Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 17: Expert MS-DOS Programming

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **Defining Segments**
- Runtime Program Structure
- Interrupt Handling
- Hardware Control Using I/O Ports

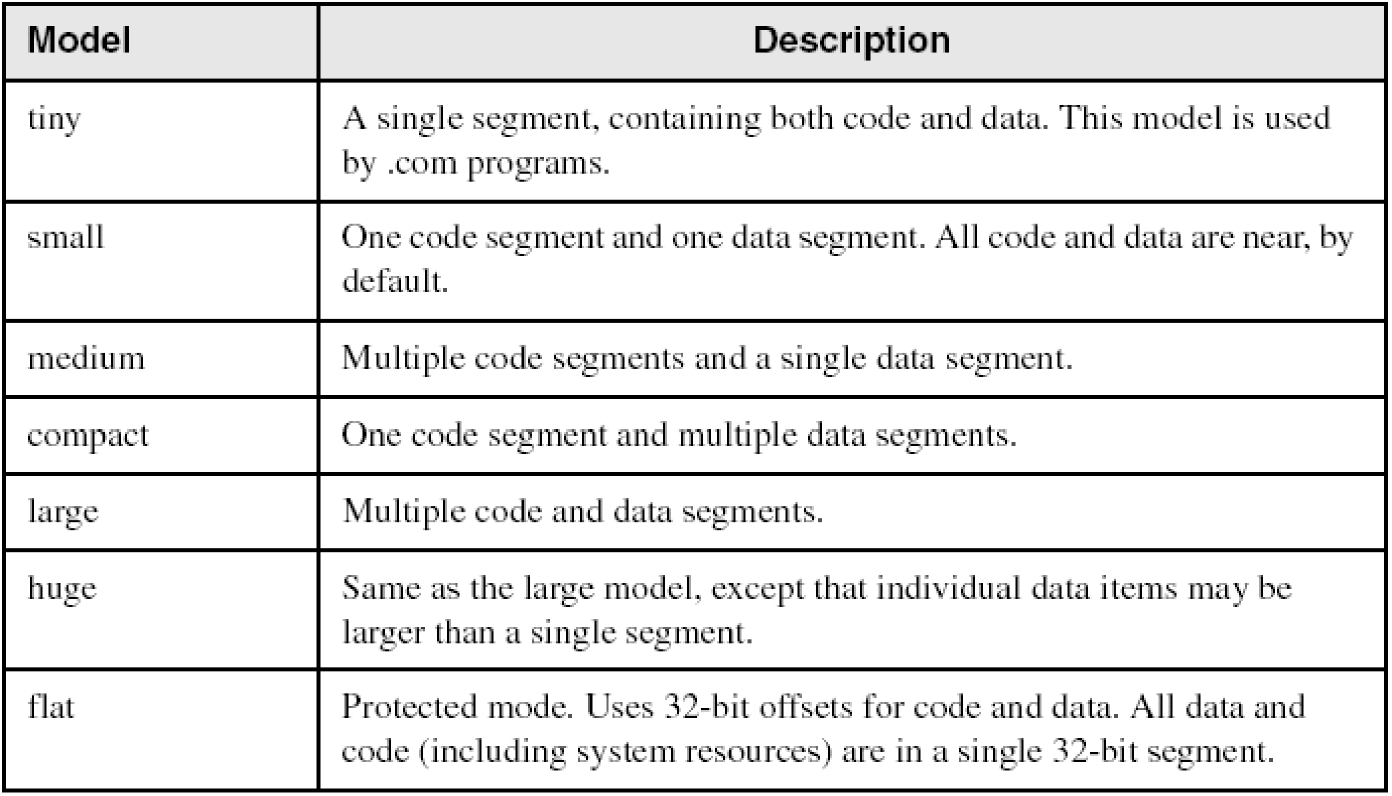
Defining Segments

- Simplified Segment Directives
- Explicit Segment Definitions
- Segment Overrides
- Combining Segments

