Chapter 5.1: Interrupts

Microprocessors

What an interrupt is · Kinds of interrupts · Vector table · INT and IRET · Exceptions: INT 0, 1, 3, 4 · Hardware interrupts · 8259A and masking · Your own handler

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/microprocessors/slides/Chapter_05_01_Interrupts.html

Learning Outcomes

Concept

Explain what an interrupt is, how it differs from polling, and the kinds of interrupts.

Mechanism

Use the interrupt vector table to work out where `INT n` goes.

Stack

Trace how `INT` and `IRET` push and pop FLAGS, CS and IP.

Hardware interrupts

Follow INTR acknowledgment and masking with the 8259A; install your own handler and restore the old vector.

What Is an Interrupt?

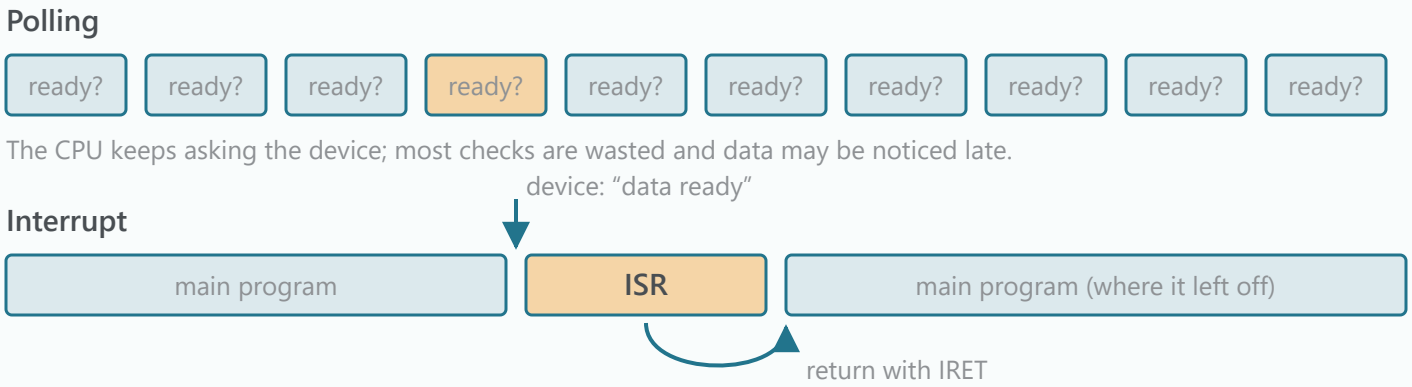
- An **interrupt** is an event that makes the processor set aside its current work and run a special routine.
- That routine is the **interrupt service routine** (**ISR**).
- When the ISR finishes, the processor resumes at the instruction where it left off; the main program never notices.
- Software interrupts are **ready-made services**: instead of writing code to print a character, you call an interrupt.
- There are interrupt functions for the screen, keyboard, disk drives, memory management and other hardware.
- Interrupts can also be triggered by **hardware** (a key press, the timer).

An everyday analogy

You are reading when the phone rings: you mark the page (save the return address), answer (the ISR), and after the call carry on from the marked page (IRET).



Polling or Interrupts?



Polling

- – Simple; no extra hardware.
- – Wastes CPU time.
- – An event may be noticed late, depending on the polling interval.

Interrupts

- + The CPU is busy only when an event occurs.
- + Fast response to events.
- + Needs a vector table, a stack and an ISR.

Kinds of Interrupts

Software interrupt

Called deliberately by an `INT n` instruction in the program. BIOS and DOS services are used this way. `INT 3` (breakpoint) and `INTO` (INT 4 if OF=1) are software interrupts too.

Hardware interrupt

An external device drives a pin. **INTR** is maskable (ignored while IF=0); **NMI** is non-maskable (e.g. memory parity error, power failure).

Internal interrupt (exception)

Generated by the processor itself: divide by zero / quotient overflow → **INT 0**; single step after every instruction while TF=1 → **INT 1**.

Type	Source	Description
`00h`	internal	Divide error (`DIV` / `IDIV` by zero, or the quotient does not fit)
`01h`	internal	Single step (TF = 1), used by debuggers
`02h`	hardware	NMI pin (non-maskable)
`03h`	software	Breakpoint; the one-byte `INT 3` (CCh)
`04h`	software	`INTO`: overflow interrupt if OF = 1
`05h–FFh`	—	Intel reserves 05h–1Fh; on the PC they are used by the BIOS, DOS and user programs

Comparing the Three Sources

	Software (INT n)	Hardware (INTR / NMI)	Internal (exception)
Triggered by	an instruction in the program	an external device, a pin	the processor itself
Timing	synchronous: always at the same point	asynchronous: after any instruction	synchronous: at the faulting instruction
Type number	in the instruction	INTR: from the 8259A, via INTA; NMI: always 2	fixed: 0, 1
While IF = 0	still runs	INTR waits, NMI runs	still runs
Typical use	BIOS/DOS services, INT 3, INTO	keyboard, timer, disk	divide error, single step

A common path

Once the type number is known, all three follow the same steps: FLAGS, CS, IP onto the stack; IF = TF = 0; CS:IP ← vector.

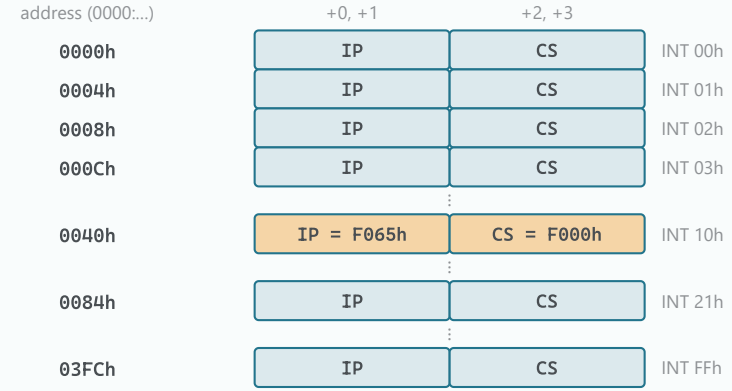
So what differs?

Only in where the type number comes from and when the interrupt occurs.

CLI disables only the INTR input; software interrupts and exceptions are not affected by IF.

The Interrupt Vector Table (IVT)

- The first **1 KB** of memory: `00000h–003FFh`.
- For each of the 256 interrupt types it holds a **4-byte vector** (the address of the ISR).
- The low word is **IP** (offset), the high word **CS** (segment); both little-endian.
- The vector for type *n* is at address **n × 4**.
- Changing a vector redirects the interrupt to another routine.



256 entries × 4 bytes = 1024 bytes (0000:0000 – 0000:03FF)

The F000:F065 shown for the INT 10h vector is the address of the video routine in the original IBM PC BIOS (an example).

Vector Address Step by Step

`INT 10h`

interrupt type n = 10h = 16

The program executes `INT 10h`. The type number is **n = 10h** (16).

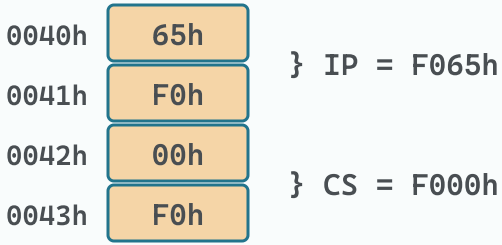
Vector Address Step by Step

$$10h \times 4 = 40h$$

16 × 4 = 64 → address 0000:0040h

Vector address **n × 4**: 10h × 4 = **40h**. (×4 is a shift left by two bits.)

Vector Address Step by Step



These 4 bytes are read: low word **IP = F065h**, high word **CS = F000h** (little-endian: low byte first).

Vector Address Step by Step

physical address = CS × 16 + IP

F0000h + F065h = FF065h

After the return information is pushed, CS:IP = F000:F065 is loaded; the ISR starts at physical address **FF065h**.

Demo: Vector Calculator

Enter an interrupt type (00h–FFh) to see its place in the table and its known role.

INT 21 h **compute** **00h** **10h** **16h** **FFh**

Calculation	`21h × 4 = 0084h` (33 × 4 = 132)
IP	`0000:0084h` and `0000:0085h`
CS	`0000:0086h` and `0000:0087h`
Role	DOS: general services (selected by AH)

`INT 21h` → `vector 0000:0084h`

The calculation never changes: `address = n × 4`. IP is at that address, CS at `n × 4 + 2`.

How Does INT n Work?

1. FLAGS is pushed: `SP ← SP − 2`, `[SS:SP] ← FLAGS`.

Step 1 / 6

A total of **6 bytes** (3 words) goes onto the stack. The old IF is saved with FLAGS, so IRET restores it too.

How Does INT n Work?

1. FLAGS is pushed: `SP ← SP − 2`, `[SS:SP] ← FLAGS`.

2. IF = 0 and **TF = 0**: the ISR is not interrupted by other maskable interrupts or by single-stepping.

A total of **6 bytes** (3 words) goes onto the stack. The old IF is saved with FLAGS, so IRET restores it too.

How Does INT n Work?

1. FLAGS is pushed: `SP ← SP − 2`, `[SS:SP] ← FLAGS`.

2. IF = 0 and **TF = 0**: the ISR is not interrupted by other maskable interrupts or by single-stepping.

3. CS is pushed.

A total of **6 bytes** (3 words) goes onto the stack. The old IF is saved with FLAGS, so IRET restores it too.

How Does INT n Work?

1. FLAGS is pushed: `SP ← SP − 2`, `[SS:SP] ← FLAGS`.

2. IF = 0 and **TF = 0**: the ISR is not interrupted by other maskable interrupts or by single-stepping.

3. CS is pushed.

4. IP (the address of the instruction *after* INT) is pushed.

A total of **6 bytes** (3 words) goes onto the stack. The old IF is saved with FLAGS, so IRET restores it too.

How Does INT n Work?

1. FLAGS is pushed: `SP ← SP − 2`, `[SS:SP] ← FLAGS`.
2. IF = 0 and **TF = 0**: the ISR is not interrupted by other maskable interrupts or by single-stepping.
3. CS is pushed.
4. IP (the address of the instruction *after* INT) is pushed.
5. `IP ← [n×4]`, `CS ← [n×4+2]`: the ISR begins.

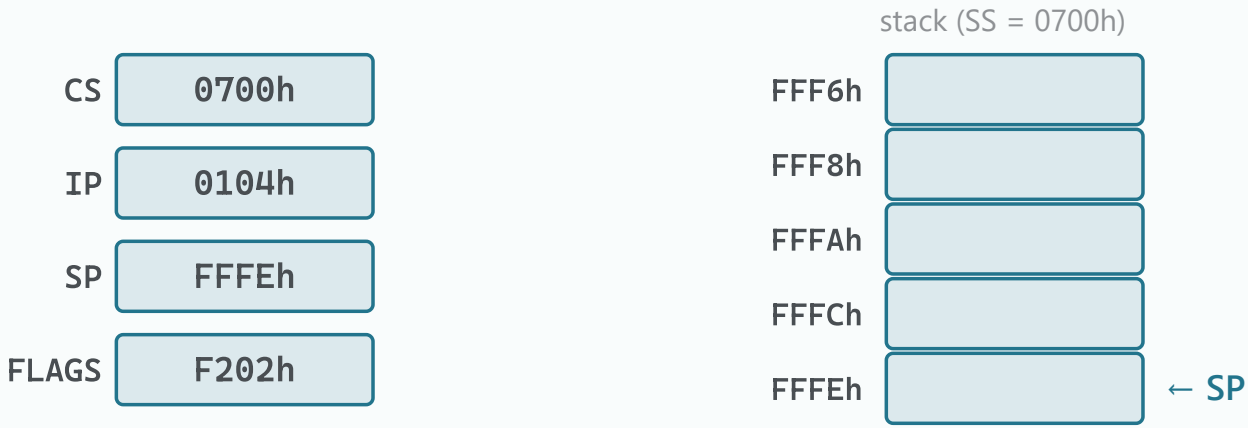
A total of **6 bytes** (3 words) goes onto the stack. The old IF is saved with FLAGS, so IRET restores it too.

