

Chapter 8

I/O Devices

8.1 Overview

In this chapter I/O instructions are discussed in the context of the interface requirements of common I/O devices such as the M6845 chip used for video control functions, the I8259 Programmable Interrupt Controller, and the I8253 Programmable Interval Timer (PIT). Interrupts and the Interrupt Vector Table are also discussed.

8.2 I/O Instructions

I/O devices are connected to the computer through I/O circuits. Each of these circuits contain I/O registers denoted as *I/O ports*. I/O ports function as transfer *gates* between the CPU and I/O devices. Alternatively, several microprocessors support *memory-mapped I/O* where a memory location is assigned to be an input or output port from and to a specific I/O device. Thus, I/O operations are treated as reading or writing operations to the assigned memory locations. All processors in the IA family access external devices through I/O ports. I/O instructions exchange data, control signals, or device status information with ports through the *al* or the *ax* register; therefore, the size of data items transferred is 8 bits or 16 bits. Most devices transfer 8 bits at a time for which the *al* register is used. Slow devices use serial communication that transfer data one bit at a time. Examples of slow devices include the keyboard, modems, etc. Parallel communication devices transfer one data item at a time; examples of parallel devices include disk drives, printer controllers, etc.

The *cpu* addresses ports numbered from 0 to $2^{16} - 1$ and a particular device may use one or more pre-assigned ports. Examples of devices with pre-assigned ports are shown in Table 8.1.

Table 8.1: Common pre-assigned I/O Ports

| Device | Port numbers |
|-------------------------|--------------|
| Video Adapter (CGA) | 3D0h – 3DFh |
| Video Adapter (EGA) | 3C0h – 3CFh |
| I8259 PIC | 20h – 21h |
| Keyboard Controller | 60h – 63h |
| I8253 PIT | 40h – 43h |
| Serial Port (COM1) | 3F8h – 3FFh |
| Serial Port (COM2) | 2F8h – 2FFh |
| Parallel Printer port 1 | 378h – 37Fh |
| Hard disk | 320h – 32Fh |

Intel provides two I/O instructions to access ports: *in* and *out*. The syntax for the use of the *in* instruction is as follows:

$$\text{in } al/ax, dx/k \quad ; \quad al/ax \leftarrow port$$

For every input transfer the information is placed through the port into the *al* register for an 8-bit transfer or into *ax* for a 16-bit transfer. The port number is specified directly in the source field as an 8-bit constant or previously moved into the *dx* register. Besides holding port numbers up to $2^{16} - 1$ the *dx* register can be used to process, decrement, increment consecutive ports. The syntax of the *out* instruction is as follows:

$$\text{out } dx/k, al/ax \quad ; \quad port \leftarrow al/ax$$

For every output transfer the information is first moved into *al* or *ax* and send to the port which is specified as an 8-bit value in the destination field or, alternatively, previously moved into the *dx* register.

8.3 Video Display Adapters

Video adapter boards provide the interface between the motherboard and the video display monitor. The information displayed on the monitor is information that has

been written to video ram (VRAM) by the *cpu*. The video adapter controller (CRT controller) reads the information from *vram* and converts it to the appropriate color and brightness signals to be displayed on the screen. These signals are sent to the screen in synchronized horizontal and vertical movements that scan the entire screen at fixed frequencies. Video boards can be programmed in graphics and text mode. In graphics mode the unit of display is a *pixel* while in text mode an entire alphanumeric character is displayed (character box) requiring each, a given number of pixels that changes from adapter to adapter. The IBM PC introduced in 1981 supported an MDA (monochrome display adapter) and a CGA (color display adapter). The MDA was designed for text display only and the CGA provided both text and color graphics on the screen. The CGA displays a maximum of 80 characters per line and 25 lines and each character requires an 8×8 box. It used the Motorola 6845 CRT controller. Seven different video modes (00h – 06h) supported by the CGA allow a display resolution from 320×200 to 640×200 pixels. Video memory starts at *B8000* and takes 16K bytes. Since a screen requires 4K bytes then 4 pages of text can be displayed at a given time.

The EGA (enhanced graphics adapter) was introduced in 1985 emulates the MDA and CGA but with improved resolution up to 640×350 and additional video modes (0Dh, 0Eh, 0Fh, 10h). The M6485 was replaced by a set of LSI chips as the CRT controller.

The VGA (video graphics array) introduced in 1987 is a single chip that integrated the same set of functions performed by LSI chips on the EGA. The term VGA is used to refer to the entire adapter. The VGA generates analog RGB output (color signals) with a resolution of up to 720×400 for text modes and 640×480 for graphics modes. The VGA emulates all the display modes of the MDA, CGA, and EGA adaptors plus it provides the additional modes: 11h, 12h, and 13h. A color look-up table allows 256 different colors to be displayed on the screen at one time. Up to 1 megabyte extra of memory can be used for graphics applications, i.e., to store pixels and attributes. Since the VGA is used to emulate the CGA text, the starting address for video is *B8000h* this is achieved by selecting the video mode 03h; for the MDA the video buffer starts at *B0000h* with the selection of the video mode 07h.

