Real Time Interrupts and Hardware Interfacing

9.1. Hardware Interrupts

- There is a single pin outside the processor called the INT pin that is used by external hardware to generate interrupts.
- There are many external devices that need the processor’s attention like the keyboard, hard disk, floppy disk, sound card. All of them need real time interrupts at some point in their operation.
- For example if a program is busy in some calculations for three minutes the key strokes that are hit meanwhile should not be wasted. Therefore when a key is pressed, the INT signal is sent, an interrupt generated and the interrupt handler stores the key for later use. Similarly when the printer is busy printing we cannot send it more data. As soon as it gets free from the previous job it interrupts the processor to inform that it is free now.
- There are many other examples where the processor needs to be informed of an external event. If the processor actively monitors all devices instead of being automatically interrupted then it there won’t be any time to do meaningful work.

Programmable Interrupt Controller (PIC)

- Since there are many devices generating interrupts and there is only one pin going inside the processor and one pin cannot be technically derived by more than one source a controller is used in between called the Programmable Interrupt Controller (PIC).
- It has eight input signals and one output signal.
- It assigns priorities to its eight input pins from 0 to 7 so that if more than one interrupt comes at the same times, the highest priority one is forwarded and the rest are held till that is serviced. The rest are forwarded one by one according to priority after the highest priority one is completed.
- The original IBM XT computer had one PIC so there were 8 possible interrupt sources. However IBM AT and later computers have two PIC totaling 16 possible interrupt sources. They are arrange is a special cascade master slave arrangement so that only one output signal comes towards the processor. However we will concentrate on the first interrupt controller only.

Example of priority management

- Consider eight parallel switches which are all closed and connected to form the output signal. When a signal comes on one of the switches, it is passed on to the output and this switch and all below it are opened so that no further signals can pass through it. The higher priority switches are still closed and the signal on them can be forwarded.
- When the processor signals that it is finished with the processing the switches are closed again and any waiting interrupts may be forwarded.

Interrupt Requests

- The eight input signals to the PIC are called Interrupt Requests (IRQ).
- The eight lines are called IRQ 0 to IRQ 7. These are the input lines of the PIC.
- For example IRQ 0 is derived by a timer device. The timer device keeps generating interrupts with a specified frequency.
- IRQ 1 is derived by the keyboard when generates an interrupts when a key is pressed or released.
- IRQ 2 is the cascading interrupt connected to the output of the second 8451 in the machine.
- IRQ 3 is connected to serial port COM 2 while IRQ 4 is connected to serial port COM 1.
- IRQ 5 is used by the sound card or the network card or the modem.
- IRQ 6 is used by the floppy disk drive while IRQ 7 is used by the parallel port.
- An IRQ conflict means that two devices in the system want to use the same IRQ line.

IRQ to INT mapping

- Each IRQ is mapped to a specific interrupt in the system. This is called the IRQ to INT mapping.
- IRQ 0 to IRQ 7 are consecutively mapped on interrupts 8 to F. This mapping is done by the PIC and not the processor.
- The actual mechanism fetches one instruction from the PIC whenever the INT pin is signaled instead of the memory.
- From the perspective of an assembly language programmer an IRQ 0 is translated into an INT 8 without any such instruction in the program.
- Therefore an IRQ 0, the highest priority interrupt, is generated by the timer chip at a precise frequency and the handler at INT 8 is invoked which updates the system time.
- A key press generates IRQ 1 and the INT 9 handler is invoked which stores this key.
- To handle the timer and keyboard interrupts one can replace the vectors corresponding to interrupt 8 and 9 respectively.

EOI

- The interrupt request from a device enters the PIC as an IRQ, from there it reaches the INT pin of the processor.
- The processor receives the interrupt number from the PIC, generates the designated interrupt, and finally the interrupt handler gain control and can do whatever is desired.
- At the end of servicing the interrupt the handler should inform the PIC that it is completed so that lower priority interrupts can be sent from the PIC. This signal is called an End Of Interrupt (EOI) signal and is sent through the I/O ports of the interrupt controller.

