CS253 Laboratory session 5

Part 1: Looking at Interrupts

Preamble: Interrupts provide a mechanism to allow hardware to temporarily redirect the CPU from running the main code associated with an application or operating system so as to run a short routine to process information relating to the hardware.

For example, pressing a key on the keyboard creates a hardware interrupt that stop the CPU running the main code. The key board service routine then runs and the key being pressed is identified and a character associated with the key-press is put in a keyboard buffer. It is the buffer (not the keyboard) that is accessed by a system call from an application program via the API (Application Programmers Interface). When you use *getchar()* you are not directly reading the keyboard, you are looking at the buffer to see if anything new has arrived.

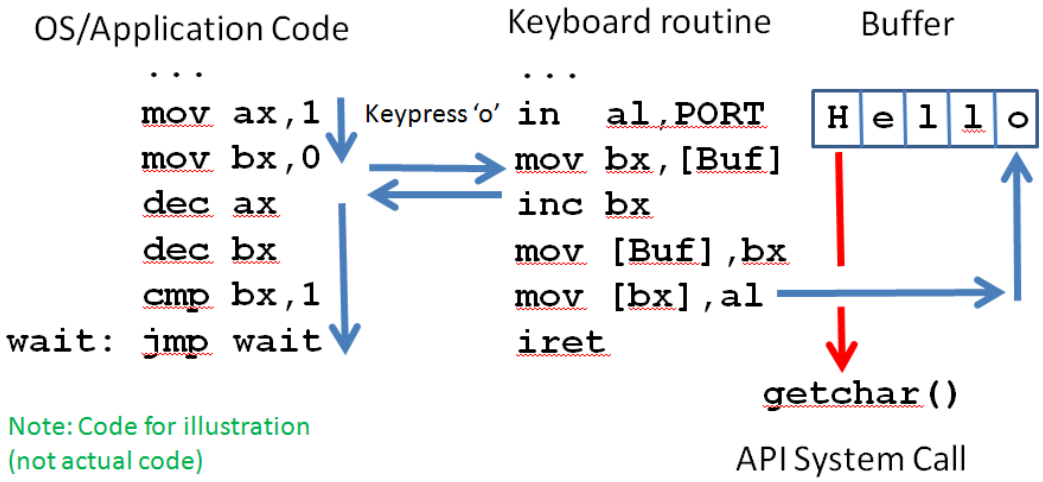


Figure 1: Illustration of Interrupt concept

Interrupts are at the centre of every operating system and for this reason they are very difficult to get at due to the security and reliability consequences if they are improperly accessed. Modern CPU architecture provides OS designers with robust methods of preventing user mode access to them.

We could use the Windows Driver Development Kit to study Interrupts, but considering we only have few hours to study this section of the module this would be a bit too ambitious.

It is still possible to get some basic information about the interrupts from within Windows, for example running ControlPanel/DeviceManager allows you to see which Interrupts are being used by each piece of hardware. For example these are the details provided for the mouse.

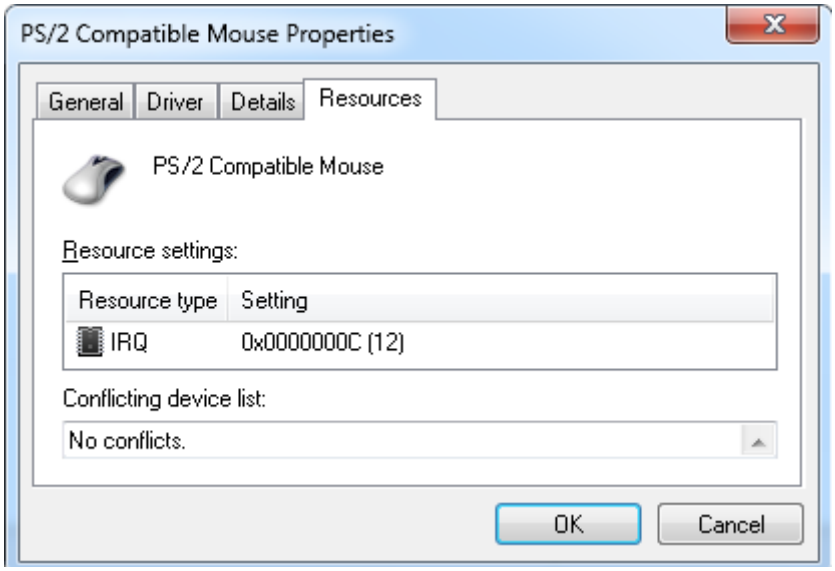


Figure 2: Using the Control Panel Device Manager to view the Mouse Properties.

