11. Interrupts

Background

Two major uses of interrupts are:

1. Scheduling regularly occurring actions -- for example, multi-tasking operating systems uses a timer interrupt to share CPU time between different tasks.
2. I/O handling – interrupts are more efficient at handling sporadically occurring input/output than polling methods.

This lab examines both types of interrupts.

Objectives:

Understand:

A. How to write interrupt service routines.

B. Gain experience with the BIOS timer interrupt.

C. Gain experience with event-driven I/O by writing an event handler for the mouse.

Pre-Lab

Review lecture notes on polled versus interrupt driven I/O. Also, review chapter 9 (Interrupt Driven I/O in the Uffenbeck text).

1. What is an interrupt service routine?
2. We will be using the BIOS user timer interrupt which has an interrupt number of "1Ch". What is the physical address in the interrupt vector table for this interrupt number?
3. What is the difference between *iret* and *ret*? What would happen if I used an *iret* instruction for a normal procedure return?

Lab

A.

The program below (*inttst.asm*) illustrates how install an interrupt service routine for the BIOS user-timer interrupt 1Ch.

```
.model small
.586
.stack 200h
.data
oldvec  dd 0  ;; old interrupt vector

.code

main    proc
      mov     ax,@data
      mov    ds,ax


      mov    ah,35h
      mov    al,1Ch   ;; get old interrupt vector for timer
      int    21h       ;; returns in ES:BX
      mov    word ptr [oldvec],bx  ;; save offset
      mov    bx,es
      mov    word ptr [oldvec+2],bx ;; save segment

      mov    ah,25h
      mov    al,1Ch     ;; use timer interrupt
      mov    dx,offset mytmr
      push   cs
      pop    ds
      int    21h           ;set new 1C interrupt

      mov     ax,@data
      mov    ds,ax
      ;; do wait...
      mov    cx,1000
      call   mywait

      ;; restore old interrupt vector before exit
      mov    dx,word ptr [oldvec]
      mov    ax,word ptr [oldvec+2]
      push   ax
      pop    ds
      mov    ah,25h
      mov    al,1ch
      int    21h

      ;; now exit
      Mov    ax, 4c00h    ;exit to DOS
      Int    21h


main   endp
```

```
mytmr  proc
       pushad       ;; save all registers (32-bit)
       mov    ah,2
       mov    dl,'A'
       int    21h
       popad        ;; pop all registers (32-bit)

       iret

mytmr  endp

;; my wait now uses stack for storage, so reentrant.

 CLKFREQ     EQU     800 ; clock frequency in MHZ
TICS_MS      EQU CLKFREQ*1000
;; will delay # of milliseconds specified in CX. Register CX destroyed

mywait       proc
             push   eax
             push   edx
             enter  8,0    ;reserve 8 bytes on the stack
                           ;for timer value
mywaitlp2:
             call   timget
             mov    [bp-4],eax   ;; save low value
             mov    [bp-8],edx   ;; save high value
mywaitlp1:
             call   timget
             sub    eax,[bp-4]   ;; subtract low value
             sbb    edx,[bp-8]   ;; subtract high value
       ;; edx:eax has delta time. Compare to TICS_MS
             sub    eax,TICS_MS
             sbb    edx,0
             jc     mywaitlp1
             loop   mywaitlp2
             leave
             pop    edx
             pop    eax
             ret
mywait       endp

;;; procedure that returns the Pentium+ 64 bit timer value
;;;  in EDX:EAX

timget       proc
             rdtsc    ;; read timestamp counter
             ret
timget       endp

end     main
```

The software interrupt 1Ch is called by the BIOS interrupt timer service routine (int 08h) which is triggered 18.2 times/second. Normally, interrupt 1Ch is an empty subroutine but can be replaced by a user-defined subroutine. The above program installs the procedure *mytmr* as the ISR for interrupt 1Ch, and then waits for 1 second before exiting. This allows the *mytmr* function to execute approximately 18 times.

How does this program work?

