BIOS and DOS Interrupts

Module 15

Dr. Tim McGuire

CS 272

Sam Houston State University

- These notes roughly correspond to text chapter 10, although the approach is considerably different

Overview

- Previously, we used the **`INT`** (interrupt) instruction to call system routines
- In this module, we discuss different kinds of interrupts and take a closer look at the operation of the **`INT`** instruction
  - We will discuss the services provided by various BIOS (basic input/output system) and DOS interrupt routines
- To demonstrate the use of interrupts, we will write a program that displays the current time on the screen

Hardware Interrupt

- Whenever a key is pressed, the CPU must be notified to read a key code into the keyboard buffer
- The general ***hardware interrupt*** goes like this:
  - a device that needs service sends an ***interrupt request signal*** to the processor
  - the CPU suspends the current task and transfers control to an interrupt routine
  - the ***interrupt routine*** services the hardware device by performing some I/O operation
  - control is transferred back to the original executing task at the point where it was suspended

Questions to be Answered

- How does the CPU find out a device is signaling?
- How does it know which interrupt routine to execute?
- How does it resume the previous task?

Acknowledging an Interrupt

- Because an interrupt signal may come at any time, the CPU checks for the signal after executing each instruction
- On detecting the interrupt signal, the CPU acknowledges it by sending an ***interrupt acknowledge signal***
- The interrupting device responds by sending an eight-bit number on the data bus, called an ***interrupt number***
- Each device uses a different interrupt number to identify its own service routine
- This process is called ***hand-shaking***

Transferring to an Interrupt Routine

- The process is similar to a procedure call
- Before transferring control to the interrupt routine, the CPU first saves the address of the next instruction on the stack; this is the return address
- The CPU also saves the **`FLAGS`** register on the stack; this ensures that the status of the suspended task will be restored
- It is the responsibility of the interrupt routine to restore any registers it uses

Software Interrupt

- Software interrupts are used by programs to request system services
- A ***software interrupt*** occurs when a program calls an interrupt routine using the **`INT`** instruction
- The format of the **`INT`** instruction is

  `INT` ***`interrupt_number`***

- The 8086 treats this interrupt number in the same way as the interrupt number generated by a hardware device
- We have already seen a number of examples of this using **`INT 21h`** and **`INT 10h`**

Processor Exception

- There is a third kind of interrupt, called a ***processor exception***
- A processor exception occurs when a condition arises inside the processor, such as divide overflow, that requires special handling
- Each condition corresponds to a unique interrupt type
- For example, divide overflow is type 0, so when overflow occurs in a divide instruction the CPU automatically executes interrupt 0 to handle the overflow condition

Interrupt Numbers

- The interrupt numbers for the 8086 are unsigned byte values -- therefore 256 types of interrupts are possible
- Not all interrupt numbers are used
- BIOS interrupt service routines are stored in ROM
- DOS interrupt routines (`int 21h`) are loaded into memory when the machine is started
- Some additional interrupt numbers are reserved by the manufacturer for further use; the remaining numbers are available for the user

Interrupt Types

Interrupt Types	***Description***
0h-1Fh	BIOS Interrupts
20h-3Fh	DOS Interrupts
40h-7Fh	reserved
80h-F0h	ROM BASIC
F1h-FFh	not used

Interrupt Vector

- The CPU does not generate the interrupt routine's address directly from the interrupt number
  - Doing so would mean that a particular interrupt routine must be placed in exactly the same location in every computer
  - Instead, the CPU uses the interrupt number to calculate the address of a memory location that contains the actual address of the interrupt routine
- This means that the routine may appear anywhere, so long as its address, called an ***interrupt vector***, is stored in a predefined memory location

Interrupt Vector Table

- All interrupt vectors are placed in an interrupt vector table, which occupies the first 1KB of memory
- Each interrupt vector is given as segment:offset and occupies four bytes
  - The first four bytes of memory contain interrupt vector 0

0003Fh	Segment of INT FF
003FCh	Offset of INT FF
	• • •
00006h	Segment of INT 1
00004h	Offset of INT 1
00002h	Segment of INT 0
00000h	Offset of INT 0

Accessing the Vector

- To find the vector for an interrupt routine, multiply the interrupt number by 4
  - This gives the memory location containing the offset of the routine
  - The segment number of the routine is in the next word