The IRQ referred to gives the position in the Interrupt Descriptor Table of the details required to run the mouse service routine. On a modern PC each interrupt IDT entry consists of 8 bytes.

To find the IDT entry for interrupt "12" you would take the base address for the vector table, IDTR, (on 8086 it is always 0, but on anything after 80286 it can be set).

IDT(Address) = IDTR + 8xInterruptNo

The 8 bytes at the IDT location contain the information giving the location of the code to be run so as to service the interrupt. Each of these eight bytes that identifies an interrupt service routine is known as a "gate". The 8 bytes are used in the following way.

```
typedef struct _CALL_GATE
{
        USHORT OffsetLow;               // address of ISR (low 16 bits)
        USHORT Selector;
        UCHAR NumberOfArguments:5;
        UCHAR Reserved:3;
        UCHAR Type:5;                   // Trap,Interrupt,Task...Gate
        UCHAR Dpl:2;                    // 0 exceptions/hw, 2 software
        UCHAR Present:1;                // Present, Interrupt active?
        USHORT OffsetHigh;              // address of ISR (low 16 bits)
}CALL_GATE,*PCALL_GATE;
```

To Do 1: To identify the interrupts in use on your computer, open a terminal window and launch control panel as follows,

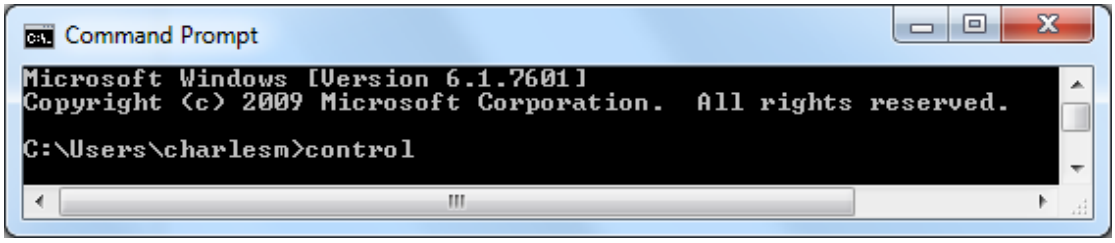


Figure3: Obtaining Control Panel on a laboratory PC.

Select "System" and then select "Device Manager", accept the warning that you can't change the system with being an adminstrator.

Some devices will have a –ve IRQ number these are Message Signaled Interrupts rather than true wired Harware Interrupts we studied in the lecture.

On modern systems the PCI bus supports an interrupt protocol whereby a device requiring an interrupt service routine can write a "message" to a piece of shared memory that can inform the processor of the request. This is called a Message Signaled Interrupt. To sperate MSI interrupts from conventional Harware Interrupts, Microsoft given them –ve values (or if not using two's complement, high binary values).

The devices that make use of Interrupts on your PC in the laboratory are the following,

Intel(R) 8 Series/C220 Serial USB EHCI #1 – 8C26
Intel(R) 8 Series/C220 Serial USB EHCI #2 – 8C26
Intel(R) Ethernet Connection I217-LM
Intel(R) 8 Series/C220 SATA AHCI Controller – 8C02
Intel(R) HD Graphics 4400
Intel(R) 3.0 eXtensible Host Controller

Expand Windows description of the hardware connected to the computer so that it looks something like that shown below.

