ECE291

Lecture 9

Interrupt implementation

(part of the magic explained!)

Lecture outline

- Interrupt vectors
- Software interrupts
- Hardware interrupts
- 8259 Programmable Interrupt Controller
- Writing your own handlers
- Installing handlers

Interrupts

- Triggers that cause the CPU to perform various tasks on demand
- Three kinds:
  - Software interrupts – initiated by the INT instruction
  - Hardware interrupts – originate in peripheral hardware
  - Exceptions – occur in response to error states in the processor
- Regardless of source, they are handled the same
  - Each interrupt has a unique interrupt number from 0 to 255. These are called interrupt vectors.
  - For each interrupt vector, there is an entry in the interrupt vector table.
  - The interrupt vector table is simply a jump table containing segment:offset addresses of procedures to handle each interrupt
  - These procedures are called *interrupt handlers* or *interrupt service routines (ISR's)*

Interrupt vectors

- The first 1024 bytes of memory (addresses 00000 – 003FF) always contain the interrupt vector table. Always. Never anything else.
- Each of the 256 vectors requires four bytes—two for segment, two for offset

Memory address (hex)	Interrupt function pointer
003FC	INT 255
4 * x	INT x
00008	INT 2
00004	INT 1
00000	INT 0

Software interrupts

- Essentially just function calls using a different instruction to do the calling
- Software interrupts give you access to “built-in” code from BIOS, operating system, or peripheral devices
- Use the INT instruction to “call” these procedures

The INT and IRET instructions

- Syntax: INT imm8
- Imm8 is an interrupt vector from 0 to 255
- INT does the following:
  - Pushes flag register (pushf)
  - Pushes return CS and IP
  - Far jumps to [0000:(4*imm8)]
  - Clears the interrupt flag disabling the interrupt system
- IRET is to INT what RET is to CALL
  - Pops flag register
  - Performs a far return

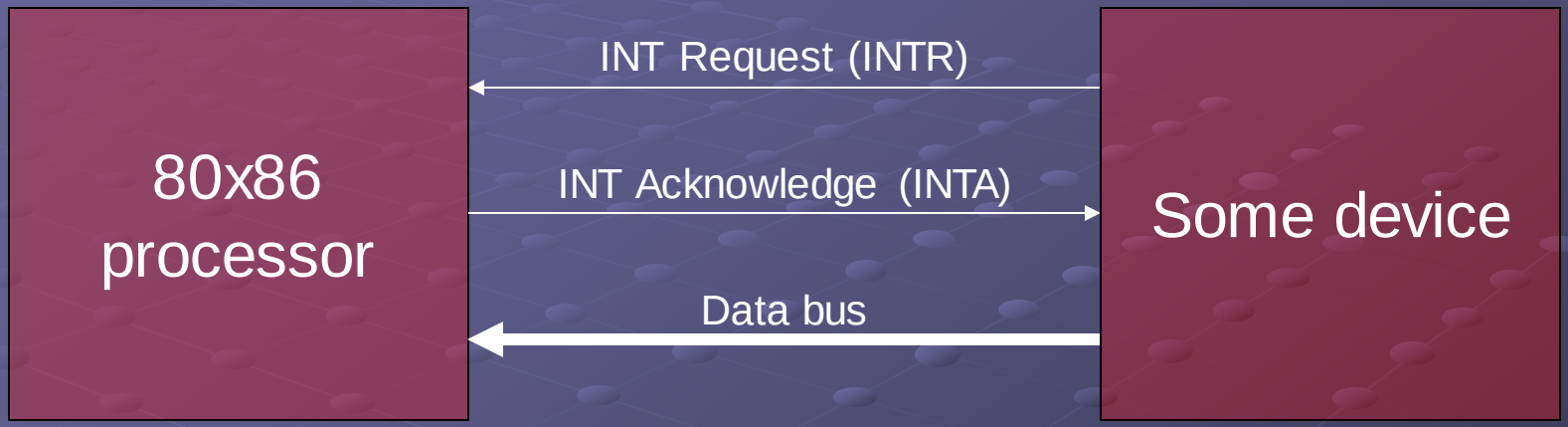
Things to notice

- The interrupt vector table is just a big permanently located jump table
- The values of the jump table are pointers to code provided by bios, hardware, the os, or eventually 291 students
- Interrupt service routines preserve the flags – the state of the computer should be completely unaltered by an ISR