Simplified Segment Directives

- .MODEL – program memory model
- .CODE – code segment
- .CONST – define constants
- .DATA – near data segment
- .DATA? – uninitialized data
- .FARDATA – far data segment
- .FARDATA? – far uninitialize data
- .STACK – stack segment
- .STARTUP – initialize DS and ES
- .EXIT – halt program

Memory Models



Model	Description
tiny	A single segment, containing both code and data. This model is used by .com programs.
small	One code segment and one data segment. All code and data are near, by default.
medium	Multiple code segments and a single data segment.
compact	One code segment and multiple data segments.
large	Multiple code and data segments.
huge	Same as the large model, except that individual data items may be larger than a single segment.
flat	Protected mode. Uses 32-bit offsets for code and data. All data and code (including system resources) are in a single 32-bit segment.

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

NEAR and FAR Segments

- NEAR segment
  - requires only a 16-bit offset
  - faster execution than FAR
- FAR segment
  - 32-bit offset: requires setting both segment and offset values
  - slower execution than NEAR

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

.MODEL Directive

- The .MODEL directive determines the names and grouping of segments
- .model tiny
  - code and data belong to same segment (NEAR)
  - .com file extension
- .model small
  - both code and data are NEAR
  - data and stack grouped into DGROUP
- .model medium
  - code is FAR, data is NEAR

.MODEL Directive

- .model compact
  - code is NEAR, data is FAR
- .model huge & .model large
  - both code and data are FAR
- .model flat
  - both code and data are 32-bit NEAR

.MODEL Directive

- Syntax:

  .MODEL *type*, *language*, *stackdistance*
- *Language* can be:
  - C, BASIC, FORTRAN, PASCAL, SYSCALL, or STDCALL (details in Chapters 8 and 12).
- *Stackdistance* can be:
  - NEARSTACK: (default) places the stack segment in the group DGROUP along with the data segment
  - FARSTACK: stack and data are **not** grouped together

.STACK Directive

- Syntax:

  .STACK [*stacksize*]
- *Stacksize* specifies size of stack, in bytes
  - default is 1024
- Example: set to 2048 bytes:
  - .stack 2048

.CODE Directive

- Syntax:

  .CODE [*segname*]

  - optional *segname* overrides the default name
- Small, compact memory models
  - NEAR code segment
  - segment is named _TEXT
- Medium, large, huge memory models
  - FAR code segment
  - segment is named *modulename*_TEXT

Whenever the CPU executes a FAR call or jump, it loads CS with the new segment address.

Calling Library Procedures

- You must use .MODEL small, stdcall
  - (designed for the small memory model)
- You can only call Irvine16 library procedures from segments named _TEXT.
  - (default name when .CODE is used)
- Advantages
  - calls and jumps execute more quickly
  - simple use of data—DS never needs to change
- Disadvantages
  - segment names restricted
  - limited to 64K code, and 64K data

Multiple Code Segments

Example, p. 585, shows calling Irvine16 procedures from main, and calling an MS-DOS interrupt from Display.

```
.code

main PROC
   mov   ax,@data
   mov   ds,ax
   call  WriteString
   call  Display
   .exit
main ENDP
```

```
.code OtherCode

Display PROC
   mov  ah,9
   mov  dx,OFFSET msg2
   int  21h
   ret
Display ENDP
```

Near Data Segments

- .DATA directive creates a Near segment
  - Up to 64K in Real-address mode
  - Up to 512MB in Protected mode (Windows NT)
  - 16-bit offsets are used for all code and data
  - automatically creates segment named DGROUP
  - can be used in any memory model
- Other types of data:
  - .DATA? (uninitialized data)
  - .CONST (constant data)

Far Data Segments

- .FARDATA
  - creates a FAR_DATA segment
- .FARDATA?
  - creates a FAR_BSS segment
- Code to access data in a far segment:

```
.FARDATA
myVar
.CODE
   mov ax,SEG myVar
   mov ds,ax
```

The SEG operator returns the segment value of a label. Similar to @data.