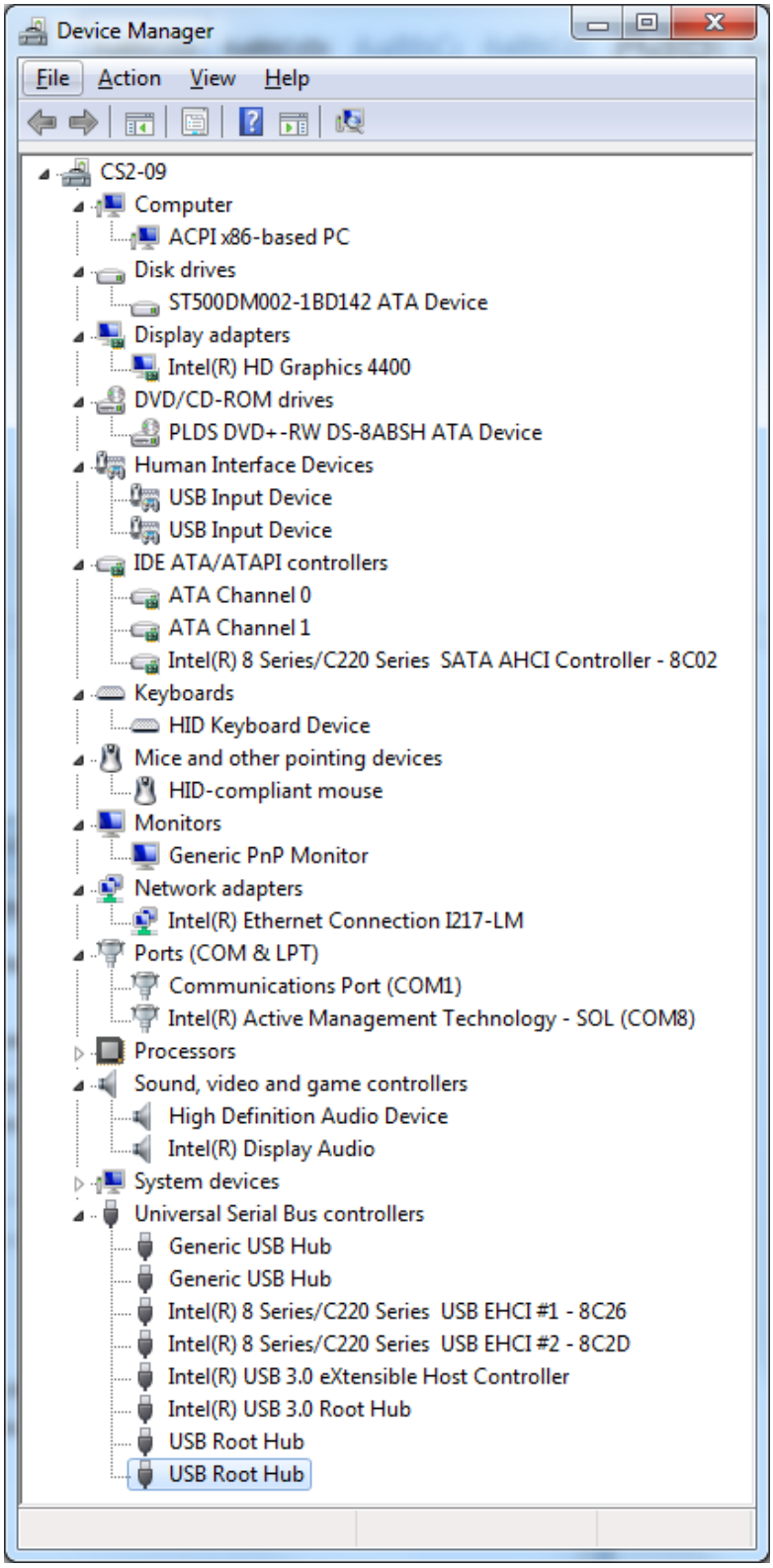


Figure 4: Device Manager display for a typical Laboratory PC.

For each of the listed devices identify the IRQ number defining its identity. Answers on **quiz Q1**.

Part 2: Interrupts on DOS, we need to write directly to screen memory

Aside: To do work with interrupts in the time available we will need to return to an older operating system that doesn't lock us out from experimenting with interrupts. The concepts and ideas are still valid on a modern computer.

The OS (operating system) may appear to be just another program that loads into memory and is then used to load into memory other programs and cause then to execute. The flow of control would appear to be from OS to the Application and then back to the OS. Although true, the reality is slightly different. The OS loads up lots of useful routines to do various things with the hardware of the machine. In some cases you can just call these routines as part of your program to do the things you want to do. An example of this would be the INT 0x21 used to print characters to the screen. However the computer also needs to be able to react to events that are occurring that are not under its control. Examples may include the mouse being moved or key being pressed (INT 0x09). When these occur, the CPU must drop everything and see how the mouse has moved or which key was pressed before the moment passes and the chance to gather information is lost. Thus an OS and CPU is essentially a reactive system, waiting for events to occur and acting on them. Interrupts are the mechanism that allows this to happen. In effect the OS will idle with "*label: jmp label*" until interrupted to do something different. Modern processor can save power while idling. It should be noted that some simple microprocessors do not have an interrupt system.

We have already looked at "event driven programming" last week where created code that responded to timer events and mouse click events. This reaction to an event is central to the understanding of interrupts.

When interrupted (by hardware or software) the microprocessor does the follow three things,

1/ suspend execution of the main program,

2/ calls a procedure that services the interrupt (interrupt handler or interrupt service routine),

3/ returns control to the mainline program.

The mainline program is essentially unaware that the service routine occurred. Although it may make use of data gathered by the interrupt service routine, or be delayed very slightly by the event.