XGA was developed by IBM as a standard for high-performance desktops and workstations. XGA boards support displaying resolutions up to 1024×768 . The super VGA adapters (SVGA) support resolutions in the range from 640×400 to 1280×1024 . There was no standardization of the additional graphics modes on the new SVGA boards, and a separate driver software was needed for every graphics to use these enhanced modes. The lack of a widely accepted standard was addressed by the Video Electronics Standards Association (VESA), a consortium of video adapter and monitor manufacturers created to standardise video protocols, and a family of

video standards was developed to support backward compatibility with VGA and greater resolution modes. VESA's SXGA standard supports 1280×1024 resolutions in monitors with a standard ratio of 5:4; however the traditional 4:3 aspect ratio found in the majority of computer monitors is provided by VGA, SVGA, XGA and UXGA.

Pixels are smaller at higher resolutions which combined with the ability to scale objects, and the option to use different font sizes, make it possible to adjust resolutions with larger screens; for example, it is possible to use 17in monitors at screen resolutions of up to 1600×1200 pixels and even 21in monitors with 1800×1440 pixels with UXGA adapters.

8.3.1 Important BIOS data

Text and graphics modes can be easily programmed using BIOS functions via INT 10h software interrupts. For example, to change the video mode with INT 10h, the code function 00h is selected and loaded in *ah*; the video mode chosen is moved into *al*. This flexibility is illustrated with the following code that selects the CGA text mode and displays a character throughout the entire 25×80 screen.

```

..start:
    mov ah, 00h           ;CGA text mode
    mov al, 03h           ;of 80x25
    int 10h

    mov ah, 09h           ;display
    mov bh, 00h           ;on page 0
    mov al, 'X'           ;a character
    mov cx, 2000h         ;on the entire screen
    mov bl, 47h           ;with this attribute
    int 10h

    mov ax, 4c00h         ;return to dos
    int 21h

```

A set of BIOS locations are also updated each time BIOS display functions are called. It is possible to program video adapters directly but this also requires knowledge of which BIOS locations must be updated. A direct programming of some I/O devices are discussed in the next subsections that require access to the BIOS data described in Table 8.2. This information is transcribed from Wilton's Programmer's Guide to PC Video Systems [4].

Table 8.2: Important BIOS data

| Address | Name | Type | Description |
|------------|-------------|------|---|
| 0000:0449h | CRT_MODE | Byte | Current video mode |
| 0000:044Eh | CRT_START | Word | Offset of current page (in bytes) |
| 0000:0050h | CURSOR_POS | Word | Array of eight words containing the cursor position (row and column) for each of eight possible video pages High order byte: row Low order byte: column |
| 0000:0460h | CURSOR_MODE | Word | starting and ending lines of cursor High order byte: starting line (top) Low order byte: ending line (bottom) |
| 0000:0462h | ACTIVE_PAGE | Byte | Currently displayed video page number |
| 0000:0463h | ADDR_6845 | Word | I/O port of the CRTC's address register monochrome CRTC:03b4h CGA:03d4h |

8.3.2 Programming the M6845 CRTC

The M6845 has 19 8-bit internal data registers. The following 6 registers can be used to illustrate direct programming applications:

| | |
|-----------------------------------|-----|
| Cursor start: | 0Ah |
| Cursor end: | ABh |
| High byte page offset (in words): | 0Ch |
| Low byte page offset (in words): | 0Dh |
| High byte cursor location: | 0Eh |
| Low byte cursor location: | 0Fh |

An update of internal data registers as well as an update of the BIOS data is necessary if direct programming applications such as a change of page, cursor location, and cursor size are implemented. For example, consider a function to change the active page with the following prototype:

```
change_page(int page)
```

This function will need to update the following BIOS data at 0000 : 0462h (ACTIVE_PAGE) with the number of the new active page, and update the contents of 0000 : 044Eh (CRT_START) with the offset in bytes of the new page. The internal registers that must be updated are 0Ch and 0dh with the offset of the new

page in words. The following are general steps for an assembly implementation of *change_page*:

1. Set up access to stack frame where the new *page* will be stored,
2. Get new page number from the stack frame,
3. Update ACTIVE_PAGE,
4. Calculate offset in words: $2048 \times \text{page}$,
5. Update internal registers with the new page offset.

(a) Transfer low-byte:

```

mov dx, word es:[ADDR_6845]      ;port address
mov al, 0dh                      ;select internal register
out dx, al
inc dx                          ;data port
mov al, low_byte                 ;transfer data
out dx, al

```

(b) Transfer high-byte:

```

dec dx                          ;port address
mov al, 0ch                     ;select internal register
out dx, al
inc dx                          ;data port
mov al, high_byte               ;transfer data
out dx, al

```

6. Update CRT_START (in bytes)

Note that the statements *mov al, low_byte* and *mov al, high_byte* refer to an 8-bit transfer of a constant value, or from a 16-bit register or memory location.

A second illustration involves programming the cursor position. This procedure will have to update the BIOS entry *CURSOR_POS* and the internal registers *0eh* and *0fh* with the offset of the cursor position in video memory. The prototype of the function is given as follows:

```
CUR_POS(int r, int c, int p)
```

where *r* and *c*, define the coordinates (row and column) of the cursor position, and *p* is the number of the display page. The following steps describe the implementation of *CUR_POS*:

1. Set up the stack frame access,
2. Get the row, column, and page from the stack,
3. Compute the cursor offset in words: $80r + c + 2048p$,
4. Transfer low byte to *0eh*,
5. Transfer high byte to *0fh*,
6. Update *CURSOR_POS* at 0000 : 0450h. Update the entry $2p$, which corresponds to page p .