How Does INT n Work?

1. FLAGS is pushed: `SP ← SP − 2`, `[SS:SP] ← FLAGS`.
2. IF = 0 and **TF = 0**: the ISR is not interrupted by other maskable interrupts or by single-stepping.
3. CS is pushed.
4. IP (the address of the instruction *after* INT) is pushed.
5. `IP ← [n×4]`, `CS ← [n×4+2]`: the ISR begins.
6. The ISR ends with **IRET**: IP, CS and FLAGS are popped in that order and the program resumes where it left off.

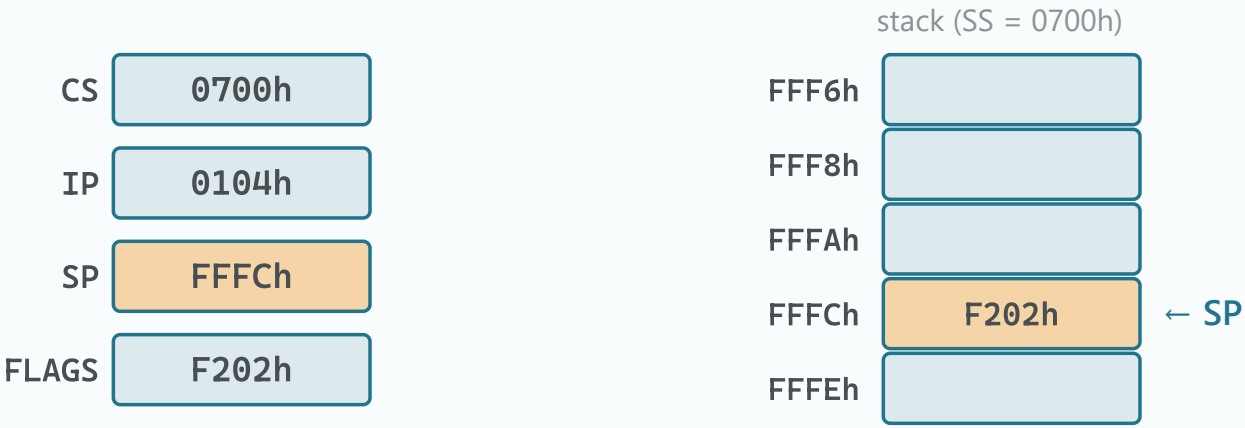
A total of **6 bytes** (3 words) goes onto the stack. The old IF is saved with FLAGS, so IRET restores it too.

INT and IRET: The Stack Step by Step



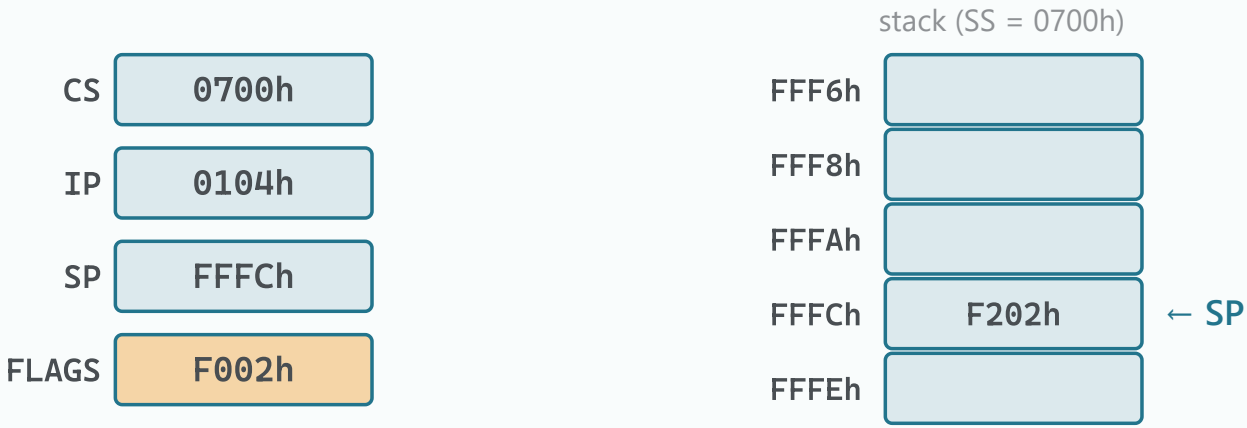
`INT 10h` at 0700:0104. SP = FFFEh, FLAGS = F202h (IF = 1; on the 8086 bit 1 and bits 12–15 always read 1).

INT and IRET: The Stack Step by Step



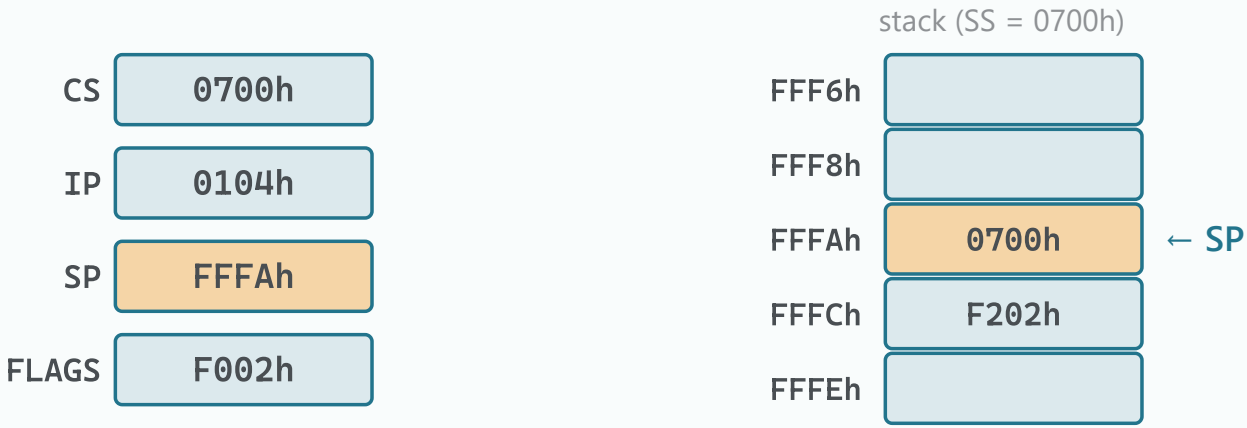
FLAGS is pushed: SP = FFFCh, [FFFCh] = F202h.

INT and IRET: The Stack Step by Step



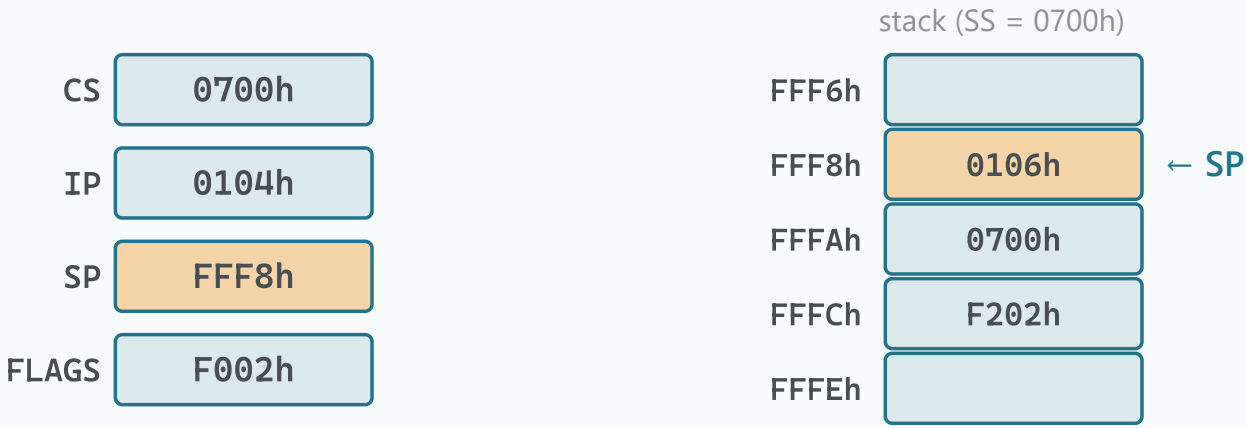
IF = 0, TF = 0 → FLAGS = F002h. The old value is safe on the stack.

INT and IRET: The Stack Step by Step



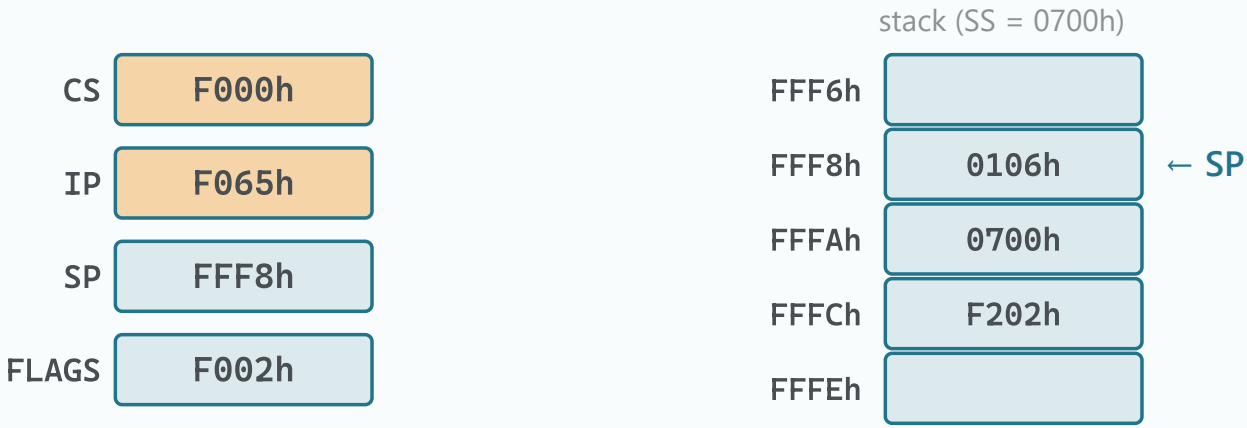
CS is pushed: SP = FFFAh, [FFFAh] = 0700h.

INT and IRET: The Stack Step by Step



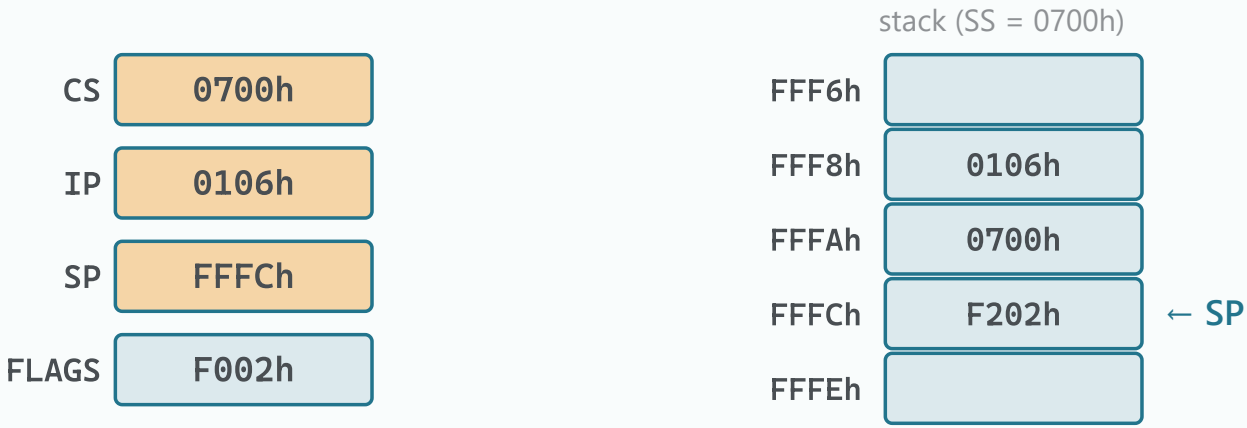
The return address is pushed: INT 10h is 2 bytes, so **0106h**. SP = FFF8h.

