

Hardware Interrupts

Thorne : 21.1, 21.3
(Irvine Edition IV : Section 16.4)

The Particular Challenges of **I/O Programming**

- Reference : Fundamentals of Embedded Software : Where C and Assembly Meet, Daniel Lewis. Chapter 6.
- **With “memory” programming :**
 1. Data transfers between memory occur as a result of an instruction **fetch-execute cycle**
 1. Time to complete is in the order of microseconds
 2. A program runs **synchronously** : Instructions are fetched then executed.
 - CPU **controls** the data transfers between memory.

The Particular Challenges of I/O Programming

- **With I/O programming**
 1. Input and output from/to external I/O devices (keypads, sensors, switches) often involve **physical movement**
 - Response times are determined by physical nature of device (eg. Bouncing of switches, A/D conversion, movement of disk head)
 - Response times are an **order of magnitude** slower than an instruction execution
 2. I/O devices operate **asynchronously** from the processor (and the program being run)
 - Availability of data for input OR device for output is **not under the control of CPU**
 - To transfer any data, the processor and I/O device must **synchronize** or “**handshake**”

Polled Input/Output Waiting Loops – Busy Waiting

CPU controls the synchronization with I/O device

- Execution is simple : sequential control flow

Example : Lab Switches

```
while ( ! (getSwitches() && 1000 0000B) ) { }  
processSwitch()
```

Is Switch E up?

Example : Hypothetical Simple Keyboard Device :

- When key pressed, character is put in data port and bit 0 in the status port is set to indicate “Key Ready”.
- When character is read from data port, status bit is cleared.

```
while (status && 0000 0001b == 0) { }
```

Read Keystroke from Data port

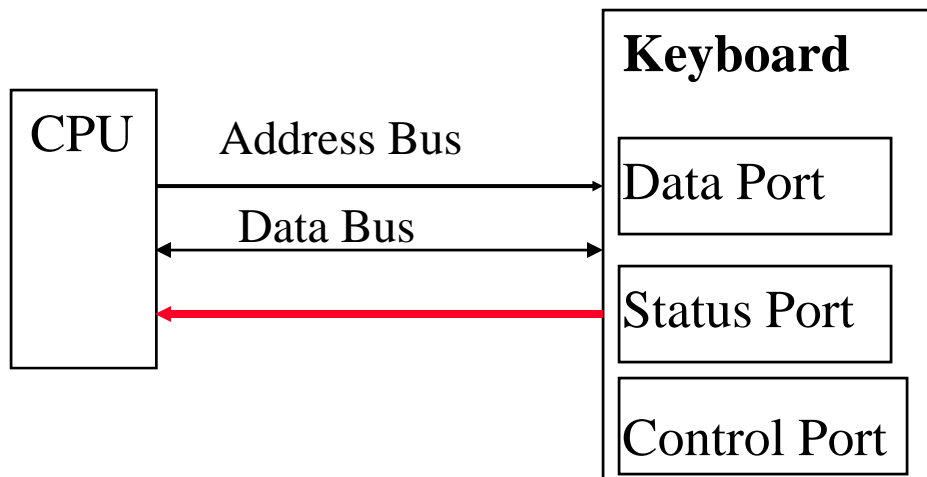
Is bit 0 in the status port set?

(Hardware) Interrupts : A Device-centric Handshake

- Example : Lewis
 - You are working at home doing taxes
 - The phone rings (an **interrupt**)
 - You stop work on the taxes and answer the phone (You **accept the interrupt**)
 - A friend wants to know a mechanic's phone number to arrange service
 - You give the phone number (You process the interrupt immediately)
 - You hang up and go back to your taxes.
- Example : Classroom questions

Example : Interrupt-Driven Keyboard

- **Simple Keyboard model:**
 - When key pressed, character is put in data port and **bit 0 in the status port** is set to indicate “Key Ready” ... and an **interrupt is sent to the CPU**
 - When character is read from data port, status bit is cleared



• **Hardware Interrupts** require a **hardware signal** from the device to the processor.

Hardware Interrupts

- The **interrupted processing** doesn't "know" it was interrupted
 - The processor just:
 1. temporarily suspended current **thread of control**
 2. ran **Interrupt Service Routine (ISR, later!)**
 3. resumed suspended thread of control
- **Polling**: CPU asks the peripheral devices whether there is anything to do now ?
 - Programming is **sequential** : Next instruction is determined by the fetch-execute cycle and by control transfer instructions.
- **Interrupts**: Device tells CPU it is time to do something NOW
 - **External hardware** signals spontaneously cause transfer of control (cause interruption in default sequential program).
 - No busy waiting, no polling.
 - Programming now requires an "**event-driven**" mindset

Learning the Event-Driven Mindset

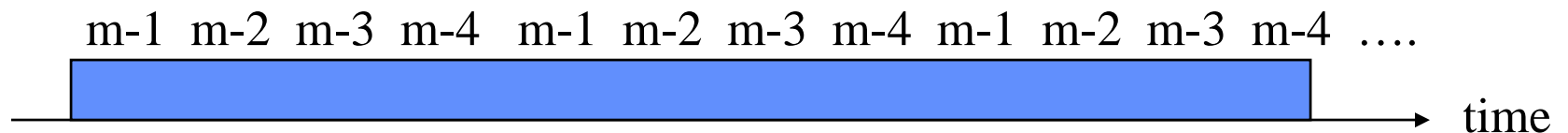
- We will show two ways to depict the **runtime behaviour** of an interrupt-driven program
- Let's set up an example to demonstrate :

```
main PROC  
  
here:  
    MOV AX,BX    ; m-1  
    MOV BX, CX   ; m-2  
    MOV CX, DX   ; m-3  
    JMP here     ; m-4  
  
main ENDP
```

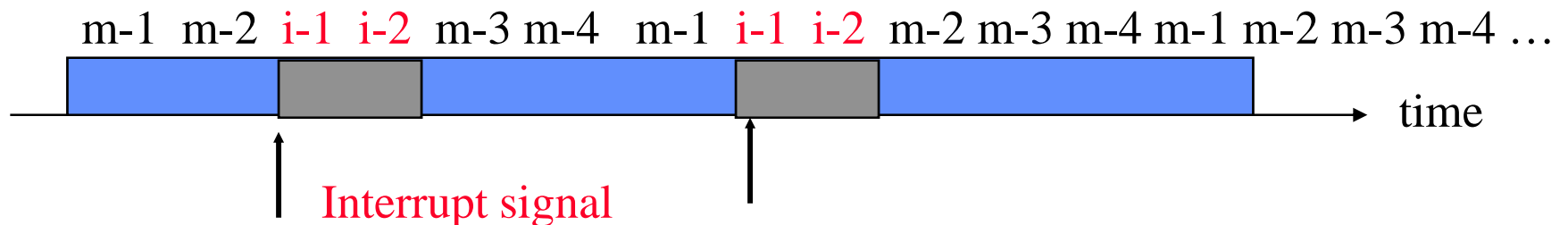
```
ISR PROC  
    MOV AX,1     ; i-1  
    IRET         ; i-2  
  
ISR ENDP
```


Learning the Event Driven Mindset : CPU Utilization Diagram

- Diagram depicts what CPU does as time progresses to the right
- If **no interrupts** happen (and interrupts don't have to happen!)
 - Main thread of control completely consumes the CPU

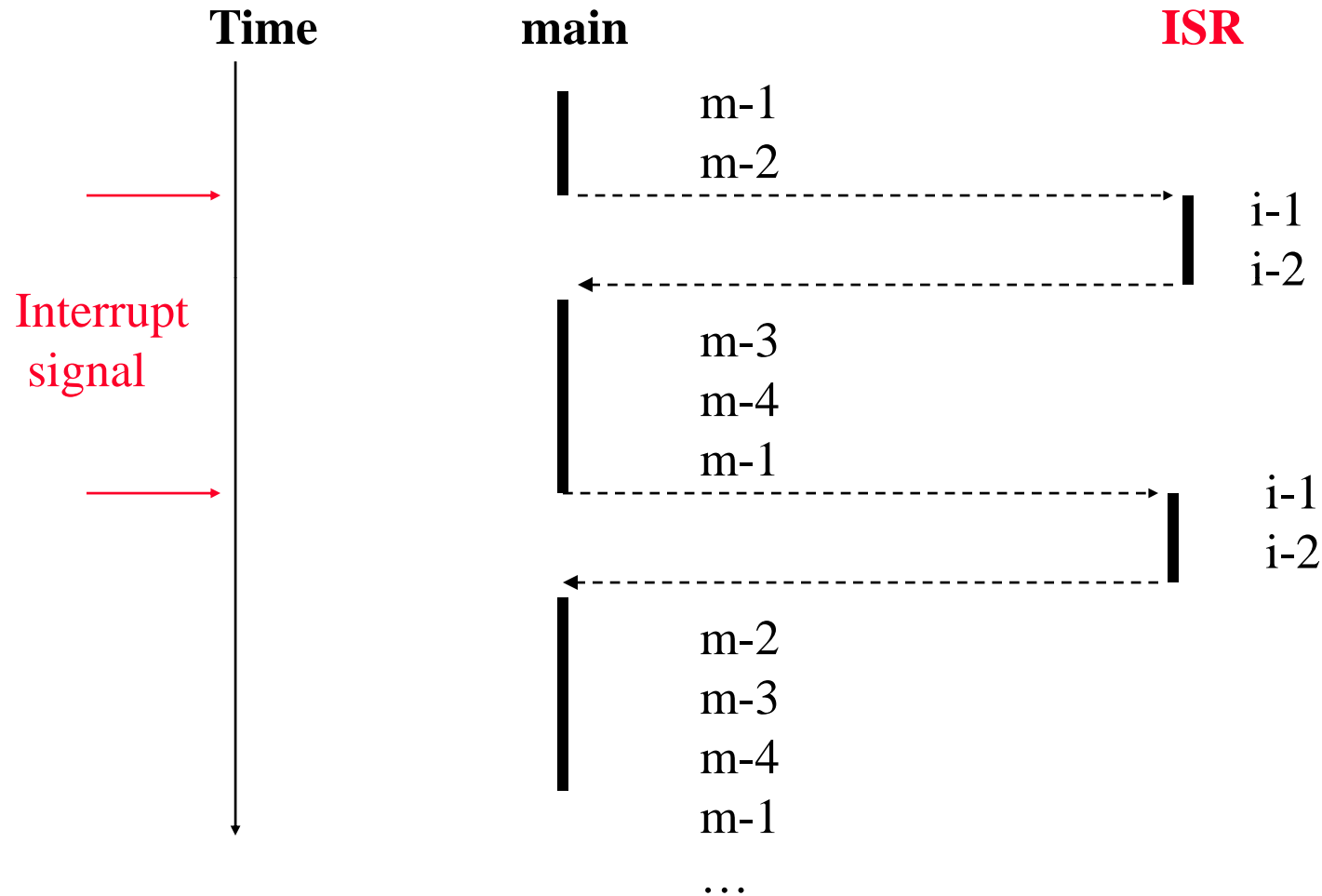


- If **interrupts happen**
 - CPU is used by **ISR** whenever the interrupt occurs.