Data-Related Symbols

- @data returns the group of the data segment
- @DataSize returns the size of the memory model set by the .MODEL directive
- @WordSize returns the size attribute of the current segment
- @CurSeg returns the name of the current segment

Explicit Segment Definitions

- Use them when you cannot or do not want to use simplified segment directives
- All segment attributes must be specified
- The ASSUME directive is required

SEGMENT Directive

Syntax:

```
name SEGMENT [align] [combine] ['class']
    statements
name ENDS
```

- *name* identifies the segment; it can either be unique or the name of an existing segment.
- *align* can be BYTE, WORD, DWORD, PARA, or PAGE.
- *combine* can be PRIVATE, PUBLIC, STACK, COMMON, MEMORY, or AT *address*.
- *class* is an identifier used when identifying a particular type of segment such as CODE or STACK.

Segment Example

```
ExtraData SEGMENT PARA PUBLIC 'DATA'
    var1 BYTE 1
    var2 WORD 2
ExtraData ENDS
```

- name: ExtraData
- paragraph align type (starts on 16-bit boundary)
- public combine type: combine with all other public segments having the same name
- 'DATA' class: 'DATA' (load into memory along with other segments whose class is 'DATA')

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

ASSUME Directive

- Tells the assembler how to calculate the offsets of labels
- Associates a segment register with a segment name

Syntax:

```
ASSUME segreg:segname [,segreg:segname] ...
```

Examples:

```
ASSUME cs:myCode, ds:Data, ss:myStack
ASSUME es:ExtraData
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

Multiple Data Segments (1 of 2)

```
cseg SEGMENT 'CODE'
ASSUME cs:cseg, ds:data1, es:data2, ss:mystack

main PROC
   mov ax,data1               ; DS points to data1
   mov ds,ax
   mov ax,SEG val2            ; ES points to data2
   mov es,ax
   mov ax,val1                ; data1 segment assumed
   mov bx,val2                ; data2 segment assumed

   mov ax,4C00h               ; (same as .exit)
   int 21h
main ENDP
cseg ENDS
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

Multiple Data Segments (1 of 2)

```
data1 SEGMENT 'DATA'
    val1 WORD 1001h
data1 ENDS

data2 SEGMENT 'DATA'
    val2 WORD 1002h
data2 ENDS

mystack SEGMENT PARA STACK 'STACK'
    BYTE 100h DUP('S')
mystack ENDS

END main
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

Segment Overrides

- A segment override instructs the processor to use a different segment from the default when calculating an effective address
- Syntax:

```
segreg:segname
segname:label
```

```
cseg SEGMENT 'CODE'
ASSUME cs:cseg, ss:mystack

main PROC
    ...
    mov ax,ds:val1
    mov bx,OFFSET AltSeg:var2
```

Combining Segments

- Segments can be merged into a single segment by the linker, if . . .
  - their names are the same,
  - and they both have combine type PUBLIC,
  - . . . even when they appear in different source code modules
- Example:
  - cseg SEGMENT PUBLIC 'CODE'
- See the program in the Examples\ch16\Seg2\ directory

What's Next

- Defining Segments
- **Runtime Program Structure**
- Interrupt Handling
- Hardware Control Using I/O Ports

Runtime Program Structure

- COM Programs
- EXE Programs

When you run a program, . . .

MS-DOS performs the following steps, in order:

1. checks for a matching internal command name
2. looks for a matching file with .COM, .EXE, or .BAT extensions, in that order, in the current directory
3. looks in the first directory in the PATH variable, for .COM, .EXE, and .BAT file
4. continutes to second directory in the PATH, and so on

Program Segment Prefix (PSP)

- 256-byte memory block created when a program is loaded into memory
- contains pointer to Ctrl-Break handler
- contains pointers saved by MS-DOS
- Offset 2Ch: 16-bit segment address of current environment string
- Offset 80h: disk transfer area, and copy of the current MS-DOS command tail

COM Programs

- Unmodified binary image of a program
- PSP created at offset 0 by loader
- Code, data, stack all in the same segment
- Code entry point is at offset 0100h, data follows immediately after code
- Stack located at the end of the segment
- All segments point to base of PSP
- Based on TINY memory model
- Linker uses the /T option
- Can only run under MS-DOS

Sample COM Program

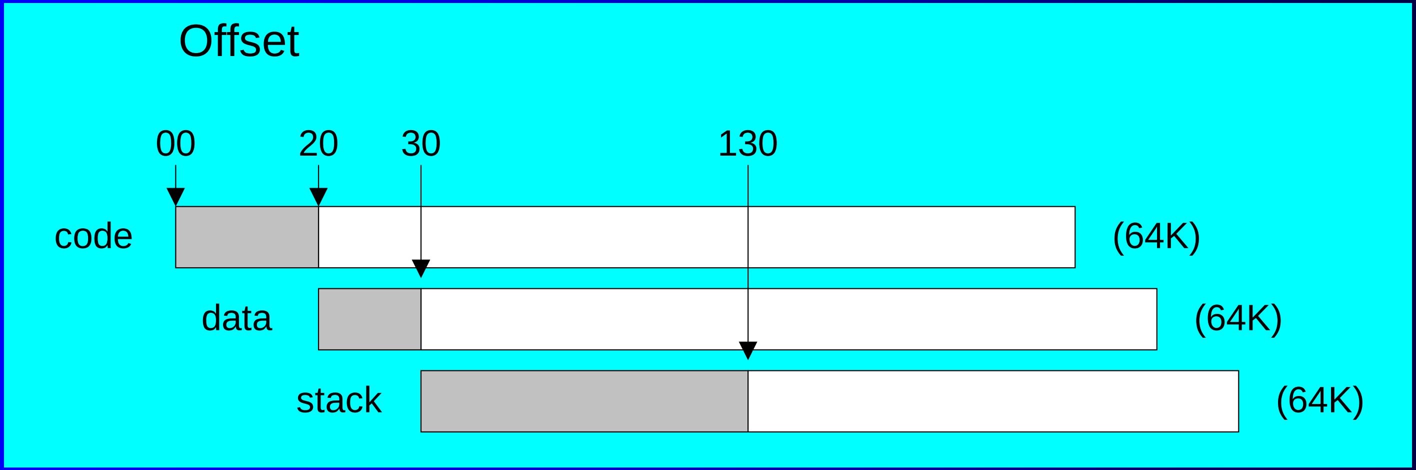
```
TITLE Hello Program in COM format   (HelloCom.asm)

.MODEL tiny
.code
ORG 100h                           ; must be before main
main PROC
    mov  ah,9
    mov  dx,OFFSET hello_message
    int  21h
    mov  ax,4C00h
    int  21h
main ENDP

hello_message BYTE 'Hello, world!',0dh,0ah,'$'