A third application involves determining the cursor size. The prototype of this function is as follows:

```
CUR_SIZE (int start, int end)
```

The implementation of *CUR_SIZE* is summarized in the following steps:

1. Set up the stack frame access,
2. Get "start" from the stack,
3. Move it to register *0ah* in the M6845:

```
mov ax, word es:[ADDR_6845]
mov dx, ax
mov al, 0Ah
out dx, al
inc dx
mov al, start
out dx, al
```

4. Get "end" from the stack,
5. Move it to register *0bh* in the M6845,
6. Update *CURSOR_MODE*.

8.4 Interrupts: 16-bit Real Mode

An executing program can be interrupted by an external device, or through the direct call of interrupt subroutines. No instruction is interrupted during execution, however, before the fetching of the next instruction occurs, the interrupt flag is checked to see the existence of pending interrupts. If pending interrupts exists, the next instruction of the current program is not fetched and control is automatically transferred to a location in memory where an interrupt service routine (interrupt handler) is stored.

Processing requests via interrupts have the desired effect of speeding up service and minimize *cpu* idle time. Suppose a microprocessor reads a character in 10^{-5} seconds. If the microprocessor waits for example 10 seconds for a user to type a character then $\frac{(10-10^{-5})}{10} = .99999$ implies that 99.99% of the time the *cpu* is idle. With I/O interrupts the *cpu* is interrupted only when the character is ready. Another advantage of the use of interrupts is that several devices are interfaced and their requests queued and prioritized. Thus *external* interrupts are initiated by hardware devices external to the *cpu* and *internal* interrupts are initiated by a program, or exceptions such as division by zero, overflow. single step execution, or breakpoints. Interrupts may *maskable*, i.e., can be temporarily disabled, or *nonmaskable*, i.e., a dedicated hardwired signal cannot be disabled.

8.4.1 Interrupt Service

The following steps summarize the actions that a 8086/88 processor executes to service an interrupt:

1. Store the flag register in the stack (*pushf*):

$$[sp] \leftarrow \textit{Flag register}$$

2. Disable interrupts (CLI): $IF \leftarrow 0$
3. Saves return address (CS:IP):

$$\begin{aligned} [sp] &\leftarrow cs \\ [sp + 2] &\leftarrow ip \end{aligned}$$

4. Locates interrupt vector in the interrupt vector table and updates the CS:IP with the address of the interrupt handler:

$$\begin{aligned}cs &\leftarrow IVT_{16}[4 \times n] \\ip &\leftarrow IVT_{16}[4 \times n + 2]\end{aligned}$$

where n is an integer identifying the interrupt type and can be provided by an external hardware device or through the software call *int n*. Steps 1-3 are indivisible, i.e., no interrupt will be served if it occurs during this short period of time.

8.4.2 Interrupt Handler

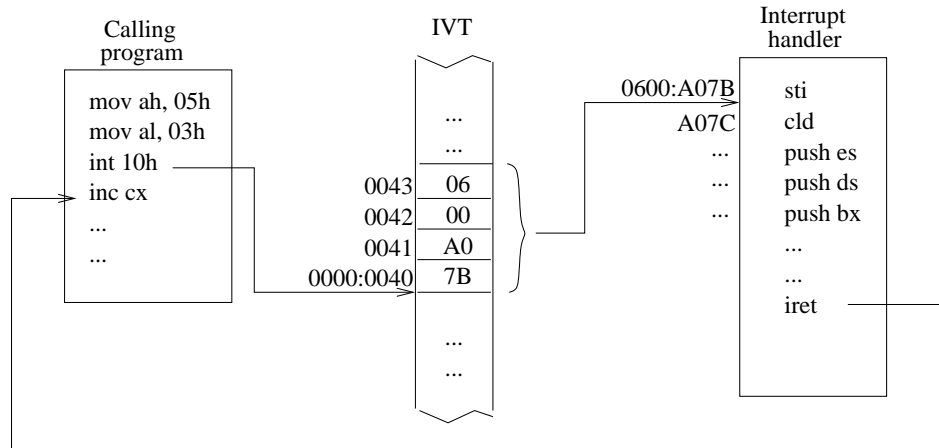
Steps:

1. Enables interrupts (STI)
2. Saves working registers
3. Executes code to service interrupt
4. Restores working registers
5. Executes an IRET:
 - restores return address; old CS:IP,
 - restores flags register (POPF)

8.4.3 Interrupt Vector Table (IVT)

The interrupt vector table refers to a memory section from 00000h to 003FFh in real-mode. The table consists of 256 entries; each entry stores an interrupt vector, which is a pointer to the location of the interrupt handler activated by an *int n* instruction. As part of the *cpu* interrupt service of an *int n* the n th entry of the IVT is accessed and its contents are placed in the *cs:ip* registers before the interrupt handler takes control of the *cpu*. Since each entry consists of four bytes, the integer n , which identifies the interrupt *type* is multiplied by 4 to form the correct index where the interrupt vector is located. For example *int 21h* will result in an index $21h \times 4 = 84$, therefore it will access the corresponding entry in the IVT at location 0000:0084h, and fetch the offset part of the pointer at 0084h and 0085h, and the segment part at 0086h and 0087h; the pair *segment:offset* fetched from the IVT points to the start of the interrupt service routine. Fig. 8.1 illustrates the interrupt execution cycle for *int 10h*.