On the 8086 processor there are three types of interrupt. These are

Hardware Interrupts:
An external electrical signal applied to the INTR (interrupt) pin or NMI (interrupt) pin of the CPU causes an interrupt. For example the keyboard controller interrupts the CPU each time you press a key causing IRQ1 (INT9) to run, this routine detects the key being pressed and updates the keyboard buffer. Although there is only one interrupt line on the 8086, Intel provide a support chip (8259) that allows 8 Interrupt signals to be multiplexed into the CPU. Most modern PCs contain two such chips and allow 16 different hardware devices to interrupt the CPU. Some hardware can share an interrupt but in such cases the service routine must check with each device to see which needs servicing. This can be inefficient.

Software Interrupts
Caused by the execution of an assembly language INT command (part of a program). It can be considered as an indirect call to a function in the OS. For example INT 21h can be used to print a character contain in dl on a text screen (when ah=2). The hardware may be different on various PCs, but they will all have an INT 21h routine that will do exactly the same thing (assuming that they are PCs).

Exception Interrupts
Internal errors such as divide by zero, or parity errors in memory mean that the normal flow of the program can not continue. These trigger a non-maskable hardware interrupt. The service routine for this would normally stop the process from proceeding and display some error message. Similar to the BSD (Blue Screen of Death) that you sometimes see in Windows when something internal goes wrong. Exception interrupts are essentially hardware interrupts that are triggered by faults.

How do Interrupts work on the 8086?

Some definitions first:

Interrupt Service Routine: A piece of machine code that is designed to interact with the hardware. There are service routines to control the hard disk, mouse, keyboard, screen, printer, etc. All service routines end with the instruction IRET.

The Vector Table: The Vector Table is a look up table that allows the CPU to identify the location in memory of the start of a service routine. The CPU is passed INT/IRQ number and looks up the location of the corresponding service routine in the vector table. The Interrupt Vector Table is located in RAM at 0000:0000 to 0000:03FF, i.e. the first 1024 bytes of RAM/Address space. Each interrupt vector is described by 4 bytes, two for the Code Segment (CS) and two for the Instruction Pointer (IP). A total of 256 Interrupt Vectors can be stored in the table (i.e. 16 Hardware and 240 Software).

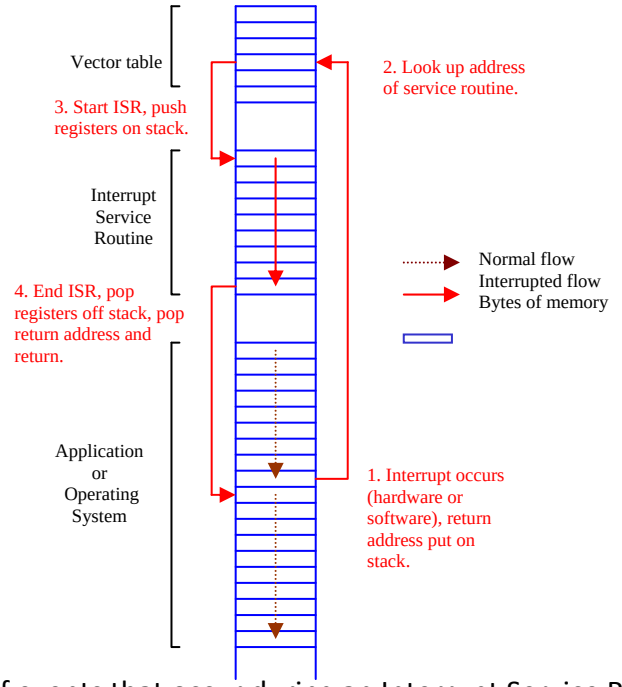


Figure 5: Sequence of events that occur during an Interrupt Service Routine call.

Hardware Interrupt: The CPU is executing an application or waiting in the OS. A hardware event (say the keyboard is touched) causes the CPU to suspend normal execution of the application. The CPU then looks up the starting address of the (keyboard) service routine. The CPU then runs the service routine. Finally the CPU returns to executing the application at the point where it was interrupted

Software Interrupt: The CPU encounters an INT command in a program, put there by the programmer. It then executes the service routine specified by the interrupt number in the INT command. Control is then returned to the program. It is very

similar to a CALL command except that the vector-table species the location of the sub-routine rather than it being part of the instruction.

TSRs

A normal program such as 'Hello World' is loaded into memory, executed and then removed from memory. However Service Routines need to be left in memory so that they can be run when hardware events occur. To leave a programme behind in memory is referred to as terminate and stay resident or TSR. Terminate and stay resident programs have very many uses. They were originally used in DOS based applications to provide some element of multitasking. Using TSR's it is possible to create pop up programs such as Sidekick or Spell checkers that run in the background of existing software. Since they remain resident they should be used with care. After developing a TSR do not do anything critical until you reboot your machine. Routines in TSRs are normally invoked by interrupts.

