Hardware Interrupts

Hardware Interrupts

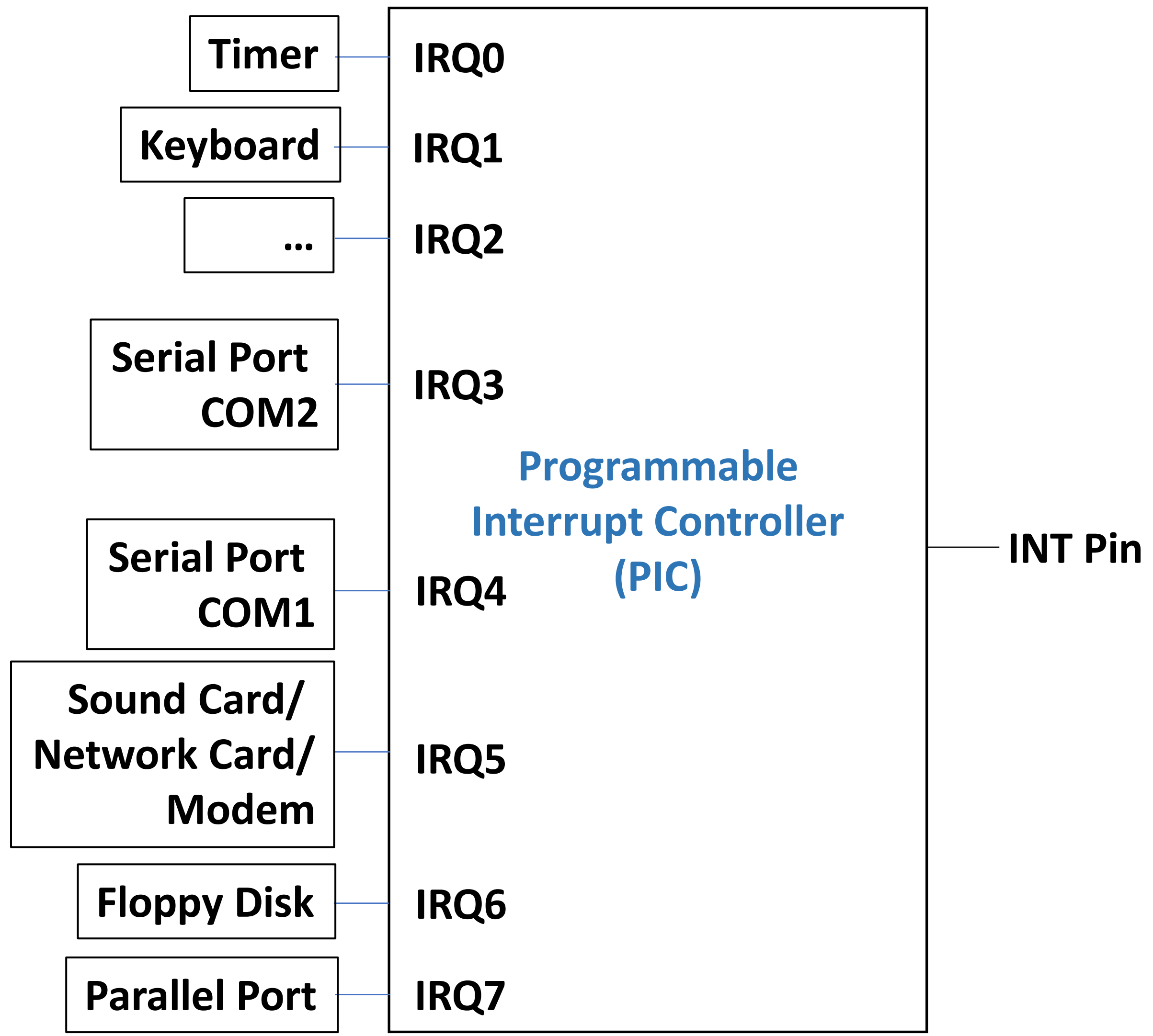
- A hardware interrupt request (IRQ) is an electronic signal issued by an external hardware device, to communicate that it needs attention from the CPU.
- Such external devices may be part of the computer (e.g., disk controller) or they may be external peripherals. For example, pressing a keyboard key or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position.
- Occasionally, programs must disable hardware interrupts when performing sensitive operations on segment registers and the stack. The CLI (clear interrupt flag) instruction disables interrupts, and the STI (set interrupt flag) instruction enables interrupts

Hardware Interrupts

- A single pin on the processor chip, called the INT pin is used by external hardware to generate interrupts.
- There are many devices generating interrupts and there is only one pin going inside the processor.
  - One pin cannot be technically derived by more than one source
  - Therefore a controller called the Programmable Interrupt Controller (8259 PIC) is used to schedule the interrupts (priority scheduling)