Example

- The keyboard interrupt routine is interrupt 9
- The offset address is stored in location **`9x4 = 36 = 00024h`**
- The segment address is found in location **`24h`** + **`2 = 00026h`**
- BIOS initializes its interrupt vectors when the computer is turned on, and the DOS interrupt vectors are initialized when DOS is loaded

Interrupt Routines

- When the CPU executes an INT instruction, it first saves the flags by pushing the contents of the FLAGS register onto the stack
- Then it clears the control flags IF (interrupt flag) and TF (trap flag)
  - The reason for this action is explained later
- Finally, it uses the interrupt number to get the interrupt vector from memory and transfers control to the interrupt routine by loading CS:IP with the interrupt vector
  - The 8086 transfers to a hardware interrupt routine or processor exception in a similar fashion
- On completion, an interrupt routine executes an **IRET** (interrupt return) instruction that restores the IP, CS, and FLAGS registers

The Control Flag TF

- When TF is set, the 8086 generates a processor exception (interrupt 1)
  - This interrupt is used by debuggers to "single step" through a program
  - To trace an instruction, the debugger first sets TF, and then transfers control to the instruction to be traced
- After the instruction is executed, the processors generates an interrupt type 1 because TF is set
  - The debugger uses its own interrupt 1 routine to gain control of the processor

The Control Flag IF

- IF is used to control hardware interrupts
  - When IF is set, hardware devices may interrupt the CPU
  - External interrupts may be disabled (masked out) by clearing IF
  - Actually, there is a hardware interrupt, called **NMI** ***(nonmaskable interrupt***) that cannot be masked out
- Both TF and IF are cleared by the processor before transferring to an interrupt routine so that the routine will not be interrupted.
  - Of course, an interrupt routine can change the flags to enable interrupts during its execution

BIOS Interrupts

- Interrupt types 0 - 1Fh are BIOS interrupts whose service routines reside in ROM segment **`F000h`**
- Interrupt 0 -- Divide Overflow: generated when a **`DIV`** or **`IDIV`** operation produces an overflow
  - The interrupt 0 routine displays the message "**`DIVIDE OVERFLOW`**" and returns control to DOS
- Interrupt 1 -- Single Step: generated when the **`TF`** is set
- Interrupt 2 -- Nonmaskable Interrupt: cannot be masked out by clearing the **`IF`**
  - The IBM PC uses this interrupt to signal memory and I/O parity errors that indicate bad chips

BIOS Interrupts

- Interrupt 3 -- Breakpoint: used by debuggers to set up breakpoints
- Interrupt 4 -- Overflow: generated by the instruction **`INTO`** (interrupt if overflow) when **`OF`** is set
  - Programmers may write their own interrupt routine to handle unexpected overflows
- Interrupt 5 -- Print Screen: The BIOS interrupt 5 routine sends the video screen information to the printer
  - An INT 5 instruction is generated by the keyboard interrupt routine (INT 9) when the PrtScr key is pressed
- Interrupts 6&7 are reserved by Intel

BIOS Interrupts

- Interrupt 8 -- Timer: A timer circuit generates an interrupt once every 54.92 milliseconds
  - The BIOS interrupt 8 routine services the timer circuit
    - It users the timer signals to keep track of the time of day
- Interrupt 9 -- Keyboard: generated by the keyboard whenever a key is pressed or released
  - The service routine reads a scan code and stores it in the keyboard buffer
- Interrupt E -- Diskette Error: The BIOS interrupt routine Eh handles diskette errors

Interrupt Types 10h - 1Fh

- The interrupt routines 10h - 1Fh are software interrupts which can be called by application programs to perform various I/O operations and status checking
- Interrupt 10h -- Video: The BIOS interrupt 10h routine is the video driver
  - Details have been covered in a other units
- Interrupt 11h -- Equipment Check: returns the equipment configuration of the particular PC
  - The return code is placed in AX
  - The table on the next slide shows how to interpret AX

Equipment Check

15-14	Number of printers installed
13	= 1 if internal modem installed
12	= 1 if game adapter installed
11-9	Number of serial ports installed
8	not used
7-6	Number of floppy drives (if bit 0=1) 00=1, 01=2KB, 10=3, 11=4
5-4	Initial video mode 00=not used, 01=40x25 color, 10=80x25 color, 11=80x25 monochrome
3-2	System board RAM size (original PC) 00=16KB, 01=32KB, 10=48KB, 11=64KB
1	= 1 if math coprocessor installed
0	= 1 if floppy drive installed