END main
```

EXE Programs

- Use memory more efficiently than COM programs
- Stored on disk in two parts:
  - EXE header record
  - load module (code and data)
- PSP created when loaded into memory
- DS and ES set to the load address
- CS and IP set to code entry point
- SS set to the beginning of the stack segment, and SP set to the stack size

EXE Programs

Sample EXE structure shows overlapping code, data, and stack segments:



EXE Header Record

- A relocation table, containing addresses to be calculated when the program is loaded.
- The file size of the EXE program, measured in 512-byte units.
- Minimum allocation: min number of paragraphs needed above the program.
- Maximum allocation: max number of paragraphs needed above the program.
- Starting IP and SP values.
- Displacement (in paragraphs) of the stack and code segments from the beginning of the load module.
- A checksum of all words in the file, used in catching data errors when loading the program into memory.

What's Next

- Defining Segments
- Runtime Program Structure
- **Interrupt Handling**
- Hardware Control Using I/O Ports

Interrupt Handling

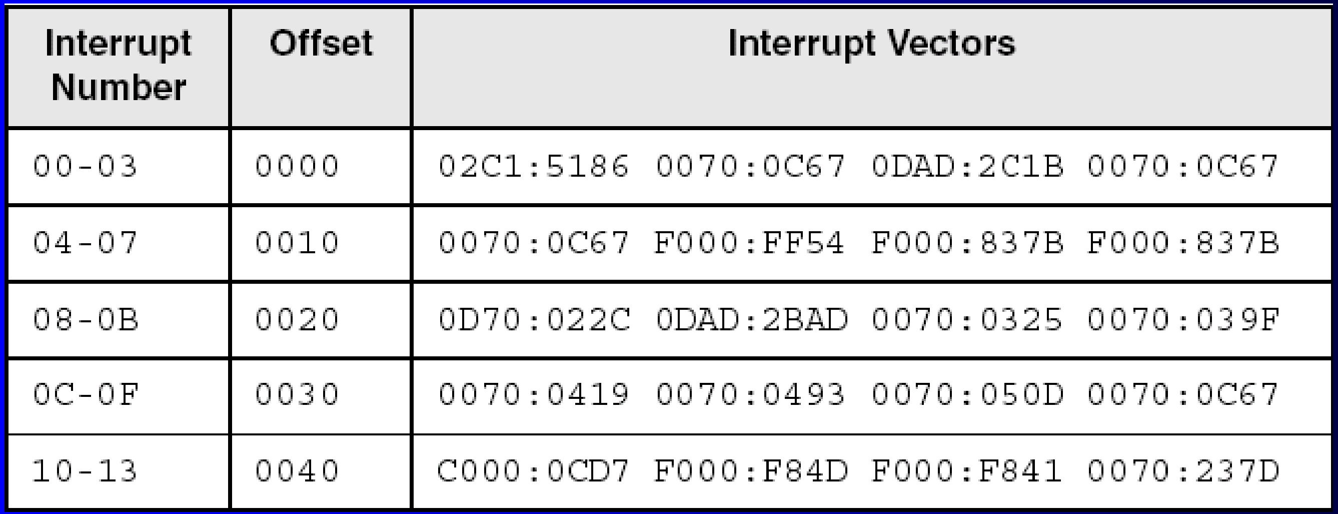
- Overview
- Hardware Interrupts
- Interrupt Control Instructions
- Writing a Custom Interrupt Handler
- Terminate and Stay Resident Programs
- The No_Reset Program

Overview

- Interrupt handler (interrrupt service routine) – performs common I/O tasks
  - can be called as functions
  - can be activated by hardware events
- Examples:
  - video output handler
  - critical error handler
  - keyboard handler
  - divide by zero handler
  - Ctrl-Break handler
  - serial port I/O

Interrupt Vector Table

- Each entry contains a 32-bit segment/offset address that points to an interrupt service routine
- Offset = *interruptNumber* * 4
- The following are only examples:



Interrupt Number	Offset	Interrupt Vectors
00-03	0000	02C1:5186 0070:0C67 0DAD:2C1B 0070:0C67
04-07	0010	0070:0C67 F000:FF54 F000:837B F000:837B
08-0B	0020	0D70:022C 0DAD:2BAD 0070:0325 0070:039F
0C-0F	0030	0070:0419 0070:0493 0070:050D 0070:0C67
10-13	0040	C000:0CD7 F000:F84D F000:F841 0070:237D

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

Hardware Interrupts

- Generated by the Intel 8259 Programmable Interrupt Contoller (PIC)
    - in response to a hardware signal
- Interrupt Request Levels (IRQ)
    - priority-based interrupt scheduler
    - brokers simultaneous interrupt requests
    - prevents low-priority interrupt from interrupting a high-priority interrupt

Common IRQ Assignments

0	System timer
1	Keyboard
2	Programmable Interrupt Controller
3	COM2 (serial)
4	COM1 (serial)
5	LPT2 (printer)
6	Floppy disk controller
7	LPT1 (printer)

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

Common IRQ Assignments

8	CMOS real-time clock
9	modem, video, network, sound, and USB controllers
10	(available)
11	(available)
12	mouse
13	Math coprocessor
14	Hard disk controller
15	(available)

Interrupt Control Instructions

- STI – set interrupt flag
  - enables external interrupts
  - always executed at beginning of an interrupt handler
- CLI – clear interrupt flag
  - disables external interrupts
  - used before critical code sections that cannot be interrupted
  - suspends the system timer

Writing a Custom Interrupt Handler

- Motivations
  - Change the behavior of an existing handler
  - Fix a bug in an existing handler
  - Improve system security by disabling certain keyboard commands
- What's Involved
  - Write a new handler
  - Load it into memory
  - Replace entry in interrupt vector table
  - Chain to existing interrupt hander (usually)

Get Interrupt Vector

- INT 21h Function 35h – Get interrupt vector
  - returns segment-offset addr of handler in ES:BX