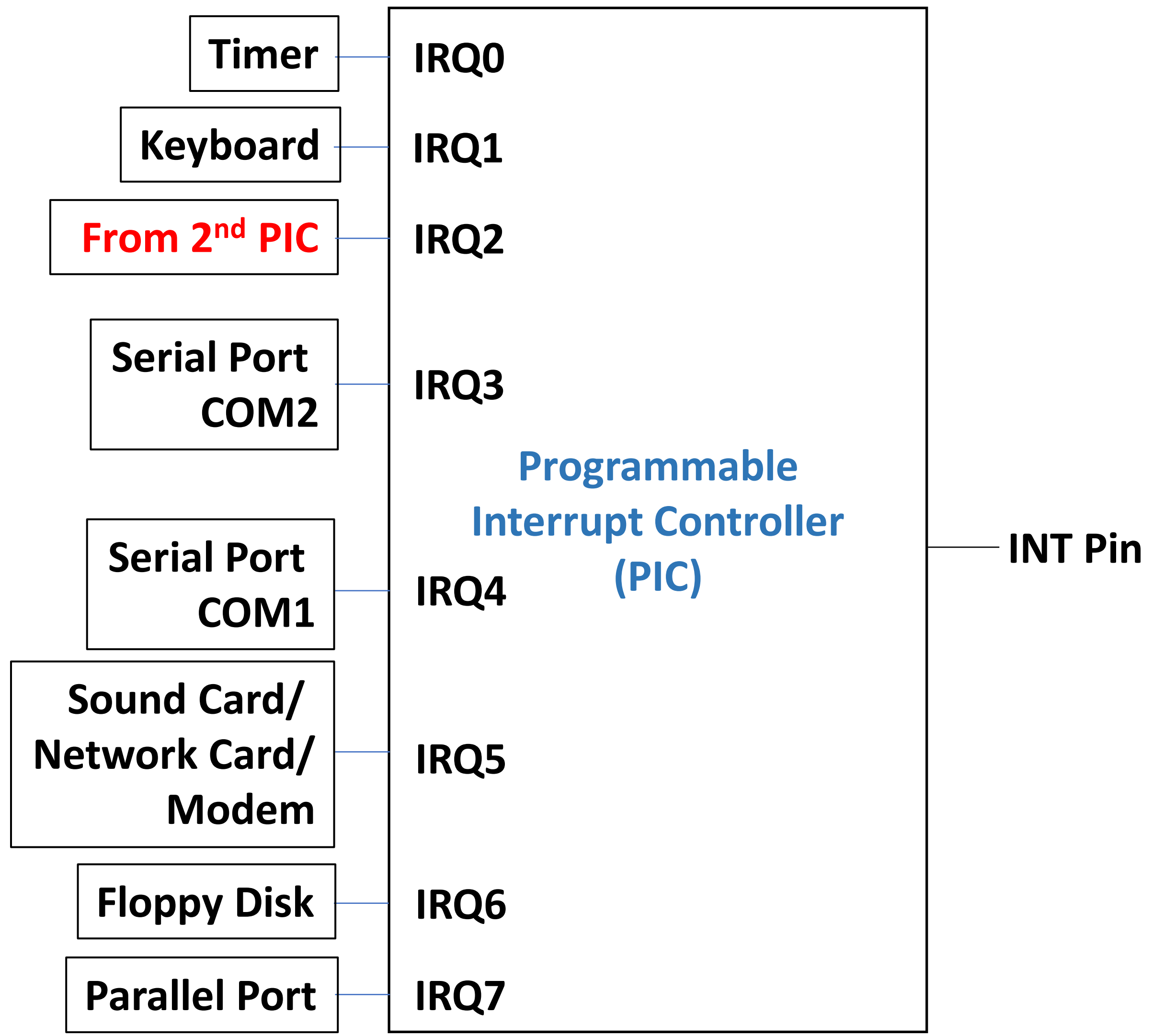
Programmable Interrupt Controller (PIC)

- It has eight input signals and one output signal.
- It assigns priorities to its eight input pins from 0 to 7 to handle if more than one interrupt arrives at the same time
- 0 has the highest priority and 7 has the lowest
- A lower-level interrupt cannot interrupt a higher-level one still in progress.



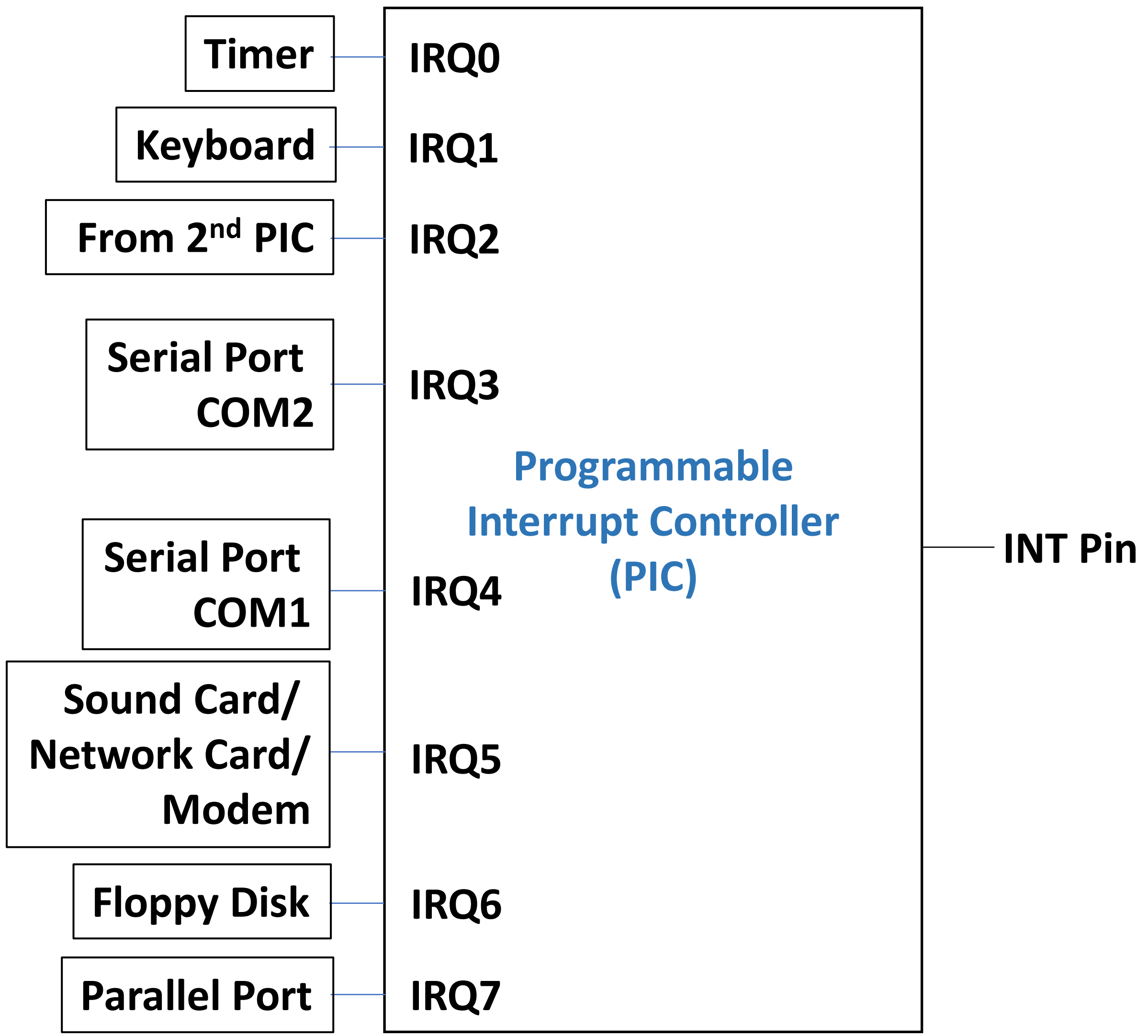
PIC cascading

- The original IBM XT computer had one PIC so there were 8 possible interrupt sources.
- However IBM AT and later computers have two PICs totaling 15 possible interrupt sources.