Operating systems such as Windows do things a little differently; they have an event manager that notifies each application when a relevant event has occurred. The relevant application then services the event. The event manager is different to an interrupt in that the request to service an event is put on a queue. However events can be put on the queue by interrupts.

To Do 2: When using studying service routines we can't use int 0x21 to put text on the screen. The hardware interrupts we are interested in have a lower interrupt number and are so of high priority. To get some text on screen we will need to write directly to memory. The graphics card allocates memory to the console as follows,

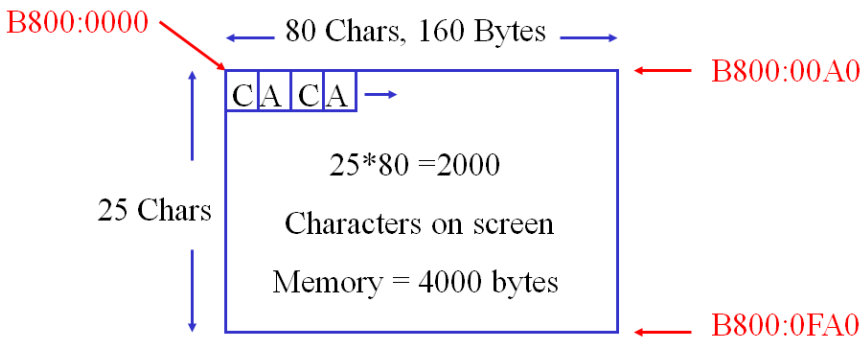


Figure 6: Memory used by graphics card to map console screen.

The base address is at 0x8000 and each character location on the screen is defined by two bytes, the first byte is the ASCII character code and the second byte is an attribute defining how the text should appear. There are 80x25 characters on a text screen.

To locate the memory addresses for a specific character position (x,y) on screen you would need to do the following

Character $Address = 0x8000 + (2*80*y) + x$

Attribute $Address = 0x8000 + (2*80*y) + x + 1$

The following MASM program allows you to my initials to the screen in the top left hand corner of the screen. Modify the code so that it writes your initials on line 10, column 12 (assuming top left hand corner is line=0, column=0) the first initial should be yellow on a black background; the second initial should be bright blue on a yellow background (it may appear a bit orange).

The attribute byte is coded as follows by the graphics card.

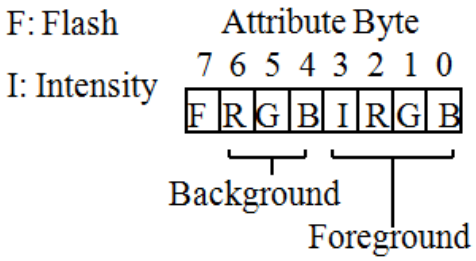
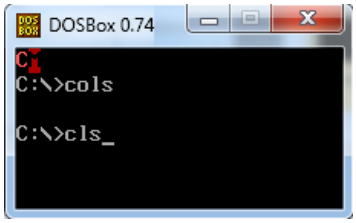


Figure 7: Coding of attribute byte (e.g. bright red on a black background would be 4+8=12)

You will need to Clear Screen before you run the code as the text will scroll off the top the screen. Just type “cls <ret>”.




```
        .MODEL    small
        .STACK
        .DATA
        .CODE

        .STARTUP

        mov   ax,0b800h   ; Base address of screen memory
        mov   es,ax   ; Use extra segment register to access 0x800...
        mov     bx,0      ; Screen position x=0, y=0, (2*((80*y)+x))=0

        mov     al,'C'     ; Print 'Hello' on the screen
        mov     es:[bx],al
        inc     bx
        mov     al,12      ; Bright red on a black background
        mov     es:[bx],al

        inc     bx        ; Black on a red background
        mov     al,'M'
        mov     es:[bx],al
        inc     bx
        mov     al,64
        mov     es:[bx],al
quit:       .EXIT
END
```

Figure 8: Code writes directly to screen memory

Answers on **quiz Q2**.

Part 3: Interrupts on DOS, investigation of the Timer and Keyboard interrupts

The timer interrupt on DOS OS machine is called 18.2 times per second (1092 per minute). On a modern Windows PC the interrupt timer interrupt is called about 64 times per second. The keyboard interrupt is called each time a key is pressed. Both the keyboard and timer are examples of hardware interrupts.

Compile and run the t1.asm program in the DOS Box emulator. This code will produce a counter on the display that increments each time the timer interrupt is activated.