```
.data
int9Save LABEL WORD
DWORD ?             ; store old INT 9 address here
.code
mov ah,35h                 ; get interrupt vector
mov al,9                   ; for INT 9
int 21h                    ; call MS-DOS
mov int9Save,BX            ; store the offset
mov [int9Save+2],ES        ; store the segment
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2014.

Set Interrupt Vector

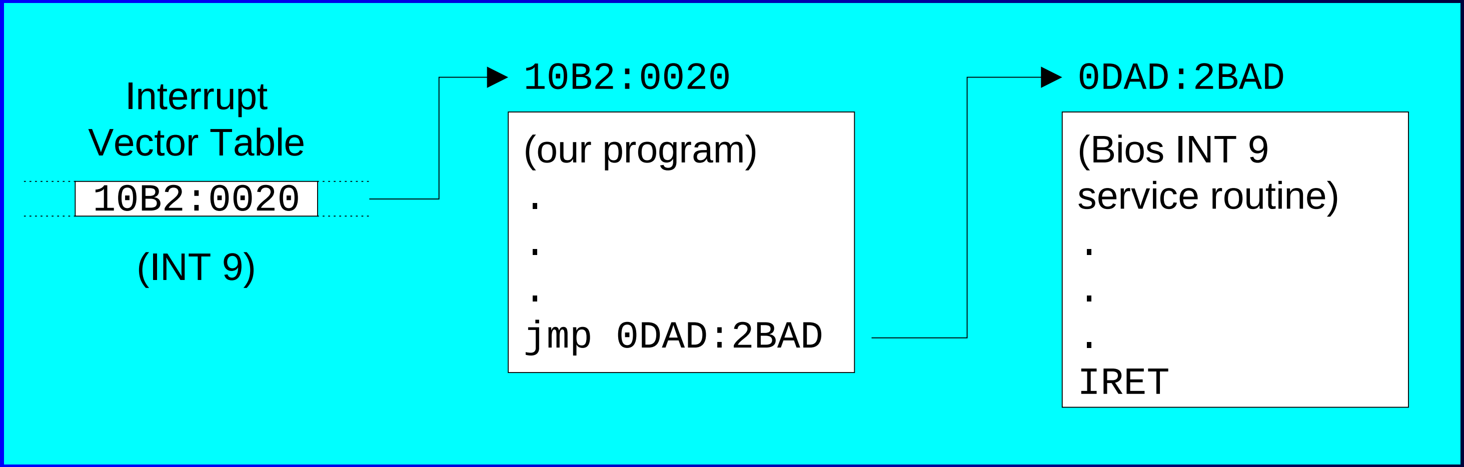
- INT 21h Function 25h – Set interrupt vector
  - installs new interrupt handler, pointed to by DS:DX

```
mov ax,SEG kybd_rtn          ; keyboard handler
mov ds,ax                    ; segment
mov dx,OFFSET kybd_rtn       ; offset
mov ah,25h                   ; set Interrupt vector
mov al,9h                    ; for INT 9h
int 21h
.
.
kybd_rtn PROC                ; (new handler begins here)
```

See the CtrlBrk.asm program.