9.2. I/O ports

- The processor needs to communicate with the peripheral devices, give and take data from them.
- Memory has a totally different purpose. It contains the program to be executed and its data. It does not control any hardware.
- For communicating with peripheral devices the processor uses I/O ports. There are only two operations with the external world possible, read or write.
- Similarly with I/O ports the processor can read or write an I/O port. When an I/O port is read or written to, the operation is not as simple as it happens in memory. Some hardware changes it functionality or performs some operation as a result.

I/O ports

- IBM PC has separate memory address space and peripheral address space.
- Some processors use memory mapped I/O in which case designated memory cells work as ports for specific devices.
- In case of Intel a special pin on the control bus signals whether the current read or write is from the memory address space or from the peripheral address space.
- The same address and data buses are used to select a port and to read or write data from that port.
- However with I/O only the lower 16 bits of the address bus are used meaning that there are a total of 65536 possible I/O ports.
- The keyboard has special I/O ports designated to it, PIC has others, DMA, sound card, network card, each has some ports.

IN and OUT instructions

- We have the IN and OUT instructions to read or write from the peripheral address space.
- When MOV is given the processor selects the memory address space, when IN is given the processor selects the peripheral address space.
- The IN and OUT instructions have a byte form and a word form but the byte form is almost always used.
- The source register in OUT and destination register in IN is AL or AX depending on which form is used.
- The port number can be directly given in the instruction if it fits in a byte otherwise it has to be given in the DX register.
- Port numbers for specific devices are fixed by the IBM standard.
- For example 20 and 21 are for PIC, 60 to 64 for Keyboard, 378 for the parallel port etc.

Examples

- A few examples of IN and OUT are below:
    - in al, 0x21
    - mov dx, 0x378
    - in al, dx

    - out 0x21, al
    - mov dx, 0x378
    - out dx, al

PIC Ports

- Programmable interrupt controller has two ports 20 and 21.
- Port 20 is the control port while port 21 is the interrupt mask register which can be used for selectively enabling or disabling interrupts.
- Each of the bits at port 21 corresponds to one of the IRQ lines.
- We first write a small program to disable the keyboard using this port.
- As we know that the keyboard IRQ is 1, we place a 1 bit at its corresponding position. A 0 bit will enable an interrupt and a 1 bit disables it. As soon as we write it on the port keyboard interrupts will stop arriving and the keyboard will effectively be disabled. Even Ctrl-Alt- Del would not work; the reset power button has to be used.

Example 9.1.

- After this three line mini program is executed the computer will not understand anything else. Its ears are closed. No keystrokes are making their way to the processor.
- Ports always make something happen on the system.
- In ports every bit has a meaning that changes something in the system.
- Every interrupt handler invoked because of an IRQ must signal an EOI otherwise lower priority interrupts will remain disabled.

Example 9.1

```
; disable keyboard interrupt in PIC mask register
[org 0x0100]
              in    al, 0x21          ; read interrupt mask register
              or    al, 2             ; set bit for IRQ2
              out   0x21, al          ; write back mask register

              mov   ax, 0x4c00        ; terminate program
              int   0x21
```

Keyboard Controller

- We will discuss how the keyboard controller communicates with the processor.
- Keyboard is a collection of labeled buttons and every button is designated a number (not the ASCII code).
- This number is sent to the processor whenever the key is pressed. From this number called the scan code the processor understands which key was pressed.
- For each key the scan code comes twice, once for the key press and once for the key release.
- Both are scan codes and differ in one bit only. The lower seven bits contain the key number while the most significant bit is clear in the press code and set in the release code.
- The IBM PC standard gives a table of the scan codes of all keys.