Interrupt Types 10h - 1Fh

- Interrupt 12h -- Memory Size: returns in AX the amount of ***conventional memory***
  - Conventional memory refers to memory circuits with address below 640K -- the unit for the return value is in kilobytes
  - Example:
    - Suppose a computer has 512KB of conventional memory. What will be returned in AX if the instruction INT 12h is executed?
    - 512 = 200h, hence AX = 200h
- Interrupt 13h -- Disk I/O: The BIOS interrupt 13h routine is the disk driver; it allows application programs to do disk I/O
  - Most file operations are done through DOS INT 21h, functions 39h - 42h, however; these utilize the BIOS INT 13h routine

Interrupt Types 10h - 1Fh

- Interrupt 14h -- Communications: The communications driver that interacts with the serial ports
- Interrupt 15h -- Cassette: Used by the original PC for the cassette interface
- Interrupt 16h -- Keyboard: the keyboard driver, discussed in a previous unit
- Interrupt 17h -- Printer I/O: the printer driver
  - supports 3 functions, given by AH=0,1, or 2
    - Function 0: writes character to the printer
    - Function 1: initializes a printer port
    - Function 2: gets printer status

Interrupt Types 10h - 1Fh

- Interrupt 18h -- BASIC: transfers control to ROM BASIC
- Interrupt 19h -- Bootstrap: reboots the system
- Interrupt 1Ah -- Time of Day: allows a program to get and set the timer tick count
- Interrupt 1Bh -- Ctrl-Break: called by the INT 9 routine when Ctrl-Break is pressed
  - The BIOS routine is a stub; it contains only an IRET instruction
  - Users may write their own routines to handle the Ctrl-Break key
- Interrupt 1Ch -- Timer Tick: called by INT 8 each time the timer circuit interrupts -- as in INT 1Bh, the routine is a stub
- Interrupts 1Dh-1Fh: These interrupt vectors point to data instead of instructions (video parameters, diskette parameters, and video graphics characters, respectively)

DOS Interrupts

- The interrupt types 20h-3Fh are serviced by DOS routines that provide high-level service to hardware as well as system resources such as files and directories
- The most useful is INT 21h, which provides many functions for doing keyboard, video, and file operations

DOS Interrupts 20h-27h

- Interrupt 20h -- Program Terminate: Terminates program, but it is better to use INT 21h, function 4Ch
- Interrupt 21h -- Function Request: Functions 0h-5Fh
  - These functions may be classified as character I/O, file access, memory management, disk access, networking, etc.
- Interrupt 22h-26h: These handle critical errors and direct disk access
- Interrupt 27h -- Terminate and Stay Resident: allows programs to stay in memory after termination

A Time Display Program

- As an example of using interrupt routines, we now write a program that displays the current time
  - We will write three versions, each more complex
- The first version simply displays the current time in hours, minutes, and seconds
- The second version will show the time updated every second
- The third version will be a *memory resident program* that can display the time while other programs are running

Clock at Power-up

- When the computer is powered up, the current time is usually supplied by a real-time clock circuit that is battery powered
  - If there is no real-time clock, DOS prompts the user to enter a time
- This time value is kept in memory and updated by a timer circuit using interrupt 8
- A program can call DOS interrupt 21h, function 2Ch, to access the time

INT 21h, Function 2Ch

- Time Of Day
  - Input:
    - AH = 2Ch
  - Output:
    - CH = hours (0 - 23)
    - CL = minutes (0 - 59)
    - DH = seconds (0 - 59)
    - DL = 1/100 seconds (0 - 99)
- Returns the time: hours, minutes, seconds, and hundredths of seconds

How the Program Works

- Three steps
  - obtains the current time (procedure GET_TIME)
  - converts the hours, minutes, and seconds into ASCII digits (ignore the fractions of seconds) (procedure CONVERT)
  - display the ASCII digits
- A time buffer, TIME_BUF, is initialized with the message of 00:00:00
- The main procedure calls GET_TIME to store the current time in the time buffer

How the Program Works

- The main procedure then calls INT 21h, function 9 to print out the string in the time buffer
- GET_TIME calls INT 21h, function 2Ch to get the time, then calls CONVERT to convert the time to ASCII characters
- CONVERT divides the input number in AL by 10; this will put the ten's digit in AL and the one's digit in AH
- The second step is to convert the digits into ASCII
- The program displays the time and terminates

