LAB 3: Exploring the Input/Output (I/O) Subsystem

1. Objectives:

In this experiment we will learn how to:

1. input from and output to the input/output (I/O) peripherals of the IBM personal computer (PC),
2. write a program to control the speaker of the IBM PC, and
3. program the 8253 programmable interval timer (PIT) for speaker tone control.

2. Introduction:

In the IBM personal computer (PC), peripherals such as: the 8255A programmable peripheral interface (PPI), the 8253 programmable interval timer (PIT), the 8237A programmable direct memory access controller (PDMAC), and the 8259A programmable interrupt controller (PIC), are located in the input/output (I/O) address space of the microprocessor unit (MPU). For this reason, they are accessed using the IN and OUT instructions. In this experiment, we will explore the processes of reading from and writing to (using IN and OUT instructions, respectively) the registers of two important peripheral devices, namely, the PPI and the PIT devices, in order to control the tone and the ON/OFF switching of the PC speaker.

The Block diagram in Fig. 1 shows the circuitry of the IBM (and compatible computers) PC speaker. It is important to note here that in order to correctly operate this speaker, the corresponding outputs of both the PPI and the PIT devices need to be appropriately controlled. As is shown in Fig. 1, these outputs are:

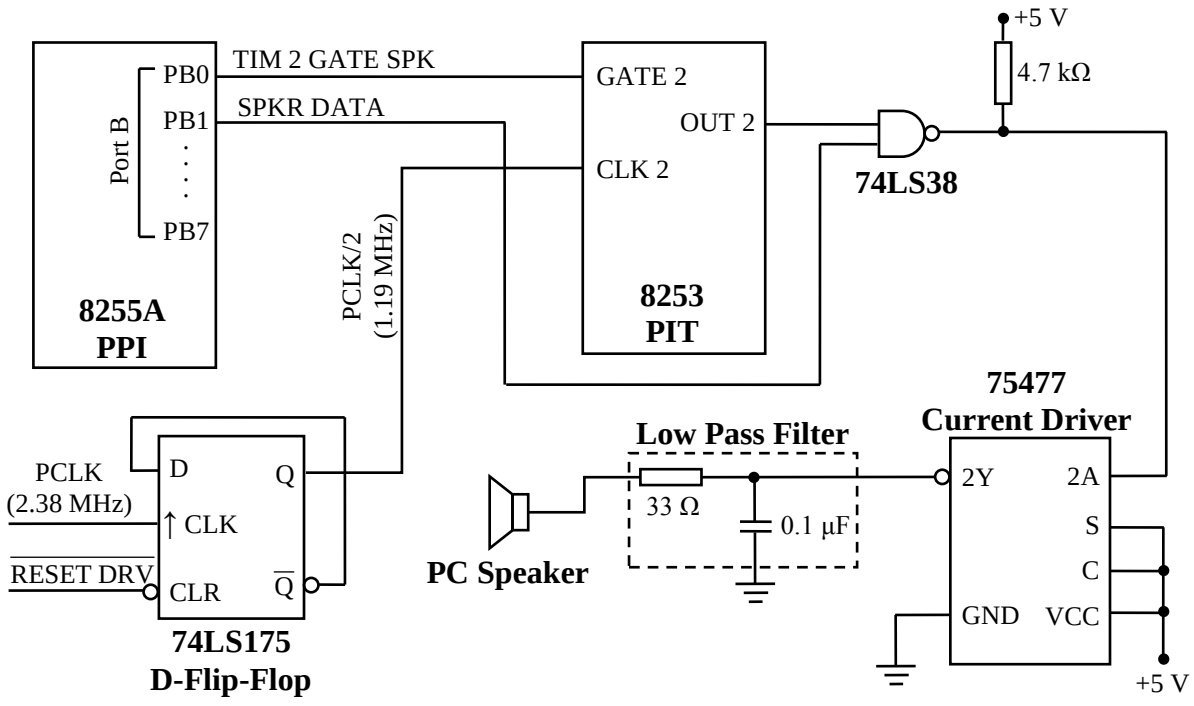


Fig. 1: A block diagram of the PC speaker circuitry in the IBM PC. Note that the I/O ports A, B, and C and the control register of the 8255A PPI device are mapped here onto the I/O addresses $60_H - 63_H$, respectively, while counters 0, 1, and 2 and the control register of the 8253 PIT device can be accessed through I/O addresses $40_H - 43_H$, respectively.