Some relevant entries in the IVT are listed in Table 8.3.

Figure 8.1: The interrupt execution cycle for *int 10h*

8.4.4 Interrupt Vector Replacement

User's interrupt subroutines can replace the interrupt handlers provided by the system by simply replacing the contents of the selected entry in the IVT. Alternatively, the functionality of current handlers can be improved by executing, upon an interruption, the user code followed by the existing handler. Pointers to user's interrupt subroutines can also be inserted into available entries in the IVT.

The temporary replacement of existing handlers is carried out by implementing the following steps:

1. Fetch old interrupt vector,
2. Save it,
3. Insert new vector, and
4. Restore the old vector after execution

There are two alternatives to implement these steps: *int 21h* functions, or by direct access of the IVT.

Use of int 21h functions

1. To get the old pointer the code function *35h* is pre-loaded into the *ah* register; the interrupt type *n* is also pre-loaded into the *al* register. The interrupt call (*int 21h*) returns the old vector in the pair *es:bx*.

Table 8.3: Selected interrupt vector assignments

| Type | Offset | Operation | Type | Offset | Operation |
|------|-------------|-----------------|-------|-------------|------------------|
| 0 | 0000 – 0003 | Divide Overflow | 10 | 0040 – 0043 | Video services |
| 1 | 0004 – 0007 | Single step | 13 | 004C – 004F | Disk I/O |
| 2 | 0008 – 000B | NMI) | 14 | 0050 – 0053 | Serial port |
| 3 | 000C – 000F | Breakpoint | 16 | 0058 – 005B | Kboard services |
| 4 | 0010 – 0013 | Overflow | 17 | 005C – 005F | Printer services |
| 5 | 0014 – 0017 | Print screen | 19 | 0064 – 0067 | Bootstrap loader |
| 8 | 0020 – 0023 | Timer | 1A | 0080 – 0083 | Time of day |
| 9 | 0024 – 0027 | Keyboard | 21 | 0084 – 0087 | DOS services |
| A | 0028 – 002B | Reserved | 23 | 008C – 008F | CTRL-BRK |
| B | 002C – 002F | COM1 | 24 | 0090 – 0093 | Critical error |
| C | 0030 – 0033 | COM2 | 33 | 00CC – 00CF | Mouse interrupts |
| D | 0034 – 0037 | Hard disk | 60–6B | | User programs |
| E | 0038 – 003B | Floppy disk | 6C–7F | | Reserved |
| F | 003C – 003F | Printer | F1–FF | | User programs |

2. The old vector now in *es:bx* is saved into the stack or into a memory location determined by the user.
3. Before an interrupt call is made to insert the new vector, the user pre-loads *ah* with the function code 25h, the register *al* with the interrupt type *n*, and the pair *ds:dx* with the pointer to the user's interrupt handler.
4. After the handler executes, the old interrupt vector is restored by implementing an insertion into the IVT as in step 3.

The following code outlines an implementation using the *int 21h* functions:

```

segment    bss
            oldvector resw 2
segment    code
            :
            mov ah, 35h                ; fetch old vector
            mov al, n
            int 21h
            mov word [oldvector], bx    ;save it
            mov word [oldvector+2], es
            mov ax, seg ihandler        ;insert new vector
            mov ds, ax

```

```

        mov ds, ihandler
        mov ah, 25h
        mov al, n
        int 21h
        :
        mov ax, word [oldvector +2] ;restore old handler
        mov ds, ax
        mov dx, word [oldvector]
        mov ah, 25h
        mov al, n
        int 21h
        :
        :
ihandler: sti                                ; interrupt handler code
        :
        push registers used
        process interrupt
        pop registers used
        iret

```

Direct access to the IVT

Consider the handler for *int 0* which is generated internally each time an application runs into an instance of a division by zero. One possible reason to replace the vector for *int 0* is to avoid the return to the operating system each time a division overflow occurs. If the user is interested in returning to the interrupted program instead, then a user version to replace *int 0* can be implemented to support an interaction sequence that controls the outcome of a division overflow. A possible implementation is outline as follows:

```

segment    code
    msg1 db                                "enter dividend:"
    msg2 db                                "enter divisor"
    msg3 db                                "result:"
    msg4 db                                "divide overflow, 13h, 10h, try again"
    dividend resw 1
    divisor resw 1
    quotient resw 1
    remainder resw 1

segment    code

```

```

        ⋮
        mov di, 0                ; fetch old vector
        mov es, di
        push word [es:di]        ; and save it into the stack
        push word [es:di+ 2]

        mov ax, zero_div         ; insert new pointer
        mov word [es:di], ax
        mov ax, seg zero_div
        mov word[es:di + 2], ax

goagain:  mov bx, 0              ; used to flag an interrupt
          display msg1
          display msg2
          perform division
          cmp bx,0                ; check if interrupt occurred
          jne goagain
          display result

          mov di, 0                ; restore old vector
          mov es, di
          pop [es:di+2]
          pop [es:di]
          ret
          ⋮
          ⋮
          ;new handler code

zero_div: sti
          ⋮
          display msg4
          mov bx, 1
          ; flag interrupt
          ⋮
          iret