Interrupt Requests (IRQ)

- The eight input signals to the PIC are called Interrupt Requests (IRQ).
- The eight lines are called IRQ 0 to IRQ 7
- Each IRQ is mapped to a specific interrupt in the system
- This mapping is done by the PIC and not the processor.



IRQ to Interrupt Mapping

IRQ	Interrupt	Description
0	8h	System timer (18.2 times/second)
1	9h	Keyboard
2	0Ah	Programmable Interrupt Controller
3	0Bh	COM2 (serial port 2)
4	0Ch	COM1 (serial port 1)
5	0Dh	LPT2 (parallel port 2) used for sound card or the network card or the modem
6	0Eh	floppy disk drive
7	0Fh	LPT1 (parallel port 1)

IRQ Mechanism

- The actual mechanism fetches one instruction from the PIC whenever the INT pin is signaled instead of the memory.
- From the perspective of an assembly language programmer, an IRQ 0 is translated into an INT 8 without any such instruction in the program.
- Therefore an IRQ 0, the highest priority interrupt, is generated by the timer chip at a precise frequency and the handler at INT 8 is invoked which updates the system time.

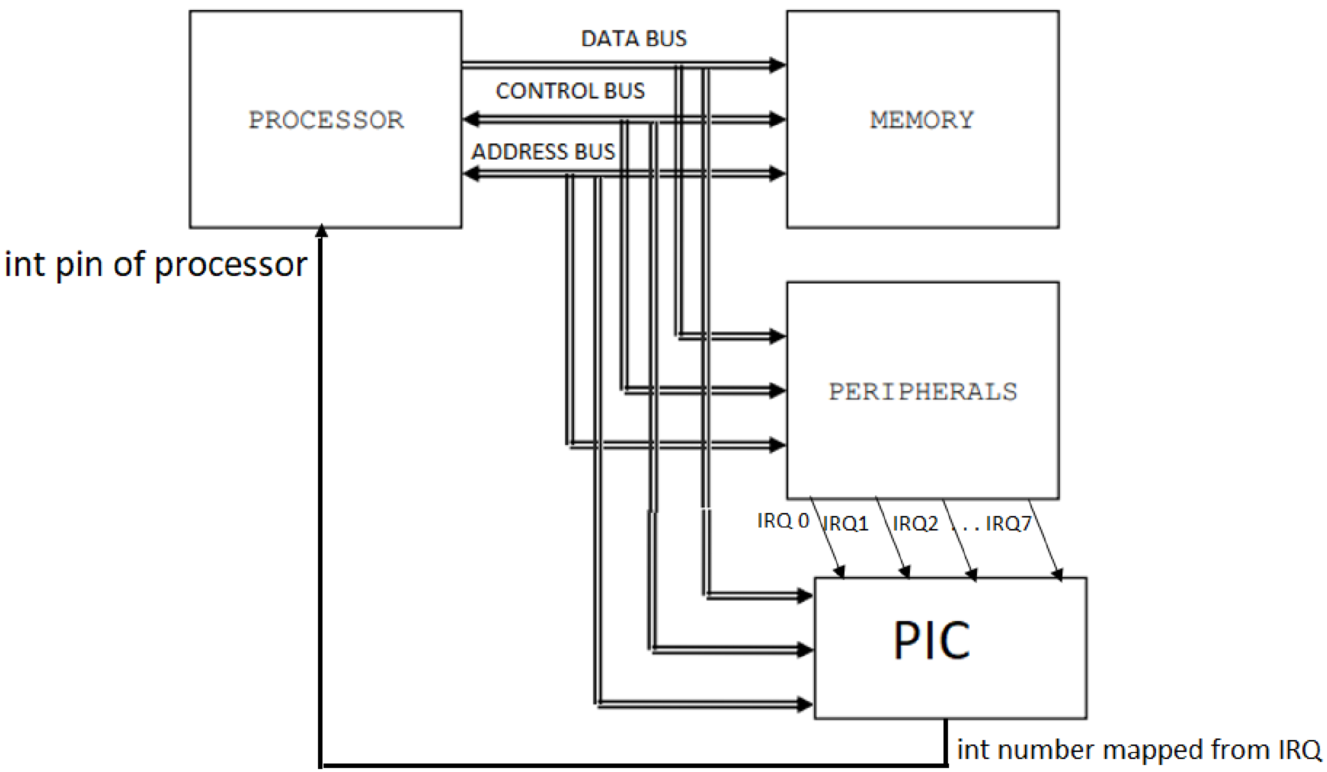
IRQ Mechanism

- A key press generates IRQ 1 and the INT 9 handler is invoked which stores this key.
- To hook the timer and keyboard interrupts one can replace the vectors corresponding to interrupt 8 and 9 respectively.

End Of Interrupt (EOI) signal

- When processor received a hardware interrupt number from PIC, it will do the same thing to handle it as any other interrupt.
- That is, push old values of flags, CS and IP, read IVT to jump to ISR of interrupt and execute it.
- At the end of servicing the interrupt, the handler should inform the PIC that it is completed so that lower priority interrupts can be sent next.
- This signal is called an End Of Interrupt (EOI) signal and is sent through the I/O ports of the PIC.
  - I/O ports discussed next

CPU and PIC Interconnection



- To get processors' attention peripherals send IRQ to PIC
- PIC then sends INT to processor
- Processor can then run the ISR and data is usually read or written to devices by the ISR code
- Data is read/written using BUS
- EOI is informed to PIC also via BUS

I/O Ports

- PIC is used to get the attention of processor in request of a device. But data is transferred to/from device using BUSes.
- Each device has an 8 or 16 bit port number.
- To communicate to a device, port number is placed on address bus
- A pin is set on control bus to show that the communication is to be between I/O device and CPU (not CPU and memory)
- Lower 16 bits of the address bus as used to specify port number
  - there can be total of 65536 possible I/O ports
- For example, port numbers 20h and 21h are for PIC, 60h to 64h for keyboard, 378h for the parallel port etc.