Hardware interrupts

- Alert the processor of some hardware situation that needs tending to
  - A key has been pressed
  - A timer has expired
  - A network packet has arrived
- Same software calling protocol
- Additional level of complexity with the interrupt "call" not coming from your program code
- Can happen at any time during the execution of your program

The 80x86 interrupt interface

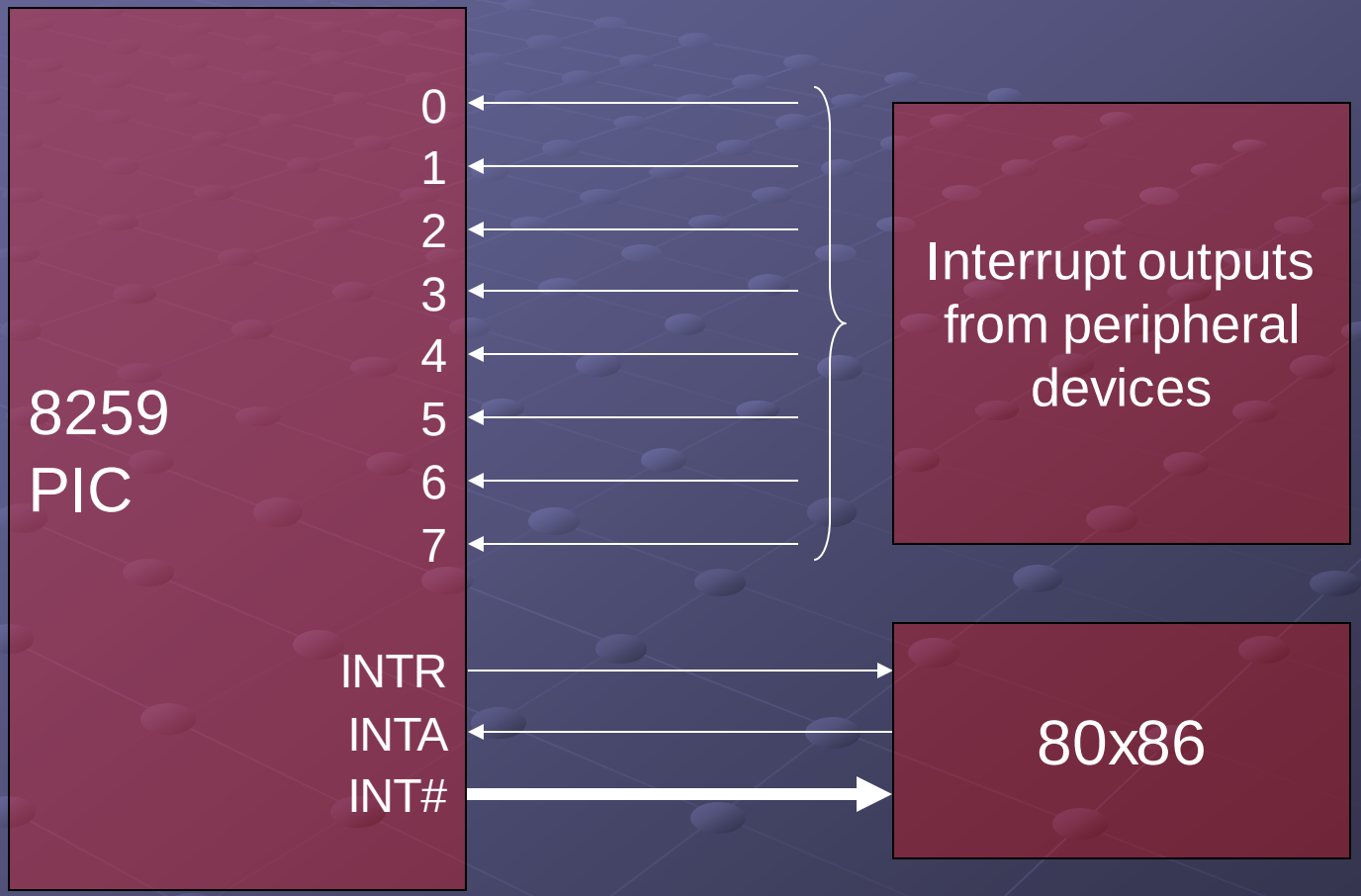


- Device generates request signal
- Device supplies interrupt vector number on data bus
- Processor completes the execution of current instruction and executes ISR corresponding to the interrupt vector number on the data bus
- ISR upon completion acknowledges the interrupt by asserting the INTA signal