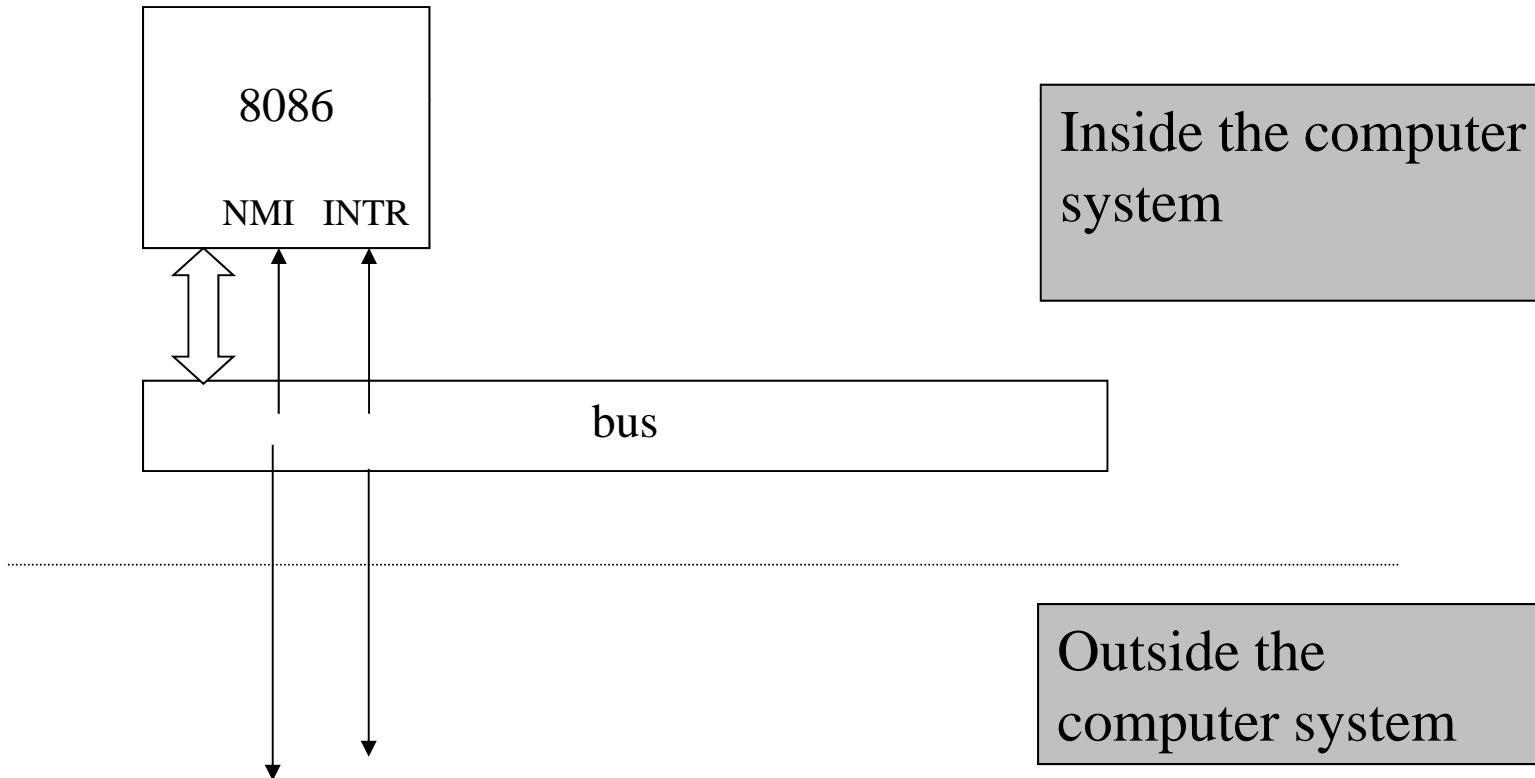
Learning the Event Driven Mindset : Thread Diagrams

- Diagram depicts the “lifelines” of each thread of control



The Interrupt Mechanism on the Intel 8086

- The 8086 processor has **two hardware interrupt** signals
 - **NMI** non-maskable interrupt
 - **INTR** Interrupt request (maskable interrupt)



Interrupt Maskability

- **Maskable** interrupts can be ignored by the CPU
 - Must be enabled before interrupting the CPU
 - 8086 Example : **INTR** is maskable
- **NonMaskable** interrupts cannot ever be ignored by the CPU
 - 8086 Example : **NMI** is non-maskable
 - Used for catastrophic errors (e.g. RAM failure
Power failure, etc).

An Interrupt request can be *pending*: the signal is active but it has not yet serviced

- For maskable interrupts, it may or may not be serviced until/if it is enabled

8086 Instructions for Interrupt Masking

Masking of **INTR** is done with the *control* flag **IF** in FLAGS register

CLI - clears IF bit in flags register (IF = 0)

- **disables** (masks) interrupts at the processor
- processor does **not monitor** INTR line while IF = 0

STI - sets IF bit in flags register (IF = 1)

- **enables** (unmasks) interrupts at the processor
- processor **monitors** INTR line while IF = 1

❑ State of IF does not affect **NMI**, **software interrupts** or **dedicated interrupts** (0..4)

❑ Later, we shall say : CLI/STI instructions disable/enable interrupts *at the processor*

The Interrupt Mechanism on the Intel 8086

Interrupt signals can occur anytime.

- When does processor consider interrupt signals?
- What ISR does processor execute ? **Where is this ISR ?**

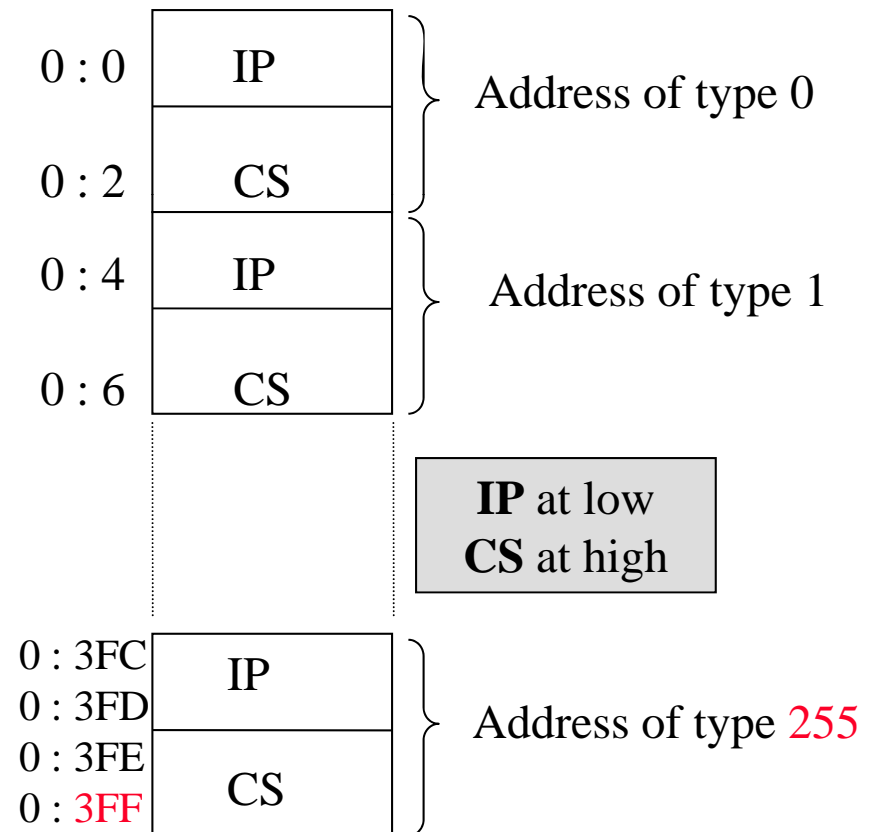
The complete instruction execution cycle :

1. Fetch instruction & adjust IP
2. Decode instruction
3. Execute instruction
4. **Check NMI** : If NMI asserted, perform “*related behaviour*”
5. **If IF = 1 check INTR** :
If INTR asserted, perform “*related behaviour*”

8086 Vector Table

Just another example of
Array Programming

- An array of **256** entries located at the reserved memory location **0000:0000**
 - Each entry is **“address”** of an **interrupt service routine (ISR)**.
 - The address is a FAR Pointer (**CS:IP** pair) (32-bits = 4bytes)
 - The array occupies addresses from **0:0** to **0:3FF** (256*4 bytes)
- Each entry is identified by unique **“interrupt-type”** (number), ranging from **0** to **255**
 - If $\text{interrupt-type} = i$ then the offset to relevant entry in the vector table = $0000H + 4 * i$



Interrupt Types : Deciding which ISR to run

- **Auto-vectored** interrupts : The interrupt-type (the vector) is predefined as part of the processor design
 - For a given hardware signal, the CPU automatically goes to a particular interrupt-type in the vector table.
 - **8086 Example : NMI is auto-vectored to Interrupt-type 2**
 - Whenever NMI is **asserted**, the 8086 **always** executes ISR at Interrupt **Type = 2**
 - The address of the **NMI ISR is always at 0000:0008h**