Keyboard Controller

- If we press Shift-A resulting in a capital A on the screen, the controller has sent the press code of Shift, the press code of A, the release code of A, the release code of Shift and the interrupt handler has understood that this sequence should result in the ASCII code of ‘A’.
- The ‘A’ key always produces the same scan code whether or not shift is pressed. It is the interrupt handler’s job to remember that the press code of Shift has come and release code has not yet come and therefore to change the meaning of the following key presses. Even the caps lock key works the same way.

Keyboard Controller

- An interesting thing is that the two shift keys on the left and right side of the keyboard produce different scan codes. The standard way implemented in BIOS is to treat that similarly. That's why we always think of them as identical.
- If we leave BIOS and talk directly with the hardware we can differentiate between left and right shift keys with their scan code. Now this scan code is available from the keyboard data port which is 60. The keyboard generates IRQ 1 whenever a key is pressed so if we hook INT 9 and inside it read port 60 we can tell which of the shift keys was hit.
- Our first program will output an L if the left shift key was pressed and R if the right one was pressed.

Example 9.2

```
; differentiate left and right shift keys with scancodes
[org 0x0100]
                jmp  start

; keyboard interrupt service routine
kbisr:          push ax
                push es

                mov  ax, 0xb800
                mov  es, ax              ; point es to video memory

                in   al, 0x60            ; read a char from keyboard port
                cmp  al, 0x2a            ; is the key left shift
                jne  nextcmp             ; no, try next comparison

                mov  byte [es:0], 'L'    ; yes, print L at top left
                jmp  nomatch             ; leave interrupt routine

nextcmp:        cmp  al, 0x36            ; is the key right shift
                jne  nomatch             ; no, leave interrupt routine

                mov  byte [es:0], 'R'    ; yes, print R at top left

nomatch:        mov  al, 0x20
                out  0x20, al            ; send EOI to PIC

                pop  es
                pop  ax
                iret

start:          xor  ax, ax
                mov  es, ax              ; point es to IVT base
                cli                      ; disable interrupts
                mov  word [es:9*4], kbisr ; store offset at n*4
                mov  [es:9*4+2], cs      ; store segment at n*4+2
                sti                      ; enable interrupts

l1:             jmp  l1                  ; infinite loop
```

033-036	CLI clears the interrupt flag to disable the interrupt system completely. The processor closes its ears and does not care about the state of the INT pin. Interrupt hooking is done in two instructions, placing the segment and placing the offset. If an interrupt comes in between and the vector is in an indeterminate state, the system will go to a junk address and eventually crash. So we stop all interruptions while changing a real time interrupt vector. We set the interrupt flag afterwards to renewable interrupts.
038	The program hangs in an infinite loop. The only activity can be caused by a real time interrupt. The kbisr routine is not called from anywhere; it is only automatically invoked as a result of IRQ 1.

When the program is executed the left and right shift keys can be distinguished with the L or R on the screen. As no action was taken for the rest of the keys, they are effectively disabled and the computer has to be rebooted. To check that the keyboard is actually disabled we change the program and add the INT 16 service 0 at the end to wait for an Esc key press. As soon as Esc is pressed we want to terminate our program.

Example 9.3

```
; attempt to terminate program with Esc that hooks keyboard interrupt
[org 0x0100]
              jmp  start

;;;;; COPY LINES 005-029 FROM EXAMPLE 9.2 (kbisr) ;;;;;

start:        xor  ax, ax
              mov  es, ax                  ; point es to IVT base
              cli                          ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs          ; store segment at n*4+2
              sti                          ; enable interrupts

l1:           mov  ah, 0                   ; service 0 – get keystroke
              int  0x16                    ; call BIOS keyboard service

              cmp al, 27                   ; is the Esc key pressed
              jne l1                       ; if no, check for next key

              mov ax, 0x4c00               ; terminate program
              int 0x21
```

When the program is executed the behavior is same. Esc does not work. This is because the original IRQ 1 handler was written by BIOS that read the scan code, converted into an ASCII code and stored in the keyboard buffer. The BIOS INT 16 read the key from there and gives in AL. When we hooked the keyboard interrupt BIOS is no longer in control, it has no information, it will always see the empty buffer and INT 16 will never return.

