MOV WORD PTR VAR+2,BX       ;Store Segment Number
```

When an existing system interrupt is being redefined, it is better to disable maskable interrupts. Hardware interrupts can occur at any time. For this reason, interrupts should be disabled while making changes in the vector table, in order to avoid the risk of an interrupt taking place while there is no valid address for a service routine. The

CLI and STI instructions are used for this purpose (Table 10. 3). A typical installation routine is given in program 10.3.

Instruction	Meaning	Flags Affected
CLI	Clear Interrupt Flag; Disable Maskable Interrupts	IF = 0
STI	Set Interrupt Flag; Re-enable Maskable Interrupts	IF = 1

Table 10. 3: Interrupt Flag Set and Clear Instructions

Typical Case, Interception of the Keyboard Interrupt:

The keyboard port:
When a key is pressed, the keyboard controller sends an 8-bit scan code to to port 60H. The key stroke triggers a hardware interrupt, which prompts the CPU to cal INT 09H. This interrupt inputs then the scan code from the port.

The DOS keyboard status flag:
The keyboard status flag is located at 0040:0017H. This status flag can be obtained either by:

1 – Using BIOS INT 16H, function 02H:

```
MOV AH, 02H
INT 16H
;Status flag returned in AL register
```

or:

2 – Directly reading segment 40H:

```
MOV AX, 0040H
MOV ES, AX
MOV DI, 0017H
MOV AL, ES:[DI]
```

To test for the status of a bit use the TEST instruction.

```
TEST AL, 00100000B  ; Test Num Lock
```

7	6	5	4	3	2	1	0

Bit 0: Right Shift Key Down
Bit 1: Left Shift Key Down
Bit 2: Ctrl Key Down
Bit 3: Alt Key Down
Bit 4:Insert On
Bit 5: Caps Lock On
Bit 6: num Lock On
Bit 7: Scrol Lock On

Figure 10. 6: Keyboard Status Register

Program 10.4 is a TSR that beeps the speaker once when the numeric keypad is used, provided that the Num Lock is on. Note that the scan codes of the keys in the numeric keypad area are from 71 to 83.

Important Note:

All ISR or TSR programs are COM (not EXE) files. To generate a COM file, assemble your program either using the option: assemble as COM in the PWB environment, or using the command: ML /at.

Pre Lab Work:

1- Write, assemble link and run program 10.1. Use the command ML /AT to generate a COM file.
2- Write, assemble link and run program 10.2.
3- Change Program 10.1 using the code given to display a number in AX in decimal format.
4- Write a program that displays a number in decimal format using INT 60H.
5- Bring your work to the lab.

Lab Work:

1- Show programs 10.1 and 10.2 to your lab instructor.
2- Write a Program that checks for the validity of your password, and beeps the speaker once if the password is right, and twice if not.

Guidelines:

- First use the program already developed in previous experiment to check for password validity.
- Second use macros as much as you can.
- Third make the password checking routine as a TSR program. Call it INT 60H. When invoked this procedure checks for password validity and returns AL = 00H if the password is correct, and AL = 0FFH is the password is incorrect.

TITLE 'Program 10.1'

```
;This program installs an ISR at INT 60H. Whenever invoked, as in
;Program 10.2, INT 60H displays the message 'Welcome To The
;Interesting World Of TSR Routines'

.MODEL TINY
.CODE
        ORG 100H              ;Force code at Offset 100H
.STARTUP
        JMP INSTALL
ISR_60H PROC FAR
        PUSH AX
        PUSH DX
        PUSH DS
        MOV  AX,  CS
        MOV  DS,  AX
        MOV  DX,  OFFSET MSG
        MOV  AH,  09H
        INT  21H
        POP  DS
        POP  DX
        POP  AX
        IRET
ISR_60H ENDP

MSG  DB 'Welcome To The Interesting World Of TSR
Routines',0DH,0AH,'$'

INSTALL PROC
        MOV  AX,  CS
        MOV  DS,  AX
        MOV  AH,  25H
        MOV  AL,  60H
        MOV  DX,  OFFSET ISR_60H
        INT  21H
        MOV  AH,  31H
        MOV  AL,  00H
        MOV  DX,  OFFSET INSTALL
        MOV  CL,  04
        SHR  DX,  CL
        INC  DX
        INT  21H
INSTALL ENDP
END
```

```
TITLE 'Program 10.2'
; This program uses the newly installed INT 60H
.MODEL TINY
.CODE
.STARTUP
    MOV  CX, 5          ; Loop 5 times
L1: INT  60H            ; Display the message
    LOOP L1
.EXIT
END
```

TITLE 'Program 10.3'
;The following code redefines an existing ISR

```
VAR DD ?                        ;Used to store the Old Interrupt Vector
...
INSTALLATION_ROUTINE PROC
            CLI                                 ;Clear IF to prevent Hardware Interrupt
            ;GET OLD INTERRUPT VECTOR
            MOV AH, 35H
            MOV AL, INTERRUPT_NUMBER
            INT 21H

            ;SAVE THE INTERRUPT VECTOR
            MOV WORD PTR VAR,BX         ;Store Offset Address
            MOV WORD PTR VAR+2,ES               ;Store Segment Number

            ;INSTALL NEW INTERRUPT VECTOR
            MOV AX, CS
            MOV DS, AX
            MOV AH, 25H
            MOV AL, INTERRUPT_NUMBER
            MOV DX, OFFSET NEW_ISR
            INT 21H

            ;TERMINATE AND STAY RESIDENT
            MOV AH, 31H
            MOV AL, 00H
            STI                                 ; Set IF to enable Hardware Interrupt
            INT 21H
INSTALLATION_ROUTINE ENDP
```

TITLE 'Program 10.4'
;

```
.MODEL TINY
.CODE
        ORG 100H              ;Force code at Offset 100H
.STARTUP
        JMP INSTALL
NEW_09H_ISR PROC FAR
            PUSHF
            PUSH AX
            PUSH ES
            PUSH DI
            PUSH DX
            ;Point ES:DI to the keyboard flag byte
            MOV  AX,  40H
            MOV  ES,  AX
            MOV  DI,  17h
            MOV  AL,  ES:[DI]
            TEST AL, 00100000B      ;NUM LOCK STATE ?
            JZ LAST
            IN AL, 60H
            CMP AL, 71H
            JL LAST
            CMP AL, 83H
            JG LAST
            MOV AH, 02H
```

```
            MOV DL, 07H
            INT  21H
LAST:       POP  DI
            POP  ES
            POP  AX
            POPF
            IRET
            JMP CS:OLD_09H_VECTOR
NEW_09H_ISR ENDP

OLD_09H_VECTOR DD ?
INSTALL     PROC
            MOV  AH,  35H
            MOV  AL,  09H
            INT 21H
            ;SAVE OLD VECTOR
            MOV WORD PTR OLD_09H_VECTOR, BX
            MOV WORD PTR OLD_09H_VECTOR + 2, ES
            ;INSTALL NEW INT. VECTOR
            MOV AX, CS
            MOV DS, AX
            MOV AH, 25H
            MOV AL, 09H
            MOV DX, OFFSET NEW_09H_ISR
            INT 21H
            MOV AX, 3100H
            MOV  DX,    OFFSET INSTALL
            MOV  CL,    04
            SHR  DX,    CL
            INC  DX
            INT  21H
INSTALL ENDP
END
```