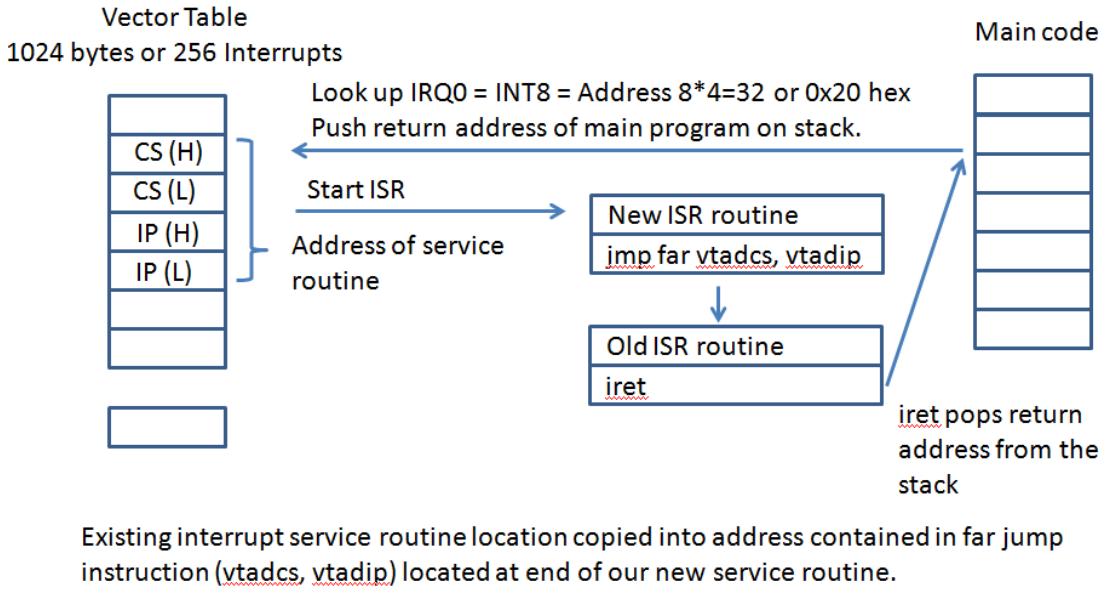


Figure 9: Overview of t1.asm

Compile and run the k1.asm program in the Dos Box emulator. This code will print "Hello" on the screen each time the 'h' key is pressed on the keyboard. The keyboard interrupt number is IRQ1 or INT9 found at address 0x24 in the vector table. Hardware interrupts are normally given an IRQ number on a DOS machine the IRQ number is the interrupt number less 8.

Both programs are well commented. Your task is to do the following...

1/ Create a new program called kt1.asm that will increment the counter each time the <space> key is pressed on the keyboard (required). I would suggest starting with k1.asm and resaving it as kt1.asm and then adding the code you need from t1.asm. See the listing later in this handout for help but use the downloaded asm files to avoid rogue characters appearing.

2/ Reset the counter to 0000 when the <ret> is pressed (optional).

When working this new code will run in the background program and keep track on the paragraph word count as you work on other programs within the DOS window.

A few words of warning: Once you redirect the vector table the effect is permanent, you may need to regularly restart DOS Box to produce a clean vector table. Be prepared to start again a few times. The key-codes used by the keyboard are not ASCII so look at the table at the end of this document.

Show your kt1.asm program working to a demonstrator and complete the quiz.

Well done!

```
;/* k1.asm Detects keypress from the keyboard      */
;/* If the key presses is 'h' or'H'                */
;/* the routine prints 'Hello' on the screen       */

; By  C.Markham
; Revised  March 2016

        .MODEL   small
        .STACK
        .DATA                       ; DS will may change when TSR operates
        .CODE

        .STARTUP

        push    es                  ; Save existing value of ES.
        mov     ax,0000h            ; Set ES (extra segment) to page 0
        mov     es,ax               ; Note page 0 is vector address table
        mov     bx,024h             ; Vector address for keyboard INT 9, IRQ1
        mov     ax,es:[bx]          ; ax=[0000:0020], CS to jump to at end of our new ISR
        mov     vtabip,ax           ; Store the CS, 1st two bytes of table
        inc     bx
        inc     bx
        mov     ax,es:[bx]          ; ax=[0000:0020], IP to jump to
        mov     vtabcs,ax           ; Store the IP, 2nd two bytes
        pop     es                  ; Restore the value of es

        push    es                  ; Change vector address to ISR routine
        mov     ax,0000h            ; Vector address table segment
        mov     es,ax
        mov     bx,024h
        mov     ax,OFFSET isr       ; Set IP of vector table to new ISR
        mov     es:[bx],ax
        inc     bx
        inc     bx
        mov     ax,cs               ; The ISR is in the same segment as the program.
        mov     es:[bx],ax          ; New CS in vector table is the code segment of
        pop     es                  ; this program.

        mov     ah,031h             ; TSR Entry code ah=31h,al=0h
        mov     al,00h
        mov     dx,1024             ; DX=no of bytes to remain resident
        int     021h                ; Terminate and stay resident.

        jmp     quit                ; Historic code, never reached


isr:    push    ax                  ; Start of code you wish to call ever 50mS (18ms PC).
        push    bx                  ; Interrupts don't store general registers, you must.
        push    cx
        push    dx
        push    di
        push    si
        push    bp
        push    es
        push    ds

        mov     dx,60h              ; Read the last scan code
        in      al,dx
        and     al,127              ; Mask MSB 1=Key Down, 0=Key Up remove if just up or down
        cmp     al,23h              ; Was a h/H pressed on the XT keyboard
        jnz     ovr1
```