Interrupt Chaining

- We can transfer control to the original BIOS ISR in the end of our routine.
- This way the normal functioning of INT 16 can work as well.
- We can retrieve the address of the BIOS routine by saving the values in vector 9 before hooking our routine.
- In the end of our routine we will jump to this address using a special indirect form of the JMP FAR instruction.

Example 9.4.

027-028	EOI is no longer needed as the original BIOS routine will have it at its end.
033	IRET has been removed and an unconditional jump is introduced. At time of JMP the stack has the exact formation as was when the interrupt came. So the original BIOS routine's IRET will take control to the interrupted program. We have been careful in restoring every register we modified and retained the stack in the same form as it was at the time of entry into the routine.

- When the program is executed L and R are printed as desired and Esc terminates the program as well.
- Normal commands like DIR work now and shift keys still show L and R as our routine did even after the termination of our program.
- Now start some application like the editor, it open well but as soon as a key is pressed the computer rashes.
- Actually our hooking and chaining was fine. When Esc was pressed we signaled DOS that our program has terminated. DOS will take all our memory as a result. The routine is still in memory and functioning but the memory is free according to DOS.
- As soon as we load EDIT the same memory is allocated to EDIT and our routine as overwritten.
- Now when a key is pressed our routine's address is in the vector but at that address some new code is placed that is not intended to be an interrupt handler. That may be data or some part of the EDIT program. This results in crashing the computer.

```
001   ; another attempt to terminate program with Esc that hooks
002   ; keyboard interrupt
003   [org 0x100]
004                 jmp   start
005
006   oldisr:       dd    0                ; space for saving old isr
007
008   ; keyboard interrupt service routine
009   kbisr:        push ax
010                 push es
011
012                 mov   ax, 0xb800
013                 mov   es, ax           ; point es to video memory
014
015                 in    al, 0x60         ; read a char from keyboard port
016                 cmp   al, 0x2a         ; is the key left shift
017                 jne   nextcmp          ; no, try next comparison
018
019                 mov   byte [es:0], 'L' ; yes, print L at top left
020                 jmp   nomatch          ; leave interrupt routine
021
022   nextcmp:      cmp   al, 0x36         ; is the key right shift
023                 jne   nomatch          ; no, leave interrupt routine
024
025                 mov   byte [es:0], 'R' ; yes, print R at top left
```

```
nomatch:        ; mov  al, 0x20
                ; out  0x20, al

                pop es
                pop ax
                jmp far [cs:oldisr]        ; call the original ISR
                ; iret

start:          xor  ax, ax
                mov  es, ax                ; point es to IVT base
                mov ax, [es:9*4]
                mov [oldisr], ax           ; save offset of old routine
                mov ax, [es:9*4+2]
                mov [oldisr+2], ax         ; save segment of old routine
                cli                        ; disable interrupts
                mov  word [es:9*4], kbisr ; store offset at n*4
                mov  [es:9*4+2], cs        ; store segment at n*4+2
                sti                        ; enable interrupts

l1:             mov  ah, 0                 ; service 0 - get keystroke
                int  0x16                  ; call BIOS keyboard service

                cmp al, 27                 ; is the Esc key pressed
                jne l1                     ; if no, check for next key

                mov ax, 0x4c00             ; terminate program
                int 0x21
```

Example 9.5 – Unhooking Interrupts

We now add the interrupt restoring part to our program.