1. **OUT 2:** the output of counter 2 of the PIT device, representing the data signal that is (indirectly) applied to the PC speaker to produce the corresponding tone. Note here that in order to generate a tone with a specific frequency, counter 2 must be first set to work in mode 3, the square wave generator mode, through outputting the appropriate control word to the PIT control register located at the I/O address 43_H (see Fig. 2 for more details about the format of this control word). This control word should also prepare counter 2 to accept the 16-bit or 8-bit count (frequency divisor) that is to be loaded (output) to this counter (located at the byte-wide I/O address 42_H) next. Note that the base frequency of the square wave at the output of counter 2 (OUT 2) is simply the frequency of the clock input (CLK 2) divided by this count (hence the name frequency divisor).
2. **TIM 2 GATE SPK:** produced by bit 0 of port B (PB0) (located at the I/O address 61_H) of the PPI device, which is connected to (GATE 2) input of counter 2 of the PIT device. Switching (and maintaining) the logic level of bit (PB0) into logic 1 is necessary to initiate and enable the work of counter 2, and hence, is necessary to start and continue producing the corresponding signal at the output of this counter (OUT 2).
3. **SPKR DATA:** produced by bit 1 of port B (PB1) of the same PPI device. This bit is applied to the second input of the NAND gate (shown in Fig. 1), and works as an enable/disable switch to the data signal that passes to the PC speaker through this gate. Switching the logic level of bit (PB1) to logic 1 would enable the complement of the data signal (produced by counter 2) at the other input of the NAND gate to be passed to the PC speaker to produce the corresponding tone (note that the 75477 current driver would again complement the signal, received from the NAND gate, at its output, which should restore the original signal produced by counter 2). On the other hand, switching the logic level of this same bit to logic 0 would pull the output of the NAND gate to logic 1 regardless of the signal at its other input, hence, allowing no signal (the complement of the NAND output, again) to be passed to the PC speaker, and no audible sound would be produced.

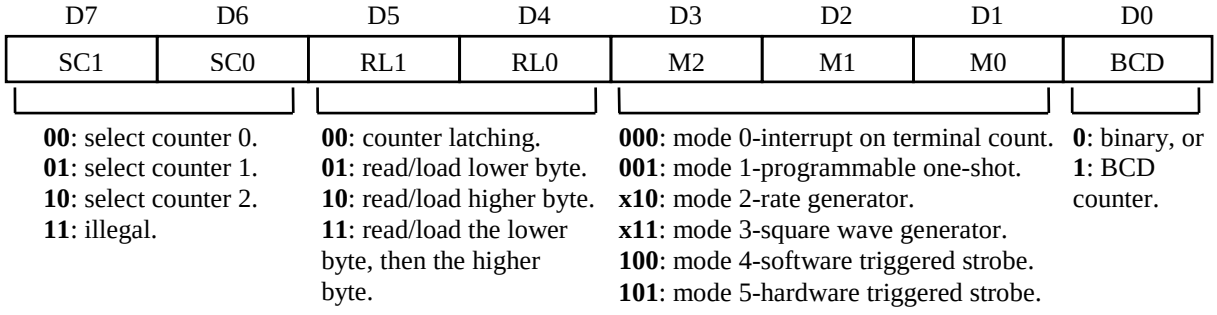


Fig. 2: PIT control word format.

3. Writing a Speaker Control Program:

A flowchart for an example program that produces a single short tone at the PC speaker is shown in Fig. 3. However, before proceeding to code this program, let us first calculate the count (the divisor) that needs to be loaded into the PIT counter 2 to set the tone frequency to *f* Hz. This is done using the expression:

$$\text{The divisor } (N) = \frac{\text{frequency of the clock signal at CLK 2 input to counter 2}}{\text{desired frquency of the tone}} = \frac{1.19\ \text{MHz}}{f}$$

Thus to generate a tone with a frequency of 1.5 kHz, for example, we must load the counter with the divisor *N*, where:

$$N = \frac{1.19\,\text{MHz}}{1.5\,\text{kHz}} = \frac{1.19*10^6}{1.5*10^3} = 793 = 319_{\text{H}}$$

In order to set counter 2 to work in the square wave generator mode, and program it to accept the 16-bit (2-byte-wide) divisor, we first need to output the appropriate control word to the control register of the PIT device. As can be seen from Fig. 2 this control word is ($B6_{\text{H}}$) here:

```
MOV   AL , B6
OUT   43 , AL
```

Now, the divisor is loaded to counter 2 starting by the lower byte followed by the higher byte:

```
MOV   AX , 319
OUT   42 , AL
MOV   AL , AH
OUT   42 , AL
```

Next, we enable both counter 2 and the NAND gate by setting bits 0 and 1 of port B of the PPI device to logic 1. Note here that since we should not change the state of the other bits of this port, we will need to read the content of this port first, save this content, set the first two bits of it to logic 1, and then output back the new content of this port, as follows:

```
IN    AL , 61
MOV   AH , AL
OR    AL , 3
OUT   61 , AL
```

Here, the first two bits of the content of port 61_{H} are set to logic 1 by *ORing* it with a *masking pattern* that has only its first two bits set to logic 1. Note also that the current content of port 61_{H} has been saved in the AH register, and can be restored back from this register later, as long as this register is not updated.