Change to <space> scan code rather than 'h'. Remove the 'and al,127'

```
        mov     ax,0b800h           ; Base address of screen memory
        mov     es,ax
        mov     bx,140      ; Screen position (1,70)

        mov     al,'H'              ; Print 'Hello' on the screen
        mov     es:[bx],al

        inc      bx
        inc     bx
        mov     al,'e'
        mov     es:[bx],al

        inc      bx
        inc     bx
        mov      al,'l'
        mov     es:[bx],al

        inc     bx
        inc     bx
        mov     al,'l'
        mov     es:[bx],al

        inc     bx
        inc      bx
        mov     al,'o'
        mov     es:[bx],al

        jmp     ovr2

ovr1:   mov     ax,0b800h     ; If not an H overwrite top right with ' '
        mov     es,ax
        mov     bx,140        ; Screen position (1,70)
        mov     cx,5
again:  mov     al,' '
        mov      es:[bx],al
        inc     bx
        inc     bx
        loop    again


ovr2:   mov   dx,020h       ; Non Specific end of interrupt command
        mov   al,020h       ; to sent 8259 #1 controller.
    out dx,al
        pop   ds            ; Restore all registers previously save on
        pop   es            ; the stack.
        pop   bp
        pop   si
        pop   di
        pop   dx
        pop   cx
        pop   bx
        pop   ax

        db      0EAh                ; jump far  (instruction created on the fly...)
vtabip  dw 0                        ; IP offset
vtabcs  dw 0                        ; CS segment

quit:   .EXIT                       ; Historic code, never reached

END
```

Remove this service routine and replace it with a counter.

```
; T1.asm Test code to redirect Timer Interrupt to our
; own Service routine, old service routine is
; run after ours.  The program terminates and
; stay resident.  The demo service routine is
; a four digit counter.

; By C.Markham
; March 2016
; Requires Win95/DOS (not NT), 8086+

     .MODEL          small
     .STACK
     .DATA                          ; DS will may change when TSR operates
     .CODE

     .STARTUP

     jmp ov1                        ; jump over our data so it is never run

dseg   db    '0','0','0','0'       ; Store variables in code segment (TSR)

;The actual location of these variables from part of a long jump
;to the existing service routine.
;vtabip dw          0               ; Vector table,      IP
;vtabcs dw          0               ; Vector table,      CS

ov1: push           es              ; Save existing value of ES.
     mov            ax,0000h        ; Set ES (extra segment) to page 0
     mov            es,ax           ; Note page 0 is vector         address table
     mov            bx,020h         ; Vector address for timer INT 8, IRQ0
     mov            ax,es:[bx]      ; ax=[0000:0020], CS to jump to after our ISR has run
     mov            vtabip,ax       ; Store the CS, 1st two bytes of table
     inc            bx
     inc            bx
     mov            ax,es:[bx]      ; ax=[0000:0020], IP to jump to
     mov            vtabcs,ax       ; Store the IP, 2nd two bytes
     pop            es              ; Restore the value of es

     push           es              ; Change vector address to ISR routine
     mov            ax,0000h        ; Vector address table segment
     mov            es,ax
     mov            bx,020h
     mov            ax,OFFSET isr   ; Set IP of vector table to new ISR
     mov            es:[bx],ax
     inc            bx
     inc            bx
     mov            ax,cs           ; The ISR is in the same segment as the program.
     mov            es:[bx],ax       ; New CS in vector table is the code segment of
     pop            es              ; this program.


;*********************


; TSR Code keeps program memory resident
     mov             ah,031h         ; TSR Entry code ah=31h,al=0h
     mov            al,00h
     mov            dx,1024          ; DX=no of bytes to remain resident
     int            021h            ; Terminate and      stay resident.
     jmp            quit            ; Historic code, never reached
```