Program Listing (timedsp1.asm)

```
%TITLE   "TIME_DISPLAY_VER_1"
;program that displays the current time
   IDEAL
   MODEL   small
   STACK   100h
   DATASEG
TIME_BUF        DB      '00:00:00$'      ;time buffer hr:min:sec
   CODESEG
Start:
   mov      AX,@data
   mov      DS,AX               ;initialize DS
;get and display time
   lea      BX,[TIME_BUF]       ;BX points to TIME_BUF
   call     GET_TIME            ;put current time in TIME_BUF
   lea      DX,[TIME_BUF]       ;DX points  to TIME_BUF
   mov      AH,09h              ;display time
   int      21h
;exit
   mov      AH,4Ch              ;return
   int      21h                 ;to DOS
```

Procedure GET_TIME

```
PROC    GET_TIME        NEAR
;get time of day and store ASCII digits in time buffer
;input:  BX = address of time buffer
   mov     AH,2Ch              ;gettime
   int     21h                 ;CH = hr, CL =  min, DH = sec
;convert hours into ASCII and store
   mov     AL,CH               ;hour
   call    CONVERT             ;convert to ASCII
   mov     [BX],AX             ;store
;convert minutes into ASCII and store
   mov     AL,CL               ;minute
   call    CONVERT             ;convert to ASCII
   mov     [BX+3],AX           ;store
;convert seconds into ASCII and store
   mov     AL,DH               ;second
   call    CONVERT             ;convert to ASCII
   mov     [BX+6],AX           ;store
   ret
ENDP    GET_TIME
```

Procedure CONVERT

```
PROC    CONVERT
;converts byte number (0-59) into ASCII digits
;input:  AL = number
;output: AX = ASCII digits, AL = high digit, AH = low digit
   mov      AH,0                ;clear AH
   mov      DL,10               ;divide AX by 10
   div      DL                  ;AH has remainder, AL has quotient
   or       AX,3030h            ;convert to ASCII, AH has low digit
   ret                          ;AL has high digit
ENDP    CONVERT
;
   END      Start
```

User Interrupt Procedures

- To make the time display program more interesting, let's write a second version that displays the time and updates it every second
  - One way to continuously update the time is to execute a loop that keeps obtaining the time via INT 21h, function 2Ch and displaying it
  - The problem here is to find a way to terminate the program
  - Instead of pursing this approach, we will write a routine for interrupt 1Ch

INT 8 and INT 1Ch

- Interrupt 1Ch is generated by the INT 8 routine which is activated by a timer circuit about 18.2 times per second
- We will write a new interrupt 1Ch routine so that when it is called, it will get the time and display it
- Our program will have a main procedure that sets up the interrupt routine and when a key is pressed, it will deactivate the interrupt routine and terminate

Set Interrupt Vector

- To set up an interrupt routine, we need to
  - save the current interrupt vector
  - place the vector of the user procedure in the interrupt vector table, and
  - restore the previous vector before terminating the program
- We use INT 21h, function 35h to get the old vector and function 25h to set up the new interrupt vector

INT 21h, Function 25h

- Set Interrupt Vector

  Store interrupt vector into vector table

  - Input:
    - AH = 25h
    - AL = Interrupt number
    - DS:DX = interrupt vector
  - Output:
    - none

INT 21h, Function 35h

- Get Interrupt Vector
  - Obtain interrupt vector from vector table
  - Input:
    - AH = 35h
    - AL = Interrupt number
  - Output:
    - ES:BX = interrupt vector

Procedure SETUP_INT

- The procedure SETUP_INT in program listing setupint.asm saves an old interrupt vector and sets up a new vector
- It gets the interrupt number in AL, a buffer to save the old vector at DS:DI, and a buffer containing the new interrupt vector at DS:SI
- By reversing the two buffers, SETUP_INT can also be used to restore the old vector

Cursor Control

- Each display of the current time by INT 21h, function 9, will advance the cursor
  - If a new time is displayed, it appears at a different screen position
  - So, to view the time updated at the same screen position we must restore the cursor to its original position before we display the time
  - This is achieved by first determining the current cursor position; then, after each print string operation, we move the cursor back
- We use INT 10h, functions 2 and 3, to save the original cursor position and to move the cursor to its original position after each print string operation