https://wiki.osdev.org/I/O_Ports

IN and OUT instructions

- Used to read and write data to ports — just like mov is used to read or write to memory
- Version 1, with 8 bit port number
  - `IN al, <8 bit port #>   ; read a byte from port to al`
  - `IN ax, <8 bit port #>   ; read a word from port to ax`
  - `OUT <8 bit port #>, al ; write a byte in al to port`
  - `OUT <8 bit port #>, ax ; write a word in ax to port`
- Version 2, with 16 bit port number (place port # in dx)
  - `MOV dx, <16 bit port #>`
  - `IN al, dx    ; read a byte from port to al`
  - `IN ax, dx    ; read a word from port to ax`
  - `OUT dx, al   ; write a byte in al to port`
  - `OUT dx, ax   ; write a word in ax to port`

Only AL or AX can be the destination for IN and source for OUT instruction

Examples

- To read data from keyboard
  - `in al, 0x60   ; read scan code in AL`
  - `in ax, 0x60   ; read ascii in AH, scan code in AL`
- Other examples
  - `mov dx, 0x389 ; DX can contain a port number`
  - `in ax, dx     ; input word from port # in DX`
  - `out dx, ax    ; output word to the same port`
  - `in ax, 0xFFFF ; will give error`

PIC Ports

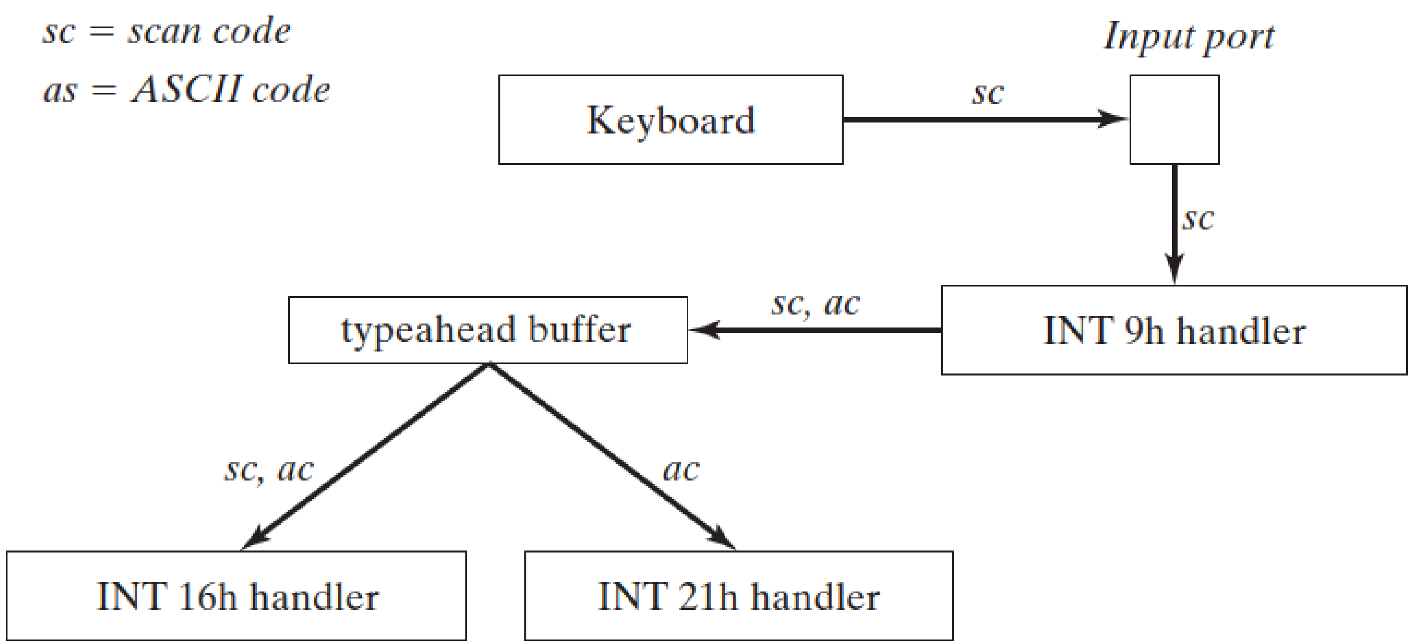
- Programmable interrupt controller has two ports 20 and 21.
- Port 20 is the control port while port 21 is the interrupt mask register which can be used for selectively enabling or disabling interrupts.
  - A 0 bit will enable an interrupt and a 1 bit disables it
- Program to disable the keyboard using this port is given below.
  - After running this program your keyboard will be disabled.
  - Restart DOSBOX to restore it to default setting

```
; disable keyboard interrupt in PIC mask register
[org 100h]

in al, 21h         ; read interrupt mask register
or al, 00000010b   ; disable IRQ-1 by setting bit 1
out 21h, al        ; write back mask register

mov ax, 0x4c00
int 21h
```

How keyboard works

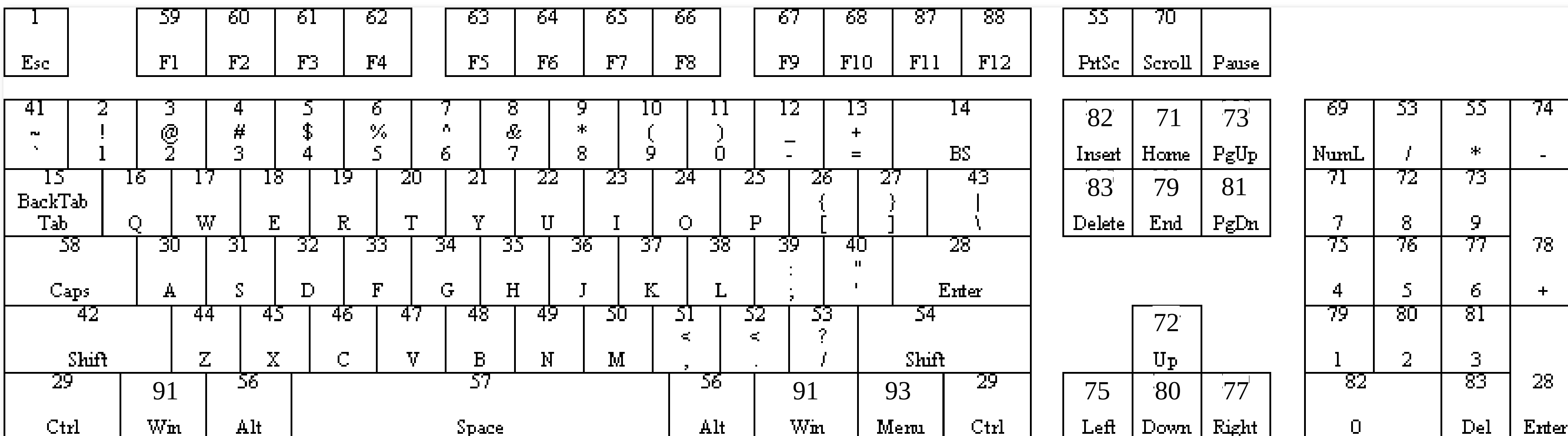
- We have seen earlier int 16h, a software interrupt that is **called by programmer** to wait and read a keyboard stroke
- int 9h is the hardware interrupt that is generated when a key is pressed.
- When int 9h is invoked, scan code and decoded ascii of pressed key are moved to buffer.