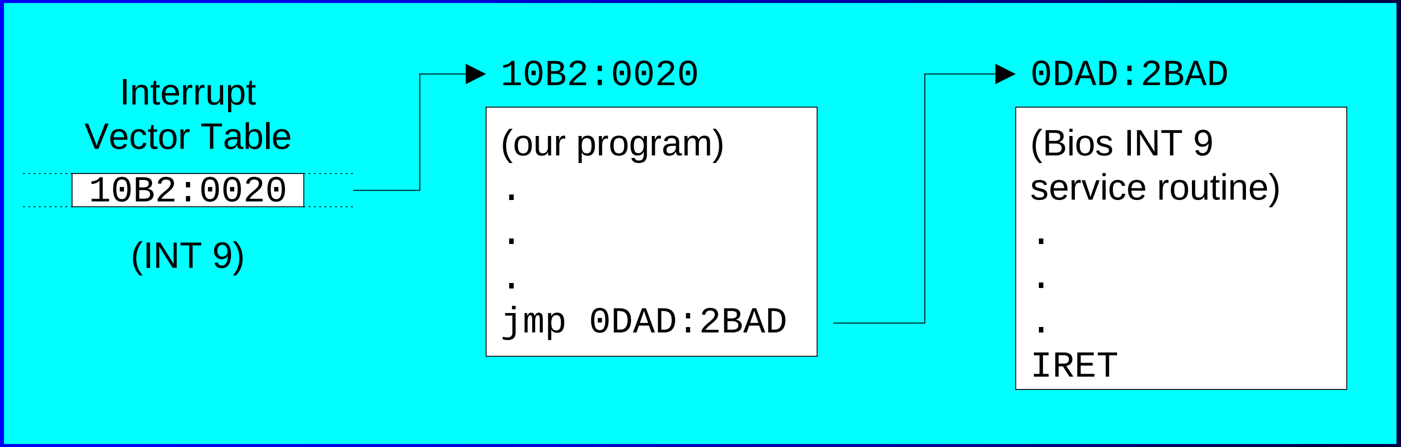
Keyboard Processing Steps

1. Key pressed, byte sent by hardward to keyboard port
2. 8259 controller interrupts the CPU, passing it the interrupt number
3. CPU looks up interrupt vector table entry 9h, branches to the address found there



Keyboard Processing Steps

4. Our handler executes, intercepting the byte sent by the keyboard
5. Our handler jumps to the regular INT 9 handler
6. The INT 9h handler finishes and returns
7. System continues normal processing



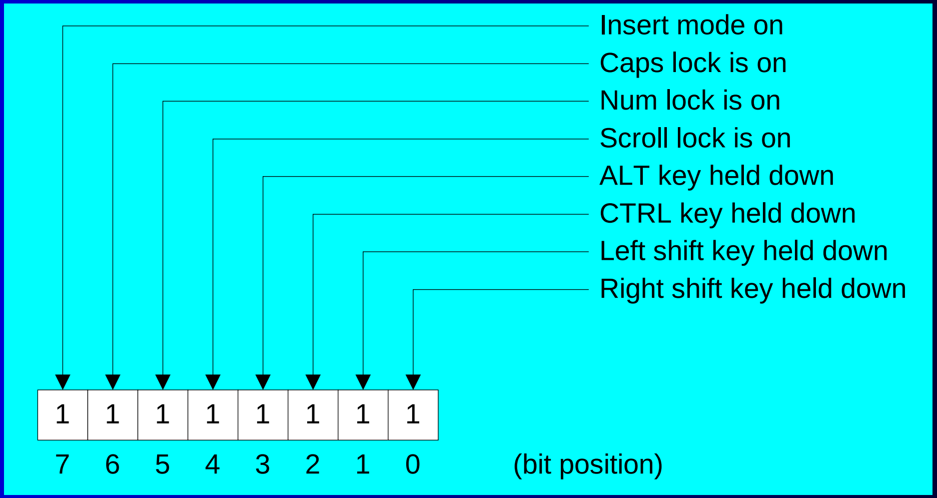
Terminate and Stay Resident Programs

- (TSR): Installed in memory, stays there until removed
  - by a removal program, or by rebooting
- Keyboard example
  - replace the INT 9 vector so it points to our own handler
  - check, or filter certain keystroke combinations, using our handler
  - forward-chain to the existing INT 9 handler to do normal keyboard processing