INT 10h, Function 2

Described in I/O module, repeated here for convenience

- **Move Cursor**
  - Input:
    - AH = 2
    - DH = new cursor row (0-24)
    - DL = new cursor column (0-79 for 80x25 mode)
    - BH = page number
  - Output: none

INT 10h, Function 3

Described in I/O module, repeated here for convenience

- **Get Cursor Position and Size**
  - Input:
    - AH = 3
    - BH = page number
  - Output:
    - DH = cursor row
    - DL = cursor column
    - CH = cursor starting scan line
    - CL = cursor ending scan line

Interrupt Procedure

- When an interrupt procedure is activated, it cannot assume that the DS register contains the program's data segment address
- Thus, if it uses any variables it must first reset the DS register
- The DS register should be restored before ending the interrupt routine with IRET

DISPTIME2.ASM

- Program listing timedsp2.asm contains a main procedure and the interrupt procedure TIME_INT
- the steps in the main procedure are
  - save the current cursor position
  - set up the interrupt vector for TIME_INT
  - wait for a key input, and
  - restore the old interrupt vector and terminate

Setting and Restoring the Interrupt Vector

- To set up the interrupt vector, we use the pseudo-ops OFFSET and SEG to obtain the offset and segment of the TIME_INT routine
  - The vector is then stored in the buffer NEW_VEC
- SETUP_INT is called to set up the vector for interrupt type 1Ch (timer tick)
- INT 16h, fcn 0 is used for key input
- SETUP_INT is again used for to restore the old interrupt vector
  - this time SI points to the old vector and DI points to the vector for TIME_INT

The TIME_INT Routine

- The steps in TIME_INT are
  - set DS
  - get new time
  - display time
  - restore cursor position, and
  - restore DS

Outline of the Program

- The program operates like this:
  - After setting up the cursor and interrupt vectors, the main procedure just waits for a keystroke
  - In the meantime, the interrupt routine (TIME_INT) keeps updating the time whenever the timer circuit ticks
  - After a key is hit, the old interrupt vector is restored and the program terminates

Assembling and Linking

- The modules must be separately assembled and then linked with:

```
tlink  timedsp2 setupint gettime
```

Memory Resident Program

- We will write the third version of DISPLAY_TIME as a ***TSR (terminate and stay resident) program***
- Normally, when a program terminates, the memory occupied by the program is used by DOS to load other programs
- However, when a TSR program terminates, the memory occupied is not released
- Thus, a TSR program is also called a ***memory resident program***

Terminating a TSR

- To return to DOS, a TSR program is terminated by using either INT 27h or INT 21h, function 31h
  - Our program uses INT 27h
- We write our program as a .COM program because to use interrupt 27h, we need to determine how many bytes are to remain memory resident
- The structure of a .COM program makes this easy, because there is only one program segment
- .COM programs also are smaller, so they save space for TSRs

INT 27h

- Terminate and Stay Resident
  - Input:
    - DS:DX = address of byte beyond the part that is to remain resident
  - Output:
    - none

- Once terminated, a TSR program is not active
  - It must be activated by some external activity, such as a certain key combination or by the timer
- The advantage of a TRS program is that it may be activated while some other program is running
  - Our program will become active when the Ctrl and right shift keys are pressed
- To keep the program small, it will not update the time

- The program has two parts:
  - an initialization part that sets up the interrupt vector, and
  - the interrupt routine itself
- The procedure INITIALIZE initializes the interrupt vector 9 (keyboard interrupt) with the address of the interrupt procedure MAIN and then calls INT 27h to terminate
- The address is passed to INT 27h is the beginning address of the INITIALIZE procedure
  - this is possible because the instructions are no longer needed
- The procedure INITIALIZE is shown in the program listing INITLZE.ASM

Procedure INITIALIZE

```
%TITLE  "INITLZE: SET UP TSR PROGRAM"
   EXTRN    MAIN:NEAR,SETUP_INT:NEAR
   EXTRN    NEW_VEC:WORD,OLD_VEC:DWORD
   PUBLIC   INITIALIZE
   IDEAL
SEGMENT C_SEG PUBLIC
   ASSUME   CS:C_SEG
PROC INITIALIZE                        ; setup interrupt vector
   mov      [NEW_VEC],offset MAIN      ; store address
   mov      [NEW_VEC+2],cs             ; segment
   lea      di,[OLD_VEC]               ; DI points  to vector buffer
   lea      si,[NEW_VEC]               ; SI points to new vector
   mov      al,09h                     ; keyboard interrupt
   call     SETUP_INT                  ; set interrupt vector
;exit to DOS
   lea      dx,[INITIALIZE]
   int      27h                        ; terminate and stay resident
ENDP INITIALIZE
ENDS    C_SEG

   END
```