But wait there's more…

- The above setup could get complicated with multiple devices generating multiple interrupts connected to the same few pins on the processor
- Enter the 8259 Programmable Interrupt Controller

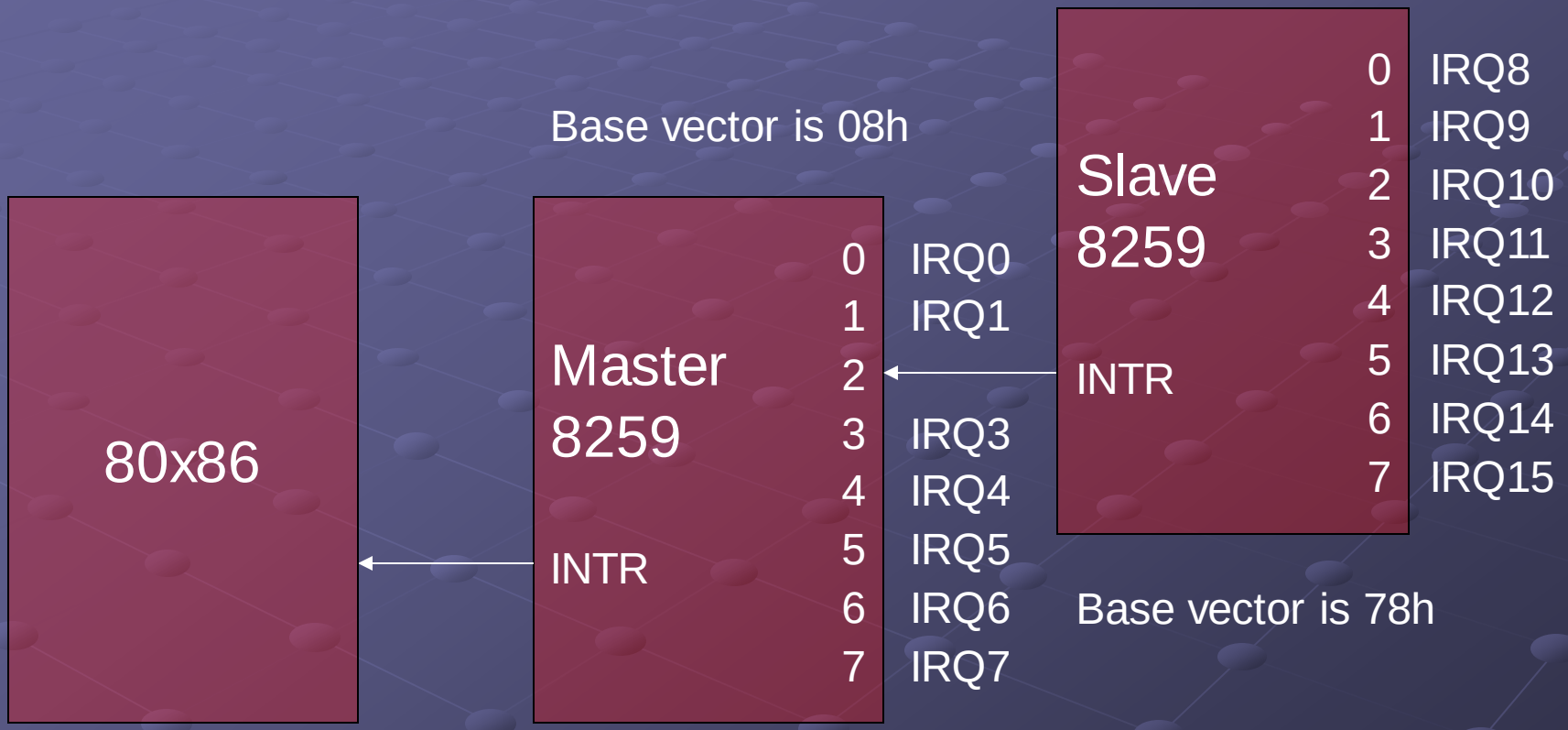
The 8259 PIC


8259
PIC
0
1
2
3
4
5
6
7
INTR
INTA
INT#
Interrupt outputs from peripheral devices
80x86

The 8259 PIC

- The PIC is programmed with a base interrupt vector number
  - “0” corresponds to this base
  - “1” corresponds to this base + 1
  - *“n”* corresponds to this base + *n*
- For example, if the PIC is programmed with a base interrupt vector of 8, then “0” corresponds to interrupt vector 8

The whole enchilada



Typical IRQ assignments

- IRQ 0: Timer (triggered 18.2/second)
- IRQ 1: Keyboard (keypress)
- IRQ 2: Slave PIC
- IRQ 3: Serial Ports (Modem, network)
- IRQ 5: Sound card
- IRQ 6: Floppy (read/write completed)
- IRQ 8: Real-time Clock
- IRQ 12: Mouse
- IRQ 13: Math Co-Processor
- IRQ 14: IDE Hard-Drive Controller

Interrupt priority

- Lower interrupt vectors have higher priority
- Lower priority can't interrupt higher priority
  - ISR for INT 21h is running
    - Computer gets request from device attached to IRQ8 (INT 78h)
    - INT 21h procedure must finish before IRQ8 device can be serviced
  - ISR for INT 21h is running
    - Computer gets request from Timer 0 IRQ0 (INT 8h)
    - Code for INT 21h gets interrupted, ISR for timer runs immediately, INT21h finishes afterwards

Servicing an interrupt

- Complete current instruction
- Preserve current context
  - PUSHF Store flags to stack
  - Clear Trap Flag (TF) & Interrupt Flag (IF)
  - Store return address to stack PUSH CS, PUSH IP
- Identify Source
  - Read 8259 PIC status register
  - Determine which device (N) triggered interrupt
- Activate Interrupt Service Routine
  - Use N to index vector table
  - Read CS/IP from table
  - Jump to instruction
- Execute Interrupt Service Routine