Keystroke processing sequence



These handlers read data from buffer only, NOT from input port

Recap - Scan codes of keys



How keyboard works

- For each key, the scan code is sent twice, once for the key press and once for the key release. Both scan codes differ in one bit only.
  - The lower seven bits contain the key number while the MSB is clear in the press code and set in the release code.
- You can read the ascii and scan code of pressed key using keyboard port
  - `in al, 0x60  ; read scan code in al`
  - `in ax, 0x60  ; read ascii in ah, scan code in al`

How keyboard works

- If we press Shift-A resulting in a capital A on the screen, the controller has sent the press code of Shift, the press code of A, the release code of A, the release code of Shift and the interrupt handler has understood that this sequence should result in the ASCII code of ‘A’.
- The ‘A’ key always produces the same scan code whether or not shift is pressed. It is the interrupt handler’s job to remember that the press code of Shift has been received and release code has not been received and therefore to change the meaning of the following key presses.
  - Caps-lock key works the same way.
- You can hook int 9h and change what happens when a key is pressed

Hooking int 9

- Create an ISR that shows scancode at every key press (and release!)
- Run this program to see that scan code changes on every key press and unpress

```
[org 0x0100]
jmp start

;;;;; insert clrscr subroutine   ;;;;;;
;;;;; insert printnum subroutine ;;;;;;

start:
    call clrscr

    ; Hook into int 9h
    xor ax, ax
    mov es, ax
    cli
    mov word [es:9*4], kbIsr
    mov [es:9*4+2], cs
    sti

    ; infinite loop
    ; interrupted by keyboard events
l2: jmp l2
```

```
; keyboard interrupt service routine
kbIsr:
        push ax

        call clrscr
        in al, 60h    ; read scancode from keyboard port
        mov ah, 0     ; clear upper half of ax
        push ax       ; print this scancode
        call printnum

        mov al, 20h
        out 20h, al   ; send EOI signal to PIC
        pop ax
        iret
```

Comparison with int 16h

- Run the program below and compare its bhaviour with the previous one.

```
[org 0x0100]
jmp start

;;;;;;; insert clrscr subroutine ;;;;;;;
;;;;;;; insert printnum subroutine ;;;;;

start:
    mov ah, 0     ; wait keystroke service
    int 16h       ; returns scancode in AH

    mov bl, ah    ; put scancode in BX
    mov bh, 0
    push bx       ; and push it to stack for printing
    call clrscr
    call printnum

    jmp start     ; infinite loop
```

Question

- What is the difference between int 9h and int 16h?

9h is hardware interrupt. Its handler reads data directly from keyboard hardware. This interrupt should be hooked if fine control of keystroke response is required.

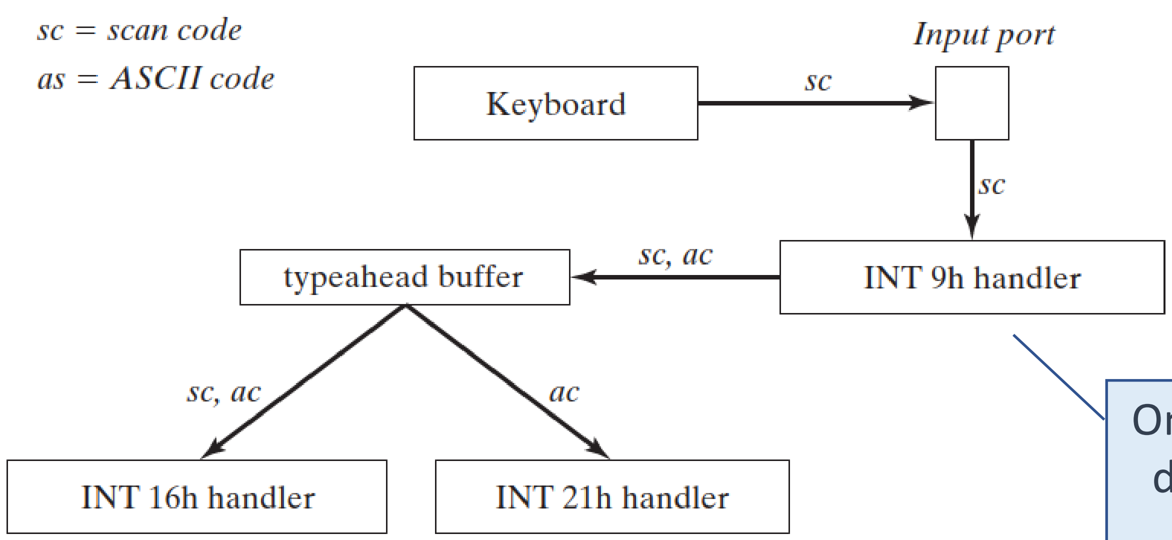
16h is software interrupt, it is called by programmer to get data from the typeahead buffer. If buffer is empty, it will keep waiting until a scan code (+ ascii code) is placed in buffer.