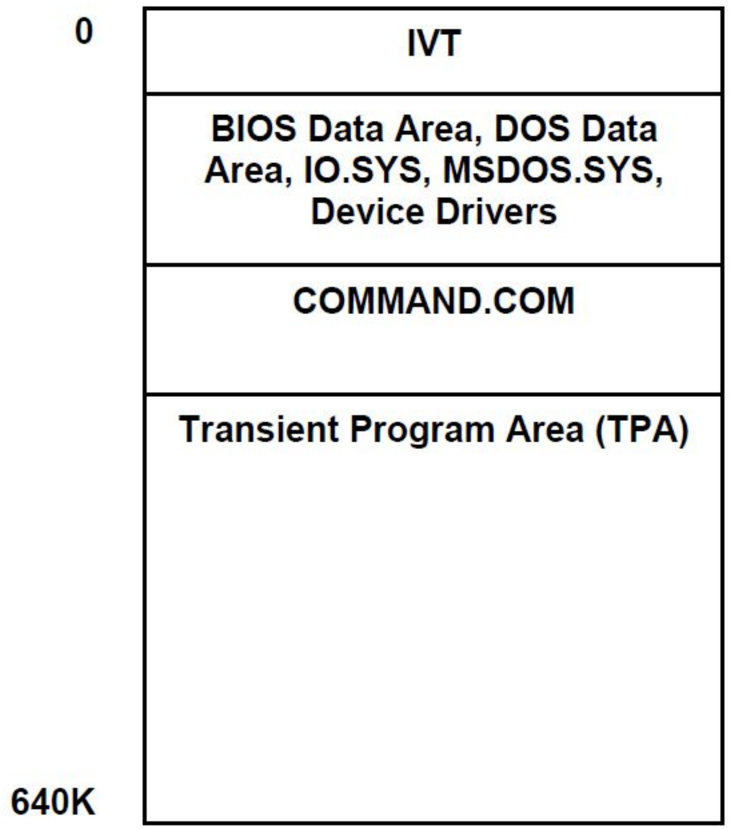
This code resets
the interrupt vector to the value it had before the start of our program.

Example 9.5

```
; terminate program with Esc that hooks keyboard interrupt
[org 0x100]
              jmp  start

oldisr:       dd   0                    ; space for saving old isr

;;;;; COPY LINES 005-029 FROM EXAMPLE 9.4 (kbisr) ;;;;;

start:        xor  ax, ax
              mov  es, ax               ; point es to IVT base
              mov ax, [es:9*4]
              mov [oldisr], ax          ; save offset of old routine
              mov ax, [es:9*4+2]
              mov [oldisr+2], ax        ; save segment of old routine
              cli                       ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs       ; store segment at n*4+2
              sti                       ; enable interrupts

l1:           mov  ah, 0                ; service 0 – get keystroke
              int  0x16                 ; call BIOS keyboard service

              cmp al, 27                ; is the Esc key pressed
              jne l1                    ; if no, check for next key

              mov ax, [oldisr]          ; read old offset in ax
              mov bx, [oldisr+2]        ; read old segment in bx
              cli                       ; disable interrupts
              mov [es:9*4], ax          ; restore old offset from ax
              mov [es:9*4+2], bx        ; restore old segment from bx
              sti                       ; enable interrupts

              mov ax, 0x4c00            ; terminate program
              int 0x21
```

9.3. Terminate and Stay Resident

- DOS memory formation and allocation procedure is as follows.
- At physical address zero is the interrupt vector table. Then are the BIOS data area, DOS data area, IO.SYS, MSDOS.SYS and other device drivers. In the end there is COMMAND.COM command interpreter.
- The remaining space is called the transient program area as programs are loaded and executed in this area and the space reclaimed on their exit. A freemem pointer in DOS points where the free memory begins. When DOS loads a program the freemem pointer is moved to the end of memory, all the available space is allocated to it, and when it exits the freemem pointer comes back to its original place thereby reclaiming all space. This action is initiated by the DOS service 4C.



