Memory Organization

- ✓ The assembler uses two basic formats for developing software. One method uses **memory models** and the other uses **full-segment definitions**. MASM uses memory models. The TASM also uses memory models, but they differ somewhat from the MASM models
- ✓ The full-segment definitions are common to most assemblers, including Intel assembler, and are often used for software development. The models are easier to use for simple task
- ✓ There are many models available to the MASM assembler from tiny to huge

Example 4-18: Write a program to copy the contents of a 100-byte block of memory (LISTA) into a second 100-byte block of memory(LISTB)

```
.MODEL SMALL                        ; Small model
.STACK 100H                         ; define stack
.DATA                               ; start data segment
        LISTA DB 100 DUP(?)
        LISTB DB 100 DUP(?)
.CODE                               ; start code segment
HERE:   MOV AX,@DATA                ; load ES and DS
        MOV ES,AX
        MOV DS,AX
        CLD                         ; move data
        MOV SI,OFFSET LISTA
        MOV DI,OFFSET LISTB
        MOV CX,100
        REP MOVSB
.EXIT 0                             ; exit to DOS
END HERE
```

- ✓ The program shows how to define the stack, data, and code
- ✓ The .EXIT 0 directive returns to DOS with an error code of 0 (no error)
- ✓ @DATA is a special directive used to identify various segments
- ✓ If the .STARTUP directive is used (MASM version 6.X), the MOV AX,@DATA followed by MOV DS,AX segments can be eliminated
- ✓ .STARTUP directive also eliminates the need to store the starting address next to the END label

Full-Segment Definitions

```
STACK_SEG  SEGMENT ' STACK'
           DW 100H DUP(?)
STACK_SEG  ENDS
DATA_SEG   SEGMENT 'DATA'
           LISTA DB 100  DUP(?)
           LISTB DB 100 DUP(?)
DATA_SEG   ENDS
CODE_SEG   SEGMENT  'CODE'
ASSUME     CS:CODE_SEG, DS:DATA_SEG
ASSUME     SS:STACK_SEG
MAIN PROC   FAR
           MOV AX,DATA_SEG
           MOV ES,AX
           MOV DS,AX
           CLD
           MOV SI,OFFSET LISTA
           MOV DI,OFFSET LISTB
           MOV CX,100
           REP MOVSB
           MOV AH,4CH
           INT 21H
MAIN       ENDP
CODE_SEG   ENDS
           END MAIN
```

- ✓ Example 4-19 appears longer than the one pictured in Example 4-18
- ✓ Bit it is more structured
- ✓ The first segment defined is the STACK_SEG, which is clearly described with the SEGMENT and ENDS directive
- ✓ Within these directives, DW 100H DUP (?) sets aside 100H words for the stack segment. Because the word STACK appears next to SEGMENT, the assembler and linker automatically load stack register (SS) and stack pointer (SP)
- ✓ The data are defined in the DATA_SEG
- ✓ Here, two arrays of data appear as LISTA and LISTB
- ✓ Each array contains 100 bytes of space for the program
- ✓ The names of the segments in this program can be changed to any name
- ✓ CODE_SEG is organized as a far procedure
- ✓ Before the program begins, the code segment contains ASSUME statement
- ✓ ASSUME statement tells the assembler and linker that the names used for the code segment (CS) is CODE_SEG, it also tells the assembler and linker that the data segment is DATA_SEG and stack segment is STACK_SEG

Description

- ✓ After the program loads both the extra segment register and data segment register with the location of the data segment, it transfers 100 bytes from LISTA to LISTB
- ✓ Following this is a sequence of two instructions that return control back to DOS (Disk Operating system)
- ✓ Note that the program loader does not automatically initialize DS and ES. These registers must be loaded with the program
- ✓ The last statement in the program is END MAIN
- ✓ MAIN (i.e procedure starts label) is needed after END

A Sample Program

Example 4-20: Write a program using full-definitions, that reads a character from the keyboard and displays it on the CRT screen. Note that @ key ends the program

Note: This program illustrates the use of a few DOS function calls. The BIOS function calls allow the use of the keyboard, printer, disk drives, and everything else that is available in your computer system

```
CODE_SEG    SEGMENT 'CODE'
ASSUME      CS:CODE_SEG
MAIN        PROC FAR
            MOV AH,06H          ;reads a key
            MOV DL,0FFH
            INT 21H
            JE MAIN             ; if no key typed
            CMP AL,'@'
            JE MAIN1            ; if an @ key
            MOV AH,06H          ; display key (echo)
            MOV DL,AL
            INT 21H
            JMP MAIN            ; repeat
MAIN1:
            MOV AH, 4CH
            INT 21H
            MAIN ENDP
CODE_SEG    ENDS
            END MAIN
```