  - usually the handler immediately re-enables the interrupt system (to allow higher priority interrupts to occur) (STI instruction)
  - process the interrupt
- Indicate End-Of-Interrupt (EOI) to 8259 PIC

```
mov   al, 20h
out   20h, al
```

;transfers the contents of AL to I/O port 20h

- Return (IRET)
  - POP IP (Far Return)
  - POP CS
  - POPF (Restore Flags)

Interrupt service routines

- Reasons for writing your own ISR's
  - to supersede the default ISR for internal hardware interrupts (e.g., division by zero)
  - to chain your own ISR onto the default system ISR for a hardware device, so that both the system's actions and your own will occur on an interrupt (e.g., clock-tick interrupt)
  - to service interrupts not supported by the default device drivers (a new hardware device for which you may be writing a driver)
  - to provide communication between a program that terminates and stays resident (TSR) and other application software

Interrupt service routines

- DOS facilities to install ISRs

Function	Action
INT 21h Function 25h	Set Interrupt vector
INT 21h Function 35h	Get Interrupt vector
INT 21h Function 31h	Terminate and stay resident

- Restrictions on ISRs
  - Currently running program should have no idea that it was interrupted.
  - ISRs should be as short as possible because lower priority interrupts are blocked from executing until the higher priority ISR completes

Interrupt Service Routines

- ISRs are meant to be short
  - keep the time that interrupts are disable and the total length of the service routine to an absolute minimum
  - remember after interrupts are re-enabled (STI instruction), interrupts of the same or lower priority remain blocked if the interrupt was received through the 8259A PIC
- ISRs can be interrupted
- ISRs must be in memory
  - Option 1: Redefine interrupt only while your program is running
    - the default ISR will be restored when the executing program terminates
  - Option 2: Use DOS Terminate-and-Stay-Resident (TSR) command to load and leave program code permanently in memory