- The second method to legally terminate a program and give control back to DOS is using the service 31.
- Control is still taken back but the memory releasing part is modified. A portion of the allocated memory can be retained.
- So the difference in the two methods is that the freemem pointer goes back to the original place or a designated number of bytes ahead of that old position.
- Remember that our program crashed because the interrupt routine was overwritten. If we can tell DOS not to reclaim the memory of the interrupt routine, then it will not crash.
- In the next program we shall tell the DOS to make a number of bytes resident. It becomes a part of the operation system, an extension to it. Just like DOSKEY§ is an extension to the operation system

- The number of paragraphs to reserve is given in the DX register.
- Paragraph is a unit just like byte, word, and double word. A paragraph is 16 bytes. Therefore we can reserve in multiple of 16 bytes.
- To calculate the number of paragraphs a label is placed after the last line that is to be made resident. The value of that label is the number of bytes needed to be made resident. A simple division by 16 will not give the correct number of paras as we want our answer to be rounded up and not down.
- For example 100 bytes should need 7 pages but division gives 6 and a remainder of 4. A standard technique to get rounded up integer division is to add divisor-1 to the dividend and then divide. So we add 15 to the number of bytes and then divide by 16. We use shifting for division as the divisor is a power of 2. We use a form of SHR that places the count in the CL register so that we can shift by 4 in just two instructions instead of 4 if we shift one by one.

Example 9.6.

- We change the display to show L only while the left shift is pressed and R only while the right shift is pressed to show the use of the release codes.
- We also changed that shift keys are not forwarded to BIOS. The effect will be visible with A and Shift-A both producing small ‘a’ but caps lock will work.
- There is one major difference from all the programs we have been writing till now.
- The termination is done using INT 21 service 31 instead of INT 21 service 4C.
- The effect is that even after termination the program is there and is legally there.

Example 9.6

```
; TSR to show status of shift keys on top left of screen
[org 0x0100]
              jmp  start

oldisr:       dd   0                  ; space for saving old isr

```

```
; keyboard interrupt service routine
kbisr:        push ax
              push es

              mov  ax, 0xb800
              mov  es, ax             ; point es to video memory

              in   al, 0x60           ; read a char from keyboard port
              cmp  al, 0x2a           ; has the left shift pressed
              jne  nextcmp            ; no, try next comparison

              mov  byte [es:0], 'L'   ; yes, print L at first column
              jmp  exit               ; leave interrupt routine
```

```
nextcmp:        cmp  al, 0x36              ; has the right shift pressed
                jne  nextcmp2              ; no, try next comparison

                mov  byte [es:0], 'R'      ; yes, print R at second column
                jmp  exit                  ; leave interrupt routine

nextcmp2:       cmp  al, 0xaa              ; has the left shift released
                jne  nextcmp3              ; no, try next comparison

                mov  byte [es:0], ' '      ; yes, clear the first column
                jmp  exit                  ; leave interrupt routine

nextcmp3:       cmp  al, 0xb6              ; has the right shift released
                jne  nomatch               ; no, chain to old ISR

                mov  byte [es:2], ' '      ; yes, clear the second column
                jmp  exit                  ; leave interrupt routine

nomatch:        pop  es
                pop  ax
                jmp  far [cs:oldisr]       ; call the original ISR
```

```
exit:           mov  al, 0x20
                out  0x20, al              ; send EOI to PIC

                pop  es
                pop  ax
                iret                       ; return from interrupt

start:          xor  ax, ax
                mov  es, ax                ; point es to IVT base
                mov  ax, [es:9*4]
                mov  [oldisr], ax          ; save offset of old routine
                mov  ax, [es:9*4+2]
                mov  [oldisr+2], ax        ; save segment of old routine
                cli                        ; disable interrupts
                mov  word [es:9*4], kbisr ; store offset at n*4
                mov  [es:9*4+2], cs        ; store segment at n*4+2
                sti                        ; enable interrupts

                mov  dx, start             ; end of resident portion
                add  dx, 15                ; round up to next para
                mov  cl, 4
                shr  dx, cl                ; number of paras
                mov  ax, 0x3100            ; terminate and stay resident
                int  0x21
```

- When this program is executed the command prompt immediately comes.
- DIR can be seen. EDIT can run and keypresses do not result in a crash. And with all that left and right shift keys shown L and R on top left of the screen while they are pressed but the shift keys do not work as usual since we did not forwarded the key to BIOS. This is selective chaining.