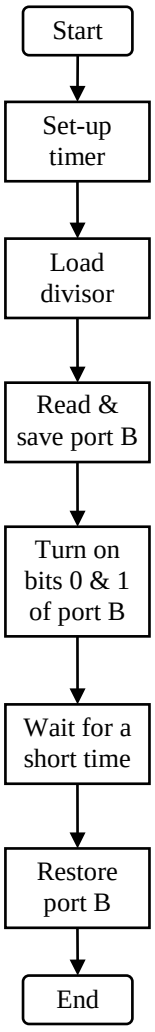


Fig. 3: Flowchart for a short tone generation program.

Now the speaker is turned on and the 1.5 kHz tone is being generated. However, the tone here is required to be produced for only a short time. For this reason, a software delay can be inserted at this point in the program. This delay is actually necessary to make the time between turning the PC speaker on and off enough for us to recognize the produced audible tone. A software delay can be coded using loops with a relatively large number of iterations through a set of dummy instructions, such as the no operation NOP instruction, as follows:

```
        MOV   DX , iter1
delay1: MOV   CX , iter2
delay2: NOP
        NOP
        LOOP  delay2
        DEC   DX
        JNZ   delay1
```

Note here that the amount of time delay achieved by this program segment depends on the total number of iterations (*iter1 * iter2*, for this example) and on the instructions used within the dummy loop (two NOP instructions). However, it is not possible to accurately determine

the exact time consumed by a given software delay in advance, due to the instruction queues used within the fetching and execution cycles, which are aimed at enabling pipelined processing. It is important to mention here, too, that the instructions within dummy loops should not destroy any data that might be needed later in the program.

Now after the delay is complete, we must turn the PC speaker back off. This can be done by outputting the byte saved within register AH back to port B of the PPI device, in order to restore its original state before the execution of this program, as follows:

```
MOV    AL , AH
OUT    61 , AL
```

Note that the PC speaker can also be turned off by explicitly resetting the first two bits of port B to logic 0 in a similar fashion to that used for setting them, that is by *ANDing* the port content with another masking pattern that has only its first two bits reset to logic 0.

4. Assignments:

1. Write a program that generates a variable number of discrete tones at the PC speaker, with the number of tones being defined by the content of some memory location.
2. Write a program that can generate a variable number of discrete tones with different frequencies at the PC speaker. The total number of tones and the frequency of each distinct tone should be defined by the content of some memory area, whose starting address may be passed to the program (before execution) through some available register. **Hint:** Before executing the program above, make sure that the content of the memory area used includes suitable values for the frequency divisors those are to be loaded into counter 2, since that, after all, only frequencies within the 20 Hz – 20 kHz range are perceptible.

5. References:

A. Singh, and W. A. Triebel, "*The 8088 microprocessor programming, interfacing, software, hardware, and applications*", Prentice-Hall International, Inc.

LAB. 4: Exploring the Interrupt Subsystem

1. Objectives:

In this experiment we will learn how to:

1. determine the starting address of an interrupt service routine,
2. explore the code of an interrupt service routine,
3. execute software interrupt service routines to determine the equipment attached to the PC and the amount of conventional memory,
4. execute the software interrupt service routine for system boot, and
5. use the interrupt 21_H function calls to read and set date and time.

2. Introduction:

Interrupts provide a mechanism for changing program environment, with the transfer of program control being initiated by either the occurrence of an event internal to the MPU, such as the division error, or an event in its external hardware, such as the requests made by the I/O devices for processing. Interrupts can also be made by software interrupt calls (using the instruction INT *n*, where *n* is the interrupt type number). Software interrupts can provide a wide variety of system services to the running programs, such as providing status information about the system hardware and even reading/writing from/to mass storage devices.

The MPU tests for an active interrupt during the last cycle of execution of each instruction. In response to the occurrence of an active interrupt request, the MPU suspends executing the currently running program, saves (on the top of the *stack*) the address of the next instruction of this program that is to be executed when control is returned to, determines the type number of the interrupt request made, and finally transfers the control to some special program segment called the *interrupt service routine* that is responsible for dealing with that specific request. After this service routine is run to completion, program control is returned to the point where the execution of the main program was interrupted (using the address saved by the MPU on the top of the stack earlier).

The Intel 8088 processor, whose interrupt subsystem will be explored here, is generally capable of implementing any combination of up to 256 interrupts, divided into four groups:

1. *internal interrupts*,
2. *the nonmaskable interrupt*,
3. *software interrupts*, and
4. *external hardware interrupts*.