Interrupt Types : Deciding which ISR to run

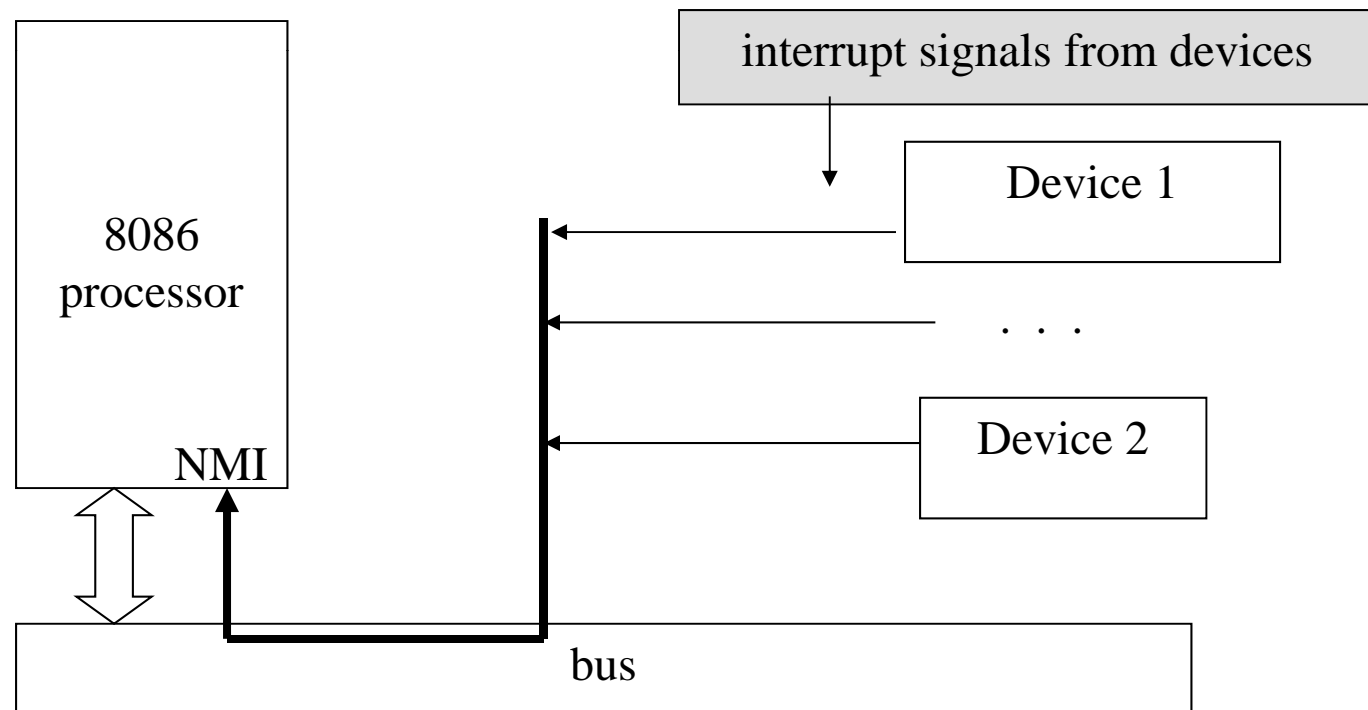
- **Vectored** interrupts : The interrupt-type (the vector) is determined during systems design and is provided to the CPU.
 - The CPU performs an “**interrupt-acknowledge**” cycle where it reads the interrupt-type from the data bus
 - **No software** is involved (More in 4601)
 - Interrupting device must provide (ie. write) the interrupt-type
 - External hardware
 - The address of the ISR is at $0000:\text{type} * 4$
 - VARIABLE
 - **8086 Example** : **INTR** is a vectored interrupt

Question : **NMI** and **INTR** – Does this mean we can have only 2 interrupt sources ?

Question : Why do we need **256** entries in the vector table ?

Multiple Interrupt Sources with **AutoVectored** Interrupts

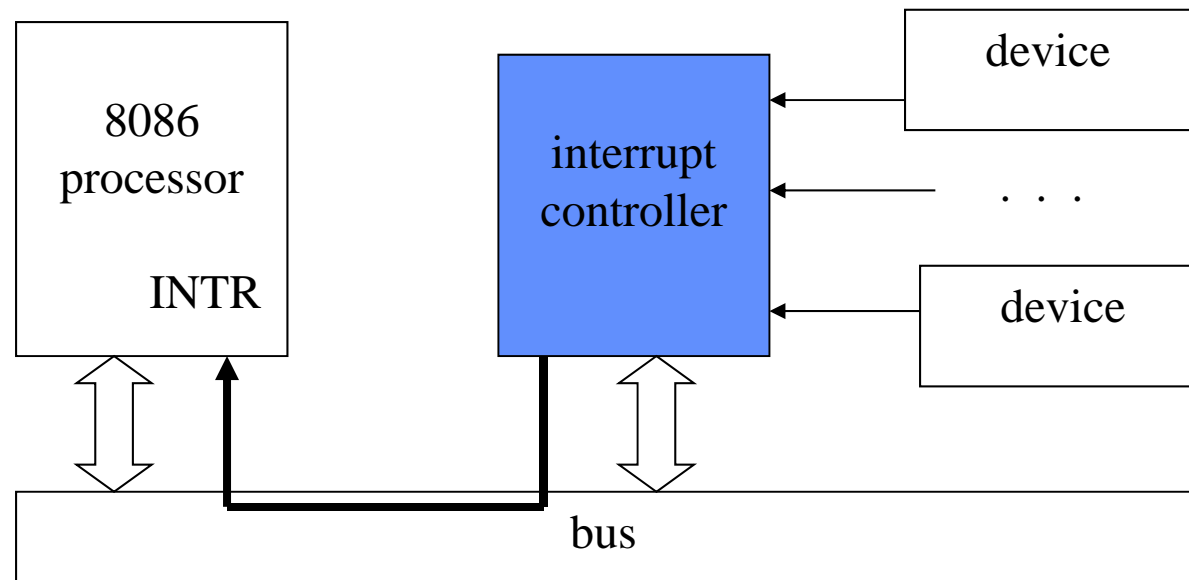
- Multiple devices share the same interrupt line
- CPU must **poll** status port on each device to determine which one generated interrupt.



Multiple Interrupt Sources with **Vectored** Interrupts

Interrupt controller acts as a funnel for multiple device interrupts

- Interrupt controller “handshakes” with CPU (**Interrupt Acknowledge Cycle**) on behalf of the device.
- Interrupt controller knows which device interrupted
- Interrupt controller “tells” the CPU by writing a unique **interrupt-type** associated with interrupting device on the data bus
- CPU uses interrupt-type to execute the appropriate **ISR**



Perform “Related Behaviour”

- **Review : Execution semantics of CALL**

- NEAR CALL target PUSH IP, IP := target
- FAR CALL target PUSH CS, PUSH IP,
 CS:IP := target

- **Execution semantics of Interrupt**

- Push FLAGS register
- Clear IF bit in flags to 0
- Clear TF bit in flags to 0
- Push CS
- Push IP

- **Fetch new CS from**
- **Fetch new IP from**

$0 : n*4 + 2$
 $0 : n*4$

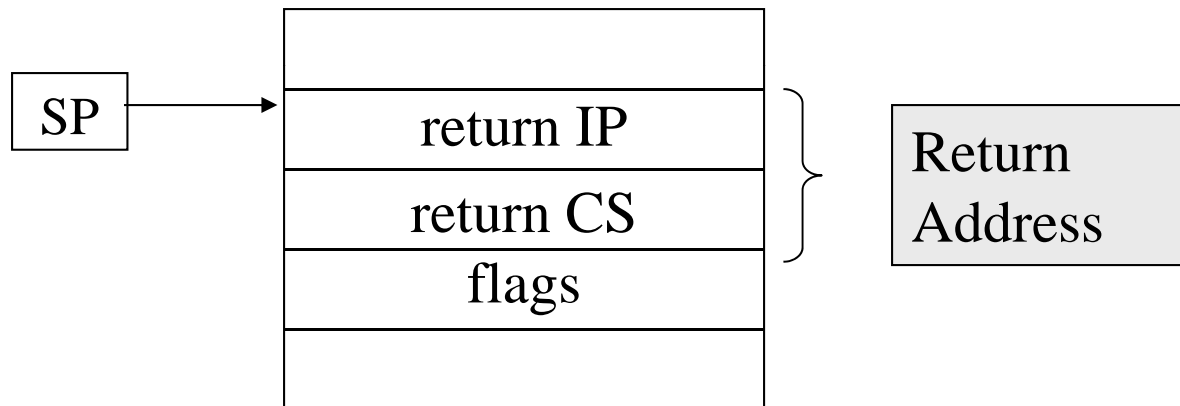
Learn this

- From Vector Table in memory!
- n: Interrupt Type!

- **Question : What does all this ?**

ISR Stack Frame

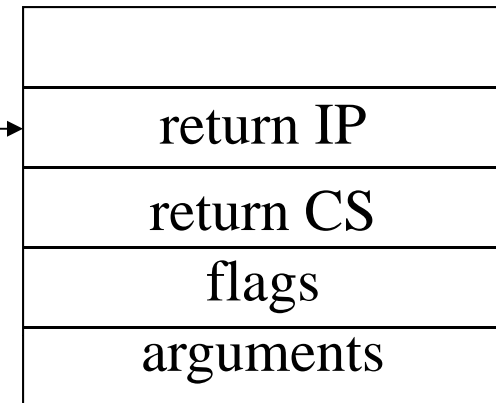
- The ISR stack frame different from subroutines!
 - **Return address** is always a FAR address **CS:IP**
 - **FLAGS** are also pushed.



Returning from an ISR

- The RET instruction will not work
 - Why not ?

SP



- When writing ISRs, you must use **IRET**.
 1. Pops 32-bit return address (CS:IP)
 2. Pops flags

Restores FLAGS value to what they were before IF and TF were cleared !

- **Example :**

```
isr PROC FAR
    IRET
isr ENDP
```

Installing an ISR

- Interrupts only work if **ISR address** has been loaded previously into correct vector!
 - This is called “installing” the ISR in the **vector table**

Dynamic Installation of an ISR

```
myint_type EQU 40 ; install as vector
```

```
myvector EQU myint_type * 4
```

```
.code
```

```
myisr
```

```
PROC FAR
```

```
IRET
```

```
myisr
```

```
ENDP
```

ISR

```
main PROC
```

```
CLI
```

```
MOV AX, 0
```

```
MOV ES, AX ; ES → vector table segment
```

```
MOV ES:[myvector], OFFSET myisr
```

```
MOV ES:[myvector+2], @code
```

```
...
```

```
STI ; NOW, interrupts can happen
```

IP of ISR

CS of ISR

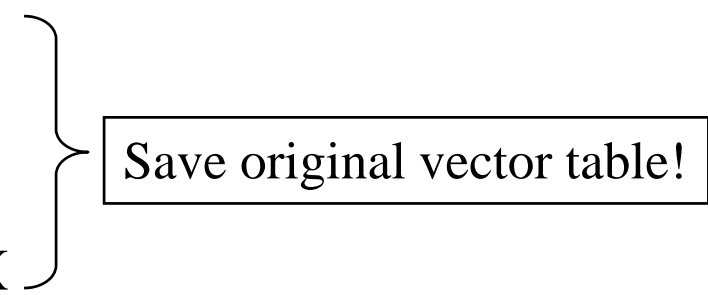
segment override for destination segment !