- There are a number of ways for the interrupt routine to detect a particular key combination
  - The simplest way is to detect the control and shift keys by checking the keyboard flags
  - When activated by a keystroke, the interrupt routine calls the old keyboard interrupt routine to handle the key input
  - To detect the control and shift keys, a program can examine the keyboard flags at the BIOS data area 0000:0417h or use INT 16h, function 2

INT 16h, Function 2

- **Get Keyboard Flags**
  - Input:
    - AH = 2
  - Output:
    - AL = key flags

bit	***meaning***
7=1	Insert on
6=1	Caps Lock on
5=1	Num Lock on
4=1	Scroll Lock on
3=1	Alt key down
2=1	Ctrl key down
1=1	Left shift key down
0=1	Right shift key down

- We will use the Ctrl and right shift key combination to activate and deactivate the clock display
  - When activated, the current time will be displayed on the upper right-hand corner
  - We must first save the screen data so that when the clock display is deactivated the screen can be restored
- The procedure SET_CURSOR sets the cursor at row 0 and the column given in DL
- The procedure SAVE_SCREEN copies the screen data into a buffer called SS_BUF, and the procedure RESTORE_SCREEN moves the data back to the screen buffer
- All three procedures are contained in SAVESCRN.ASM

Procedure SAVE_SCREEN

```
%TITLE  "SAVESCRN: SAVE SCREEN AND  CURSOR"
   IDEAL
   EXTRN    SS_BUF:BYTE
   PUBLIC   SAVE_SCREEN,RESTORE_SCREEN,SET_CURSOR
SEGMENT C_SEG PUBLIC
   ASSUME   cs:C_SEG
PROC SAVE_SCREEN
;saves 8 characters from upper right hand corner of screen
   lea      di,[SS_BUF]         ;screen buffer
   mov      cx,8              ;8 times
   mov      dl,72             ;column  72
   cld                        ;clear DF for string operation
SS_LOOP:
   call     SET_CURSOR        ;setup cursor at row 0, col DL
   mov      ah,08h            ;read char on screen
   int      10h               ;AH = attribute, AL = character
   stosw                      ;stores char and attribute
   inc      dl                ;next col
   loop     SS_LOOP
   ret
ENDP SAVE_SCREEN
```

Procedure RESTORE_SCREEN

```
PROC RESTORE_SCREEN
;restores saved screen
   lea      si,[SS_BUF]      ;SI points to buffer
   mov      di,8             ;repeat 8 times
   mov      dl,72            ;column 72
   mov      cx,1             ;1 char at a time
RS_LOOP:
   call     SET_CURSOR       ;move cursor
   lodsw                     ;AL = char, AH = attribute
   mov      bl,ah            ;attribute  to BL
   mov      ah,09h           ;function 9, write char and attribute
   mov      bh,0             ;page 0
   int      10h
   inc      dl               ;next char position
   dec      di               ;more characters?
   jg       RS_LOOP          ;yes, repeat
   ret
ENDP RESTORE_SCREEN
```

Procedure SET_CURSOR

```
PROC SET_CURSOR
;sets cursor at row 0, column DL
;input DL = column number
   mov      ah,02               ;function 2, set cursor
   mov      bh,0                ;page 0
   mov      dh,0                ;row 0
   int      10h
   ret
ENDP SET_CURSOR
;
ENDS    C_SEG
   END
```

- We are now ready to write the interrupt routine
- To determine whether to activate or deactivate the time display, we use the variable ON_FLAG, which is set to 1 when the time is being displayed
- Procedure MAIN is the interrupt routine

- The steps in procedure MAIN are:
  - save all registers used and set up the DS and ES registers
  - call the old keyboard interrupt routine to handle the key input
  - check to see if both Ctrl and right shift keys are down (if not, then exit)
  - test ON_FLAG to determine status, and if ON_FLAG is 1 then restore screen and exit
  - save current cursor position and also the display screen info, and
  - get time, display time, then exit