Installing ISRs

Let **N** be the interrupt to service

- Read current function pointer in vector table
  - Use DOS function 35h
  - Set **AL = N**
  - Call DOS Function **AH = 35h, INT 21h**
  - Returns: ES:BX = Address stored at vector N
- Set new function pointer in vector table
  - Use DOS function 25h
  - Set DS:DX = New Routine
  - Set **AL = N**
  - DOS Function **AH = 25h, INT 21h**

Installing ISR

- Interrupts can be installed, chained, or called
- Install New interrupt
  replace old interrupt

```
MyIntVector
        Save Registers
        Service Hardware
        Reset PIC
        Restore Registers
        IRET
```

- Chain into interrupt
  Service myCode first

```
MyIntVector
        Save Registers
        MyCode
        Restore Registers
        JMP CS:Old_Vector
```

- Call Original Interrupt
  Service MyCode last

```
MyIntVector
        PUSHF
        CALL CS:Old_Vector
        Save Registers
        MyCode
        Restore Registers
        IRET
```

Interrupt Driven I/O

- Consider an I/O operation, where the CPU constantly tests a port (e.g., keyboard) to see if data is available
  - CPU polls the port if it has data available or can accept data
- Polled I/O is inherently inefficient
- Wastes CPU cycles until event occurs
- *Analogy:* Checking your watch every 30 seconds until your popcorn is done, or standing at the door until someone comes by

- Solution is to provide interrupt driven I/O
- Perform regular work until an event occurs
- Process event when it happens, then resume normal activities
- *Analogy:* Alarm clock, doorbell, telephone ring

Timer interrupt example

- In this example we will replace the ISR for the Timer Interrupt
- Our ISR will count the number of timer interrupts received
- Our main program will use this count to display elapsed time in minutes and seconds

Timer interrupt - main proc skeleton

```
;====== Variables ==================
; Old Vector (far pointer to old interrupt function)
oldv        RESW    2
count       DW 0        ;Interrupt counter (1/18
    sec)
scount      DW 0        ;Second counter
mcount      DW 0        ;Minute counter
pbuf        DB 8        ;Minute counter
;====== Main procedure =====
..start
…
;----Install Interrupt Routine-----
call Install
;Main program (print count values)
.showc
Mov ax, [mcount]    ;Minute  Count
…
```

```
call pxy
mov ax, [scount]    ;Second Count
…
call pxy
mov ax,[count]      ;Interrupt Count (1/18 sec)
…
call pxy
mov ah,1
int 16h             ;Check for key press
jz   .showc         ;Quit on any key

;---- Uninstall Interrupt Routine-----
call UnInst         ;Restore original INT8
…
call mpxit
```

Timer interrupt – complete main proc

```
..start
    mov   ax, cs            ;Initialize DS=CS
    mov   ds, ax
    mov   ax, 0B800h        ;ES=VideoTextSegment
    mov   es, ax
    call  install           ;Insert my ISR
showc:
    mov   ax, [mcount]      ;Minute Count
    mov   bx, pbuf
    call  binasc
    mov   bx, pbuf
    mov   di,0              ;Column 0
    mov   ah,00001100b      ;Intense Red
    call  pxy

    mov   ax,[scount]       ;Second Count
    mov   bx,pbuf
    call  binasc
    mov   bx, pbuf
    mov   di,12             ;Column 6 (DI=12/2)
    mov   ah,00001010b      ;Intense Green
    call  pxy

    mov   ax,[count]        ;Int Count (1/18th sec)
    mov   bx,pbuf
    call  binasc
    mov   bx, pbuf
    mov   ah,00000011b      ;Cyan
    mov   di,24             ;Column 12 (DI=24/2)
    call  pxy
    mov   ah,1
    int   16h               ;Key Pressed ?
    jz    showc

    Call  UnInst            ;Restore original INT8
    mov   ax,4c00h          ;Normal DOS Exit
    int   21h
```

Timer interrupt – PXY and Install interrupt

```
;pxy (bx = *str, ah = color, di = column)
pxy
      mov   al, [bx]
      cmp   al, '$'
      je    .pxydone
      mov   es:[di+2000], ax
      inc   bx
      add   di,2
      jmp   pxy
.pxydone
      ret
;====== Install Interrupt =====
install                       ;Install new INT 8 vector
      push  es
      push  dx
      push  ax
      push  bx
```

```
      mov   al, 8         ;INT = 8
      mov   ah, 35h       ;Read  Vector Subfunction
      int   21h           ;DOS Service

      mov   word [oldv+0], bx
      mov   word [oldv+2], es
      mov   al, 8         ;INT = 8
      mov   ah, 25h       ;Set Vector Subfunction

      mov   dx, myint
                          ;DS:DX point to function
      int   21h           ;DOS Service

      pop  bx
      pop  ax
      pop  dx
      pop  es
      ret
```