```

8.5 The I8259 Programmable Interrupt Controller

As shown in Fig. 8.2 a single I8259 chip receives interrupt requests from up to 8 different sources. These sources correspond to the interrupt types 8 to 15 in the IVT described in Table 8.3. Some of the functions that the I8259 performs include the queueing and prioritizing of interrupt requests, disable (mask out) selected requests, send an interrupt signal to the CPU, and acknowledge the CPU response by sending the interrupt vector number. The CPU uses this number to service the interrupt and to access the IVT and fetch the address of the corresponding interrupt service routine.

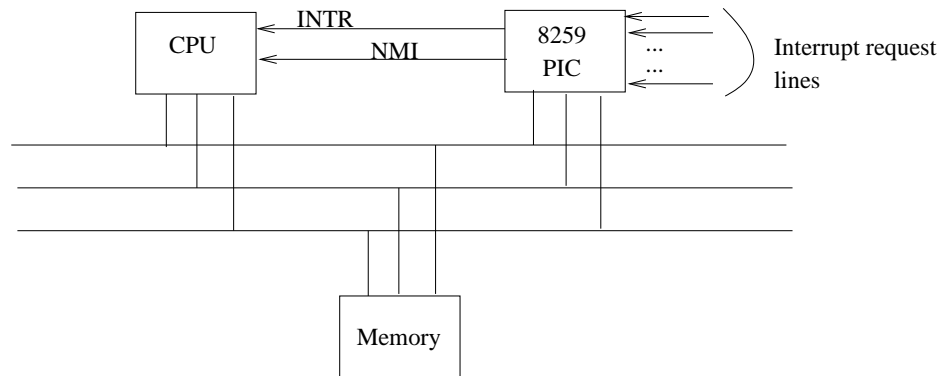


Figure 8.2: The I8259 interface with the *cpu*

Access

- The instruction: *in al, 21h* will transfer a byte from the I8259 to the *al* register. The port address *21h* is used to access the *IRQ* lines (IMR - Interrupt Mask Register).
- The instruction: *out 21h, al* will enable or disable interrupt requests according to the contents of AL. See program 9.5.
- The port *20h* is used to access the Interrupt Service Register/Interrupt Request Register.

Keyboard Interface: An Intel 8048 micro-controller inside the keyboard unit scans the keyboard for keyboard activity. A bidirectional data line (KBD DATA) carries the serial keystroke data from the keyboard. Another bidirectional line (KBD CLK) carries either the baud rate clock from the keyboard unit or a *clock disable* control signal from the system unit. The I8048 maintains a 16-keystroke buffer, and transmit each keystroke serially to the system unit at 10,000 baud over the KBD DATA line

together with the baud rate clock on the KBD CLK line. The 8 data bits are transmitted LSB first; bits 0-6 are the scan code which uniquely identifies the key by its position on the keyboard, bit 7 (MSB) is 0 for key press and 1 for key release. Holding a key down for more than half a second invokes the typematic action: key press scan codes are sent repeatedly at the rate of 10 per second without intervening key release scan codes, until the key is released.

An interrupt from the keyboard generates a series of actions triggering an *int 9* interrupt. The following steps summarize these actions:

Steps:

1. The I8259 sends an INTR signal to the CPU.
2. The CPU acknowledges that signal and the I8259 sends the interrupt number.
3. If external interrupts are not disabled, then the CPU services this interrupt:
 - PUSHF
 - CLI
 - CS:IP \rightarrow stack
 - locates entry for the *int 9* in the interrupt vector table and updates the pair *cs:ip* with this entry.
4. A BIOS routine for *int 9* takes control:
 - enables interrupts
 - It reads the scan code from port 60h,
 - sends the *clear-and-re-enable* handshaking signal to the keyboard unit over the KBD DATA line by setting bit 7 of port 61h to 1 and back to 0.
 - processes the scan code,
 - sends an *end-of-interrupt* signal to the I8259 through port 20).
 - executes an *iret*:
 - restores return address (old *cs:ip*)
 - restores flags register (*popf*)

8.6 The I8253 Programmable Interval Timer

One purpose of the I8253 is to generate an interrupt tic approximately every 54.9254 msecs. The signal generated is wired to the *IRQ₀* line of the I8259 which corresponds

to interrupt type 08h in the interrupt vector table. This is illustrated in Fig. 8.3. The interrupt handler maintains the time-of-day clock and performs other internal timing functions.