Template to hook Hardware interrupt n

```
newISR:
    store registers
    <<< body >>>
    mov al, 0x20
    out 0x20, al        ; send EOI to PIC
    restore registers
    iret
start:
    xor ax, ax
    mov es, ax                      ; load zero in es
    mov word [es:n*4], newISR ; put offset at n*4
    mov [es:n*4+2], cs        ; segment at n*4+2
```

Not included when hooking software interrupts

Interrupt hooking side affect



Once we replace the default 9h handler with our own …

… the handler for 16h will stop working (since typeahead buffer will remain empty)

Solution: Interrupt Chaining

- To keep the default functionality of an interrupt, we can “chain” rather than “replace” the existing interrupt handler (ISR).
- To make the chain
  - save the address of old ISR before hooking
  - the new ISR should not do IRET itself
  - rather it must do a far jump to the old ISR code

Interrupt chaining example

- Following program behaves the same as before: scan code is printed for each key press, program does not terminate with Esc.

```
start:
    call clrscr

    ; Hook into int 9h
    xor ax, ax
    mov es, ax                      ; point es to IVT base
    cli                             ; disable other interrupts
    mov word [es:9*4], kbIsr        ; store offset at n*4
    mov [es:9*4+2], cs              ; store segment at n*4+2
    sti                             ; re-enable other interrupts

l2: mov ah, 0          ; get keystroke service
    int 16h

    cmp ah, 1          ; quit on Esc key
    jne l2             ; otherwise keep looping

    mov ax, 0x4c00
    int 21h
```

doesn't work

Interrupt chaining example

- To make int 16h work, chain the new **kbIsr** with old ISR.

```
start:
    call clrscr

    xor ax, ax
    mov es, ax              ; point es to IVT base
    mov ax, [es:9*4]        ; save offset of original vector
    mov [oldIsr], ax
    mov ax, [es:9*4+2]      ; same with segment
    mov [oldIsr+2], ax
    ; Now hook into 9h
    cli
    mov word [es:9*4], kbIsr
    mov [es:9*4+2], cs
    sti

l2: mov ah, 0          ; get keystroke service
    int 16h
    cmp ah, 1          ; quit on esc key
    jne l2             ; otherwise keep looping

    mov ax, 0x4c00
    int 21h
```

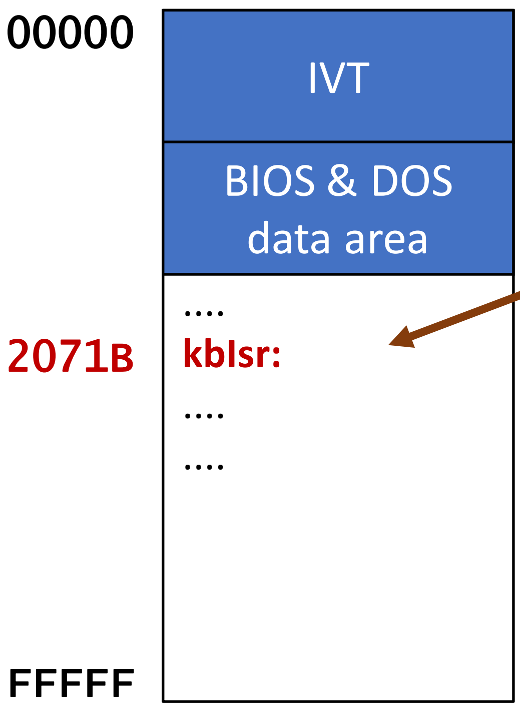
```
oldIsr:  dd  0    ; allocate two words

kbIsr:
    push ax

    call clrscr
    in al, 60h    ; read scancode from keyboard port
    mov ah, 0     ; clear upper half of ax
    push ax       ; print this scancode
    call printnum

    pop ax
    ; No need for EOI and iret
    ; original (chained) routine will do these

    jmp far [cs:oldIsr]   ; call the original ISR
```

Interrupt Unhooking

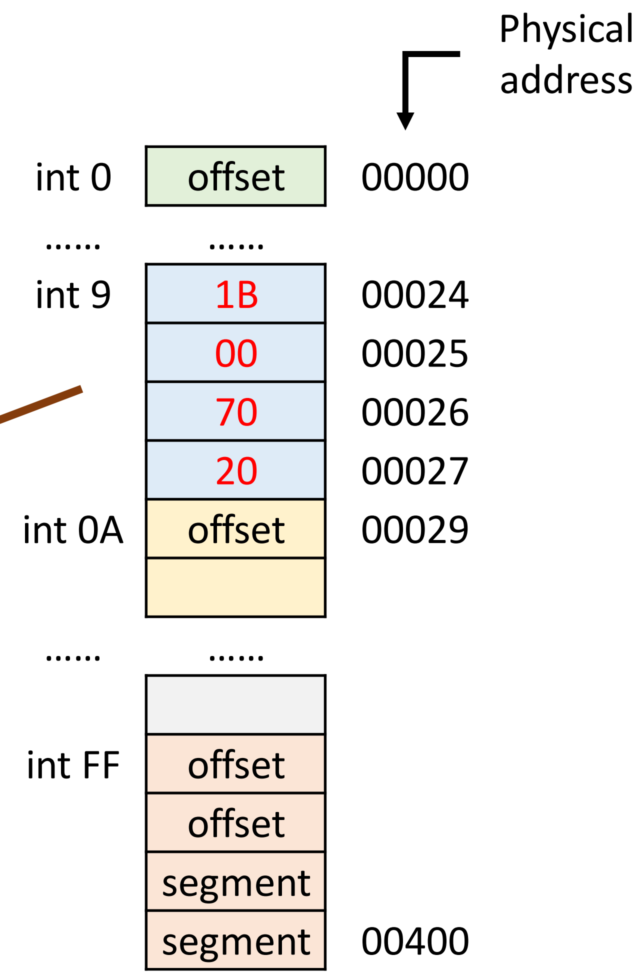
- Once you have hooked an interrupt, it will stay hooked even after your program exits.
- It can possibly crash the computer since the code of new ISR will be overwritten by code/data of a new program
- So, we should unhook our ISR before exiting our programs i.e. reset the IVT entry to its old value.