INT and IRET: The Stack Step by Step



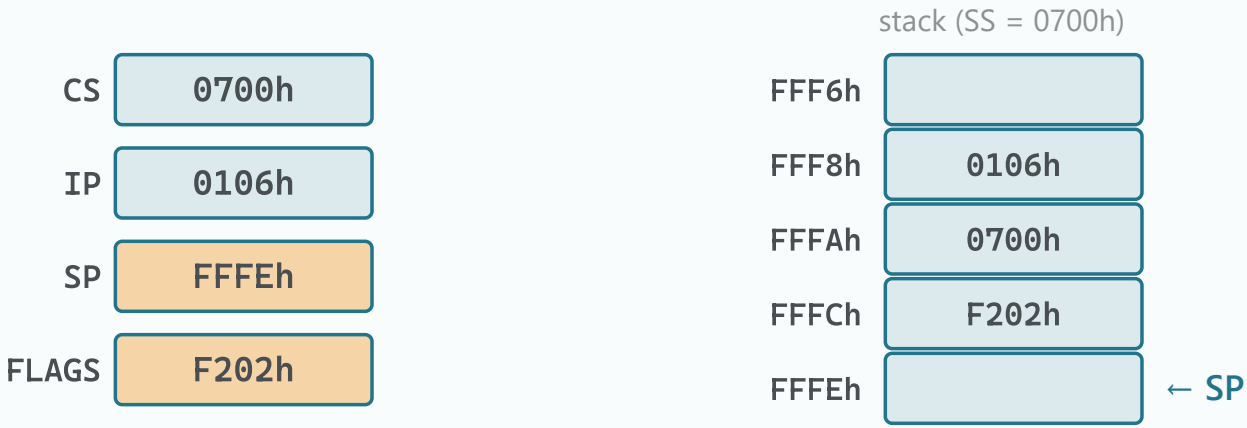
CS:IP ← vector (F000:F065). The ISR runs and prints the character.

INT and IRET: The Stack Step by Step



IRET: IP ← 0106h, CS ← 0700h (SP = FFFCh).

INT and IRET: The Stack Step by Step



FLAGS ← F202h, SP = FFFEh. The program continues at 0106h. (The old values on the stack are not erased; SP just moves back.)

INT/IRET vs. CALL/RET

	`CALL` / `RET`	`INT n` / `IRET`
Target address	in the instruction (a label)	from the vector table (n × 4)
Pushed	IP (near) or CS + IP (far)	FLAGS + CS + IP (6 bytes)
Flags	unchanged	IF = 0, TF = 0
Return	`RET` : pops IP (and CS)	`IRET` : pops IP, CS, FLAGS
Instruction length	3 bytes (near CALL)	2 bytes (`INT 3` : 1 byte)
Invoked by	only the program	the program, the processor (exception) or hardware

Ending an ISR with `RET` is a bug: a near RET pops only IP; CS and FLAGS stay on the stack (SP is off by 4 bytes) and IF is not restored. ISRs always end with **IRET**.

Code Trace: RET Instead of IRET

```
1 org 100h
▶ mov ax, 0
3 mov es, ax
4 mov word ptr es:[61h*4], offset isr
5 mov word ptr es:[61h*4+2], cs
6 int 61h
7 hlt                ; SP = ?
8 isr:
9 ret                ; BUG!
```



AX = 0: the IVT segment.

Step 1 / 7

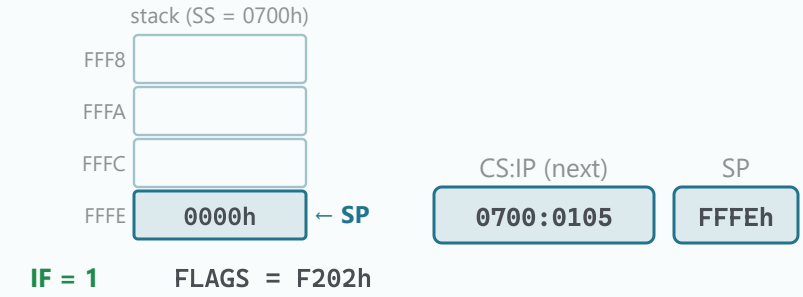
VARIABLES

`CS = 0700h` `IP = 0103h` `SP = FFFEh` `IF = 1`

Even RETF is not enough: it pops IP and CS, but FLAGS (2 bytes) stays on the stack and IF is not restored.

Code Trace: RET Instead of IRET

```
1 org 100h
2 mov ax, 0
▶ mov es, ax
4 mov word ptr es:[61h*4], offset isr
5 mov word ptr es:[61h*4+2], cs
6 int 61h
7 hlt                ; SP = ?
8 isr:
9 ret                ; BUG!
```



ES = 0000h.

Step 2 / 7

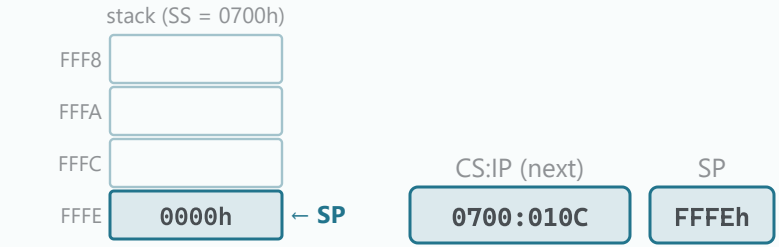
VARIABLES

CS = 0700h | IP = 0105h | SP = FFFEh | IF = 1

Even RETF is not enough: it pops IP and CS, but FLAGS (2 bytes) stays on the stack and IF is not restored.

Code Trace: RET Instead of IRET

```
1 org 100h
2 mov ax, 0
3 mov es, ax
▶ mov word ptr es:[61h*4], offset isr
5 mov word ptr es:[61h*4+2], cs
6 int 61h
7 hlt                ; SP = ?
8 isr:
9 ret                ; BUG!
```



IP of the type 61h vector: [184h] = `offset isr` = 0114h.

Step 3 / 7

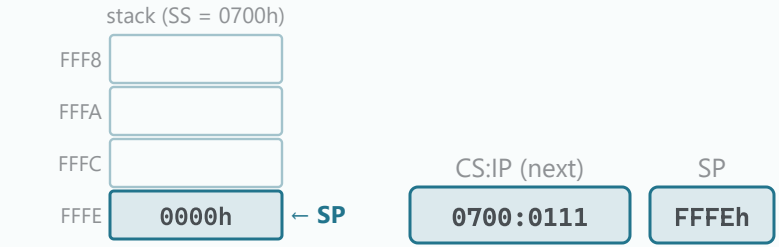
VARIABLES

`CS = 0700h` `IP = 010Ch` `SP = FFFEh` `IF = 1`

Even RETF is not enough: it pops IP and CS, but FLAGS (2 bytes) stays on the stack and IF is not restored.

Code Trace: RET Instead of IRET

```
1 org 100h
2 mov ax, 0
3 mov es, ax
4 mov word ptr es:[61h*4], offset isr
▶ mov word ptr es:[61h*4+2], cs
6 int 61h
7 hlt                ; SP = ?
8 isr:
9 ret                ; BUG!
```



CS: [186h] = 0700h. The vector is 0700:0114.

Step 4 / 7

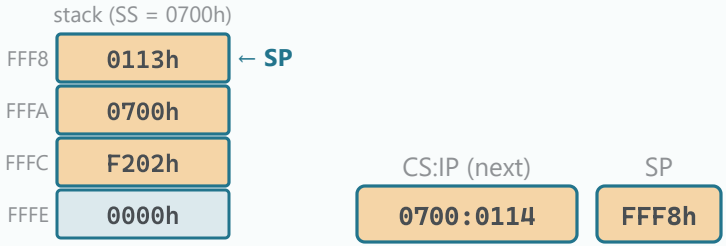
VARIABLES

CS = 0700h | IP = 0111h | SP = FFFEh | IF = 1

Even RETF is not enough: it pops IP and CS, but FLAGS (2 bytes) stays on the stack and IF is not restored.

Code Trace: RET Instead of IRET

```
1 org 100h
2 mov ax, 0
3 mov es, ax
4 mov word ptr es:[61h*4], offset isr
5 mov word ptr es:[61h*4+2], cs
▶ int 61h
7 hlt                ; SP = ?
8 isr:
9 ret                ; BUG!
```



IF = 0 FLAGS = F002h

`int 61h` : FLAGS (F202h), CS (0700h) and IP (0113h) are pushed: **6 bytes**; SP = FFF8h, IF = 0.

Step 5 / 7

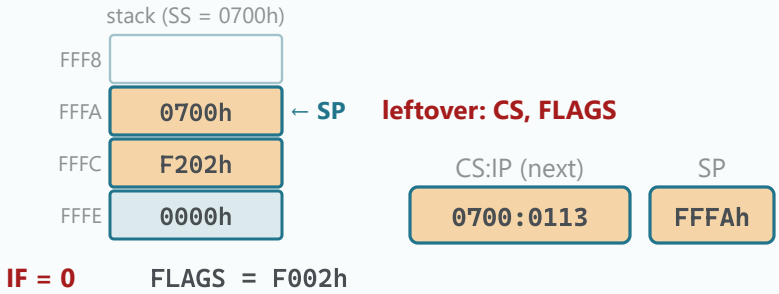
VARIABLES

CS = 0700h | IP = 0114h | SP = FFF8h | IF = 0

Even RETF is not enough: it pops IP and CS, but FLAGS (2 bytes) stays on the stack and IF is not restored.

Code Trace: RET Instead of IRET

```
1 org 100h
2 mov ax, 0
3 mov es, ax
4 mov word ptr es:[61h*4], offset isr
5 mov word ptr es:[61h*4+2], cs
6 int 61h
7 hlt                ; SP = ?
8 isr:
▶ ret                ; BUG!
```



The near `ret` in the ISR (it should be IRET) pops only **IP** (0113h): CS and FLAGS stay on the stack. SP = **FFFAh**, 4 bytes short of the expected FFFEh; IF stays 0.

Step 6 / 7

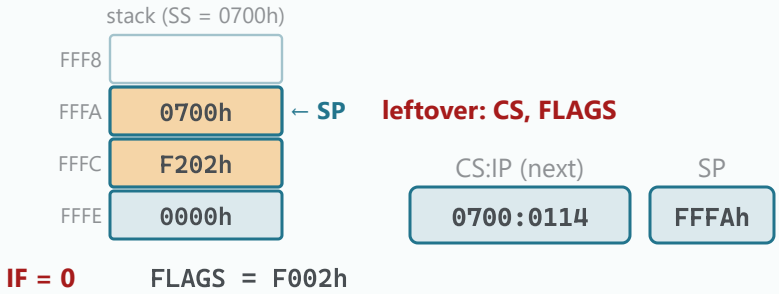
VARIABLES

`CS = 0700h` `IP = 0113h` `SP = FFFAh` `IF = 0`

Even RETF is not enough: it pops IP and CS, but FLAGS (2 bytes) stays on the stack and IF is not restored.

Code Trace: RET Instead of IRET

```
1 org 100h
2 mov ax, 0
3 mov es, ax
4 mov word ptr es:[61h*4], offset isr
5 mov word ptr es:[61h*4+2], cs
6 int 61h
▶ hlt                ; SP = ?
8 isr:
9 ret                ; BUG!
```



We reach `hlt`, but interrupts are off (IF = 0) and the stack is unbalanced: any later `ret` would take 0700h as its return address. An ISR always ends with **IRET**.

Step 7 / 7

VARIABLES

`CS = 0700h` `IP = 0114h` `SP = FFFAh` `IF = 0`

Even RETF is not enough: it pops IP and CS, but FLAGS (2 bytes) stays on the stack and IF is not restored.