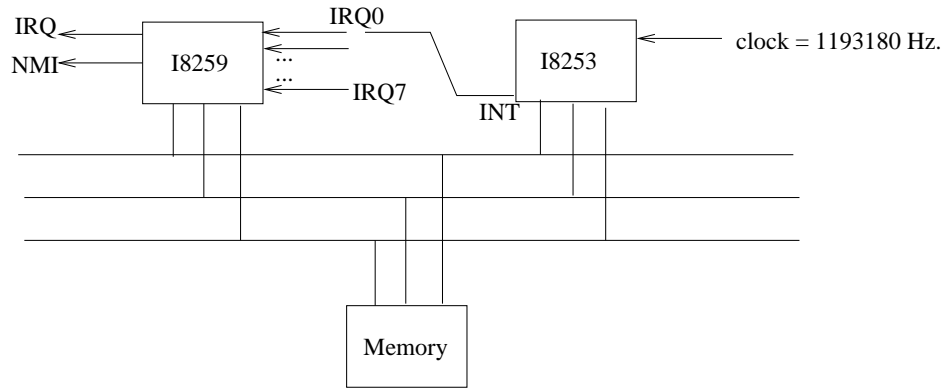


Figure 8.3: The I8253 interface with the *I8259*

As shown in Fig. 8.4, the I8253 contains three independent counters. Each counter is connected to a 1.19 Mhz *clock* input, a *gate* input for enabling/triggering the count, and a counter output (OUT) that provides a periodic output that can be programmed to generate it in a specific shape. An 8-bit internal data bus is connected to the system bus through which the CPU performs read and write operations into the I8253 internal registers. Each counter has an specific purpose and must be programmed separately. A *control* register which is common to all counters, controls the operation and the writing/reading of the counter registers. There are six possible operation modes that determine the shape of the output signal:

- Mode 0 interrupt on terminal count
- Mode 1 hardware retriggerable one-shot
- Mode 2 rate generator
- Mode 3 square wave generator
- Mode 4 software triggered strobe
- Mode 5 hardware triggered strobe

A control word sent to the control register via its port number, selects the counter, the signal shape, a read or write operation, and initializes the selected counter. The control register is loaded by writing a control byte to its assigned I/O port 43h. In addition to the port assigned to the control register, a port number is assigned to each counter. Counters registers 0, 1, and 2 can be accessed through I/O ports 40h, 41h, and 42h, respectively. To program a tic period, a *count* number that divides the input clock is loaded into the counter register. Since the counter

register is 16 bits, the number must be a 16-bit value from 1 to FFFFh and must be transferred in two separate bytes through the data bus. The selected counter is decremented and whenever a zero count is reached an output signal is generated. Depending on the mode selected, the *gate* input may act as an enable input, or as a trigger to start the down-count; similarly, the counter may automatically reload the count and repeat, or require a reload/re-trigger (one-shot operation).

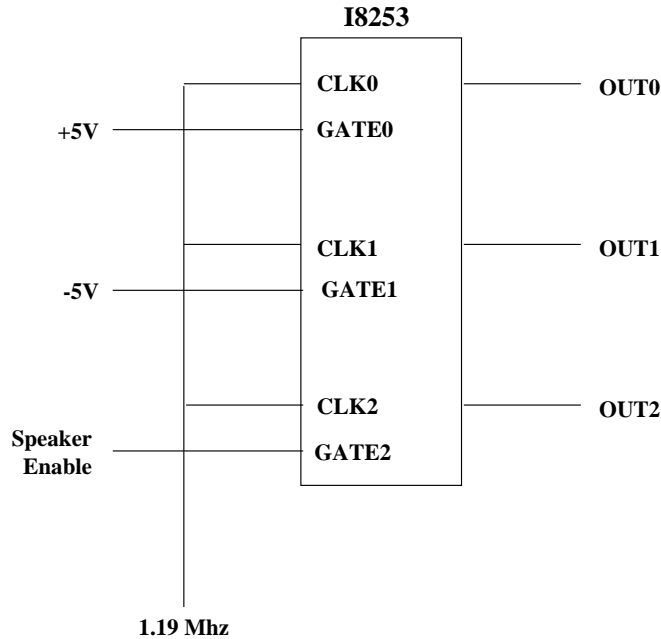


Figure 8.4: Chip connections of the I8253

Counter Register 0: The input *GATE0* maintains *CLK0* enabled to activate a clock input at the rate of 1.193 Mhz. The output *OUT0* is directly connected to *IRQ₀* and activated at a minimum rate of 18.2 Hz when the counter is loaded with the maximum value of 65536. Any tic period *T* can be programmed by loading the counter with the appropriate constant. If *count* denotes such a constant then:

$$Count = clock \times T$$

Example 1: to generate an interrupt every 54.9254 ms, i.e.. with $T = 54.9254ms$, need $Count = 1193180 \times 0.0549254 = 65536 = 2^{16}$

Loading a zero into the counter causes an interrupt tic with this period.