These interrupts are serviced on a *priority basis*. Here, priority is achieved in two ways. *First*, the interrupt processing sequence implemented in the 8088 MPU tests for the occurrence of the various groups of interrupts based on the same priority hierarchy in the list above, with the internal interrupts being the *highest priority group*, while the external hardware interrupts being the *lowest priority group*. *Second*, the various interrupts within the same group are given different *priority levels* by assigning to each an *interrupt type number*. Here, type 0 identifies the *highest priority level* and type 255 identifies the *lowest priority level*. For example, if a type 50 external hardware interrupt is in progress, it can be interrupted by all internal interrupts, the nonmaskable interrupt, any software interrupt, in addition to external hardware interrupts with type numbers less than 50 (provided that the external hardware interrupt input is enabled). That is, the external hardware interrupts with type numbers equal to or greater than 50 would be *masked out* (kept waiting for the current one to

be completely processed). The importance of priority lies in the fact that, if an interrupt service routine has been initiated to perform a function at a specific priority level, only requests of higher priorities can interrupt the active service routine, while lower priority requests can wait until this routine is completed before they are *acknowledged*. For this reason, tasks those must not be interrupted frequently should be assigned to higher priority levels than those can be more frequently interrupted.

We have mentioned earlier that in order to transfer the control to the appropriate service routine in response for a given interrupt request, the processor should first determine the type number of that request. Interrupt type number can be determined by the MPU as follows:

1. Internal interrupts are *automatically* initiated by the MPU itself, and occur as a consequence to producing an invalid result after the execution of some instruction, as in division error, which initiates interrupt type 0, or in response to setting some MPU internal *flag*, such as the *trap flag* (TF), which would cause interrupt type 1 (single step execution) to be automatically initiated after the execution of each instruction.
2. The nonmaskable interrupt is initiated by an external hardware request that is applied to the nonmaskable interrupt request input (NMI) of the MPU. A request on this input would initiate a type 2 interrupt.
3. Software interrupts are initiated by the software interrupt request instruction INT *n*, where the interrupt type number (*n*) is a part of the instruction.
4. External hardware interrupts are initiated by external hardware requests those are applied to the interrupt request input (INTR) of the MPU. Since that the requests applied to this input can be masked out by resetting the *interrupt enable flag* (IF), the MPU would only respond to external interrupt requests, when this flag is set. At the receipt of an external hardware interrupt request, the MPU acknowledges this request by sending the interrupt acknowledge signal ($\overline{\text{INTA}}$), the hardware interrupt interface should then send the interrupt type number back to the MPU on the data bus. In the IBM PC, hardware interrupt interface is implemented using the 8259A programmable interrupt controller (PIC) device. This device is programmed during system startup to map its 8 interrupt request inputs ($IR_0 – IR_7$) onto interrupt types 8 through 15.

The interrupt type number (*n*) is used to determine the specific service routine, where the program control is to be transferred in response to the received interrupt request. This is performed with the aid of an address table called the *interrupt vector table*, which includes the starting addresses of the service routines of all the 256 interrupt types supported by the system. This table is located at the lower 1K bytes of the memory address space. It starts at address 00000_H and ends at address $003FF_H$. Each of the 256 starting addresses (*interrupt vectors*) in the table consists of two words (four bytes). The higher addressed word of each vector identifies the *memory segment*, where the corresponding service routine resides, and is to be loaded into the *code segment* (CS) register of the MPU, while the lower addressed word of each vector, is the *offset* of the first instruction of the service routine from the beginning of the code segment defined by the base address to be loaded into the CS register. This offset is to be loaded into the *instruction pointer* (IP) register. Looking at the interrupt vector table in Fig. 1, we may notice that the distinct interrupt vectors there are stored at double-word (four bytes) boundary, that is, the interrupt vectors starting at addresses 00000_H, 00004_H, 00008_H … $003FC_H$ in the table, correspond to interrupt types 0, 1, 2 … 255, respectively. So, in order to access the interrupt vector corresponding to some interrupt type number (*n*), the MPU multiplies this number (*n*) by 4 and uses the result, that is, it uses (*n* * 4), as the starting address of the double-word interrupt vector to be fetched from the vectors table.

After fetching the starting address of the destination service routine from the interrupt vector table, the MPU then saves the point in the main program where execution has been suspended by *pushing* the old values of the flags register and CS and IP registers on the top of