Timer interrupt – uninstall interrupt

```
;====== Uninstall Interrupt ===========
UnInst                  ; Uninstall Routine (Reinstall old vector)
    push  ds
    push  dx
    push  ax
    mov   dx, word [oldv+0]
    mov   ds, word [oldv+2]

    mov   al, 8         ; INT = 8
    mov   ah, 25h       ; Subfunction = Set Vector
    int   21h           ; DOS Service

    pop   ax
    pop   dx
    pop   ds
    ret
```

Timer interrupt – ISR code

```
;====== ISR Code =========
myint
    push  ds          ;Save all registers
    push  ax
    mov   ax, cs      ;Load default segment
    mov   ds, ax
    pushf             ;Call Orig Function w/flags
    call  far [oldv]  ;Far Call to existing routine

    inc   word [count]
                      ;Increment Interrupt count
    cmp   word [count],18
    jne   .myintdone
    inc   word [scount]       ;Next second
    mov   word [count], 0
    cmp   word [scount], 60
    jne   .myintdone
    inc   word [mcount]       ; Next minute
    mov   word [scount], 0

.myintdone
    mov   al, 20h     ;Reset the PIC
    out   20h, al     ;End-of-Interrupt signal
    pop   ax          ;Restore all Registers
    pop   ds
    iret              ;Return from Interrupt
```

The complete code with exe

- www.ece.uiuc.edu/ece291/lecture/timer.asm
- www.ece.uiuc.edu/ece291/lecture/timer.exe

Replacing An Interrupt Handler

```
;install new interrupt vector
%macro setInt 3   ;Num, OffsetInt, SegmentInt
      push   ax
      push   dx
      push   ds

      mov    dx, %2
      mov    ax, %3
      mov    ds, ax
      mov    al, %1
      mov    ah, 25h      ;set interrupt vector
      int    21h

     pop     ds
      pop    dx
      pop    ax
%endmacro
```

```
;store old interrupt vector
%macro getInt 3   ;Num, OffsetInt, SegmentInt
     push    bx
     push   es

      mov    al, %1
      mov    ah, 35h       ;get interrupt vector
      int    21h
      mov    %2, bx
      mov    %3, es

      pop    es
      pop    bx
%endmacro
```

Replacing An Interrupt Handler

```
CR EQU  0dh
LF  EQU  0ah

SEGMENT stkseg STACK
      resb 8*64
stacktop:

SEGMENT code

Warning  DB  "Overflow - Result Set to
    ZERO!!!!",CR,LF,0
msgOK    DB  "Normal termination", CR, LF, 0
old04hOffset        RESW
old04hSegment       RESW
```

```
New04h  ;our new ISR for int 04
        ;occurs on overflow
    sti     ;re-enable interrupts

    mov   ax, Warning
    push  ax
    call  putStr      ;display message
    xor   ax, ax      ;set result to zero
    cwd               ;AX to DX:AX

    iret
```

Replacing An Interrupt Handler

```
..start
    mov   ax, cs
    mov   ds, ax

    ;store old vector
    getInt 04h, [old04hOffset], [old04hSegment]

    ;replace with address of new int handler
    setInt 04h, New04h, cs

    mov   al, 100
    add   al, al
    into  ;calls  int 04  if an overflow occurred
    test  ax, 0FFh
    jz    Error

    mov   ax, msgOK
    push  ax
    call  putStr

Error:
    ;restore original int handler
    setInt 04h, [old04hOffset], [old04hSegment]

    mov   ax, 4c00h
    int   21h
```

NOTES

- INTO is a conditional instruction that acts only when the overflow flag is set
- With INTO after a numerical calculation the control can be automatically routed to a handler routine if the calculation results in a numerical overflow.
- By default Interrupt 04h consists of an IRET, so it returns without doing anything.