Installing the ISR

- When installing an ISR, you are **over-writing** a previous value inserted by the O/S during startup
 - Example : The entry may have a “useful” DOS value .
 - Example : Even “unused” entries have an address of a default ISR just in case (eg. a simple return)
- Whenever installing vectors, first **save the existing contents** of the vector.
 - The saved values should be restored before the program exits
 - If save/restore is not done, the OS (eg. DOS) might not run properly.
 - Sound familiar ? (Hint : SYSC-3006 Subroutine Policies)

Robust Version of ISR Installation

```
.data
old_vector_offset    dw ?
old_vector_segment   dw ?
.code
...
MOV  AX, 0
MOV  ES, AX          ; ES → vector table segment
MOV  AX, ES:[myvector]
MOV  old_vector_offset, AX
MOV  AX, ES:[myvector+2]
MOV  old_vector_segment, AX
MOV  ES:[myvector] , OFFSET myisr
MOV  ES:[myvector+2] , @code
...
```

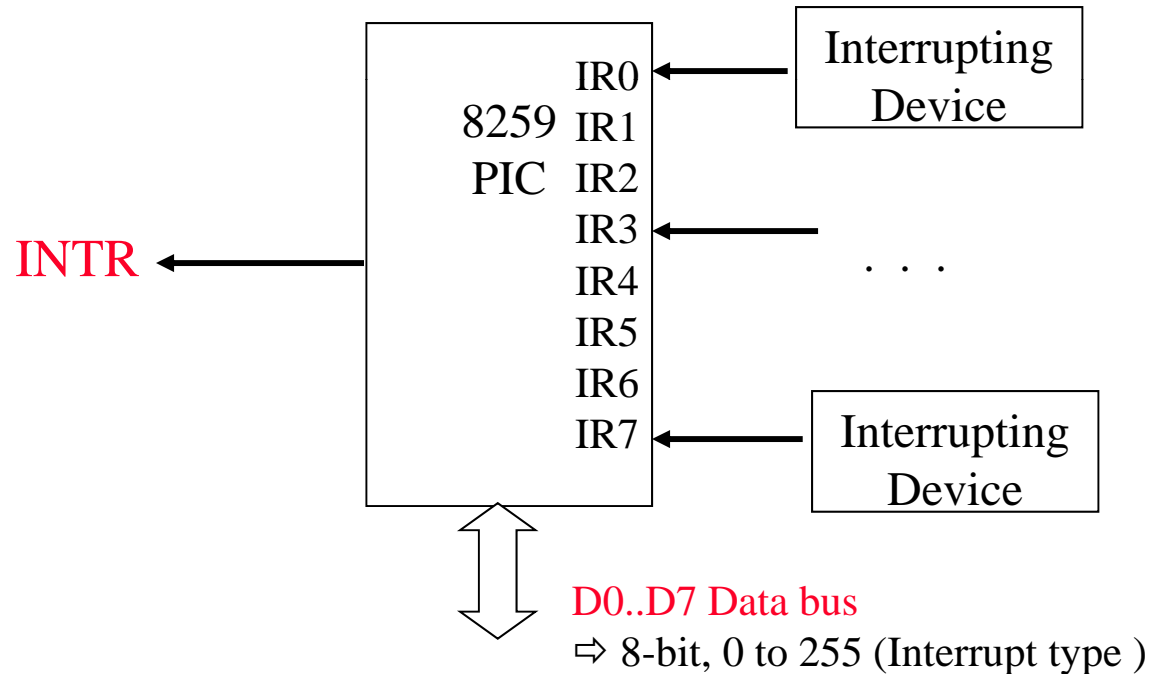


Save original vector table!

The Intel **Programmable Interrupt Controller (PIC)**

In 80x86 based PCs, interrupt controller is the **Intel 8259A**

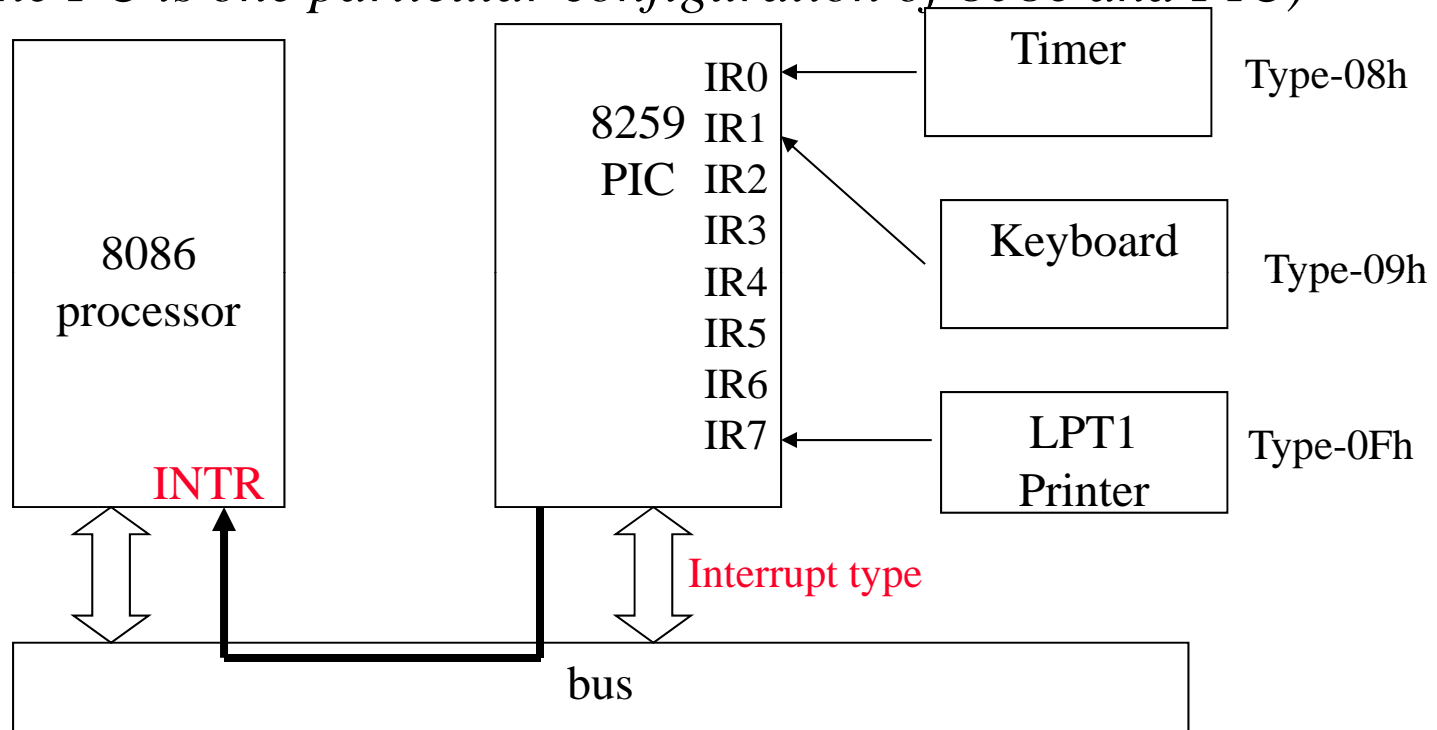
- Complex (ie. configurable / programmable) behaviour
 - Example : The specific interrupt-type to be associated with each interrupting device is often programmable
- Itself is an I/O device to be read/written.



It supports 8 device inputs : $IR_0 \rightarrow IR_7$

The PC configuration of 80x86 and 8259 PIC

(NB The PC is one particular configuration of 8086 and PIC)



During power-up, BIOS programs (initialises) the master PIC:
IR0 → IR7 mapped to interrupt types 08h → 0Fh

PC Example : Keyboard

- Assume interrupts are **enabled** (assume $IF = 1$)
- When the keyboard hardware logic asserts IR1 at PIC, the **PIC** generates **INTR** signal to the processor
 - During interrupt acknowledge, PIC identifies interrupt source as type 9
 - The CPU executes the INT 9h behaviour
 - Saves the flags
 - Clears IF and TF (Disabling interrupts at processor)
 - Saves CS and IP
 - Reads the **interrupt-type** = 9h from the **Data bus** and **vectors** to the ISR pointed to by the double word at address of $0 : 9h * 4$ in **Vector Table** (memory)
- Execution of ISR 9 is caused by hardware interrupt mechanism
 - **No software involved** in the invocation of ISR 9 !

Interrupts disabled when ISR begins execution

Some (as yet) Unanswered Questions:

1. If two devices generate interrupt signals **at the same time**, which ISR should be executed first?

order?

2. If the CPU is executing an ISR, and a second device interrupts, when should the second ISR be executed?

interrupting an ISR?

not possible unless ISR
re-enables interrupts !
i.e. $IF = 1$

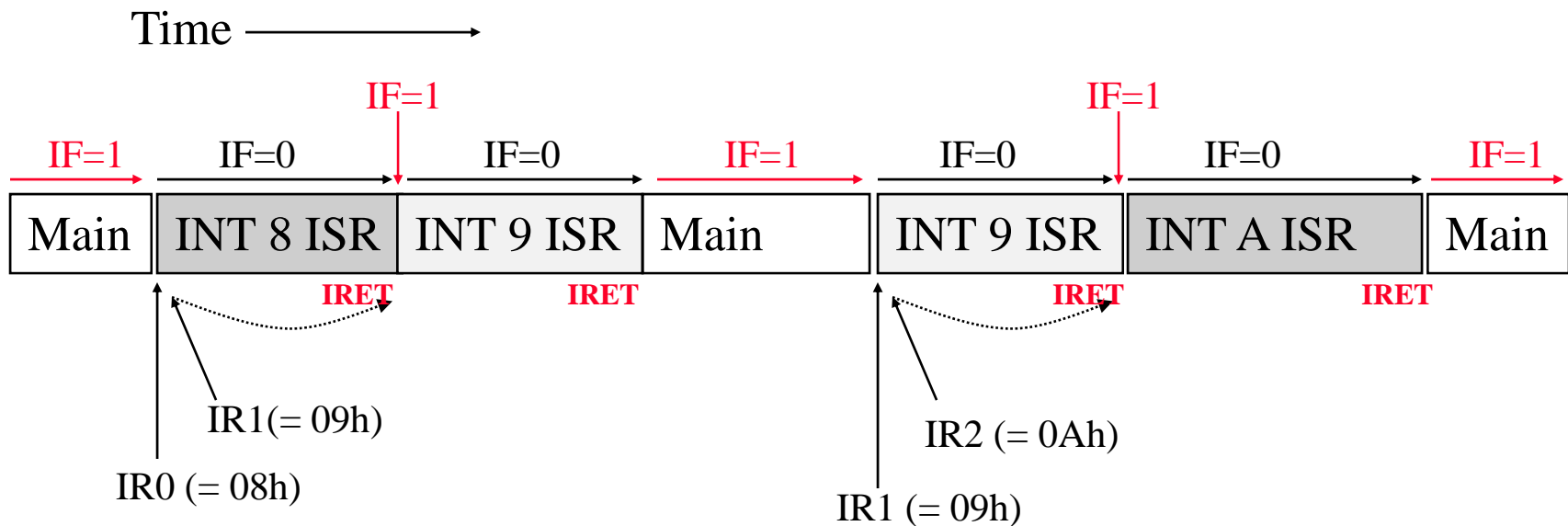
- When $IF=0$, no further (maskable) interrupts are accepted.
- NMI is always accepted regardless of IF .