The No_Reset Program (1 of 5)

- Inspects each incoming key
- If the Del key is received,
  - checks for the Ctrl and Alt keys
  - permits a system reset only if the Right shift key is also held down

The keyboard status byte indicates the current state of special keys:



Insert mode on
Caps lock is on
Num lock is on
Scroll lock is on
ALT key held down
CTRL key held down
Left shift key held down
Right shift key held down

1	1	1	1	1	1	1	1	
7	6	5	4	3	2	1	0	(bit position)

The No_Reset Program (2 of 5)

- View the source code
- Resident program begins with:

```
int9_handler PROC FAR
    sti                         ; enable hardware interrupts
    pushf                       ; save regs & flags
    push  es
    push  ax
    push  di
```

The No_Reset Program (3 of 5)

- Locate the keyboard flag byte and copy into AH:

```
L1: mov   ax,40h                   ; DOS data segment is at 40h
    mov   es,ax
    mov   di,17h                   ; location of keyboard flag
    mov   ah,es:[di]               ; copy keyboard flag into AH
```

- Check to see if the Ctrl and Alt keys are held down:

```
L2: test   ah,ctrl_key               ; Ctrl key held down?
    jz     L5                        ; no: exit
    test   ah,alt_key                ; ALT key held down?
    jz     L5                        ; no: exit
```

The No_Reset Program (4 of 5)

- Test for the Del and Right shift keys:

```
L3: in     al,kybd_port          ; read keyboard port
    cmp    al,del_key            ; Del key pressed?
    jne    L5                    ; no: exit
    test   ah,rt_shift           ; right shift key pressed?
    jnz    L5                    ; yes: allow system reset
```

- Turn off the Ctrl key and write the keyboard flag byte back to memory:

```
L4: and    ah,NOT ctrl_key       ; turn off bit for CTRL
    mov    es:[di],ah            ; store keyboard_flag
```

The No_Reset Program (5 of 5)

- Pop the flags and registers off the stack and execute a far jump to the existing BIOS INT 9h routine:

```
L5: pop    di                     ; restore regs & flags
    pop    ax
    pop    es
    popf
    jmp    cs:[old_interrupt9]   ; jump to INT 9 routine
```

What's Next

- Defining Segments
- Runtime Program Structure
- Interrupt Handling
- **Hardware Control Using I/O Ports**

Hardware Control Using I/O Ports

- Two types of hardware I/O
  - memory mapped
    - program and hardware device share the same memory address, as if it were a variable
  - port based
    - data written to port using the OUT instruction
    - data read from port using the IN instruction

Input-Ouput Ports

- ports numbered from 0 to FFFFh
- keyboard controller chip sends 8-bit scan code to port 60h
    - triggers a hardware interrupt 9
- IN and OUT instructions:

```
IN accumulator, port
OUT port, accumulator
```

    - accumulator is AL, AX, or EAX
    - port is a constant between 0 and FFh, or a value in DX betweeen 0 and FFFFh

PC Sound Program

- Generates sound through speaker
- speaker control port: 61h
- Intel 8255 Programmable Peripheral Interface chip turns the speaker on and off
- Intel 8253 Timer chip controls the frequency
- Source code

Summary

- Explicit segment definitions used often in custom code libraries
- Directives: SEGMENT, ENDS, ASSUME
- Transient programs
- Program segment prefix (PSP)
- Interrupt handlers, interrupt vector table
- Hardware interrupt, 8259 Programmable Interrupt Controller, interrupt flag
- Terminate and Stay Resident (TSR)
- Memory-mapped and port-based I/O

The End