Exceptions: Divide Error and INTO

INT 0: divide error

DIV or IDIV raises INT 0 in two cases: the divisor is 0, or the quotient does not fit in the destination. `AX = 1000`, `DIV BL` (BL = 2): the quotient 500 does not fit in AL (at most 255) → INT 0. The error occurs even though the divisor is not zero.

Return address

The 8086 pushes the address of the instruction **after** DIV; IRET resumes the program past the division. From the 80286 on, the faulting instruction's own address is pushed (so it can be retried).

INTO

A one-byte instruction (CEh). Calls INT 4 if OF = 1 and does nothing if OF = 0. Place it right after a signed add or subtract: `ADD AL, BL` · `INTO`.

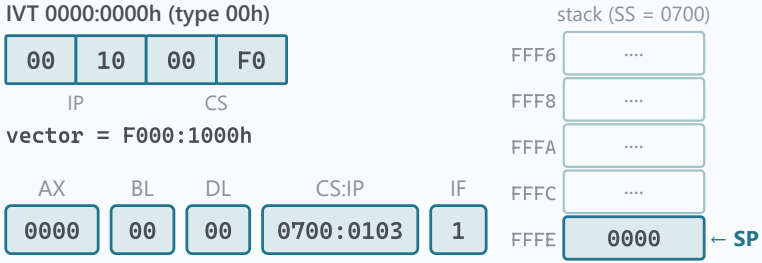
What does DOS do?

DOS installs its own routine in the INT 0 vector: it prints "Divide overflow" and terminates the program. A program can change this by installing its own handler.

Checking before a division that the divisor is nonzero and the quotient will fit (e.g. using DX:AX for a 16-bit divide) prevents INT 0 in the first place.

Code Trace: Our Own INT 0 Handler

```
1  org 100h
▶ mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



AX = 0: the IVT segment.

Step 1 / 11

VARIABLES

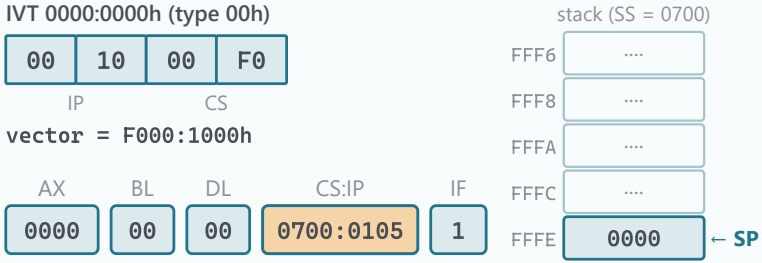
AX = **0000h** BX = **0000h** DX = **0000h**

SP = **FFFEh** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
▶  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



ES = 0000h: ES:[0] is the vector of type 0 (0 × 4 = 0).

Step 2 / 11

VARIABLES

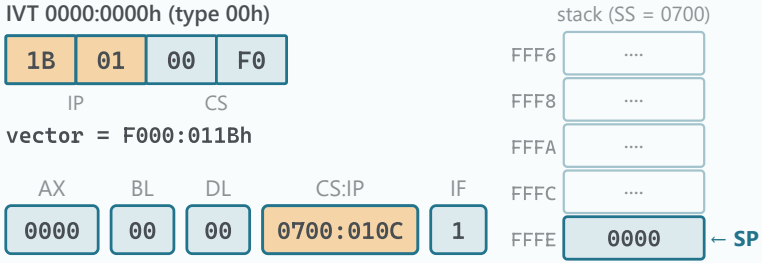
AX = **0000h** BX = **0000h** DX = **0000h**

SP = **FFFEh** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
▶  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



Low word (IP) = `offset div_err` = **011Bh**. The previous vector pointed to the emulator's default routine.

Step 3 / 11

VARIABLES

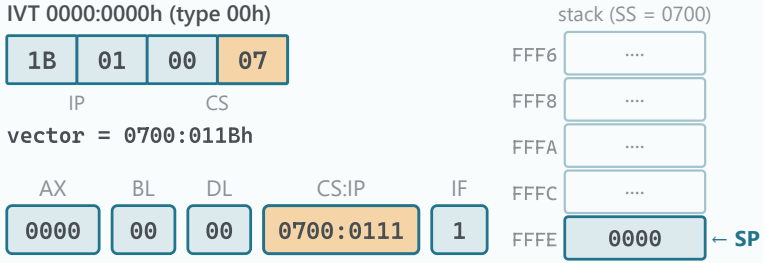
AX = **0000h** BX = **0000h** DX = **0000h**

SP = **FFFEh** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
▶  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



High word (CS) = 0700h. Type 0 now goes to our handler.

Step 4 / 11

VARIABLES

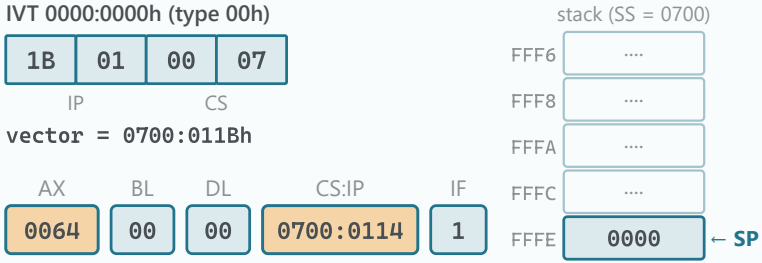
AX = **0000h** BX = **0000h** DX = **0000h**

SP = **FFFEh** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
▶  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



AX = 100 = 0064h: the dividend.

Step 5 / 11

VARIABLES

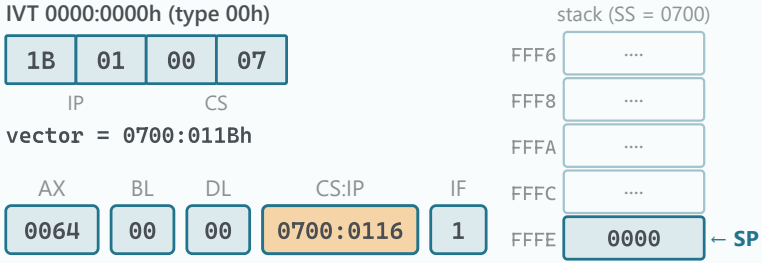
AX = **0064h** BX = **0000h** DX = **0000h**

SP = **FFFEh** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
▶  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



BL = 0: the divisor is zero.

Step 6 / 11

VARIABLES

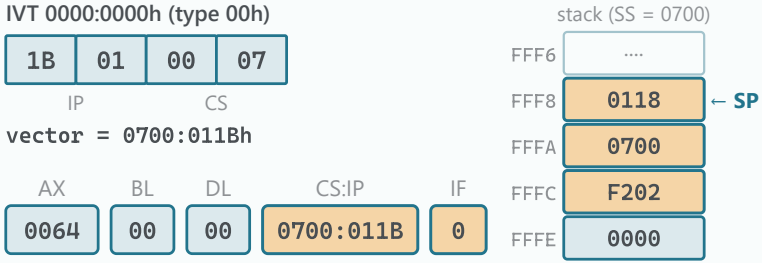
AX = **0064h** BX = **0000h** DX = **0000h**

SP = **FFFEh** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
▶  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



`div bl` → **INT 0**. FLAGS, CS and the return IP **0118h** (the instruction after DIV) are pushed; SP = FFF8h, IF = 0.

Step 7 / 11

VARIABLES

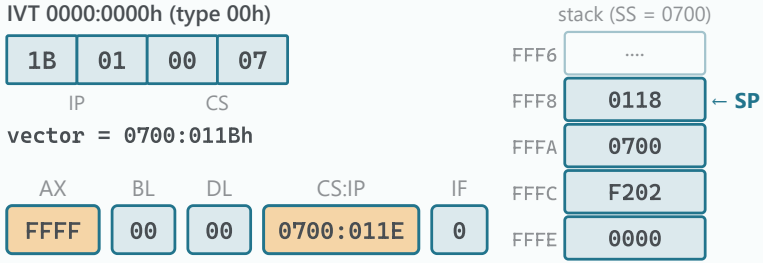
AX = **0064h** BX = **0000h** DX = **0000h**

SP = **FFF8h** IF = **0**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
▶ mov ax, 0FFFFh          ; error flag
13 iret
```



The handler runs: AX = **FFFFh**, telling the program “result invalid”.

Step 8 / 11

VARIABLES

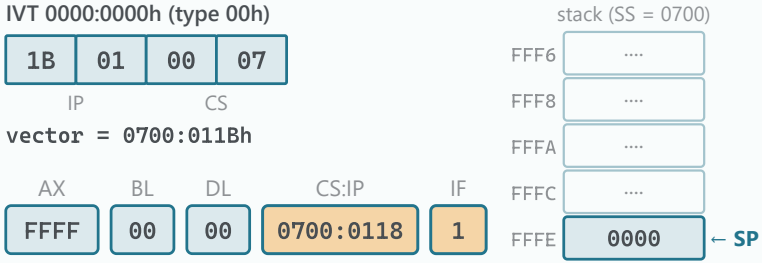
AX = **FFFFh** BX = **0000h** DX = **0000h**

SP = **FFF8h** IF = **0**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
▶  iret
```



`iret`: IP = 0118h, CS and FLAGS are restored; SP = FFFEh, IF = 1. DIV is not run again.

Step 9 / 11

VARIABLES

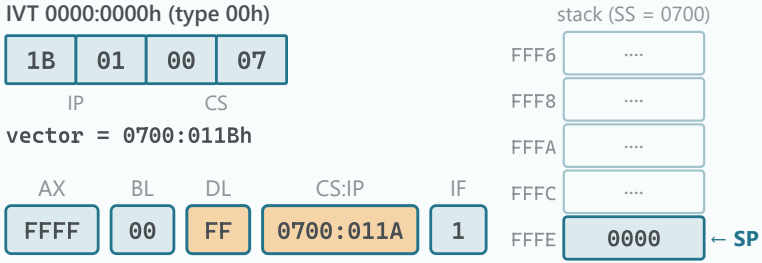
AX = **FFFFh** BX = **0000h** DX = **0000h**

SP = **FFFEh** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
▶  mov dl, al              ; returns here
10 ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



DL = AL = **FFh**: the main program sees the error from the flag value.

Step 10 / 11

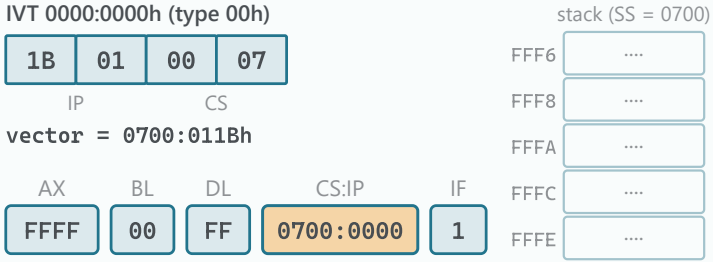
VARIABLES

AX = **FFFFh** · BX = **0000h** · DX = **00FFh** · SP = **FFFEh** · IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own INT 0 Handler

```
1  org 100h
2  mov ax, 0
3  mov es, ax              ; ES = 0: IVT
4  mov word ptr es:[0*4], offset div_err
5  mov word ptr es:[0*4+2], cs
6  mov ax, 100
7  mov bl, 0
8  div bl                  ; INT 0!
9  mov dl, al              ; returns here
▶  ret
11 div_err:
12 mov ax, 0FFFFh          ; error flag
13 iret
```



`ret` : the program returns to DOS.

Step 11 / 11

VARIABLES

AX = **FFFFh** BX = **0000h** DX = **00FFh**

SP = **0000h** IF = **1**

For brevity the old vector is not saved; a real program restores it before exiting. Values assume an emu8086-style .COM load (CS = 0700h).

A Debugger's Tools: INT 1 and INT 3

Single step (TF): while TF = 1 the processor raises INT 1 after every instruction. No instruction sets TF directly; FLAGS is edited through the stack:

```
PUSHF
POP   AX
OR    AH, 01h       ; bit 8 = TF
PUSH  AX
POPF                ; TF = 1
```

The INT 1 handler runs with TF = 0 (INT clears TF), so its own instructions are not single-stepped. IRET restores the TF = 1 saved on the stack.

Breakpoint (INT 3) has a special one-byte encoding: **CCh**. The debugger:

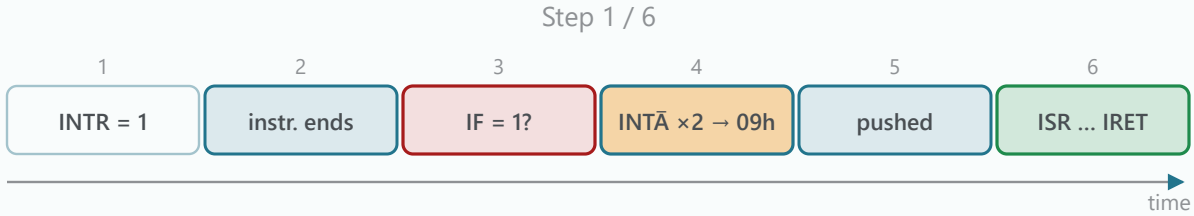
1. Saves the first byte of the instruction to stop at and writes CCh in its place.
2. When the program gets there, INT 3 runs and the registers are shown.
3. Writes the saved byte back and decrements the return IP on the stack by 1.
4. Re-executes the instruction with IRET.

Being one byte, it fits in place of even the shortest instruction; `INT 03h` would be the two-byte CD 03.

emu8086's "single step" button does the same inside the emulator; on a real processor tools such as DEBUG use the INT 1 and INT 3 vectors.

How INTR Is Acknowledged

1. A device (e.g. the keyboard) activates IRQ1; the 8259A drives the **INTR** pin high.

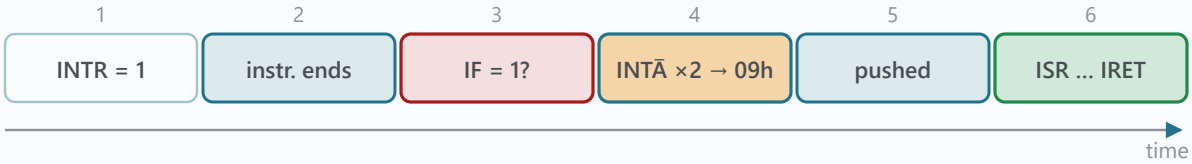


How INTR Is Acknowledged

1. A device (e.g. the keyboard) activates IRQ1; the 8259A drives the **INTR** pin high.

2. The 8086 **finishes** the current instruction; interrupts are never accepted mid-instruction.

Step 2 / 6



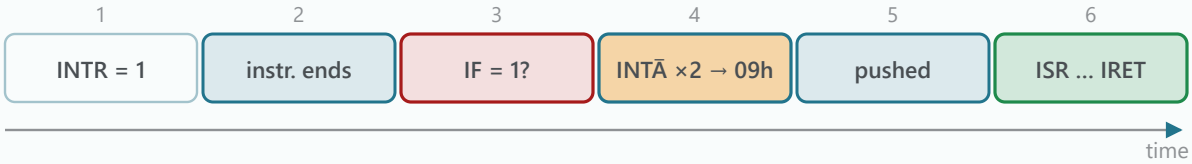
How INTR Is Acknowledged

1. A device (e.g. the keyboard) activates IRQ1; the 8259A drives the **INTR** pin high.

2. The 8086 **finishes** the current instruction; interrupts are never accepted mid-instruction.

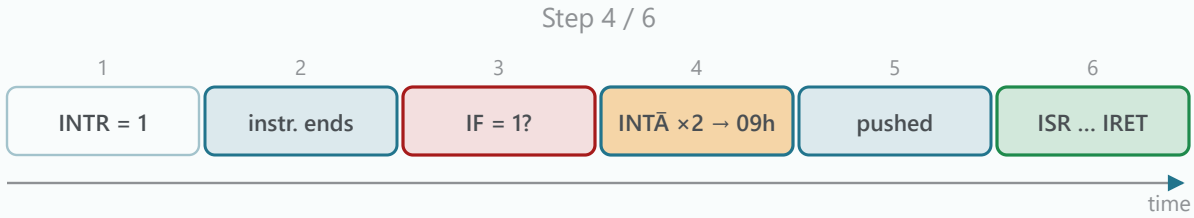
3. If IF = 0 the request waits. If IF = 1 it is accepted.

Step 3 / 6



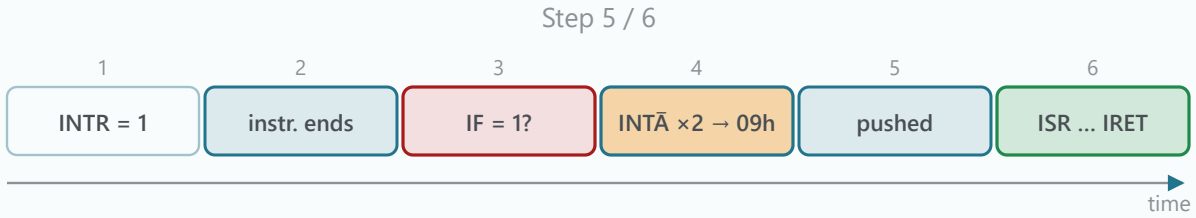
How INTR Is Acknowledged

1. A device (e.g. the keyboard) activates IRQ1; the 8259A drives the **INTR** pin high.
2. The 8086 **finishes** the current instruction; interrupts are never accepted mid-instruction.
3. If IF = 0 the request waits. If IF = 1 it is accepted.
4. The CPU issues two **INTĀ** pulses; on the second the 8259A places the type number on the data bus (`09h` for the keyboard on a PC).



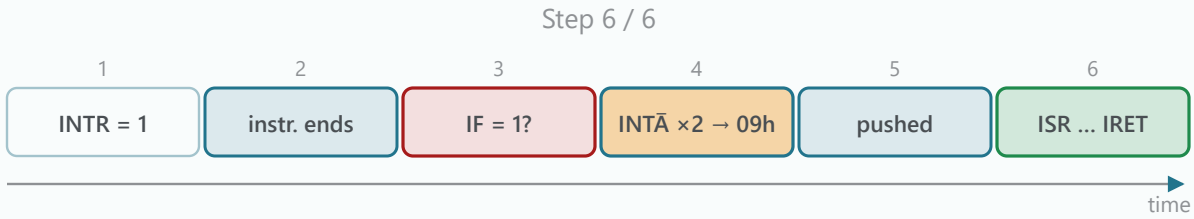
How INTR Is Acknowledged

1. A device (e.g. the keyboard) activates IRQ1; the 8259A drives the **INTR** pin high.
2. The 8086 **finishes** the current instruction; interrupts are never accepted mid-instruction.
3. If IF = 0 the request waits. If IF = 1 it is accepted.
4. The CPU issues two **INTĀ** pulses; on the second the 8259A places the type number on the data bus (`09h` for the keyboard on a PC).
5. From here on it is the same as `INT n`: FLAGS, CS, IP onto the stack; IF = TF = 0; CS:IP ← [n×4].



How INTR Is Acknowledged

1. A device (e.g. the keyboard) activates IRQ1; the 8259A drives the **INTR** pin high.
2. The 8086 **finishes** the current instruction; interrupts are never accepted mid-instruction.
3. If IF = 0 the request waits. If IF = 1 it is accepted.
4. The CPU issues two **INTĀ** pulses; on the second the 8259A places the type number on the data bus (`09h` for the keyboard on a PC).
5. From here on it is the same as `INT n`: FLAGS, CS, IP onto the stack; IF = TF = 0; CS:IP ← [n×4].
6. When the ISR is done it sends the 8259A an **EOI** (end of interrupt) and returns with `IRET`.



Priority and the PC Interrupt Map

8086 priority order

Rank	Source
1 (highest)	Divide error, `INT n`, `INTO`
2	NMI
3	INTR
4 (lowest)	Single step (TF)

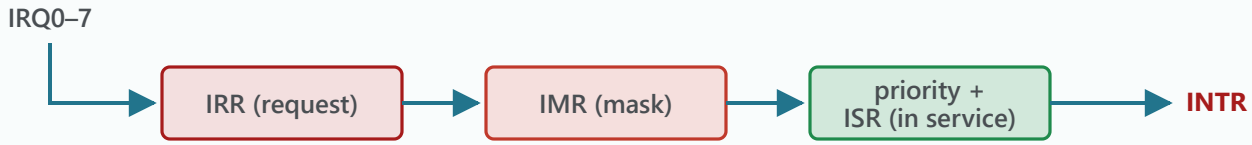
When several requests arrive at once, the highest priority is served first.

IBM PC hardware interrupts

IRQ	Device	Type
0	Timer (18.2 Hz)	`08h`
1	Keyboard	`09h`
3 / 4	Serial ports	`0Bh / 0Ch`
6	Floppy disk	`0Eh`
7	Printer	`0Fh`

A key press first runs the hardware interrupt `INT 09h`, which puts the key in a buffer; the `INT 16h` a program calls reads from that buffer.

Three Registers Inside the 8259A



INTA: the 8259A puts the type number (base + IRQ no.) on the data bus

IRR

Holds requests that have arrived but not yet been served.

IMR

Interrupt mask register: one bit per IRQ. IF disables all of INTR; the IMR disables individual lines.

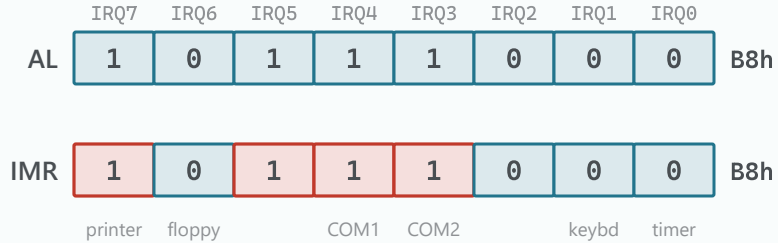
ISR (register)

Marks the IRQ being served; lower priorities wait until an EOI arrives.

Cascading eight slaves on one master gives 64 IRQs; the PC/AT connects a second chip to IRQ2 (IRQ8–15, types 70h–77h). Do not confuse the in-service register ISR with the interrupt service routine.

Code Trace: Masking a Single IRQ

```
1 org 100h
▶ in al, 21h            ; read IMR
3 or al, 00000010b      ; mask IRQ1
4 out 21h, al
5 mov cx, 3             ; critical
  work
6 in al, 21h
7 and al, 11111101b     ; unmask IRQ1
8 out 21h, al
9 ret
```



red = masked (1) · port 21h = IMR

`in al, 21h` : the IMR is read, AL = **B8h** = 1011 1000b. IRQ3, 4, 5, 7 are already masked (sample value).

Step 1 / 8

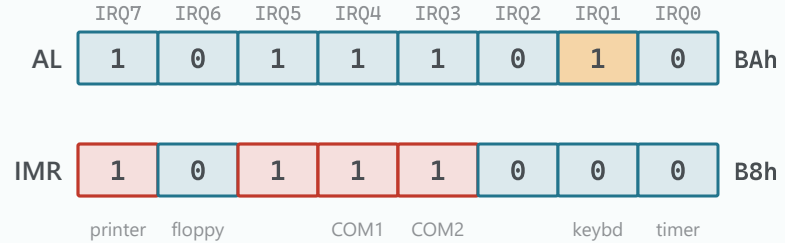
VARIABLES

`AL = B8h` `CX = 0000h` `IMR = B8h`

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

Code Trace: Masking a Single IRQ