Interrupt Priority

- Interrupting Devices are assigned **priorities**
 - Higher priority devices take precedence over lower priority devices
 - Priority is applied whenever interrupts **coincide**
 - When multiple interrupts occur **at the same time**
 - When new interrupts occur while still processing the ISR of previous interrupts.
- Typically, **interrupt controllers** manage priority issues.
 - In PC's
 - The devices have pre-configured connections to PIC
 - Timer is always IR0 and Keyboard is always IR1
 - DOS programs the 8259A to assign priority based on the device connection
 - **IR0 == highest priority and IR7 == lowest priority**

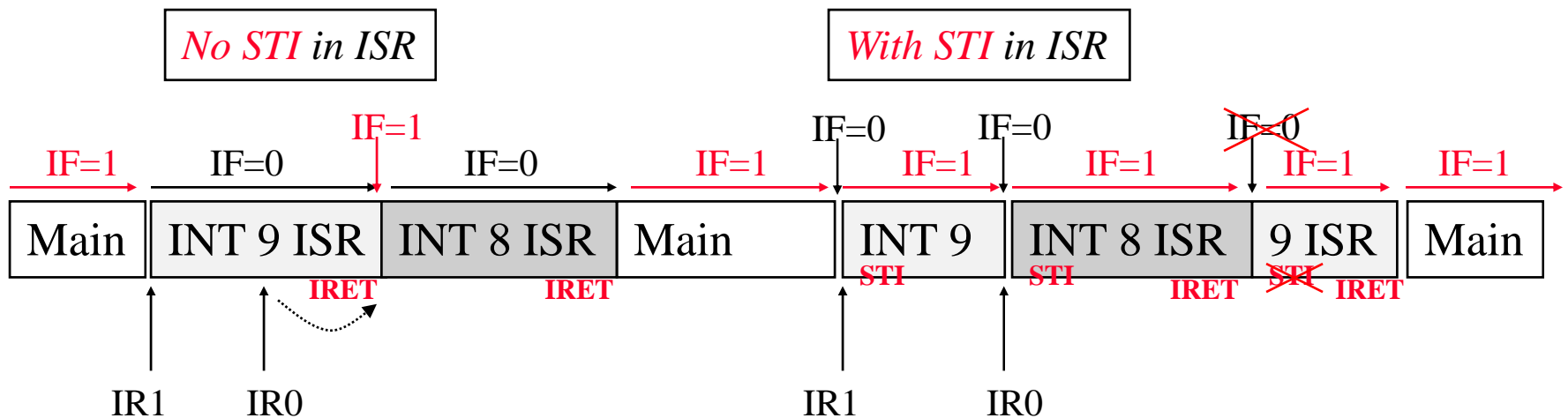
Interrupt Priority Scenarios

1. If two devices generate interrupt signals at the same time, which ISR should be executed first?



Interrupt Priority Scenarios

- If the CPU is executing an ISR, and a second device interrupts, when should the second ISR be executed?
 - If a **higher** priority interrupt follows a lower priority, **the PIC generates another interrupt.**

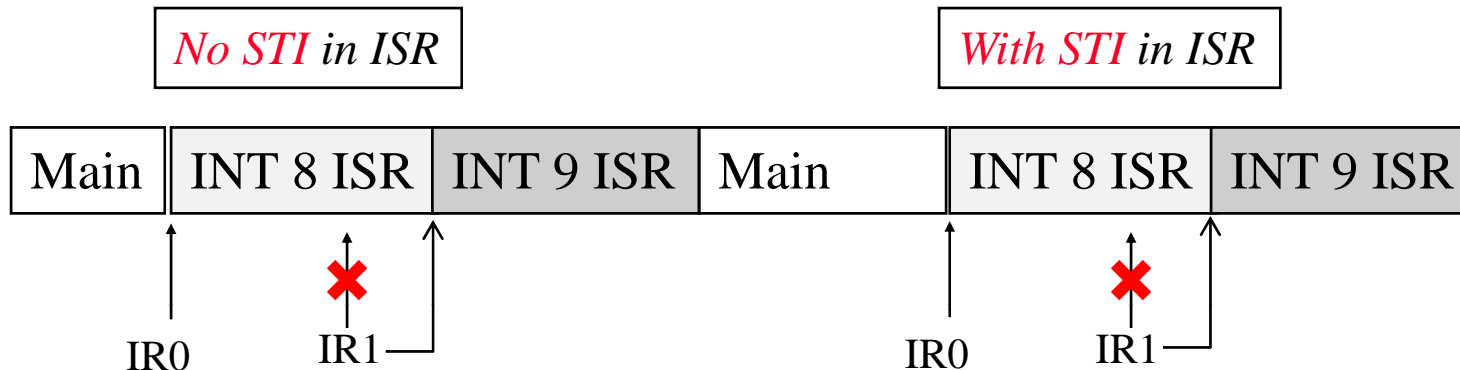


- The PIC *will try* to allow a higher priority interrupt to interrupt a lower priority ISR !
 - The second interrupt will not be recognized by the processor until interrupts are **re-enabled until IF = 1**

When is this ?

Interrupt Priority Scenarios

- If a lower priority interrupt follows a higher priority, the PIC *maintains* the priority .



- It “**remembers**” (latches) the **lower** priority interrupt **until the high priority ISR is finished** ... regardless of interrupts being enabled/disabled
- When **finished**, PIC generates another interrupt ... on behalf of the lower priority device.
- Hmmm.... Two More Questions:
 - How many interrupts can the PIC remember?
 - How does the PIC know when the **higher** priority ISR is finished?

Pending Interrupts

- Terminology: A “**Pending**” Interrupt is an interrupt signal that is latched somewhere in the system, but has not yet been acknowledged by the processor
 - Interrupts can be pending **at device** and / or **at the PIC**
- Example : The Intel 8259 has an internal **8-bit** register
 - one bit per IR input
 - When IR is asserted, the associated bit is set
 - When the interrupt on IR is acknowledged, the associated bit is cleared
 - In summary, the PIC has 1-bit memory for each IR
 - It can remember up to 1 pending interrupt for each IR

⇒ Total 8 different IRs

End-of-Interrupt (EOI)

- After sending an interrupt to processor, PIC needs to know when it is safe to generate a lower priority interrupt
 - the PIC requires feedback from the CPU
- End Of Interrupt (EOI) is a command sent **to PIC from the CPU**
 - It is not part of the INTA cycle; it is not done in hardware
 - It is a software command; ie. something **your program** must do.

PIC Programmer's Model

Simple version here –
on a need-to-know basis.
Complete details in ELEC 4601

- The PIC is an **I/O Device** so it has I/O port addresses
 - It appears as two 8-bit ports:

Interrupt Mask Register (Port 21H) read/write

- Allows us to enable/disable individual interrupts at the PIC
- bit $i = 1$ IR_i is masked (not recognized by the PIC)
- bit $i = 0$ IR_i is unmasked (recognized by the PIC)

Beware : mask at PIC \rightarrow bit = **1** mask at processor \rightarrow IF = **0**

Command Register (Port 20H) - write-only

- Write 20H to inform PIC of **end of interrupt (EOI)**

PC Keyboard : I/O Programmer's Model

- The PC keyboard is interrupt driven
 - It can't run in polled mode because there is no status port
 - It is connected to IR1 of the PIC, through 8255 Parallel Peripheral Interface (PPI)
 - The 8255 is our programming interface to the keyboard
- There are 2 interrelated 8255 PPI ports:
 - Data Port (Port PA) : I/O address 60H
 - Control Port (Port PB) : I/O address 61H

PC Keyboard : I/O Programmer's Model

- The keyboard **data port** (Port **A**) has dual functionality :
 - Dual = Different values are read from the same port!
 - The value read depends on the setting of Port **B, Bit 7!**
 - Port B, Bit 7 = **0** “**Scan Code**” is read.
(ie. identify the keystroke)
 - Port B, Bit 7 = **1** “Configuration switch data” is read
- In this course, we never use the configuration data, so why don't we just set Port PB, Bit 7 = 0 and leave it there ?

PC Keyboard : Hardware Requirement

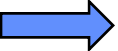
- The keyboard **will not send the next scan code until the previous scan code has been “acknowledged”**
- To acknowledge scan code:
 - **PB bit 7 must be toggled from 0 → 1 and then 1 → 0**
- **CAREFUL!** All bits in PB have important values
- To acknowledge:
 1. Read Port B : PB_value
 2. Force bit 7 = 1: PB_value OR 80H
 3. Write modified value back to Port B
 4. Write original value (with bit 7 = 0) back to Port B
- NB. The keyboard hardware is initialised when DOS boots

PC Keyboard : Scan Codes

- The scan code is a code sent from keyboard whenever its keys change state
 - Scan codes are **NOT ASCII** codes!!
- The scan codes runs from 0 – 53H
 - e.g. “A” key scan code = 1EH
- Scan codes are “make/break coded”
 - one code sent when key is **pressed** (make)
 - different code sent when key is **released** (break)
 - The only difference is the most-significant bit
 - If MSBit = 0 → key was **pressed**
 - If MSBit = 1 → key was **released**
 - Example : Letter A
 - Make ‘A’ = 1EH (0001 1110b)
 - Break ‘A’ = 9EH (1001 1110b)

Keyboard Scan Codes (Read from Port HEX 60 = DEC 96) (Keyboard Layout)

Top number ... DEC
Bottom number ... HEX

Key 

Scan code {

| | | | | | | | | | | | | | | | | | | |
|----|----|-----|----|----|----|----|----|-----|----|----|----|----|----|-----|-----|--------------------|--------------------|----------------|
| F1 | F2 | ESC | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | - | = | BkS | Num Lok | Scr Lok | |
| 59 | 60 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 69 | 70 | |
| 3B | 3C | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 45 | 46 | |
| F3 | F4 | TAB | Q | W | E | R | T | Y | U | I | O | P | [|] | | 7 ³ 8 | 9 ³ - | |
| 61 | 62 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | | 71 ³ 72 | 73 ³ 74 | |
| 3D | 3E | 0F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | | 47 ³ 48 | 49 ³ 4A | |
| F5 | F6 | CTR | A | S | D | F | G | H | J | K | L | ; | ' | ` | 28 | 4 ³ 5 | 6 ³ | |
| 63 | 64 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | | 75 ³ 76 | 77 ³ | |
| 3F | 40 | 1D | 1E | 1F | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | | 4B ³ 4C | 4D ³ | |
| F7 | F8 | Shf | \ | Z | X | C | V | B | N | M | , | . | / | Shf | Prt | 1 ³ 2 | 3 ³ + | |
| 65 | 66 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 78 ³ 80 | 81 ³ 78 | |
| 41 | 42 | 2A | 2B | 2C | 2D | 2E | 2F | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 4F ³ 50 | 51 ³ 4E | |
| F9 | F0 | A | t | | | | | pac | | | | | | Cap | Lok | I | s ³ D | l ³ |
| 67 | 68 | 5 | | | | | | 57 | | | | | | 5 | | 8 ³ 8 | ³ | |
| 43 | 44 | 3 | | | | | | 39 | | | | | | 3 | | 5 ³ 5 | ³ | |

PC Keyboard : Multiple Key Combinations

- Multiple key combinations
 - <SHIFT> ‘A’
 - <CTRL><ALT>
- Software must manage multiple key combinations.
 - Left Shift key press, make code = 2AH
 - Right Shift key press, make code = 36H
 - Ctrl key press, make code = 1DH
 - Alt key press, make code = 3AH
- Keyboard software must track the state of control keys for correct interpretation

Example : A Simple Keyboard Driver

- Requirements
 - prints **uppercase** char's representing keys pressed
 - ALT, SHIFT, CTRL keys (and a few others) are **not** managed
 - exit the program by resetting
 - ISR ignores key **released** scan codes
 - uses **lookup table** to convert key pressed **scan code** to uppercase **ASCII** representation

Example : A Simple Keyboard Driver

- Program architecture
 - The duties have been divided between the **main program** and **keyboard ISR**
 - **Keyboard ISR** gathers data as user enters keystrokes
 - **Main** prints the keystrokes
 - The data is shared between the two threads in a variable **KEYBOARD_CHARACTER (global variable)**
 - The variable is initialised to 0FFh to represent “no data”
 - (0FFh is not an ASCII code for any key)
 - The keyboard ISR puts ASCII code in the variable
 - Main program **polls** the variable until valid data is found;
 - When main reads ASCII code, it must reset the variable to “no data” value

How does
it know
when ?