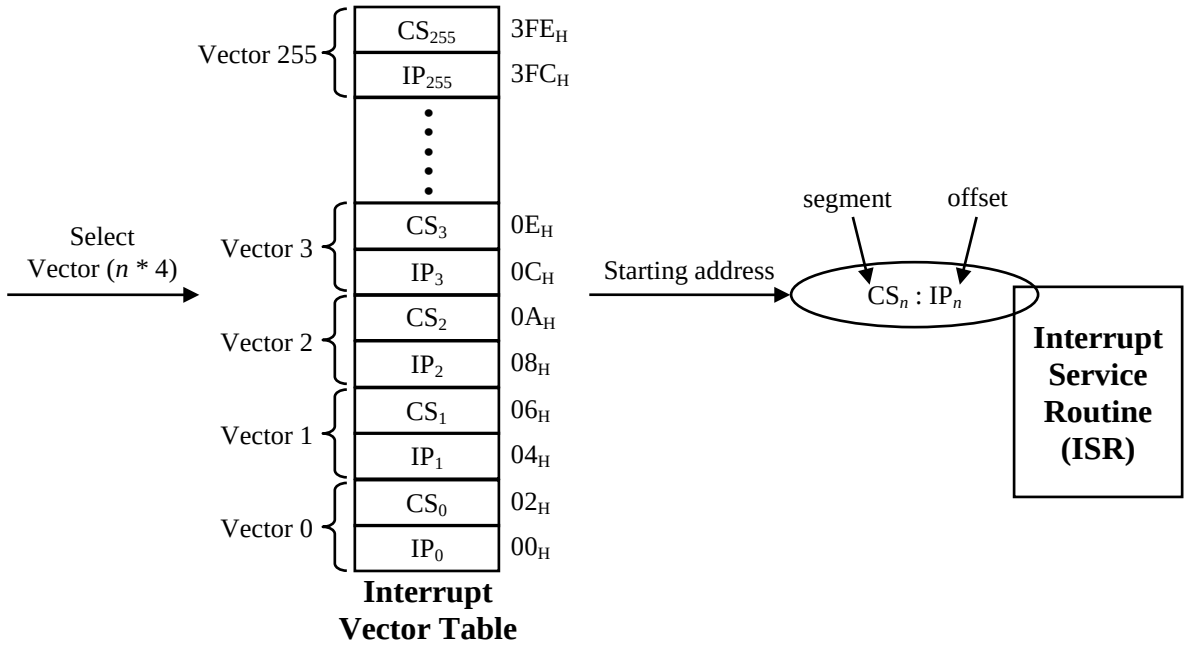


Fig. 1: Indexing the interrupt vector table using the interrupt type number (*n*) to find the starting address for the corresponding interrupt service routine.

the stack. The control is next transferred to the service routine by loading the CS and IP registers with the fetched starting address. At the end of the service routine, the interrupt return (IRET) instruction restores the old contents of the IP and CS registers and the flags register from the top of the stack, in order to resume the execution of the interrupted main program from the instruction where execution has been suspended. Note here that in order to disable the single step execution mode (if enabled) and to mask out any *pending* hardware interrupts while executing a service routine, the MPU resets both the trap flag (TF) and the interrupt enable flag (IF) before it transfers the program control to the destination service routine. However, in some cases it may be necessary to permit other higher-priority external hardware interrupts to interrupt the active service routine. In these cases, the interrupt enable flag (IF) could be set at the beginning of the active service routine using the set interrupt enable flag (STI) instruction, to re-enable the interrupt request input (INTR), otherwise, it would be automatically enabled by the interrupt return (IRET) instruction at the end of that service routine.

3. Exploring the Code of an Interrupt Service Routine:

In order to explore the code of an interrupt service routine, one of two things can be done:

1. We may use the interrupt type number (*n*) to fetch the interrupt vector corresponding to this interrupt from the vector table through *dumping* the double-word saved at the starting address (*n* * 4) in the memory. We may then list (*unassemble*) the code of the service routine that is saved in the memory area starting with the specific address defined by the fetched interrupt vector. Note here that the double-word address fetched from the memory is arranged (starting from the lower addressed byte) as the lower byte then the higher byte of the offset, followed by the lower byte and then the higher byte of the segment part of the address.
2. We can simply execute the instruction INT *n* using the single step execution mode to walk through (*trace*) the individual instructions of the service routine. Note that it could be easier to trace the instructions of some service routines up to their last instruction (interrupt return IRET) using this method, since that these routines include many branching instructions.

4. Determining the PC Hardware Equipment and Conventional RAM Size:

The service routine for interrupt 11_H determines what equipment is attached to the PC. The execution of this service routine returns a word in register AX, whose bits indicate the following:

Table 1: The format of the PC configuration word.

Bits	***Significance***
0	= 1 if any floppy disk drive is installed.
1	= 1 if the math coprocessor is installed.
3 , 2	System board read/write (R/W) memory size (PC only): (00 = 16K, 01 = 32K, 10 = 48K, 11 = 64K).
5 , 4	Video mode: (00 = reserved, 01 = 40x25 using color card, 10 = 80x25 using color card, 11 = 80x25 using monochrome card).
7 , 6	Number of floppy disk drives installed (only if bit 0 = 1): (00 = 1, 01 = 2, 10 = 3, 11 = 4).
8	Reserved.
11 , 10 , 9	Number of RS-232 ports installed.
12	= 1 if game adapter is installed.
13	= 1 if internal modem is installed (PC and PC/XT only).
15 , 14	Numberof printers installed.

Note that the service routine of interrupt 11_H simply returns the bit pattern of the PC configuration word stored at memory address $0000:0410_H$.

On the other hand, the service routine for interrupt 12_H returns the number of K bytes of conventional R/W memory attached to the system in the AX register. Note that the number returned does not reflect the size of the installed extended memory (above the 1 M bytes boundary) on the PC/AT systems. Note here also that the service routine of interrupt 12_H only returns the conventional memory size word stored at memory address $0000:0413_H$.