```
1 org 100h
2 in al, 21h            ; read IMR
▶ or al, 00000010b      ; mask IRQ1
4 out 21h, al
5 mov cx, 3             ; critical
  work
6 in al, 21h
7 and al, 11111101b     ; unmask IRQ1
8 out 21h, al
9 ret
```



red = masked (1) · port 21h = IMR

`or al, 02h` : only bit 1 is set → AL = **BAh**. The other bits are kept; nothing is written to the IMR yet.

Step 2 / 8

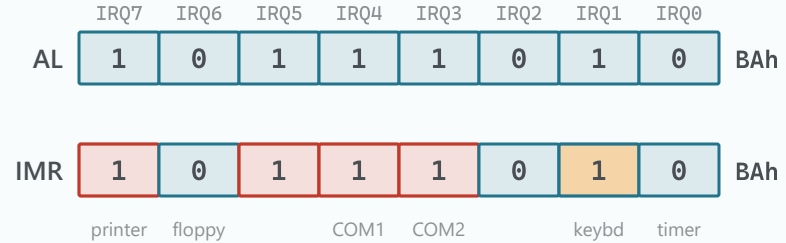
VARIABLES

`AL = BAh` `CX = 0000h` `IMR = B8h`

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

Code Trace: Masking a Single IRQ

```
1 org 100h
2 in al, 21h            ; read IMR
3 or al, 00000010b      ; mask IRQ1
▶ out 21h, al
5 mov cx, 3             ; critical
  work
6 in al, 21h
7 and al, 11111101b     ; unmask IRQ1
8 out 21h, al
9 ret
```



red = masked (1) · port 21h = IMR

`out 21h, al` : IMR = BAh. **IRQ1 (keyboard)** can no longer reach INTR; IF and the other IRQs are unaffected.

Step 3 / 8

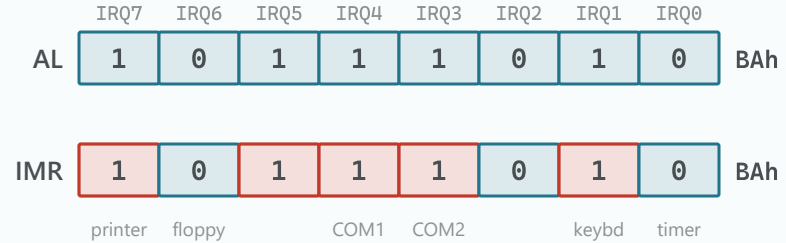
VARIABLES

`AL = BAh` `CX = 0000h` `IMR = BAh`

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

Code Trace: Masking a Single IRQ

```
1 org 100h
2 in al, 21h            ; read IMR
3 or al, 00000010b      ; mask IRQ1
4 out 21h, al
▶ mov cx, 3             ; critical
  work
6 in al, 21h
7 and al, 11111101b     ; unmask IRQ1
8 out 21h, al
9 ret
```



red = masked (1) · port 21h = IMR

Work that must not be interrupted (just a placeholder here). A key press waits in the IRR; it is not lost.

Step 4 / 8

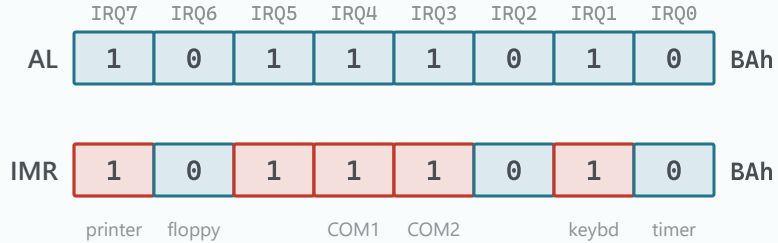
VARIABLES

`AL = BAh` `CX = 0003h` `IMR = BAh`

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

Code Trace: Masking a Single IRQ

```
1 org 100h
2 in al, 21h            ; read IMR
3 or al, 00000010b      ; mask IRQ1
4 out 21h, al
5 mov cx, 3             ; critical
  work
▶ in al, 21h
7 and al, 11111101b     ; unmask IRQ1
8 out 21h, al
9 ret
```



red = masked (1) · port 21h = IMR

The IMR is read again: AL = BAh.

Step 5 / 8

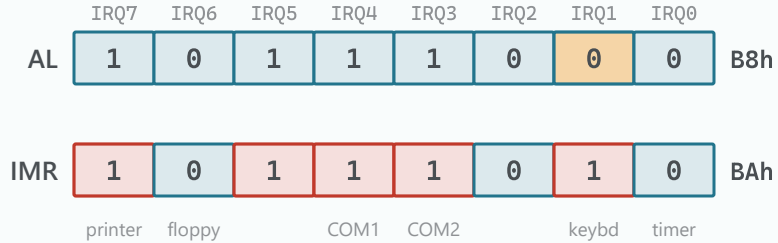
VARIABLES

`AL = BAh` `CX = 0003h` `IMR = BAh`

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

Code Trace: Masking a Single IRQ

```
1 org 100h
2 in al, 21h            ; read IMR
3 or al, 00000010b      ; mask IRQ1
4 out 21h, al
5 mov cx, 3             ; critical
  work
6 in al, 21h
▶ and al, 11111101b     ; unmask IRQ1
8 out 21h, al
9 ret
```



red = masked (1) · port 21h = IMR

`and al, 0FDh` (1111 1101b): only bit 1 is cleared → AL = **B8h**.

Step 6 / 8

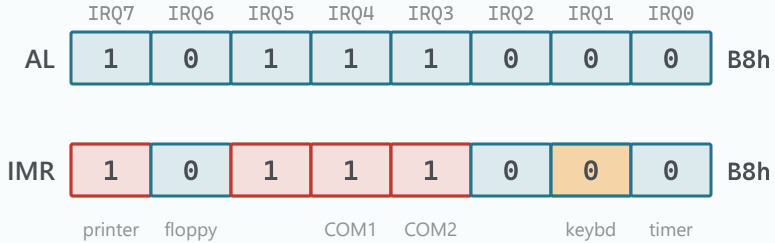
VARIABLES

AL = **B8h** | CX = **0003h** | IMR = **BAh**

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

Code Trace: Masking a Single IRQ

```
1 org 100h
2 in al, 21h            ; read IMR
3 or al, 00000010b      ; mask IRQ1
4 out 21h, al
5 mov cx, 3             ; critical
  work
6 in al, 21h
7 and al, 11111101b     ; unmask IRQ1
▶ out 21h, al
9 ret
```



red = masked (1) · port 21h = IMR

`out 21h, al` : IMR = B8h, IRQ1 is enabled; the pending keyboard request is served now.

Step 7 / 8

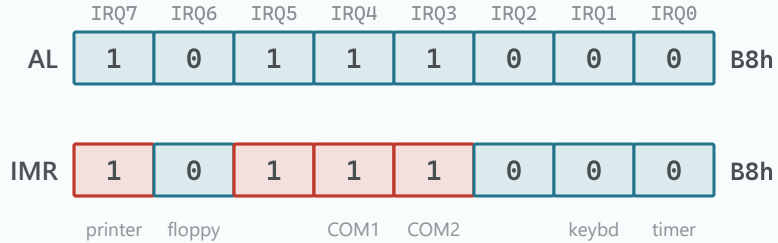
VARIABLES

`AL = B8h` `CX = 0003h` `IMR = B8h`

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

Code Trace: Masking a Single IRQ

```
1 org 100h
2 in al, 21h            ; read IMR
3 or al, 00000010b      ; mask IRQ1
4 out 21h, al
5 mov cx, 3             ; critical
  work
6 in al, 21h
7 and al, 11111101b     ; unmask IRQ1
8 out 21h, al
▶ ret
```



red = masked (1) · port 21h = IMR

`ret` : the mask is back to its original value. No one else's bits were touched.

Step 8 / 8

VARIABLES

`AL = B8h` `CX = 0003h` `IMR = B8h`

Read–modify–write: OR to mask, AND to unmask, so the other bits are preserved. Unlike CLI, only the chosen device is held off.

When Is INTR Not Accepted?

Mid-instruction

The request waits until the current instruction finishes. Prefixes (LOCK, REP, segment override) count as part of the instruction; no interrupt is accepted between a prefix and its instruction.

After loading SS, and after STI

After `MOV SS, AX` or `POP SS`, no interrupt arrives until the next instruction also finishes, so `MOV SP, …` completes safely. STI also lets one more instruction run; no interrupt can get between `sti` / `hlt` or `sti` / `iret`.

String instructions with REP

A long `REP MOVSB` can be interrupted between iterations; the return address points to the prefix, and after IRET it continues for the remaining CX. The 8086 remembers only one prefix: if `REP ES: MOVSB` is interrupted the ES: prefix is lost, so wrap such a line in CLI/STI.

HLT and WAIT

`HLT` stops the processor; an INTR or NMI wakes it up and IRET returns to the instruction after HLT. An interrupt arriving during `WAIT` is also accepted.

In short, the 8086 accepts interrupts only at instruction boundaries; in a few cases that boundary is deliberately pushed one instruction later.

Interrupt Latency

Operation (8086)	Clocks
`INT n`	51
`INT 3`	52
`INTO` (OF = 1 / OF = 0)	53 / 4
INTR acknowledge (both INTAs)	61
`IRET`	24
16-bit `IDIV` (register)	165–184

Interrupt latency: the time from the request to the start of the ISR's first instruction.

- In the worst case the current instruction must finish first: a long IDIV ≈ 184 clocks.
- Then the acknowledge: 61 clocks. Total ≈ 245 clocks.
- At 5 MHz one clock is 200 ns → ≈ **49 µs**.
- Time spent with interrupts off (CLI) adds directly to this latency; keep CLI–STI sections short.

For a timer interrupt arriving 18.2 times a second this is negligible; for fast devices such as a high-speed serial port, a short ISR prevents data loss.

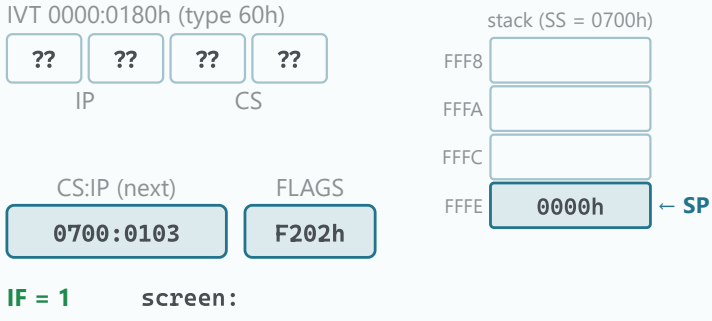
Our Own Interrupt Handler