Interrupt Unhooking - Why

Suppose we have hooked int 9h to **kbIsr** (address 2070:001B)



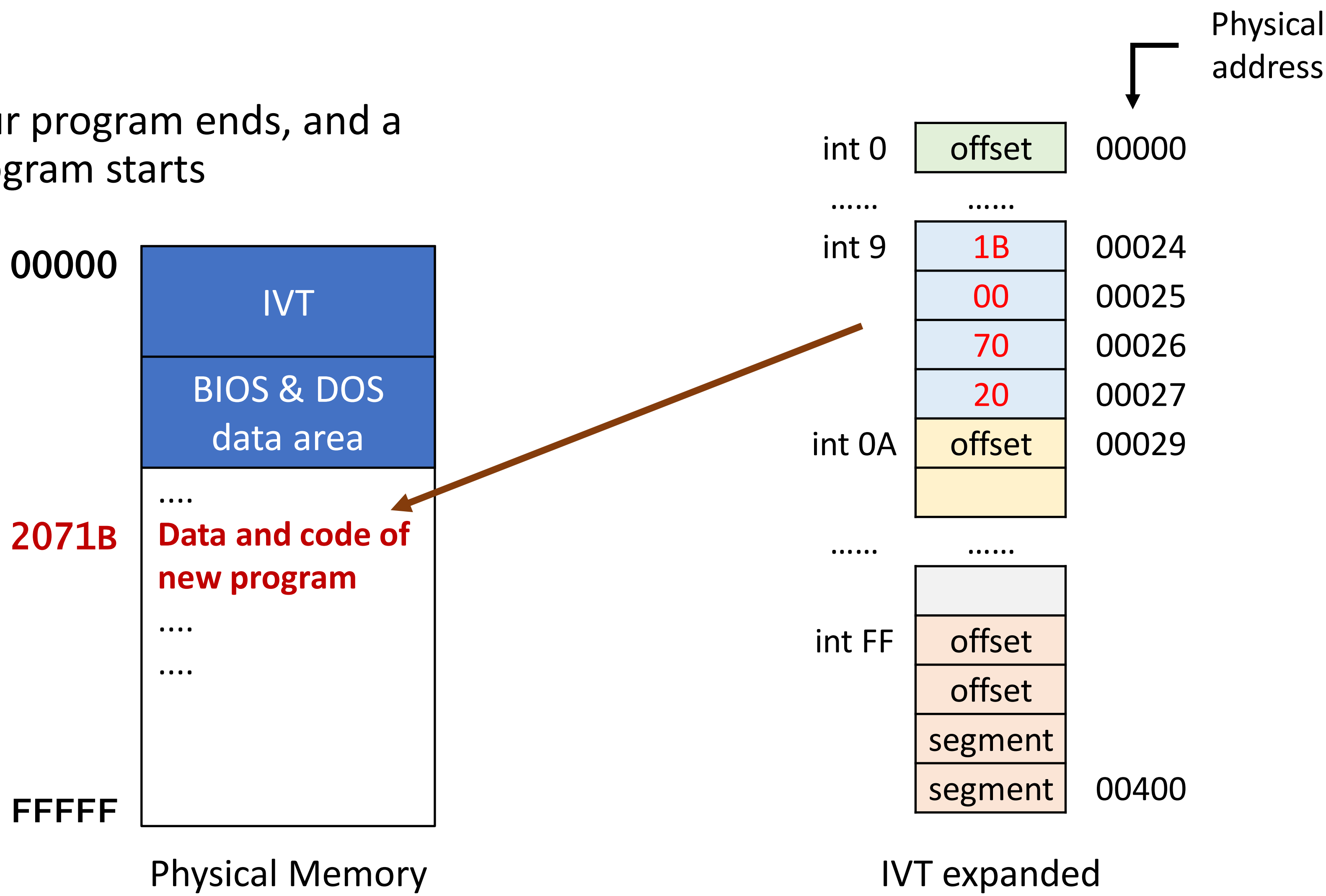
Physical Memory



IVT expanded

Interrupt Unhooking - Why

Once our program ends, and a new program starts



Unhooking example

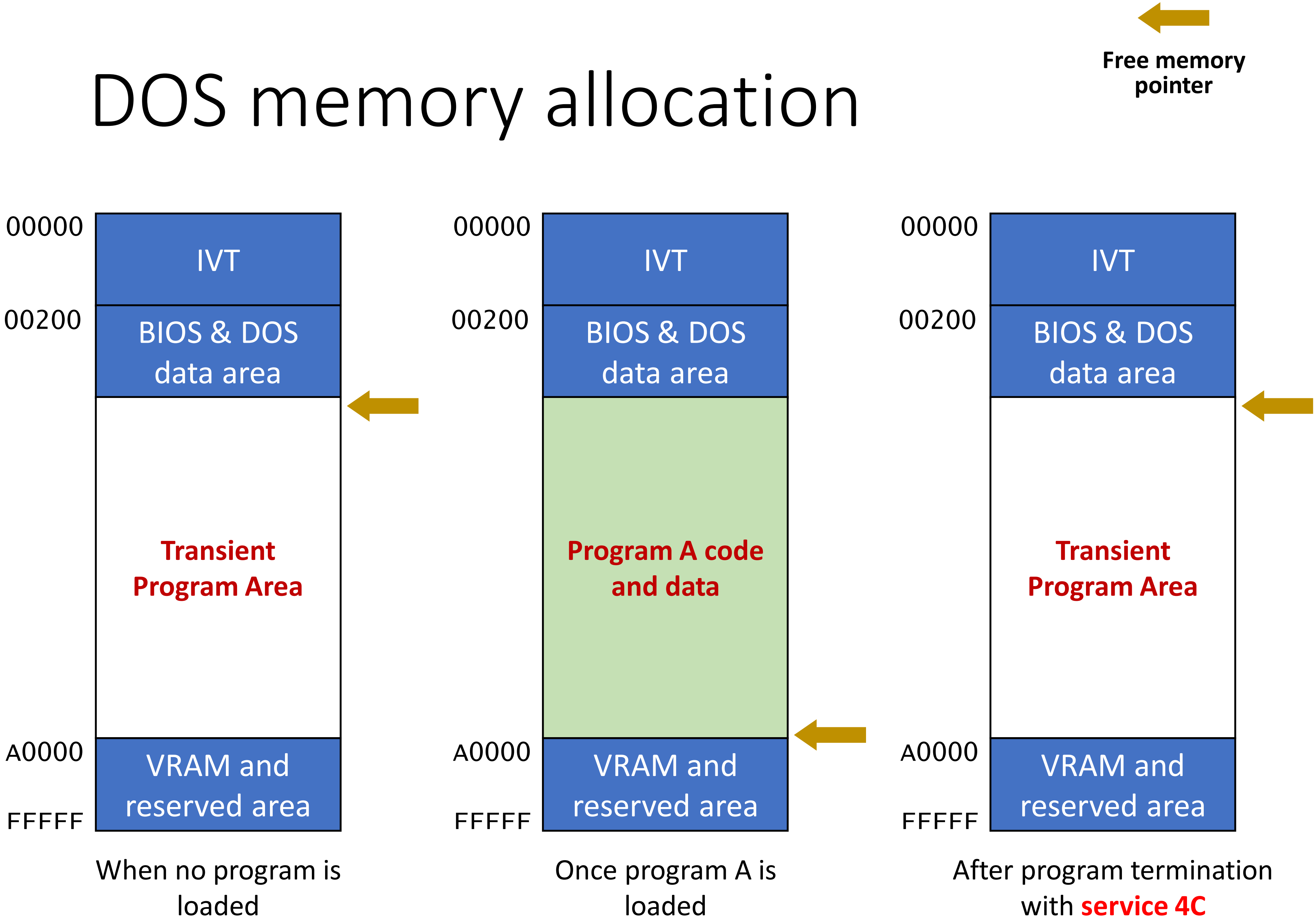
- Add the following code to previous program before exit

```
mov ax, [oldIsr]          ; read old offset in ax
mov bx, [oldIsr+2]        ; read old segment in bx
cli                       ; disable interrupts
mov [es:9*4], ax          ; restore old offset from ax
mov [es:9*4+2], bx        ; restore old segment from bx
sti                       ; enable interrupts

mov ax, 0x4c00
int 21h
```