- In step 1, to set up the registers DS and ES, we use CS
  - segment values cannot be used in a .COM program
- In step 2, we need to push the FLAGS register so that the procedure call simulates an interrupt call
- In step 6, we use the BIOS interrupt 10h instead of DOS interrupt 21h, function 9 to display the time because (from experience) INT 21h, function 9 tends to be unreliable in a TSR program

timedsp3.asm

```
%TITLE    "TIME_DISPLAY_VER_3"
;memory resident program that shows current time of day
;called by Ctrl-rt shift key combination
;
   EXTRN    INITIALIZE:NEAR,SAVE_SCREEN:NEAR
   EXTRN    RESTORE_SCREEN:NEAR,SET_CURSOR:NEAR
   EXTRN    GET_TIME:NEAR
   PUBLIC   MAIN
   PUBLIC   NEW_VEC,OLD_VEC,SS_BUF
   IDEAL
SEGMENT C_SEG    PUBLIC
   assume   cs:C_SEG, ds:C_SEG, SS:C_SEG
   org      100h
START:   jmp     INITIALIZE
;
SS_BUF          DB      16 DUP(?)        ;save screen buffer
TIME_BUF        DB      '00:00:00$'      ;time buffer hr:min:sec
CURSOR_POS      DW      ?       ;cursor position
ON_FLAG         DB      0       ;1 = interrupt procedure running
NEW_VEC         DW      ?,?     ;contains new vector
OLD_VEC         DD      ?       ;contains old vector
;
```

timedsp3.asm

```
%TITLE   "TIME_DISPLAY_VER_3"
;memory resident program that shows current time of day
;called by Ctrl-rt shift key combination
;
   EXTRN    INITIALIZE:NEAR,SAVE_SCREEN:NEAR
   EXTRN    RESTORE_SCREEN:NEAR,SET_CURSOR:NEAR
   EXTRN    GET_TIME:NEAR
   PUBLIC   MAIN
   PUBLIC   NEW_VEC,OLD_VEC,SS_BUF
   IDEAL
SEGMENT C_SEG   PUBLIC
   assume   cs:C_SEG, ds:C_SEG, SS:C_SEG
   org      100h
START:   jmp     INITIALIZE
;
SS_BUF          DB      16 DUP(?)          ;save screen buffer
TIME_BUF        DB      '00:00:00$'        ;time buffer hr:min:sec
CURSOR_POS      DW      ?       ;cursor position
ON_FLAG         DB      0       ;1 = interrupt procedure running
NEW_VEC         DW      ?,?     ;contains new vector
OLD_VEC         DD      ?       ;contains old vector
;
```

timedsp3.asm

```
%TITLE   "TIME_DISPLAY_VER_3"
;memory resident program that shows current time of day
;called by Ctrl-rt shift key combination
;
   EXTRN    INITIALIZE:NEAR,SAVE_SCREEN:NEAR
   EXTRN    RESTORE_SCREEN:NEAR,SET_CURSOR:NEAR
   EXTRN    GET_TIME:NEAR
   PUBLIC   MAIN
   PUBLIC   NEW_VEC,OLD_VEC,SS_BUF
   IDEAL
SEGMENT C_SEG    PUBLIC
   assume   cs:C_SEG, ds:C_SEG, SS:C_SEG
   org      100h
START:   jmp     INITIALIZE
;
SS_BUF          DB      16 DUP(?)         ;save screen buffer
TIME_BUF        DB      '00:00:00$'       ;time buffer hr:min:sec
CURSOR_POS      DW      ?       ;cursor position
ON_FLAG         DB      0       ;1 = interrupt procedure running
NEW_VEC         DW      ?,?     ;contains new vector
OLD_VEC         DD      ?       ;contains old vector
;
```