Is equivalent to .EXIT

Description

- ✓ This example program uses only a code segment because there is no data
- ✓ A stack segment should appear, but it has been left out because DOS automatically allocates a 128-byte stack for all programs
- ✓ The only time that the stack is used in this example is for the INT 21H instructions that call a procedure in DOS
- ✓ The stack is fewer than 128 bytes in this example
- ✓ The program uses a DOS functions 06H and 4CH. The function number is placed in AH before the INT 21H instruction executes. The 06H function reads the keyboard if DL = 0FFH, or displays the ASCII contents of DL if it is not 0FFH
  The first section of the program moves 06H into AH and 0FFH into DL, so that a key is read from the keyboard. The INT 21H tests the keyboard. If no key is typed, it returns equal. The JE instruction tests equal condition and jumps to MAIN if no key is typed
- ✓ When a key is typed, the program continues to the next step, which compares the contents of AL with an @symbol
- ✓ Upon return from the INT 21H, the ASCII character of the typed key is found in AL. In this program, if an @ symbol is typed, the program ends
- ✓ If the @ symbol is not typed, the program continues by displaying the character typed on the keyboard with the next INT 21H instruction
- ✓ The second INT 21H moves the ASCII character into DL so it can be displayed on the CRT screen
- ✓ If the @ symbol is typed, the program continues at MAIN1, where it executes the DOS function code number 4CH. This causes the program to return to the DOS prompt so that the computer can be used for other tasks

Example 4-21: Same problem using memory model

```
.MODEL TINY
.CODE
.STARTUP
MAIN:   MOV AH,6            ; read a key
        MOV DL,0FFH
        INT 21H
        JE MAIN             ; if no key typed
        CMP AL, '@'
        JE MAIN1            ; if an @ key
        MOV AH, 06H         ; display key (echo)
        MOV DL,AL
        INT 21H
        JMP MAIN            ; repeat
MAIN1:
.EXIT                       ; exit to DOS
END
```

Structure

Structure of a main module using simplified directive

```
.MODEL SMALL        ;This statement is required before you can use other simplified segment directives
.STACK              ; use default 1-Kilobyte stack (128 bytes)

.DATA               ; begin data segment

                    ; place data declarations here

.CODE               ; begin code segment
.STARTUP            ; generate start-up code

                    ; place instructions here

.EXIT               ; generate exit code
END
```

Note:
If you do not use **.STARTUP,** you must give starting address as an argument to the **END** directive

For Example:

```
.CODE
START:

  ; place executable code here


END START
```

Segment Order Directives

You can control the order in which segments appear in the executable program with three directives

.SEQ

Arranges segments in the order in which you declare them

.ALPHA

Directive specifies alphabetical segment ordering within a module

.DOSSEG

Directive specifies the MS-DOS segment-ordering convention. It places segments in the standard order required by Microsoft Language

The **.DOSSEG** directive orders segments as follows

i) Code Segment

ii) Data Segment

Initializing DS and SS

- ✓ The DS register is automatically initialized to the correct value if you use .STARTUP. If you do not use .STARTUP with the MS-DOS, you must initialize DS using the following instructions

```
MOV AX,@DATA
MOV DS,AX
```

- ✓ The SS and SP registers are initialized automatically if you use the .STACK directive. If you want SS to be equal to DS, use .STARTUP

Levels of Programming

Machine Level Programming

The language in which the instructions are represented by binary codes is called machine language.

Features

- ✓ The machine language programs developed for one processor cannot be used for another processor.
- ✓ The machine level programs are machine dependent. It is highly tedious for a programmer to write programs in the machine language

Assembly Level Programming

In ALP, instructions are written using mnemonics. If the program is developed using mnemonics then it is called ALP

Features

- ✓ µP cannot execute the assembly language programs directly. The assembly language programs have to be converted to machine language for execution. This conversion is performed using a software called assembler.
- ✓ The mnemonics of one processor will not be same as that of another processor. The ALPs are machine dependent

High Level Programming

In high level programming the instructions will be in the form of statements written using symbols, English words and phrases

Features

- ✓ The programs written in high level languages are easy to understand and machine independent
- ✓ A high level language program has to be converted into machine language programs in order to be executed by the processor. The conversion is performed by a software called compiler

Types of Assembler

- ✓ One-pass assembler
- ✓ Two-pass assembler
- ✓ Macro assembler
- ✓ Resident assembler and
- ✓ Meta assembler

The assembler usually generates 2 output files called

1. Object file and (".OBJ")
2. List file (".LIST")

   Any labels encountered are given an address and stored in a table by an assembler

One-pass assembler:

A one-pass assembler is an assembler in which the source codes are processed only once. A one-pass assembler is very fast and in one-pass assembler only backward reference may be used

Two-pass assembler:

The source codes are processed two times. In the first pass, the assembler assigns addresses to all the labels and attach values to all the variables used in the program. In the second pass, it converts the source code into machine code.

NEXT CLASS

DOS Function Call

THANK YOU