Keyboard : Code Fragments

```
LF      EQU    0AH
CR      EQU    0DH
```

.data

shared variable initialized to “no data” value

```
KEYBOARD_CHARACTER      DB      0FFH
```

```
SCAN_TABLE ; lookup table
```

```
DB      0,0,'1234567890-=',8,0
DB      'QWERTYUIOP[]',CR,0
DB      'ASDFGHJKL;',0,0,0,0
DB      'ZXCVBNM,./',0,0,0
DB      ' ',0,0,0,0,0,0,0,0,0,0,0,0,0
DB      '789-456+1230'
```

Use 0 for keys to ignore

Keyboard : Code Fragments

```
.code
MAIN PROC
    CLI                    ; Disable ints while installing ISR
    ; Install the Keyboard ISR at Interrupt Type = 9
    ; Enable keyboard interrupts at the PIC
    STI                    ; Enable interrupts at the processor
FOR_EVER:                  ; press reset to exit ☺
    CALL GET_CHAR          ; returns ASCII in AL
    PUSH AX                ; save char & pass parameter
    CALL DISPLAY_CHAR     ; displays char in AL
    POP AX                 ; restore char & clear stack
    CMP AL , CR           ; check for Enter key
    JNZ REPEAT_LOOP
    MOV AL , LF           ; if Enter – do LF too !
    PUSH AX
    CALL DISPLAY_CHAR
    ADD SP, 2
REPEAT_LOOP:
    JMP FOR_EVER
MAIN ENDP
```


Keyboard ISR : Code Fragments

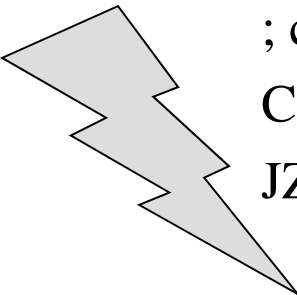
```
GET_CHAR    PROC NEAR
```

```
    ; poll until char received from ISR
```

```
    ; check for “no data” value
```

```
    CMP     KEYBOARD_CHARACTER , 0FFH
```

```
    JZ     GET_CHAR
```



| |
|---|
| Is this a critical region? Should it be protected? |
|---|

```
    ; get ASCII character
```

```
    MOV     AL , KEYBOARD_CHARACTER ; global variable from KISR
```

```
    MOV     KEYBOARD_CHARACTER , 0FFH
```

```
    RET
```

```
GET_CHAR    ENDP
```

Keyboard : Code Fragments

KISR PROC FAR

; Standard ISR Setup (**Save registers**, including DS)

MOV AX, @data

MOV DS, AX

IN **AL** , 60H ; **AL = scan code**

; **Acknowledge Keyboard : Toggle PB bit 7**

PUSH AX ; save scan code

IN AL , 61H ; read current PB value

OR AL , 80H ; set bit 7(= 1)

OUT 61H , AL ; write value back + bit 7 = 1

AND AL , 7FH ; clear bit 7 (=0) – back to original

OUT 61H , AL ; write original value back

POP AX ; restore scan code

Keyboard : Code Fragments

TEST AL , 80H ; ignore break codes

JNZ SEND_EOI

; Convert make code to **ASCII**

LEA BX , SCAN_TABLE

XLAT

CMP AL , 0 ; some keys are ignored !

JZ SEND_EOI

; Put **ASCII encoded value** in shared variable

MOV **KEYBOARD_CHARACTER** , AL

SEND_EOI:

MOV AL , 20H

OUT 20H , AL

; Standard ISR exit code (**restore** all registers)

IRET

KISR ENDP

Example : Polled **Timing** Loop

- Programs must often manage **time** (eg. timing constraints)
 - Example : In animation, motion is timing dependent
 - Example : Changing display after fixed time,
 - To clear a dialog box.
 - To turn on/off a LED for the floppy disk access light.
- In a program, what is time?
- Various schemes are used to manage time : hardware / software

Software Solution : Polled Timing Loop

- Example : A **software-only solution** for a timing loop
 - Write a loop that simply counts to waste time (busy waiting)

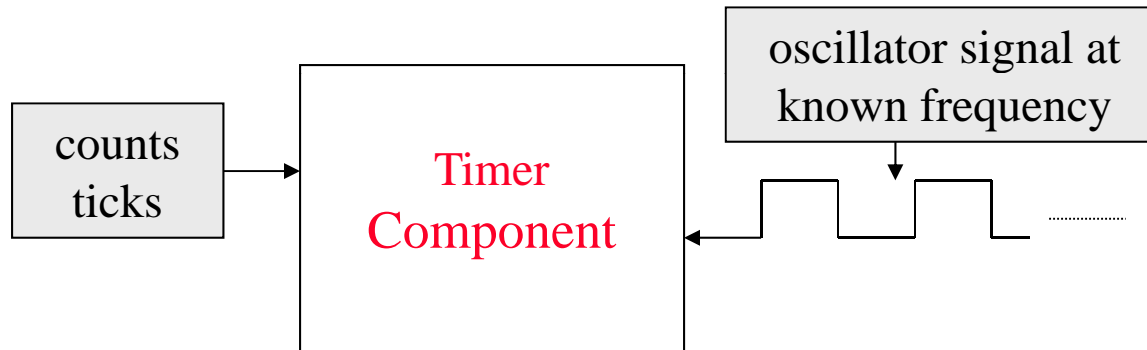
```
for ( int i = 0; i < 10000; i++ )  
{  
    for (int j = 0; j < 10000; j++)  
    { } // empty body  
}
```
- Advantage: It is **simple** software; No explicit h/w involved
- Disadvantage:
 - Timing is based on the **execution speed** of the processor.
 - It is not portable because execution speed **varies** on different machines
- download old DOS games ??

Hardware Solution : Hardware-Based Timing

- Requirement : The computer system includes hardware dedicated to managing time

- We will introduce the timing **hardware** as a separate **I/O device**
- Often, there is no external device; it is just **internal** component but still with usual programmer's model of an I/O device

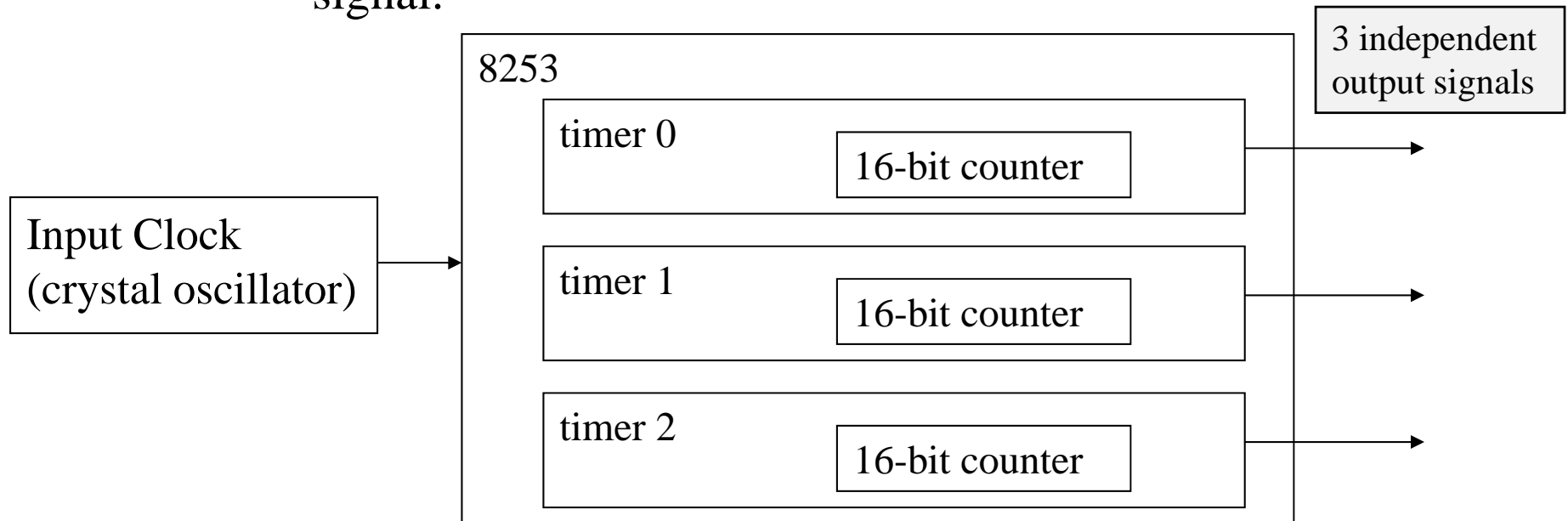
Integration



- **Timing Component :**
 - Input Signal : oscillator circuit generates “ticks” at a known frequency – e.g. square wave at some frequency
 - It “counts” ticks (edge-to-edge) on the input signal
 - Since frequency of ticks is known, then counting a fixed number represents the passage of a known amount of time

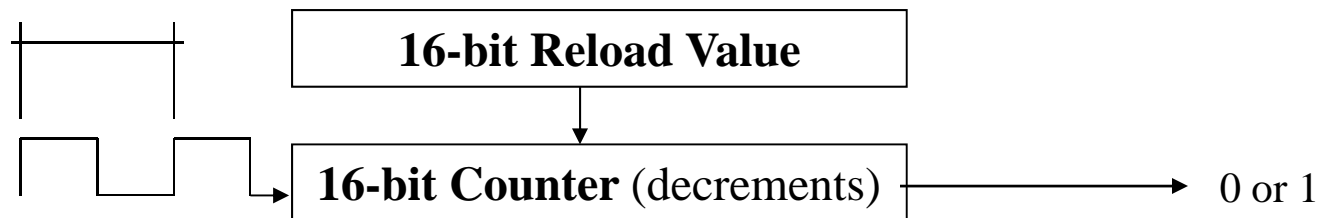
Intel 8253 Programmable Interval Timer (PIT)

- One 8253 component has **3** independent timer components
 - Each timer counts ticks on the **same master clock** input
 - Each timer generates its own output signal
 - Each timer can be programmed for one of **6 modes** (mode 0-5) that determine the shape of the output signal.