You will need to store the counter values in your new program.

```
; The New Interrupt Service Routine

isr:   push    ax          ; ISR is called 18.2 times per second (Apx 55mS).
       push    bx          ; Interrupts don't store general registers, you must.
       push    cx
       push    dx
       push    di
       push    si
       push    bp
       push    es
       push    ds


;+++++++++++++++++++++++++++++++
; Core code of the service routine (what you want it to do each
; time the timer interrupt is called)

; Four digit counter, advance +1 on each call of the routine

       mov     cx,4
       mov     bx,OFFSET dseg
back:  mov      al,cs:[bx]      ; Load al with character code in [dseg]
       inc     al              ; Increase al by 1, next character.
       mov     cs:[bx],al       ; Store the increased value
       cmp      al,':'         ; If al       equals character after 9
       jnz     skip            ; If it       is not 10 do nothing
       mov     al,'0'          ; Otherwise reset character to'0'
       mov     cs:[bx],al
       inc     bx              ; Move onto next digit
       loop    back            ; Repeat on the successive digits

skip:  mov     ax,0b800h       ; Base address of screen memory
       mov     es,ax
       mov     bx,OFFSET dseg  ; First digit
       mov     dx,140          ; Screen position (10,40)=2*(10*80+40)
       mov     cx,4


nextc: mov     al,cs:[bx]       ; Read character from our store

       mov     di,bx           ; Exchange dx,bx
       mov     bx,dx
       mov     dx,di

       mov     es:[bx],al       ; Character changed on screen to ASCII al

       mov     di,bx           ; Exchange dx,bx
       mov     bx,dx
       mov     dx,di

       inc     bx
       dec     dx
       dec     dx
       loop    nextc
;+++++++++++++++++++++++++++++++


ovr:   mov     dx,020h         ; Non Specific end of interrupt command
       mov     al,020h          ; to sent 8259 #1 controller.
       out     dx,al
       pop     ds              ; Restore all registers          previously save on
       pop     es              ; the stack.
       pop     bp
```

This code will display and increment the counter.

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

        db      0EAh            ; jump far to the existing Timer routine
vtabip  dw 0                    ; IP offset
vtabcs  dw 0                    ; CS segment

quit:   .EXIT                   ; In TSR mode this is not reached

END
```

(Keyboard scan-codes available inside keyboard isr, not ASCII)
Make: Key down, Break: Key released

Key	Make	Break	Key	Make	Break
Backspace	0E	8E	F1	3B	BB
Caps Lock	3A	BA	F2	3C	BC
Enter	1C	9C	F3	3D	BD
Esc	01	81	F4	3E	BE
Left Alt	38	B8	F7	41	C1
Left Ctrl	1D	9D	F5	3F	BF
Left Shift	2A	AA	F6	40	C0
Num Lock	45	C5	F8	42	C2
Right Shift	36	B6	F9	43	C3
Scroll Lock	46	C6	F10	44	C4
Space	39	B9	F11	57	D7
Sys Req (AT)	54	D4	F12	58	D8
Tab	0F	8F			

Keypad Keys	Make	Break
Keypad 0 (Ins)	52	D2
Keypad 1 (End)	4F	CF
Keypad 2 (Down arrow)	50	D0
Keypad 3 (PgDn)	51	D1
Keypad 4 (Left arrow)	4B	CB
Keypad 5	4C	CC
Keypad 6 (Right arrow)	4D	CD
Keypad 7 (Home)	47	C7
Keypad 8 (Up arrow)	48	C8
Keypad 9 (PgUp)	49	C9
Keypad . (Del)	53	D3
Keypad * (PrtSc)	37	B7
Keypad -	4A	CA
Keypad +	4E	CE

Key	Make	Break	Key	Make	Break
A	1E	9E	N	31	B1
B	30	B0	O	18	98
C	2E	AE	P	19	99
D	20	A0	Q	10	90
E	12	92	R	13	93
F	21	A1	S	1F	9F
G	22	A2	T	14	94
H	23	A3	U	16	96
I	17	97	V	2F	AF
J	24	A4	W	11	91
K	25	A5	X	2D	AD
L	26	A6	Y	15	95
M	32	B2	Z	2C	AC

Key	Make	Break	Key	Make	Break
1	02	82	-	0C	8C
2	03	83	=	0D	8D
3	04	84	[	1A	9A
4	05	85	]	1B	9B
5	06	86	;	27	A7
6	07	87	'	28	A8
7	08	88	`	29	A9
8	09	89	\	2B	AB
9	0A	8A	,	33	B3
0	0B	8B	.	34	B4
			/	35	B5