```
org 100h
mov ax, 0
mov es, ax          ; ES = 0: IVT segment
cli                 ; interrupts off
mov word ptr es:[90h*4], offset isr  ; IP
mov word ptr es:[90h*4+2], cs        ; CS
sti                 ; interrupts on
int 90h             ; our interrupt
ret
isr:
mov ah, 0Eh
mov al, '*'
int 10h
iret                ; not RET!
```

- Pick an unused type: **90h**. Its vector is at `90h × 4 = 240h`.
- The IVT is in segment 0000; set ES = 0.
- Protect the vector write with `CLI` / `STI`: no interrupt may arrive while the vector is half-written.
- `INT 90h` prints * and the program continues.
- Under DOS, `INT 21h` with AH = `25h` (set vector) / `35h` (get vector) is the safer way.

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
▶ 2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```



AX = 0. The interrupt vector table (IVT) is at 0000:0000; the vector for type 60h is at **60h × 4 = 180h**.

Step 1 / 12

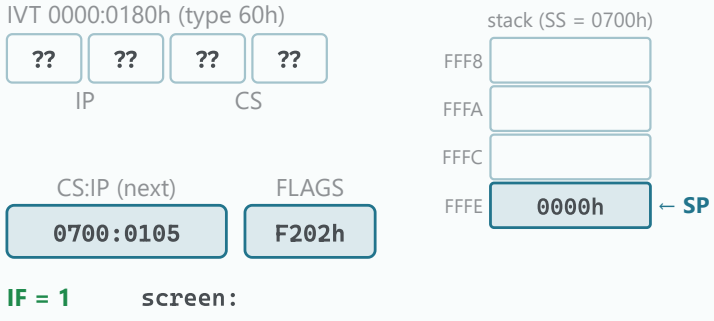
VARIABLES

`AX = 0000h` `ES = 0700h` `SP = FFFEh` `FLAGS = F202h` `IF = 1`

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
▶  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```



ES = 0000h: ES:[180h] now refers to byte 0000:0180h of the IVT.

Step 2 / 12

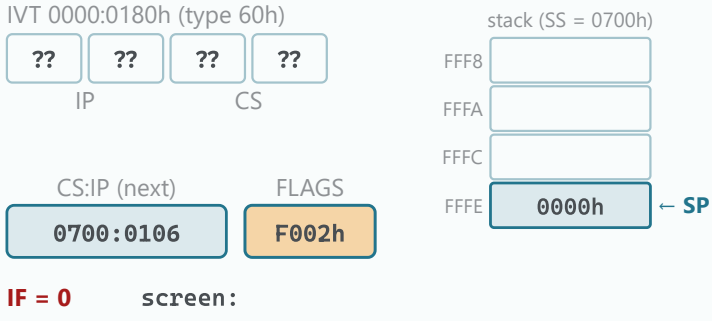
VARIABLES

AX = 0000h | ES = 0000h | SP = FFFEh | FLAGS = F202h | IF = 1

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
▶  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```



`cli` : IF = 0. No hardware interrupt may arrive while the vector is half-written. FLAGS = F002h.

Step 3 / 12

VARIABLES

AX = **0000h** · ES = **0000h** · SP = **FFFEh** · FLAGS = **F002h** · IF = **0**

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
▶  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```



Low word = IP = `offset isr` = **0118h**: 18 01 in memory (little-endian).

Step 4 / 12

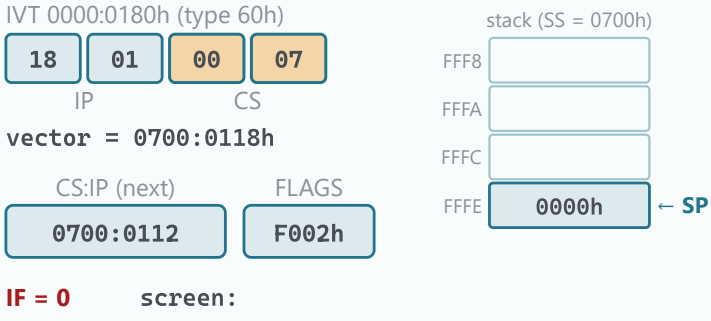
VARIABLES

AX = **0000h** · ES = **0000h** · SP = **FFFEh** · FLAGS = **F002h** · IF = **0**

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
▶  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```



High word = CS = **0700h**: 00 07. The vector is now **0700:0118**.

Step 5 / 12

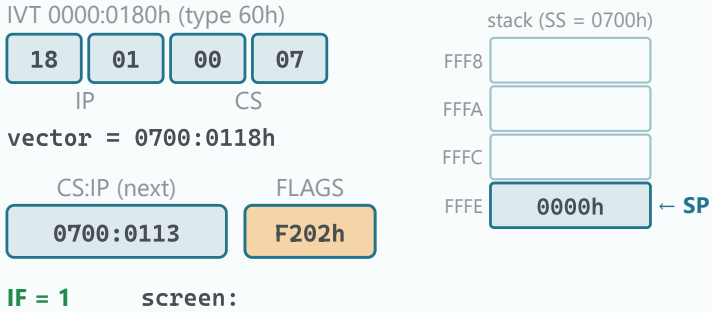
VARIABLES

`AX = 0000h` `ES = 0000h` `SP = FFFEh` `FLAGS = F002h` `IF = 0`

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
▶  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```



`sti` : IF = 1, interrupts are enabled again. FLAGS = **F202h** (on the 8086 bit 1 and bits 12–15 always read 1).

Step 6 / 12

VARIABLES

AX = **0000h** ES = **0000h** SP = **FFFEh** FLAGS = **F202h** IF = **1**

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
▶  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```

IVT 0000:0180h (type 60h)

18	01	00	07
IP		CS	

vector = 0700:0118h

CS:IP (next)	FLAGS
0700:0115	F202h

IF = 1 screen:

stack (SS = 0700h)

FFF8	
FFFA	
FFFC	
FFFE	0000h ← SP

AL = '#' (23h): a parameter passed to the ISR in a register.

Step 7 / 12

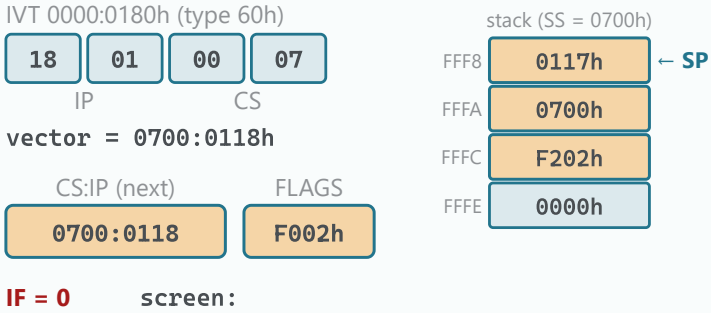
VARIABLES

AX = 0023h | ES = 0000h | SP = FFFEh | FLAGS = F202h | IF = 1

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
▶ int 60h                ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```



`int 60h` : FLAGS (F202h), CS (0700h) and the return IP (**0117h**) are pushed; IF = 0; CS:IP ← [180h] = 0700:0118.

Step 8 / 12

VARIABLES

AX = **0023h** ES = **0000h** SP = **FFF8h**

FLAGS = **F002h** IF = **0**

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
▶ mov ah, 0Eh
13 int 10h
14 iret
```

IVT 0000:0180h (type 60h)

18	01	00	07
IP		CS	

vector = 0700:0118h

CS:IP (next)	FLAGS
0700:011A	F002h

stack (SS = 0700h)

FFF8	0117h	← SP
FFFA	0700h	
FFFC	F202h	
FFFE	0000h	

IF = 0 screen: **inside the ISR**

The ISR runs: AH = 0Eh (teletype).

Step 9 / 12

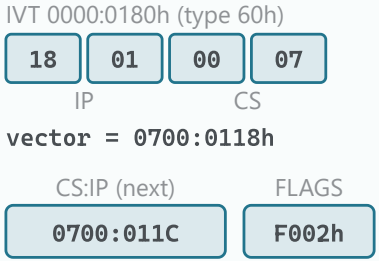
VARIABLES

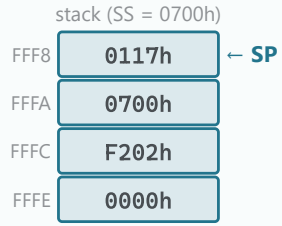
AX = 0E23h | ES = 0000h | SP = FFF8h | FLAGS = F002h | IF = 0

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
▶ int 10h
14 iret
```





IF = 0 `screen: #` **inside the ISR**

`int 10h` : # is printed. An ISR may call another interrupt.

Step 10 / 12

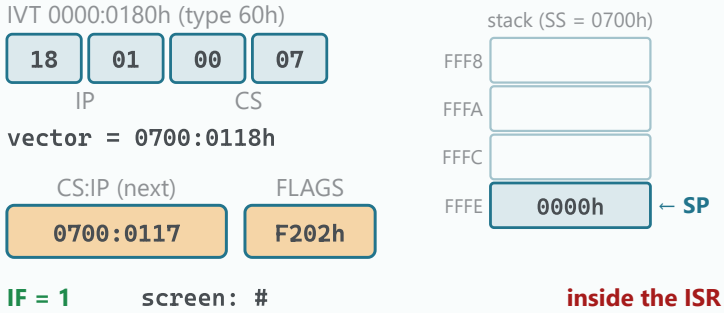
VARIABLES

`AX = 0E23h` `ES = 0000h` `SP = FFF8h` `FLAGS = F002h` `IF = 0`

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
10 ret
11 isr:
12 mov ah, 0Eh
13 int 10h
▶  iret
```



`iret` : IP = 0117h, CS = 0700h, FLAGS = F202h are restored; IF is 1 again, SP = FFFEh.

Step 11 / 12

VARIABLES

AX = **0E23h** · ES = **0000h** · SP = **FFFEh** · FLAGS = **F202h** · IF = **1**

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Our Own ISR on INT 60h

```
1  org 100h
2  mov ax, 0
3  mov es, ax            ; ES = 0: IVT
4  cli
5  mov word ptr es:[60h*4], offset isr
6  mov word ptr es:[60h*4+2], cs
7  sti
8  mov al, '#'
9  int 60h               ; our interrupt
▶ ret
11 isr:
12 mov ah, 0Eh
13 int 10h
14 iret
```

IVT 0000:0180h (type 60h)

18	01	00	07
IP		CS	

vector = 0700:0118h

CS:IP (next)	FLAGS
0700:0000	F202h

IF = 1 screen: #

stack (SS = 0700h)

FFF8	
FFFA	
FFFC	
FFFE	

`ret`: the main program resumes where it left off and returns to DOS.

Step 12 / 12

VARIABLES

AX = 0E23h | ES = 0000h | SP = 0000h | FLAGS = F202h | IF = 1

Type 60h is one of the types set aside for user programs. Values assume an emu8086-style .COM load (CS = 0700h).

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h         ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

03	10	00	F0
IP		CS	

`vector = F000:1003h`

AX	BX	CS:IP	IF
0000	0000	0700:0104	1

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	0000 ← SP

`output:`

ES = 0000h: the IVT. The type 3 vector is at 3 × 4 = **000Ch**.

Step 1 / 17

VARIABLES

AX = **0000h** BX = **0000h** SP = **FFFEh** IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx     ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h        ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

03	10	00	F0
IP		CS	

`vector = F000:1003h`

AX	BX	CS:IP	IF
0130	0000	0700:0107	1

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	0000 ← SP

`output:`

AX = `offset isr3` = **0130h**.

Step 2 / 17

VARIABLES

AX = **0130h** | BX = **0000h** | SP = **FFFEh** | IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
▶6 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h         ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	F0
IP		CS	

vector = F000:0130h

AX	BX	CS:IP	IF
1003	0000	0700:010C	1

output:

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	0000 ← SP

`xchg ax, es:[12]` : one instruction writes the new IP and brings the old IP (**1003h**) into AX.

Step 3 / 17

VARIABLES

AX = **1003h** BX = **0000h** SP = **FFFEh** IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
▶6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx     ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h        ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	F0
IP		CS	

`vector = F000:0130h`

AX	BX	CS:IP	IF
1003	0000	0700:010F	1

`output:`

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	0000 ← SP

The old IP is saved in `old_vec`.

Step 4 / 17

VARIABLES

AX = **1003h** BX = **0000h** SP = **FFFEh** IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
 6 mov old_vec, ax
▶ mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx     ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h        ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	F0
IP		CS	

vector = F000:0130h

AX	BX	CS:IP	IF
0700	0000	0700:0111	1

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	0000 ← SP

output:

AX = CS = 0700h.

Step 5 / 17

VARIABLES

AX = 0700h | BX = 0000h | SP = FFFEh | IF = 1

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h         ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	07
IP		CS	

vector = 0700:0130h

AX	BX	CS:IP	IF
F000	0000	0700:0116	1

output:

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	0000 ← SP

`xchg` : the new CS is written; the old CS = **F000h** is in AX. The vector is now 0700:0130.

Step 6 / 17

VARIABLES

AX = F000h BX = 0000h SP = FFFEh IF = 1

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
▶ mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx     ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h        ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```



The old CS is saved too: the old vector is F000:1003. Only the program calls type 3, so no CLI is needed; for hardware types these lines go between CLI/STI.