Basic Timing Function of 8253

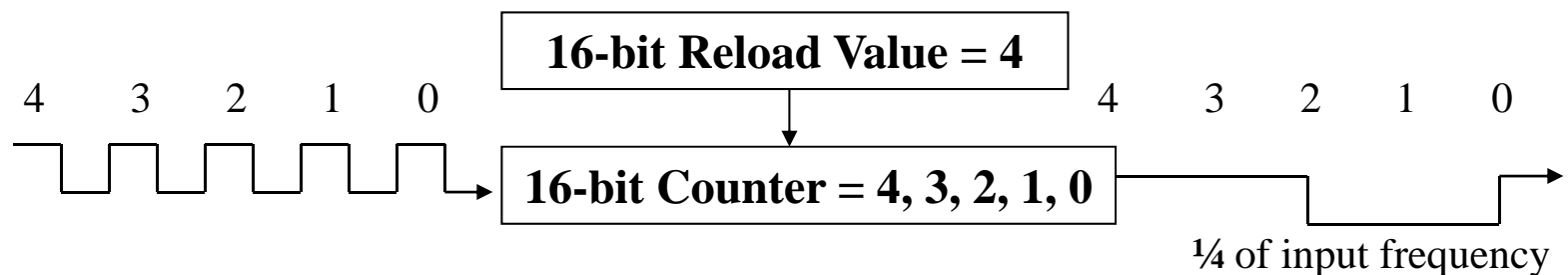
- Each timer's counter is a **16-bit unsigned** value
- Timer **decrements** counter **every tick** on the input clock (typically from crystal oscillator)
- When counter reaches zero, its output signal changes (if 1, now 0; if 0, now 1)
- Some “modes” **automatically reload counter** and start again ... leading to digital patterns.



- Hence, the counter value is related to the period of the output signal

Square Wave Generator Mode (Mode 3)

- In **mode 3**, a 8253 timer component generates a **square wave** on its output signal.
 - Square wave : approx. 50% duty cycle
 - Each time master clock input signal “ticks”, the timer’s counter is decremented
 - When the counter reaches **half** of original value, the **output signals is toggle**
 - When counter reaches **0**, the **output signal is toggled** and the counter is **reloaded**.
 - last value written to counter is used as reload (original) value



Square Wave Generator Mode (**Mode 3**)

- In Mode 3, the timer's output is a scaled down version of the input clock
 - There is **one output** cycle for **every n input** cycles where n is called the “**scaling factor**”

$$\text{output freq.} = \text{input freq.} \div \text{scaling factor}$$

- Usually, we need to find the scaling factor to *program* the timer component
 - Need to determine the initial counter value that will generate a desired output frequency

$$\text{scaling factor} = \text{input freq.} \div \text{output freq.}$$

8253 PIT Programmer's Model

- The PIT is an I/O device with **four** 8-bit ports (registers)
 - On the PC: port addresses are 40H → 43H
- There are three **8-bit Data Registers**
 - Timer 0 Counter Register 40H
 - Timer 1 Counter Register 41H
 - Timer 2 Counter Register 42H
- **Question** : How can 8-bit data registers be used to initialise **16-bit** internal counters ?

8253 PIT Programmer's Model

| RL1 | RL0 | read/load sequence |
|------------|------------|-------------------------------|
| 0 | 1 | read/load LSB only |
| 1 | 0 | read/load MSB only |
| 1 | 1 | read/load LSB first, then MSB |

M2 M1 M0 mode (6 modes)

x 1 1 **mode 3** square wave generator where x = don't care
(other modes in ELEC-4601)

BCD (binary coded decimal)

0 **16 bit binary count**

1 BCD count (4 decimal digits)

8253 Hardware Configuration and Limits on the PC

- On a PC, the 8253 is wired such that
 - master input signal = **1.19318 MHz**
 - output from timer 0 connected to **IR0** on PIC
- The scaling factor is an unsigned number from **0 ... FFFFh** (65,535) but ...0000H = 65,536 (i.e. 10000H, with implied MS bit)
- **Question** : What is the maximum frequency ?
- **Question** : What is the minimum frequency ?

Generate timer interrupts!

$$\begin{aligned}\text{min output} &= \text{input} \div \text{max scaling factor} \\ &= 1.19318 \text{ MHz} \div 65,535 \\ &= \mathbf{18.2 \text{ Hz}}\end{aligned}$$

Hardware Timing Example : DOS Time-of-Day

- The DOS maintains the time-of-day feature HH : MM : SS
 - > date
 - Provide file timestamps.
- DOS uses Timer 0 to provide a real-time clock interrupting at a frequency of 18.2 Hz.
 - When DOS boots, **Timer 0** is programmed for **Mode 3** (square wave) and an “*initial time*” is loaded
 - The **Timer 0 ISR** counts “*ticks*” (at 18.2 Hz)
 - “*initial time + ticks*” is used to calculate “current” the time-of-day

Hardware Timing Example : 20Hz Real-Time Clock

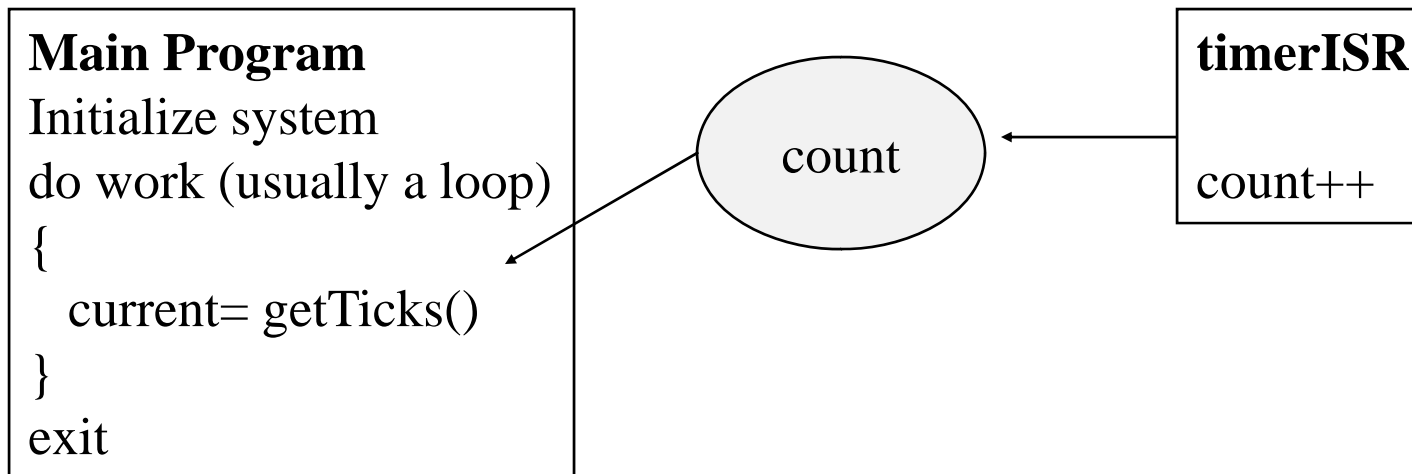
- Example : Devise a program that provides replaces the DOS time-of-day to provide a “ticksCount” = number of 20Hz ticks
 - Program timer to interrupt at 20 Hz
 - Timer ISR keeps running tick count of 20 Hz ticks
 - Main program can use tick count to provide timing information

20Hz Real-Time Clock : Before Programming

1. Develop the Program Architecture

- Two software components - main program and timerISR – will share a **32-bit unsigned count variable**
- timerISR increments count every “tick”
- Main program reads count whenever it needs to
 - Will encapsulate in a subroutine : double getTicks()

Why 32-bit ?



20Hz Real-Time Clock : Before programming

2. Determine values for programming the 8253 Timer 0:

a) Write to the timer's **control register**: (at 43H)

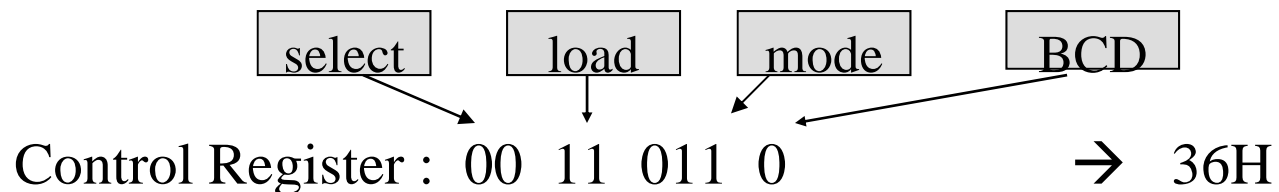
SC1 SC0 = **00** (select **timer 0**)

RL1 RL0 = **11** (**LS byte** then **MS byte**)

M2 M1 M0 = **011** (square wave): **mode 3**

– could also use M2 M1 M0 = 111

BCD = **0** (**16 bit** binary count)



20Hz Real-Time Clock : Before programming

b) Determine the initial (reload) counter value:

scaling factor = input freq. \div output freq.

= 1.19318 MHz \div 20 Hz

= 59,659 (decimal) = **0E90BH**

- After writing to control register, write **timer 0 counter value**:
(at 40H)
 - First write LSB: 0BH
 - Then write MSB: E9H

20Hz Real-Time Clock : Code Fragments

; Symbolic Definitions

; PIC registers and constants

```
PIC_COMMAND_REG    equ 20H ; Command register
PIC_IMR_REG        equ 21H ; Interrupt mask register
EOI                 equ 20H ; EOI to port 20h (Command Register)
```

; Timer registers and constants

```
TIMER_0_REG        equ 40H ; Timer 0 counter data register
TIMER_CTRL_REG     equ 43H ; Control register
TIMER_0_MODE       equ 36H ; = 0011 0110b
```

20Hz Real-Time Clock : Code Fragments

.data

; Program modifies state of system; need variables to save

; state to allow restoring of state upon exit

old_pic_imr db ?

old_int8_vector_offset dw ?

old_int8_vector_segment dw ?

; **32-bit count**: shared by ISR and main program

count_low dw ?

count_high dw ?