The BIOS call *int 1Ah* can be used to read/set the tic count as follows:

- to read the count:

- input: `ah = 00h`
- output: `cx:dx` (timer count)
- to set count – input: `ah = 01h`, `cx:dx = count`

To measure execution times via *int 1Ah* the following scheme can be used:

- Before execution starts read and save the timer count (K_1)
- Before exit read and save the timer count (K_2)
- calculate the number of tic counts during execution = $K_2 - K_1$
- Execution time = $(K_2 - K_1) \times 54.9254$

Example 2: If $T = 1ms$, then $count = 1193180 \times 0.001 = 1193$

Example 3: To obtain the tic interrupt frequency f given that the count = 5966, then $T = \frac{count}{clock} = \frac{5966}{1193180} = .005 \text{ secs.}$ and $f = \frac{1}{T} = \frac{1}{0.005} = 200 \text{ tics/sec.}$

The following code segment illustrates the programming of the counter register 0. A constant 36 loaded into the control register selects this counter with mode 3 (square wave generation):

```

mov dx, 43h      ; control register port
mov ax, 036h     ; operation mode to generate a square signal
out dx, al
mov dx, 040h     ; counter 0 port
mov ax, PIT_count
out dx, al       ; transfer low byte
xchg ah, al
out dx, al       ; transfer high byte

```

Besides keeping the time-of-day clock, additional functionality of the *IRQ₀* (type 08h) handler includes 1) turning off the motor of the floppy drive if needed, and 2) allowing the execution of user-defined applications; BIOS checks the interrupt vector table via *INT 1Ch* to fetch a pointer to a user-defined application.

Counter Register 1: As shown in Fig. 8.4 CLK1 is also wired directly to the 1.193 Mhz clock. The output OUT1 generates a periodic signal that is used to refresh DRAM memory through the 8237 DMA chip. The frequency of OUT1 is 66,278 Hz; therefore the clock signal must be divided by 18 to generate a tic every 15 μ secs. The programming of counter 1 will require the following lines of code:

```

    mov al, 54h          ; control word
    out 43h, al          ; loaded in to the control register
    mov al, 18           ; load constant
    mov 41h, al          ; into counter 1

```

The word loaded into the control register selects mode 2 that will keep the pulse high for 18 clock pulses.

Counter Register 2: The output OUT2 of counter 2 is used to generate a beep sound but it can be programmed to generate sounds at different frequencies. As shown in Fig. 8.4 the GATE2 input is enabled via the *speaker enable* signal. The beep has a frequency of 896 Hz and it is generated by loading a value of 1331 into the counter register. The following code segment selects mode 2:

```

    mov al, 0B6h         ; control word
    out 43h, al
    mov al, 33h          ; loading constant
    out 42h, al          ; requires two bytes
    mov al, 05h          ; high byte
    out 42h, al

```

8.7 Interrupts: IA32 processors

Real Mode

When the *cpu* acknowledges an interrupt request, the I8259 places a one-byte interrupt type code on the data bus. For real-mode applications in the I32 family of processors the Interrupt Descriptor Table (IDT) substitutes the IVT structure used in the 8086/88/286 processors. The IDT still consists of 256 4-byte entries as in the IVT but the physical address where the IDT is located starts at 00000000h and for 256 entries it ends at 000003FFh. Access to the IDT table is facilitated using the Interrupt Descriptor Table Register which as described in Fig. 8.5. is a 48-bit register consisting of a base address field and a length field. Initially the base address of the IDT is preloaded with the 32-bit starting address at the zero location and the length is pre-loaded with the 16-bit value 03FF. The actions taken by the processor in response to an interrupt event are summarized in the following steps:

1. When either an interrupt (software or hardware) interrupt or a software exception occurs, the *cpu* pushes the current contents of *eflags* register onto the stack, and clears the interrupt flag and disables the recognition of external interrupts. It clears also the TF flag in the *eflags* register to disable single-step mode interrupts.

2. Pushes the current contents of *cs* and *eip*
3. The interrupt type *n* is multiplied by 4 and added to the base address in the IDTR to select the correct entry in the IDT. The contents of this entry are transferred to *cs* and *eip*
4. Resumes execution using the contents of *cs* : *eip* which now point to the first instruction of the interrupt/exception handler.

Note that user can change the base address and the length using the instruction *lidt*; however, real mode applications assume the default base address and length of the IDT.

Protected mode

In protected mode the *cpu* also uses the interrupt type to index the IDT as shown in Fig. 8.5. The IDT in this case consists of up to 256 eight-byte entries. Each entry contains an *interrupt gate descriptor* with information that includes the segment selector and the offset needed to locate the interrupt handler. The IDTR provides the base address of the IDT and the interrupt type multiplied by eight provides the location of the interrupt vector.

The *cpu* service to an interrupt/exception request is summarized in the following steps:

1. Push the *eflags* register:

$$\begin{aligned} esp &\leftarrow esp - 4 \\ [esp] &\leftarrow eflags \end{aligned}$$

As in the case of the real-mode access, the top of the stack is first updated to point to a new location in the stack where the *eflags* register is then stored.

2. Disable interrupts by clearing the interrupt flag in the *eflags* register (*cli*) From this point on, the *cpu* will not permit another interrupt.
3. Push the return address:

$$\begin{aligned} esp &\leftarrow esp - 4 \\ [esp] &\leftarrow cs \\ esp &\leftarrow esp - 4 \\ [esp] &\leftarrow eip \end{aligned}$$

These transfer operations describe the transfer of the segment and the offset parts of the pointer to the next instruction in the interrupted program.

4. As described before, the first 16 bits of the IDTR contain the IDT length (or limit) and the second 32 bits contain the IDT base address. Combining the IDT base address (A) from the IDTR and the interrupt type n as shown in Fig. 8.5, an index $p = A + 8n$ where $0 \leq n \leq k$, is calculated to access the *interrupt gate* for *int* n .