9.4. Programmable Interval Timer

- Another very important peripheral device is the Programmable Interval Timer (PIT), the chip numbered 8254. This chip has a precise input frequency of 1.19318 MHz. This frequency is fixed regardless of the processor clock.
- Inside the chip is a 16bit divisor which divides this input frequency and the output is connected to the IRQ 0 line of the PIC. The special number 0 if placed in the divisor means a divisor of 65536 and not 0. The standard divisor is 0 unless we change it. Therefore by default IRQ 0 is generated 1193180/65536=18.2 times per second. This is called the timer tick. There is an interval of about 55ms between two timer ticks.
- The system time is maintained with the timer interrupt. This is the highest priority interrupt and breaks whatever is executing. Time can be maintained with this interrupt as this frequency is very precise and is part of the IBM standard. When writing a TSR we give control back to DOS so TSR activation, reactivation and action is solely interrupt based, whether this is a hardware interrupt or a software one. Control is never given back; it must be caught, just like we caught control by hooking the keyboard interrupt. Our next example will hook the timer interrupt and display a tick count on the screen.

Example 9.7

```
; display a tick count on the top right of screen
[org 0x0100]
              jmp  start

tickcount:    dw   0

; subroutine to print a number at top left of screen
; takes the number to be printed as its parameter
printnum:     push bp
              mov  bp, sp
              push es
              push ax
              push bx
              push cx
              push dx
              push di

              mov  ax, 0xb800
              mov  es, ax                ; point es to video base
              mov  ax, [bp+4]            ; load number in ax
              mov  bx, 10                ; use base 10 for division
              mov  cx, 0                 ; initialize count of digits
```

```
nextdigit:    mov  dx, 0              ; zero upper half of dividend
              div  bx                 ; divide by 10
              add  dl, 0x30           ; convert digit into ascii value
              push dx                 ; save ascii value on stack
              inc  cx                 ; increment count of values
              cmp  ax, 0              ; is the quotient zero
              jnz  nextdigit          ; if no divide it again

              mov  di, 140            ; point di to 70th column

nextpos:      pop  dx                 ; remove a digit from the stack
              mov  dh, 0x07           ; use normal attribute
              mov  [es:di], dx        ; print char on screen
              add  di, 2              ; move to next screen location
              loop nextpos            ; repeat for all digits on stack

              pop  di
              pop  dx
              pop  cx
              pop  bx
              pop  ax
```

```
                pop  es
                pop  bp
                ret  2

; timer interrupt service routine
timer:          push ax

                inc  word [cs:tickcount]; increment tick count
                push word [cs:tickcount]
                call printnum           ; print tick count

                mov  al, 0x20
                out  0x20, al           ; end of interrupt

                pop  ax
                iret                    ; return from interrupt

start:          xor  ax, ax
                mov  es, ax             ; point es to IVT base
                cli                     ; disable interrupts
                mov  word [es:8*4], timer; store offset at n*4
                mov  [es:8*4+2], cs     ; store segment at n*4+2
                sti                     ; enable interrupts

                mov  dx, start          ; end of resident portion
                add  dx, 15             ; round up to next para
                mov  cl, 4
                shr  dx, cl             ; number of paras
                mov  ax, 0x3100         ; terminate and stay resident
                int  0x21
```

Example 9.7.

- When we execute the program the counter starts on the screen. Whatever we do, take directory, open EDIT, the debugger etc. the counter remains running on the screen. No one is giving control to the program; the program is getting executed as a result of timer generating INT 8 after every 55ms.
- In the next example will hook both the keyboard and timer interrupts. When the shift key is pressed the tick count starts incrementing and as soon as the shift key is released the tick count stops. Both interrupt handlers are communicating through a common variable. The keyboard interrupt sets this variable while the timer interrupts modifies its behavior according to this variable.