20Hz Real-Time Clock : Code Fragments

```
.code
```

```
MAIN PROC
```

```
; Initialize data structures used by ISR
```

```
    SUB  AX , AX                ; trick! AX = 0 !
```

```
    MOV  count_low , AX        ; tick count = 0
```

```
    MOV  count_high , AX
```

```
CLI    ; Disable interrupts while installing and setting up timer. Why ?
```

```
; Install Timer ISR at interrupt type = 8 (As before)
```

```
; Program timer 0 to interrupt at 20 Hz
```

```
    MOV  AL , TIMER_0_MODE      ; Control Register
```

```
    MOV  DX , TIMER_CTRL_REG
```

```
    OUT  DX , AL
```

```
    MOV  AL , 0BH             ; scaling factor = E90BH
```

```
    MOV  DX , TIMER_0_REG
```

```
    OUT  DX , AL                ; write low byte
```

```
    MOV  AL , 0E9H
```

```
    OUT  DX , AL                ; write high byte
```

20Hz Real-Time Clock : Code Fragments

; Enable **TIMER** interrupts at the PIC (As before)

```
MOV  DX , PIC_IMR_REG
```

```
IN   AL , DX
```

```
MOV  old_pic_imr, AL      ; for later restore
```

```
AND  AL , 0FEH          ; clear bit 0 of IMR
```

```
OUT  DX , AL
```

; Enable interrupts at the processor

```
STI
```

```
forever: ....           ; Main loop of program
```

```
CALL  get_ticks        ; Returns tick count in dx:ax
```

```
; Use value as needed
```

```
JNE  forever
```

```
exit :   ; Restore state before returning to DOS (not shown)
```

timerisr PROC FAR

STI

Event-driven thinking !

; Re-enable ints

PUSH DS

; Save EVERY register used by ISR

PUSH DX

PUSH AX

What are implications of this?
Is it really needed?

MOV AX , @data

MOV DS , AX

ADD count_low,1

JNC t1

INC count_high

event-driven thinking !

t1:

CLI

; Lock out all ints until IRET

MOV AL , EOI

; Send EOI to PIC

MOV DX , PIC_COMMAND_REG

OUT DX , AL

POP AX

; Restore registers

POP DX

POP DS

IRET

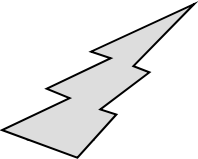
timerisr ENDP

How do interrupts get re-enabled? **STI**?

20Hz Real-Time Clock : Code Fragments

```
get_ticks PROC NEAR
```

```
    CLI ; Lock out ints while accessing shared data  
        ; (ensure mutual exclusion)
```



event-
driven
thinking !

```
    MOV DX , count_high ; Return 32 bit count  
    MOV AX , count_low
```



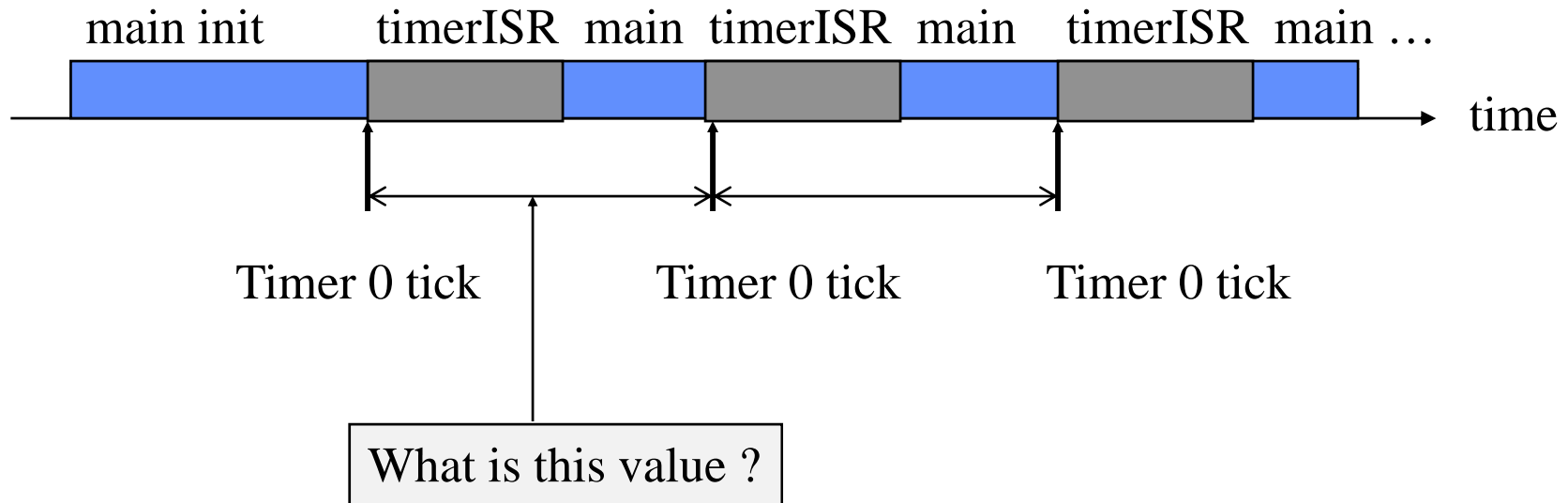
```
    STI ; Re-enable ints
```

```
    RET
```

```
get_ticks ENDP
```

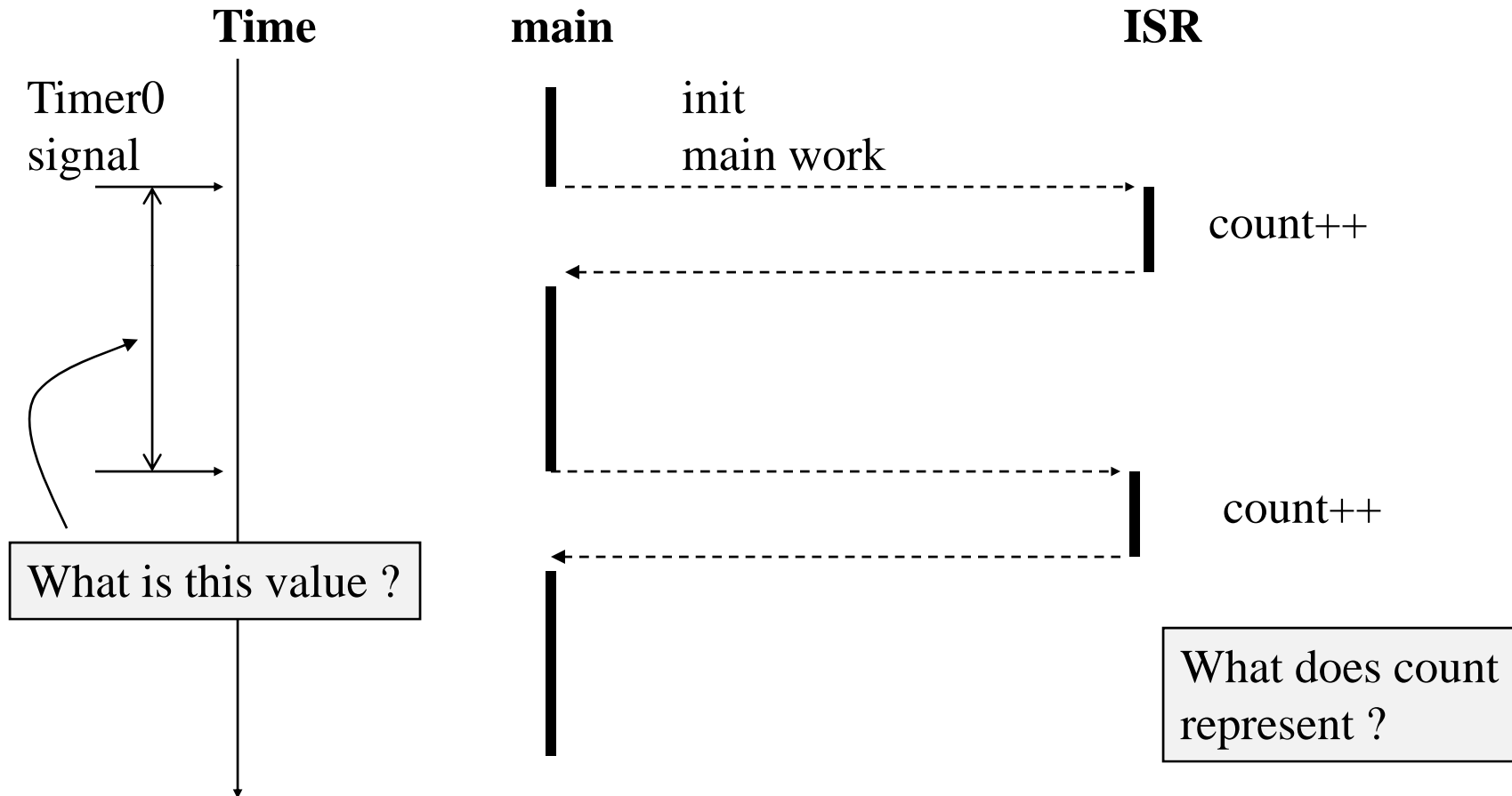
20Hz Real-Time Clock : A Timing Analysis

- CPU Utilization Diagram



20Hz Real-Time Clock : A Timing Analysis

- Thread Diagram



Event-Driven Thinking : An Interference Scenario

- Suppose **CLI / STI protection not there** , and:

count_high = 0010H

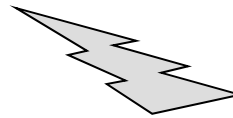
count_low = FFFFH

What's special about this value ?

- Suppose main program was executing getTicks() with the following:

```
MOV DX , count_high
```

```
MOV AX , count_low
```



- Suppose Timer interrupt occurs during /after `MOV DX , count_high`
 - At this point, `DX = 0010H`
 - For count to be correct, `AX = FFFFH`
- BUT ...

Event-Driven Thinking : An Interference Scenario

- In response to the interrupt, timerISR increments count, to become:
 - count_high = 0011H
 - count_low = 0000H
- When main program resumes, it executes MOV AX, count_low
(DX = 0010h, from before interrupt)
AX = 0000H (from after interrupt)
- The value returned from get_ticks() is:
 - count_high **before** Timer interrupt DX = 0010H
 - count_low **after** Timer interrupt AX = 0000H

I/O Program Metrics

- Definition : **Transfer Rate**
 - Number of bytes per second transferred between CPU and device
 - **Maximum** transfer rate characterizes the capability of the I/O program
- Definition : **Latency (Response time)**
 - **Delay** from the time the device is ready until the first data byte is exchanged.

Polled Input/Output

- CPU controls the synchronization with device
 - Execution is simple : sequential control flow
- Polled transfer rates are usually reasonable
 - Maximum transfer rate is the time to execute the check (once the device is ready) and then transfer
- Polled latency is usually unpredictable
 - Depends on the “other works” being done between checks
 - ... unless CPU works only on polling, in which case CPU is dedicated to I/O and cannot perform any other work.
 - What if other interesting events happen on other devices ?

Interrupt-Driven Input/Output

- Device signals CPU of a new I/O event
 - Event-driven programming
- In general, **interrupts improve the latency** of a system
 - Interrupt latency is deterministic
- Interrupt transfer rates includes **interrupt overhead**
 - Can be slower than polled
 - Depends If amount of data transferred **per** interrupt exceeds the **overhead** of switching tasks
- Advantage : CPU can be doing other work whenever there is no I/O event.
 - Software structure remains clean and task-oriented.
 - Although the time to complete the other work is marginally increased, depending on the number of interrupts.