5. System Rebooting:

The service routine for interrupt 19_H *boots* the system from a bootable disk. The service routine of this interrupt reads the *boot sector* (sector 1, track 0) of the aforementioned disk into the memory area starting at address $0000:7C00_H$ and then transfers the system control to the same address. The program read from the boot sector then takes the responsibility to load the necessary operating system files from the bootable disk and to transfer the system control to them.

6. The INT 21_H Function Calls:

Interrupt 21_H is provided to invoke DOS operations for a wide variety of functions, such as character I/O, file management, date and time setting and reading. To call any function of interrupt 21_H, the register AH should be first loaded with the specific function number before

executing the software interrupt call instruction INT 21_H. The called function could also require some parameters to be passed to it through the remaining registers and/or could return some other parameter values through these registers. If the function cannot be performed, interrupt 21_H returns the value FF_H in the register AH.

In this experiment, we will learn how to use the data and time setting/reading functions of interrupt 21_H. Table 2 below shows the specific number of each of the aforementioned functions (that should be loaded into register AH prior to executing the INT 21_H instruction) and the registers used by them to return/receive parameters.

Table 2: The time and date setting/getting functions.

Function	***Number***	***Registers used***
Get date	$2A_H$	The function returns date on registers: (AL = day of week, CX = year, DH = month, DL = day).
Set date	$2B_H$	The date should be passed to the function through registers: (CX = year, DH = month, DL = day).
Get time	$2C_H$	The function returns time on registers: (CH = hours, CL = minutes, DH = seconds, DL = hundredths of a second).
Set time	$2D_H$	The time should be passed to the function through registers: (CH = hours, CL = minutes, DH = seconds, DL = hundredths of a second).

7. Assignments:

1. Using the methods described in section 3 explore the code of the nonmaskable interrupt (interrupt type 2).
2. By executing software interrupts 11_H and 12_H (executing instructions INT 11_H and INT 12_H, respectively), determine the equipment and the size of the conventional memory of your PC. Also, check the contents of the configuration and memory size words on the memory addresses mentioned in section 4. Try to change the contents of these words and see what effect these changes could make on the values returned by interrupts 11_H and 12_H.
3. Reboot the PC from a bootable disk by executing interrupt 19_H.
4. Use the DOS commands Time and Date to check the current time and date on your PC, then use the get time and get date functions as is describe in section 6 to find the current time and date, compare the two results. You can also try to set time and date using the DOS or the corresponding interrupt 21_H function calls and see the effect each method makes on the other.

8. References:

A. Singh, and W. A. Triebel, "*The 8088 microprocessor programming, interfacing, software, hardware, and applications*", Prentice-Hall International, Inc.

LAB. 5:System Timer Interrupt

1. Objectives:

In this experiment we will:

1. explore the service routine of the system timer interrupt (interrupt 8) of the IBM PC,
2. learn how to disable and enable the system timer interrupt, and
3. use the software timer ticks counter to determine the current time-of-the-day.

2. Introduction:

A system timer is a vital auxiliary device for computer systems. The hardware unit named system timer may or may not be a clock. Actually, in many systems it is nothing more than a *pulse generator* with a *precisely controlled frequency* that is used to generate *interrupts* to the MPU of the system. In response to the incoming interrupt requests, the MPU transfers the system control to the corresponding interrupt service routine that would *update* some *software counter*, which is used to keep track of the number of *timer ticks* (interrupt requests), and hence of the time elapsed. The time interval between timer ticks, here, represents the shortest time interval known to the system.

The choice of the timer precision, that is the timer tick period, has to be a compromise between the timing accuracy required and the load on the MPU. If too short a period (high precision) is chosen, then the MPU will spend a large proportion of its time simply servicing the system timer and may not be able to perform the other tasks efficiently. If too long a period (low precision) is chosen, on the other hand, then the system will not be able to accurately schedule its tasks.

Fig. 1 shows a block diagram of the system timer circuit of the IBM PC. The timer of this system is driven by the pulses generated by an interval timer device.