1. The DOS function 21h, AH=25h can be used to set the value for an interrupt vector table entry. Register AL must specify the interrupt number, and DS:DX must point to the Interrupt Service Routine (ISR). The *main* procedure uses this function to install *mytmr* as the ISR for interrupt 1Ch. Before setting the new interrupt vector for interrupt 1Ch, the old interrupt vector for interrupt 1CH is first read via the DOS function 21h, AH=35h and stored at the memory location *oldvec* (segment/offset of vector is returned in ES:BX by int 21h, AH=35h). Before the program exits, the old vector stored at *oldvec* is restored for interrupt 1Ch by using DOS interrupt 21h, AH=25h.
2. The procedure *mytmr* prints one character ('A') to the console using the DOS 21h, AH=02 function before exiting. It is important for an ISR to save any registers that it uses, and to also use an IRET to return from the interrupt. The *mytmr* functions uses the instruction PUSHAD to push all 32-bit registers on the stack, and POPAD to pop all 32-bit registers off the stack. This is probably overkill because the DOS 21h function don't use any 32 bit registers – the instructions PUSHA/POPA (16 bit registers) would have been sufficient.

Lab Question 1: Assemble this program and execute it.

A. How many 'A's get printed to the screen? (be sure that you modify the CLKFREQ equate to match your system).

B. Modify the main procedure such that it loops printing 'A's to the screen – exit the main procedure when any key is pressed. Modify the *mytmr* procedure such that a digit ('0'-'9') is written at cursor location row=0, col=79 ; have the digit change each time the *mytmr* procedure is called such that the value cycles through '0' to '9'. You will need to use the appropriate BIOS 10h functions to read the current cursor position; position the cursor at 0,79; write the digit; and then restore the cursor position to its original location. Do not forget to save all registers in your *mytmr* procedure -- you may also need to save the DS register if you access your data segment from within the *mytmr* procedure. Include the assembled listing of this program in your lab report.

B. Interrupt Driven I/O

Interrupt driven I/O means that a device generates an interrupt whenever it has generated new data (input) or requires additional data (output). All of the I/O devices on the PC (keyboard, mouse, video, serial/parallel ports, etc) support some form of interrupt driven I/O. This type of I/O is also termed 'event-driven' I/O and the interrupt routine is called an 'event handler'.

The mouse is a good candidate for interrupt I/O since mouse clicks are an infrequent occurrence when compared to other I/O events in the system. The BIOS 33h, AX=0Ch function provides a way for a user to install a user procedure that is called on mouse events. An event mask is passed to this function that defines what events will generate a call to the user procedure. The event mask is passed in CX, and each bit defines different events. A bit value should be a '1' if the procedure is to be called when this event occurs. The bit definitions are:

1. Bit 0 (LSB): cursor position changed
2. Bit 1 : left button pressed
3. Bit 2: left button released
4. Bit 3: right button pressed
5. Bit 4: right button released
6. Bits 5 through 15: Unused

For example, if the user procedure is to be called each time the right button is either released or pressed, then CX should be the value : 00018h. The BIOS 33h, AX=0CH function expects ES:DX to point to the user procedure. The user procedure is called from the mouse interrupt service routine, so it should use a normal 'RET' instruction. The procedure must be declared as 'proc far' because the call is made using a FAR call (both CS,IP pushed on stack). The '.model medium' model statement must be used in order to declare a 'proc far' procedure.

When the user procedure is called, the registers have the following values:

1. AX: condition mask causing the call
2. CX,DX: horizontal, vertical mouse coordinates
3. DI,SI : horizontal, vertical counts (you can ignore these)
4. DS points to the mouse driver data segment
5. BX contains the button state. Bit 0 is a '1' if the left button is pressed, Bit 1 is a '1' if the right button is pressed. Bits 15-2 are unused.

The user procedure should preserve all register values the same as any interrupt service routine.

Lab Question 2: Modify the *hline.asm* example from Lab #9 such that it uses interrupt events to detect button press/release. This means that the main program should install a user defined routine for mouse events, then enter a loop waiting for an key to be pressed to exit the program. All the work of recording the mouse position, drawing the line, etc. will be done from your user-defined mouse event handler. Be careful - if you change the DS (data segment) register to point to your own data segment, it must be restored to its original value before your routine exits.

Lab Report

A. Describing What You Learned

Include the answers to all "**Lab Questions**" in your report.

B. Applying What You Learned

Demonstrate the programs you wrote for Lab Questions 1 and 2 to the TA.