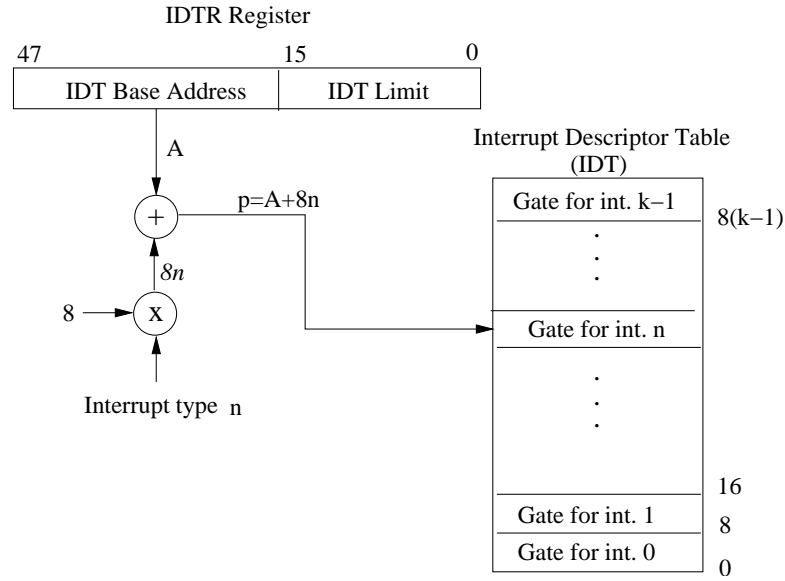


Figure 8.5: Access to the Interrupt Descriptor Table

5. The interrupt gate is a 64-bit entry in the IDT with a format as shown in Fig. 8.6. Among other information bits, it contains the segment selector and an offset that are loaded into $cs : eip$:

$$cs : eip \leftarrow idt_{64}[p]$$

The segment selector is transferred to cs and the offset is transferred to eip .

6. The actual segment of the interrupt handler is now accessed from the Global Descriptor Table (GDT) using the 13-bit segment selector in the cs register:

$$cs \leftarrow gdt_{64}[cs]$$

The actual generation of the physical address where the interrupt handler is located is shown in Fig. 8.7, which follows the protected mode address generation mechanism described in Chapter 3.

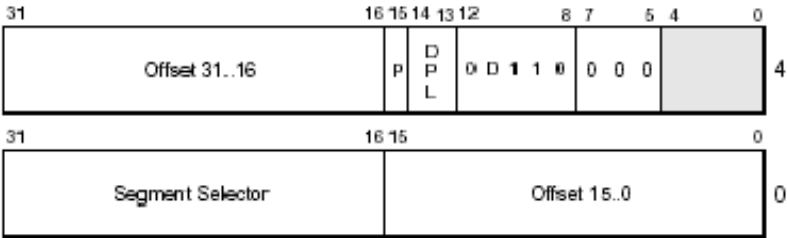


Figure 8.6: Interrupt gate description

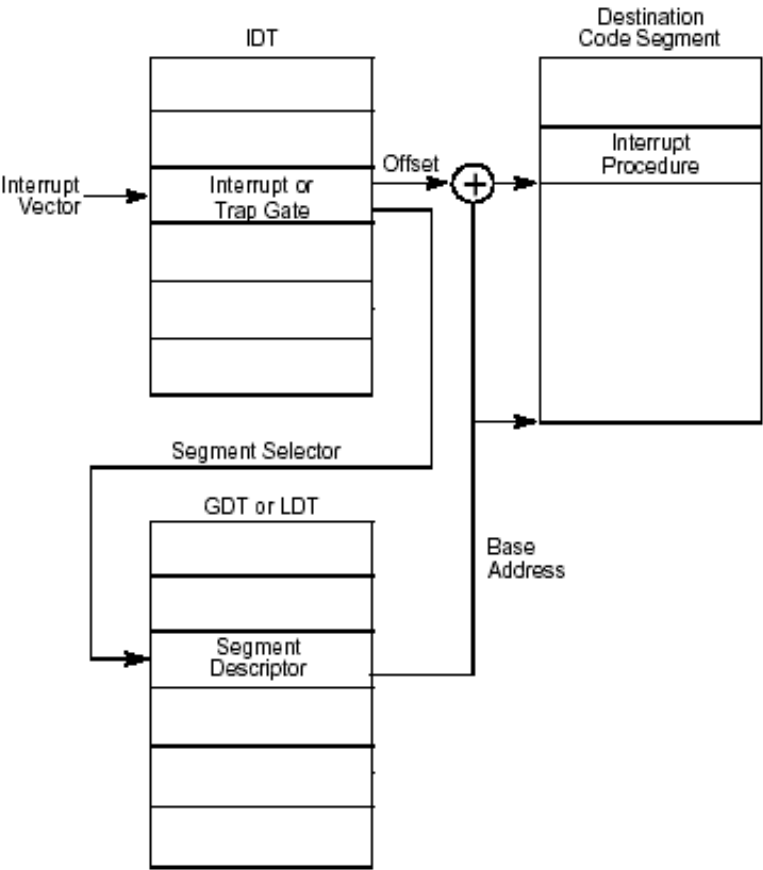


Figure 8.7: Interrupt Procedure Call