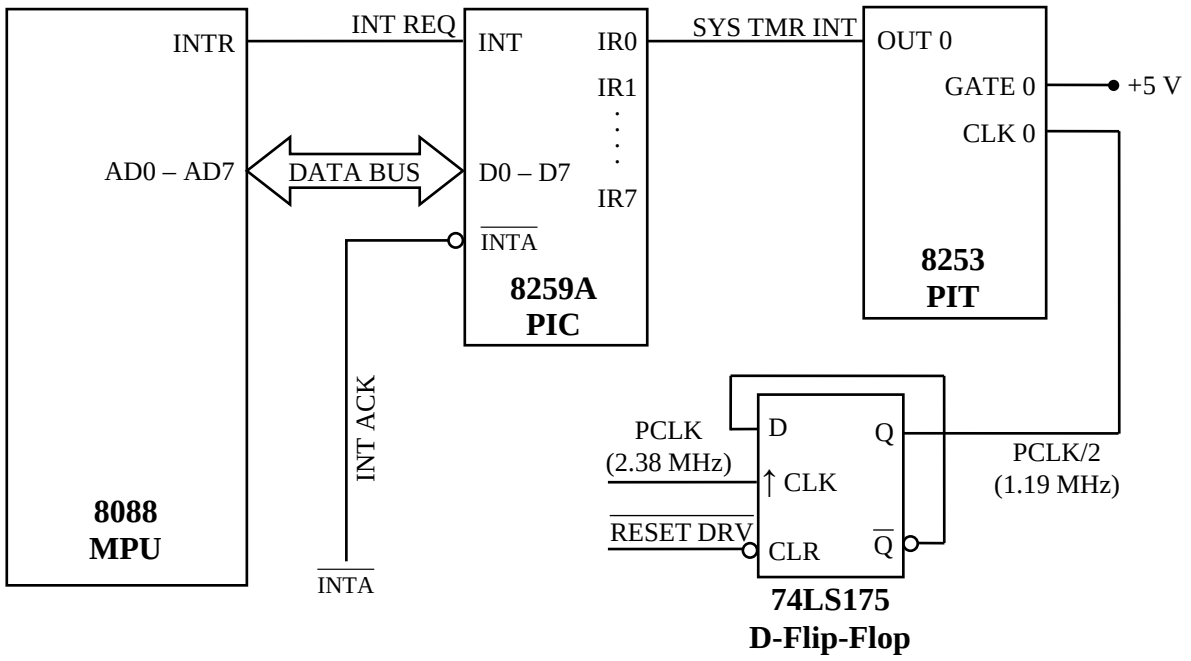


Fig. 1: A block diagram of the timer circuitry in the IBM PC. Note that the counters 0, 1, and 2 and the control register of the 8253 PIT device are mapped here onto the I/O addresses 40_H – 43_H, respectively, while the command registers of the 8259A PIC device can be accessed through I/O addresses 20_H and 21_H.

To understand how the timer circuit operates, let's consider the following signals:

1. **SYS TMR INT:** produced by the output of counter 0 (OUT 0) of the PIT device. This signal represents the interrupt request that is driving the timer in this system. It is applied to the first interrupt request input (IR0) of the PIC device, used to implement the hardware interrupt interface. In order that the timer ticks get generated at the required frequency, counter 0 is appropriately set up during system initialization to divide the frequency of the signal at the (CLK 0) counter input by the value loaded into this counter. In the IBM PC systems, a timer tick is generated every 54.936 msec.
2. **INT REQ:** produced by the interrupt request output (INT) of the PIC device. This signal is directly applied to the interrupt request input (INTR) of the MPU. The PIC device switches (and maintains) the logic level of this signal into logic 1 to initiate an interrupt request to the MPU of the system.
3. **INT ACK:** produced by the 8288 bus controller device in the 8088 based PC systems, which are operated in the maximum mode. This signal is applied to the interrupt acknowledge ($\overline{\text{INTA}}$) input of the PIC device. It is switched to logic 0 to tell the hardware interrupt interface that its interrupt request is acknowledged by the MPU. In response to this signal the PIC device should prepare the interrupt type number (corresponding to the highest priority interrupt request that is pending on the interrupt request inputs of the latter device) and then put (during the second interrupt acknowledge cycle) this number on the data bus in order to be fetched by the MPU.

As it can be seen from Fig. 2, the 8-bit interrupt type number returned by the PIC device, is actually obtained from combining the 3 bits corresponding to the number of the pending interrupt request input along with the highest 5 bits of the PIC initialization command word 2 (ICW2), passed to the PIC device during system initialization. This way, it becomes possible to map the PIC interrupt request inputs onto 8 contiguous interrupt types out of the 256 available types. In the IBM PC system, the interrupt request inputs (IR0 – IR7), of the PIC device, are mapped onto interrupt types 8 – 15, respectively. This means that the system timer interrupt request is mapped onto interrupt type 8.

The service routine of the timer interrupt (interrupt 8) handles a number of the time-related jobs, among these jobs is keeping track of the time-of-the-day. To do so, the service routine increments some software counter, represented by the content of the double-word (4 bytes) located at the memory addresses $0040{:}006C_H$ – $0040{:}006F_H$, by one, each time it is invoked (after each invocation of the system timer interrupt). Note that this way, the system can determine the time that has elapsed since it started counting timer ticks. However, in order to determine the real time-of-the-day, the system initializes this counter, during startup, with the number of timer ticks since midnight. Also note here that since the system timer interrupt is known to occur each 54.936 msec., and that the software counter can be accessed by any running program, then user programs can make use of this counter for determining the time-of-the-day or the time that has elapsed since some given instance.