Example 9.8

```
; display a tick count while the left shift key is down
[org 0x0100]
                jmp  start

seconds:        dw   0
timerflag:      dw   0
oldkb:          dd   0

;;;;; COPY LINES 007-047 FROM EXAMPLE 9.7 (printnum) ;;;;;

; keyboard interrupt service routine
kbisr:          push ax

                in   al, 0x60              ; read char from keyboard port
                cmp  al, 0x2a              ; has the left shift pressed
                jne  nextcmp               ; no, try next comparison

                cmp  word [cs:timerflag], 1; is the flag already set
                je   exit                  ; yes, leave the ISR

                mov  word [cs:timerflag], 1; set flag to start printing
                jmp  exit                  ; leave the ISR

nextcmp:        cmp  al, 0xaa              ; has the left shift released
                jne  nomatch               ; no, chain to old ISR

                mov  word [cs:timerflag], 0; reset flag to stop printing
```

```
068                 jmp  exit                  ; leave the interrupt routine
069
070 nomatch:        pop  ax
071                 jmp  far [cs:oldkb]        ; call original ISR
072
073 exit:           mov  al, 0x20
074                 out  0x20, al              ; send EOI to PIC
075
076                 pop  ax
077                 iret                       ; return from interrupt
078
079 ; timer interrupt service routine
080 timer:          push ax
081
082                 cmp  word [cs:timerflag], 1 ; is the printing flag set
083                 jne  skipall               ; no, leave the ISR
084
085                 inc  word [cs:seconds]     ; increment tick count
086                 push word [cs:seconds]
087                 call printnum              ; print tick count
088
089 skipall:        mov  al, 0x20
090                 out  0x20, al              ; send EOI to PIC
091
092                 pop  ax
093                 iret                       ; return from interrupt
094
```

006	This flag is one when the timer interrupt should increment and zero when it should not.
058-059	As the keyboard controller repeatedly generates the press code if the release code does not come in a specified time, we have placed a check to not repeatedly set it to one.
058	Another way to access TSR data is using the CS override instead of initializing DS. It is common mistake not to initialize DS and also not put in CS override in a real time interrupt handler.

When we execute the program and the shift key is pressed, the counter starts incrementing. When the key is released the counter stops. When it is pressed again the counter resumes counting. As this is made as a TSR any other program can be loaded and will work properly alongside the TSR.

```
095 start:      xor  ax, ax
096             mov  es, ax                 ; point es to IVT base
097             mov  ax, [es:9*4]
098             mov  [oldkb], ax            ; save offset of old routine
099             mov  ax, [es:9*4+2]
100             mov  [oldkb+2], ax          ; save segment of old routine
101             cli                         ; disable interrupts
102             mov  word [es:9*4], kbisr   ; store offset at n*4
103             mov  [es:9*4+2], cs         ; store segment at n*4+2
104             mov  word [es:8*4], timer   ; store offset at n*4
105             mov  [es:8*4+2], cs         ; store segment at n*4+
106             sti                         ; enable interrupts
107
108             mov  dx, start              ; end of resident portion
109             add  dx, 15                 ; round up to next para
110             mov  cl, 4
111             shr  dx, cl                 ; number of paras
112             mov  ax, 0x3100             ; terminate and stay resident
113             int  0x21
```

006	This flag is one when the timer interrupt should increment and zero when it should not.
058-059	As the keyboard controller repeatedly generates the press code if the release code does not come in a specified time, we have placed a check to not repeatedly set it to one.
058	Another way to access TSR data is using the CS override instead of initializing DS. It is common mistake not to initialize DS and also not put in CS override in a real time interrupt handler.

When we execute the program and the shift key is pressed, the counter starts incrementing. When the key is released the counter stops. When it is pressed again the counter resumes counting. As this is made as a TSR any other program can be loaded and will work properly alongside the TSR.

Reading

- 9.1. – 9.4.