timedsp3.asm

```
PROC    MAIN
;interrupt procedur
;save registers
   push    ds
   push    es
   push    ax
   push    bx
   push    cx
   push    dx
   push    si
   push    di
;
   mov     ax,cs               ;set ds
   mov     ds,ax
   mov     es,ax               ;and es to current segment
;call old keyboard interrupt procedure
   pushf                       ;save FLAGS
   call    [OLD_VEC]           ; OLD_VEC contains address of procedure
```

```
;get keyboard flags
   mov     ax,cs              ;reset ds
   mov     ds,ax
   mov     es,ax              ;and es to current segment
   mov     ah,02              ;function 2, keyboard flags
   int     16h                ;al has flag bits
   test    al,1               ;rt shift?
   je      I_DONE             ;no, exit
   test    al,100B            ;Ctrl?
   je      I_DONE             ;no, exit
;process
   cmp     [ON_FLAG],1        ;procedure active?
   je      RESTORE            ;yes, deactivate
   mov     [ON_FLAG],1        ;no, activate
;--save cursor position and screen info
   mov     ah,03              ;get cursor position
   mov     bh,0               ;page 0
   int     10h                ;dh = row, dl = col
   mov     [CURSOR_POS],dx    ;save it
   call    SAVE_SCREEN        ;save time display screen
```

```
;--position cursor to upper right corner
   mov      dl,72               ;column 72
   call     SET_CURSOR          ;position cursor in row 0, col 72
   lea      bx,[TIME_BUF]
   call     GET_TIME            ;get current time
;--display time
   lea      si,[TIME_BUF]
   mov      cx,8                ;8 chars
   mov      bh,0                ;page 0
M1:     mov     ah,0Eh               ;write char
   lodsb                        ;char in al
   int      10h                 ;cursor is moved to next col
   loop     M1                  ;loop back if more chars
   jmp      RES_CURSOR          ;
RESTORE:
;restore screen
   mov      [ON_FLAG],0           ;clears flag
   call     RESTORE_SCREEN
```

```
;restore saved cursor position
RES_CURSOR:
   mov      ah,02              ;set cursor
   mov      bh,0
   mov      dx,[CURSOR_POS]
   int      10h
;restore registers
I_DONE:
   pop      di
   pop      si
   pop      dx
   pop      cx
   pop      bx
   pop      ax
   pop      es
   pop      ds
   iret                        ;interrupt return
ENDP     MAIN
;
ENDS     C_SEG
   END      START              ;starting instruction
```

Linking the TSR

- Because the program has been written as a .COM program, we need to rewrite the file containing the GET_TIME procedure with full segment directives. The file GETTIME2.ASM contains GET_TIME, CONVERT, and SETUP_INT
- The TLINK command should be

```
tlink /t timedsp3 savescrn gettime2 initlze
```

- Notice that initlze.obj is linked last so that the procedure INITIALIZE is placed at the end of the program
- Writing TSR programs is tricky -- if there are other TSR programs on your system, your program may not function properly

Procedure GET_TIME

```
%TITLE   "GETTIME2.ASM: GET AND CONVERT TIME TO ASCII"
        PUBLIC  GET_TIME,SETUP_INT
C_SEG   SEGMENT PUBLIC
         IDEAL
        ASSUME  cs:C_SEG
PROC    GET_TIME
;get time of day and store ASCII digits in time buffer
;input:  bx = address of time buffer
        mov     ah,2Ch              ;gettime
        int     21h                 ;ch = hr, cl =  min, dh = sec
;convert hours into ASCII and store
        mov     al,ch               ;hour
        call    CONVERT             ;convert to ASCII
        mov     [bx],ax             ;store
;convert minutes into ASCII and store
        mov     al,cl               ;minute
        call    CONVERT             ;convert to ASCII
        mov     [bx+3],ax           ;store
;convert seconds into ASCII and store
        mov     al,dh               ;second
        call    CONVERT             ;convert to ASCII
        mov     [bx+6],ax           ;store
        ret
ENDP    GET_TIME
```

Procedure CONVERT

```
PROC    CONVERT
;converts byte number (0-59) into ASCII digits
;input: al = number
;output: ax = ASCII digits, al = high digit, ah = low digit
        mov     ah,0            ;clear ah
        mov     dl,10           ;divide ax by 10
        div     dl              ;ah has remainder, al has quotient
        or      ax,3030h        ;convert to ASCII, ah has low digit
        ret                     ;al has high digit
ENDP    CONVERT
```

Procedure SETUP_INT

```
PROC    SETUP_INT
;saves old vector and sets up new  vector
;input:  al = interrupt type
;       di = address of buffer for old vector
;       si = address of buffer containing new vector
;save old interrupt vector
        mov     ah,35h          ;function 35h, get vector
        int     21h             ;es:bx = vector
        mov     [di],bx         ;save offset
        mov     [di+2],es       ;save segment
;setup new vector
        mov     dx,[si]         ;dx has offset
        push    ds              ;save it
        mov     ds,[si+2]       ;ds has segment number
        mov     ah,25h          ;function 25h, set vector
        int     21h             ;
        pop     ds              ;restore ds
        ret
ENDP    SETUP_INT
ENDS    C_SEG
        END
```