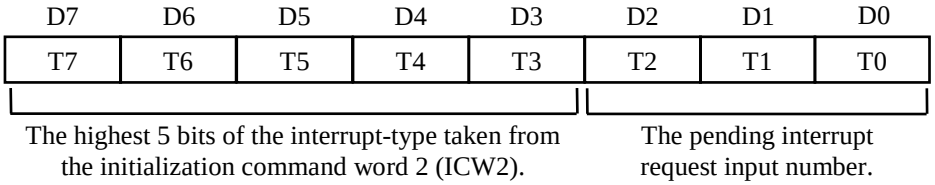


Fig. 2: The format of the 8-bit interrupt type number word put by the PIC device on the data bus during the second interrupt acknowledge cycle.

3. Changing the Frequency of Invocation of the System Timer Interrupt:

We have mentioned earlier that in the IBM PC system, the timer is driven by the interrupt signal generated at the output of counter 0 of the PIT device, and that the frequency of occurrence of this interrupt is determined by the count loaded into that counter. So, in order to change the system timer tick period, we simply need to reload that counter with an appropriate count. This count can be calculated from the expression:

$$\text{count} = \text{tick interval} * \text{ frequency of signal at CLK 0 input to counter 0 of PIT}$$

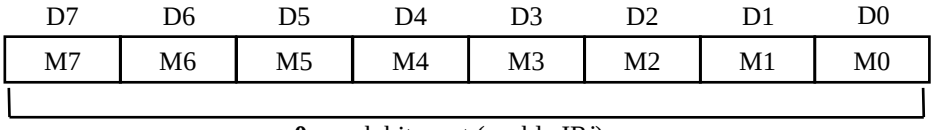
$$N = t_{\text{tick}} * 1.19\,\text{MHz}$$

Note here that when the tick period is changed, the service routine of the corresponding interrupt will not function correctly any more.

To load counter 0 of the PIT with some 16-bit (2-byte) wide count, two OUT operations to the byte-wide I/O port 40_H need to be issued, the first would load counter 0 with the lower byte of the count and the second with the higher byte of it.

4. Disabling and Enabling the System Timer Interrupt:

In order to disable the hardware interrupt interface of the MPU, a program can clear the interrupt enable flag (IF) using the (CLI) instruction. This instruction, however, would disable all hardware interrupts in the system, including the interrupts that control the interfacing with such peripheral devices as the keyboard. To disable some specific hardware interrupt without affecting the remaining ones, on the other hand, a program segment should issue an *operational command word* to the *masking register* of the PIC device. The masking register of the PIC device is an 8-bit register, whose constituting bits represent the *masking state* of the corresponding interrupt request inputs to the device, see Fig. 3. This means that if the hardware timer interrupt (interrupt 8), which is connected to the interrupt request 0 (IR0) of the PIC device, is to be disabled, then, a program segment should read the content of this register located at I/O address 21_H, set the bit 0 of the word to logic 1 (note here that setting a bit in the masking word to 1 would cause the corresponding interrupt request input to be masked or disabled), and re-output the new masking word to the masking register. To re-enable the masked interrupt request input later, the corresponding bit in the masking register of the PIC device should be reset to logic 0. Note here that setting all the masking register bits to logic 1 would disable all the hardware interrupts in the system.



0: mask bit reset (enable IR*i*).
1: mask bit set (disable IR*i*).

Fig. 3: PIC operational command word 1 (OCW1) format.

5. Assignments:

1. Using the single step execution mode, trace the instructions of the service routine of the system timer interrupt (interrupt 8). Determine the instructions, in the service routine, those are responsible for updating the content of the software counter used to keep track of the time-of-the-day.
2. By loading counter 0 of the PIT device with an appropriate count, set the system timer interrupt invocation frequency to 10 times its original value (18.203 interrupt/sec.). Check the effect of the new setting on the operation of the system timer by issuing the

DOS command Time for a few times. You can also set the system timer tick period to a specific value, for example 25 msec., by loading counter 0 with the appropriate count, and then check the effect of this setting.

3. By masking hardware interrupt 8, stop the timer of the system. You can make sure that the timer is really stopped by using the DOS command Time, or by checking the content of the timer ticks software counter for a few times. The system timer can be run again by re-enabling hardware interrupt 8. This can be checked out using the same methods above.
4. Using the same method above, stop the timer of the system and then read the content of the timer ticks software counter. Keeping in mind that the content of this counter represents the number of timer ticks since midnight, calculate the current time-of-the-day in terms of hours, minutes, seconds, and hundredths of the second. Use the DOS command Time to make sure that your calculations were correct. **Hint:** The number of seconds that has elapsed since midnight equals the product of the number of timer ticks since that instance and the timer tick period (in unit of seconds), which is $54.936 * 10^{-3}$ sec.

6. References:

[1] S. Bennett, "*Real-time computer control*", Prentice-Hall International (UK) Ltd., 1988.

[2] A. Singh, and W. A. Triebel, "*The 8088 microprocessor programming, interfacing, software, hardware, and applications*", Prentice-Hall International, Inc.