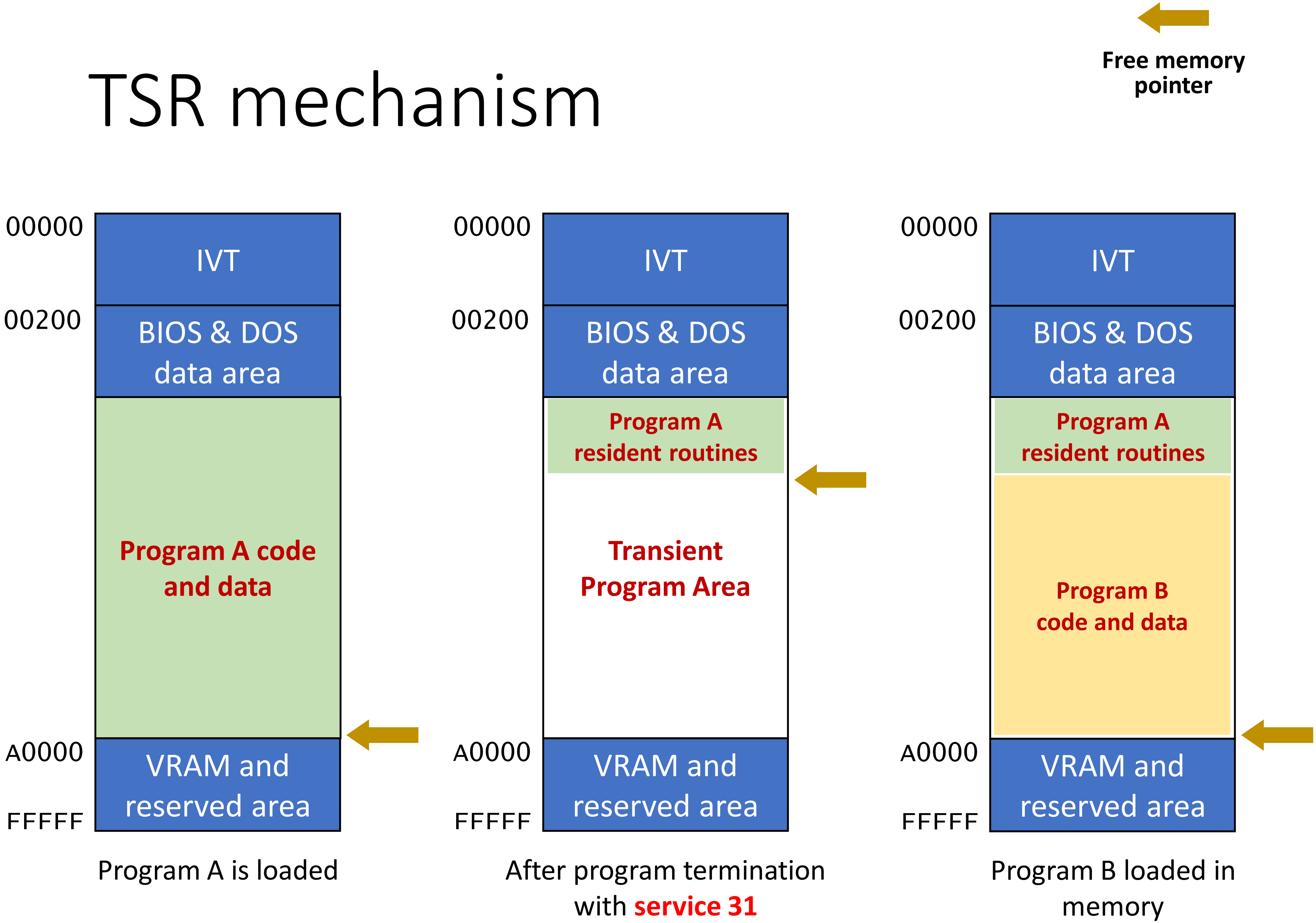
Terminate and Stay Resident (TSR)

- In case we want our hooked ISR to remain functional even after our program ends
- We need to tell DOS to retain that code in memory (i.e. not overwrite it)
- How?
    1. Tell DOS how many paragraphs of memory you want to retain
    2. Terminate the program using int 21h service 31 (instead of service 4C)

DOS memory allocation



TSR mechanism



TSR How to

- Tell DOS how many paragraphs (16 byte blocks) of memory you want to retain.
  - Put this number in DX register
  - Then call int 21h service 31
- To get number of paragraphs, place a label after the last line of resident routines.
- Divide the address (offset) of this label by 16 and round up.
- An easy way to do rounded-up division is to add 15 to offset and then shift right 4 bits (which is division by 16)

TSR example

- To make everything before the “start” label resident, use the following code to terminate the program

```
mov   dx, start      ; end of resident portion
add   dx, 15         ; round up to next para
mov   cl, 4          ; number of bits to shift
shr   dx, cl         ; div by 16 to get number of paras
mov   ax, 0x3100     ; terminate and stay resident
int   0x21
```

Controlling speakers through I/O port

- See the example given here

https://web.archive.org/web/20190917215918/http://www.intel-assembler.it/portale/5/make-sound-from-the-speaker-in-assembly/8255-8255-8284-asm-program-example.asp

References

- Chapter 9 BH, section 9.1 to 9.3