Step 7 / 17

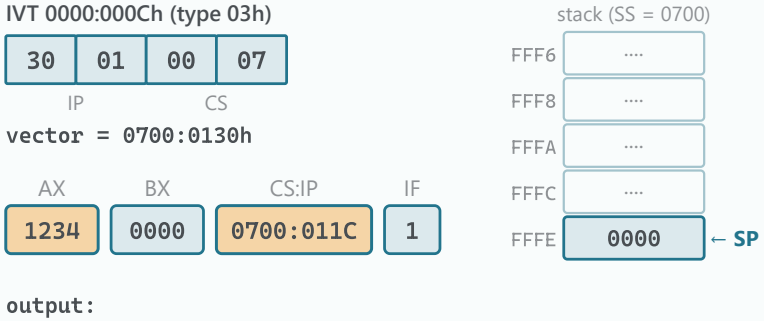
VARIABLES

AX = **F000h** BX = **0000h** SP = **FFFEh** IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
▶10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx     ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h        ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```



AX = 1234h: a value of the main program. The ISR must not destroy it.

Step 8 / 17

VARIABLES

AX = **1234h** | BX = **0000h** | SP = **FFFEh** | IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
▶ int 3                ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx    ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h       ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	07
IP		CS	

`vector = 0700:0130h`

AX	BX	CS:IP	IF
1234	0000	0700:0130	0

`output:`

stack (SS = 0700)

FFF6	....
FFF8	011D ← SP
FFFA	0700
FFFC	F246
FFFE	0000

`int 3` (one byte, CCh): FLAGS, CS and the return IP **011Dh** are pushed; CS:IP ← 0700:0130.

Step 9 / 17

VARIABLES

AX = **1234h** · BX = **0000h** · SP = **FFF8h** · IF = **0**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3               ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx    ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
▶ isr3: push ax
18 mov ax, 0E21h       ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	07
IP		CS	

vector = 0700:0130h

AX	BX	CS:IP	IF
1234	0000	0700:0131	0

output:

stack (SS = 0700)

FFF6	1234	← SP
FFF8	011D	
FFFA	0700	
FFFC	F246	
FFFE	0000	

`push ax` : the ISR first saves the AX it is about to use (1234h on the stack).

Step 10 / 17

VARIABLES

AX = 1234h | BX = 0000h | SP = FFF6h | IF = 0

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
▶ mov ax, 0E21h         ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	07
IP		CS	

`vector = 0700:0130h`

AX	BX	CS:IP	IF
0E21	0000	0700:0134	0

stack (SS = 0700)

FFF6	1234	← SP
FFF8	011D	
FFFA	0700	
FFFC	F246	
FFFE	0000	

`output:`

AX = 0E21h: AH = 0Eh (teletype), AL = 21h ('!').

Step 11 / 17

VARIABLES

AX = 0E21h | BX = 0000h | SP = FFF6h | IF = 0

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h         ; AH=0Eh, AL='!'
▶ int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	07
IP		CS	

vector = 0700:0130h

AX	BX	CS:IP	IF
0E21	0000	0700:0136	0

output: !

stack (SS = 0700)

FFF6	1234	← SP
FFF8	011D	
FFFA	0700	
FFFC	F246	
FFFE	0000	

`int 10h` : **!** is printed.

Step 12 / 17

VARIABLES

AX = **0E21h** BX = **0000h** SP = **FFF6h** IF = **0**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
1  org 100h
2  xor ax, ax
3  mov es, ax            ; ES = 0: IVT
4  mov ax, offset isr3
5  xchg ax, es:[3*4]     ; new IP ⇔ old
6  mov old_vec, ax
7  mov ax, cs
8  xchg ax, es:[3*4+2]   ; new CS ⇔ old
9  mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h         ; AH=0Eh, AL='!'
19 int 10h
▶ pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	07
IP		CS	

vector = 0700:0130h

AX	BX	CS:IP	IF
1234	0000	0700:0137	0

output: !

stack (SS = 0700)

FFF6	....
FFF8	011D ← SP
FFFA	0700
FFFC	F246
FFFE	0000

`pop ax` : AX = **1234h**, the main program's value is back.

Step 13 / 17

VARIABLES

AX = 1234h | BX = 0000h | SP = FFF8h | IF = 0

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h         ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
▶ iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

30	01	00	07
IP		CS	

vector = 0700:0130h

AX	BX	CS:IP	IF
1234	0000	0700:011D	1

output: !

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	0000 ← SP

`iret` : IP, CS and FLAGS are restored; SP = FFFEh.

Step 14 / 17

VARIABLES

AX = 1234h | BX = 0000h | SP = FFFEh | IF = 1

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                ; our ISR
▶ mov bx, old_vec
▶ mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h        ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

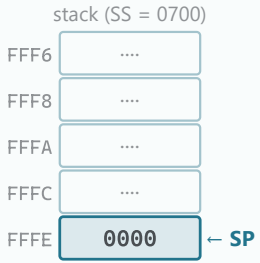
IVT 0000:000Ch (type 03h)

03	10	00	07
IP		CS	

vector = 0700:1003h

AX	BX	CS:IP	IF
1234	1003	0700:0126	1

output: !



The old IP is written back into the vector.

Step 15 / 17

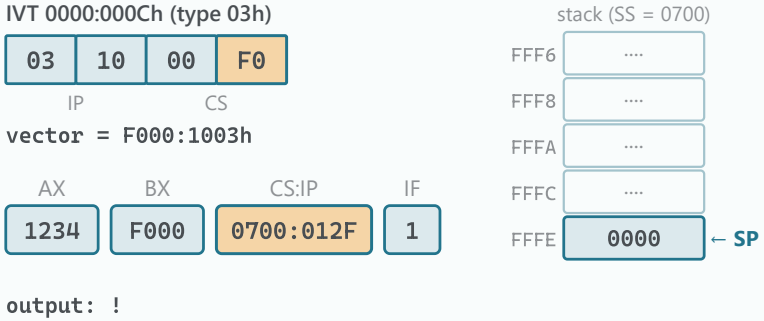
VARIABLES

AX = **1234h** | BX = **1003h** | SP = **FFFEh** | IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax           ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]    ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx     ; restore
▶ mov bx, old_vec+2
▶ mov es:[3*4+2], bx
16 ret
17 isr3: push ax
18 mov ax, 0E21h        ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```



The old CS is written back too: type 3 points to the old routine again.

Step 16 / 17

VARIABLES

AX = **1234h** | BX = **F000h** | SP = **FFFEh** | IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Code Trace: Save, Replace, Restore a Vector

```
 1 org 100h
 2 xor ax, ax
 3 mov es, ax            ; ES = 0: IVT
 4 mov ax, offset isr3
 5 xchg ax, es:[3*4]     ; new IP ⇔ old
 6 mov old_vec, ax
 7 mov ax, cs
 8 xchg ax, es:[3*4+2] ; new CS ⇔ old
 9 mov old_vec+2, ax
10 mov ax, 1234h
11 int 3                 ; our ISR
12 mov bx, old_vec
13 mov es:[3*4], bx      ; restore
14 mov bx, old_vec+2
15 mov es:[3*4+2], bx
▶ ret
17 isr3: push ax
18 mov ax, 0E21h         ; AH=0Eh, AL='!'
19 int 10h
20 pop ax
21 iret
22 old_vec dw ?, ?
```

IVT 0000:000Ch (type 03h)

03	10	00	F0
IP		CS	

`vector = F000:1003h`

AX	BX	CS:IP	IF
1234	F000	0700:0000	1

stack (SS = 0700)

FFF6	....
FFF8	....
FFFA	....
FFFC	....
FFFE	....

`output: !`

`ret` : AX = 1234h was preserved, and the vector table is as it was before the program ran.

Step 17 / 17

VARIABLES

AX = **1234h** | BX = **F000h** | SP = **0000h** | IF = **1**

The old vector F000:1003h is the emulator's default routine. Under DOS: INT 21h, AH = 35h / 25h.

Rules for Writing an ISR, and Common Mistakes

Rule	If broken
Save every register you use with PUSH/POP (the processor saves FLAGS).	The main program continues with a corrupted register; the bug shows up at random moments.
A hardware ISR sends the 8259A an EOI: `MOV AL, 20h` · `OUT 20h, AL`.	The 8259A never clears the in-service bit; that IRQ and all lower ones never come again.
Keep the ISR short; leave long work to the main program through a flag variable.	Interrupt latency grows and fast devices lose data.
Never call INT 21h from a hardware ISR.	DOS is not reentrant; the interrupted DOS call's data gets corrupted.
Write the vector under CLI or with AH = 25h, and restore the old one on exit.	After the program ends, the interrupt jumps into memory that now holds something else.
End the ISR with `IRET`.	The stack is off by 4 bytes and IF stays off.

A long ISR may use STI to let higher-priority interrupts in; the ISR itself must then be reentrant.

Lab: The Interrupt Mechanism

INT 90h: our own handler

CODE · **29 bytes**

```
 1 org 100h
 2 mov ax, 0
 3 mov es, ax                ; ES =…
 4 cli                       ; inte…
 5 mov word ptr es:[90h*4], o…
 6 mov word ptr es:[90h*4+2],…
 7 sti                       ; inte…
 8 int 90h                   ; our …
 9 ret
10 isr:
11 mov ah, 0Eh
12 mov al, '*'
13 int 10h
14 iret                      ; not …
```

REGISTERS

AX	0000	AH 00	AL 00		
BX	0000	BH 00	BL 00		
CX	001D	CH 00	CL 1D		
DX	0000	DH 00	DL 00		
SI	0000	DI 0000	BP 0000		
SP	FFFE	CS 0700	IP 0100		
DS	0700	ES 0700	SS 0700		

FLAGS

O	D	I	T	S	Z	A	P	C
0	0	1	0	0	0	0	0	0

CS:IP = 0700:0100 · step 0

MEMORY **DS = 0700h**

```
0100 B8 00 00 8E C0 FA 26 C7
0108 06 40 02 16 01 26 8C 0E
0110 42 02 FB CD 90 C3 B4 0E
```

STACK **SS:SP**

```
FFF2 0000
FFF4 0000
FFF6 0000
FFF8 0000
FFFA 0000
FFFC 0000
FFFE 0000 ◀ SP          DOS return
```

OUTPUT

Program loaded: 29 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (001Dh). The Step button executes instructions one at a time; the highlighted line is the next instruction.

SUMMARY

Summary

Interrupt
The processor sets its work aside, runs the ISR, and returns where it left off with `IRET`.

Vector table
0000:0000–03FF, 256 × 4 bytes. Type n → address n × 4 (IP), n × 4 + 2 (CS).

INT n
FLAGS, CS, IP onto the stack (SP −6), IF = TF = 0, CS:IP ← vector.

Hardware interrupt
INTR is accepted only when IF = 1, at an instruction boundary; the type number comes from the 8259A on the second INTA pulse. NMI is always INT 2.

8259A
The IMR masks individual IRQs (port 21h); the ISR ends with an EOI: `MOV AL, 20h` · `OUT 20h, AL`.

Your own ISR
Write the vector between CLI/STI, save the registers you use, end with `IRET`; restore the old vector on exit.

SUMMARY

Exercises

1. At which physical addresses is the `INT 13h` vector?
2. If SP = FFFEh before `INT 21h` executes, what is SP inside the ISR? What is on the stack?
3. With IMR = B8h, write the instructions that also mask IRQ0 (the timer). What is the new IMR?
4. The type 61h vector is to become 0700:0120h. Which bytes go to which physical addresses?
5. On a 5 MHz 8086, INTR arrives just as `MUL BX` (at most 133 clocks) starts. What is the worst-case interrupt latency?
6. What problems does ending an ISR with `RET` instead of `IRET` cause?

▶ **Hints / answers**

References

- A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., Pearson, 2008.
- Intel, *The 8086 Family User's Manual*, 1979, Interrupts chapter.
- B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009, Chapters 6 and 12.
- emu8086 help: the “Interrupts” and “Supported interrupts” pages.
- R. Brown, *Ralf Brown's Interrupt List* (BIOS/DOS interrupt reference).

The End

Chapter 5.1: Interrupts

Previous: Chapter 4.2: Variables · Next: Chapter 5.2: Interrupt Functions