Hashemite University

Faculty of Engineering and Technology

Computer Engineering Department

Microprocessors Lab Manual

Prepared By: Eng. Ezya Khader & Eng. Sara Al-shaer

Table of Content

Lab Rules	*3*
Guidelines for Writing Lab report	*4*
Review of the System Commands and Assembly Instruction Set	*6*
Developing Assembly Language Programs and Executing using Emu8086 and MASM	*22*
Programming Techniques	*28*
BIOS Interrupts Programming	*33*
DOS & Mouse Interrupts Programming	*40*
Parallel Data Input/output	*45*
I/O Applications: Dynamic Display	*48*
LCD & Keypad Interrupts	*50*
Design and conduct an Experiment	*52*
Appendix A: Communicating with MTS-8088 KIT	*54*

Lab Rules

- **General Rules:**
  - Be PUNCTUAL for your laboratory session.
  - Foods, drinks and smoking are NOT allowed.
  - The lab timetable must be strictly followed. Prior permission from the Lab Supervisor must be obtained if any change is to be made.
  - Experiment must be completed within the given time.

- **Instruments:**
  - Be careful in dealing with the lab apparatus.
  - Don't change the place of any instrument or any of its cables.
  - The main equipment that you will use in the lab experiments is the MTS-8088 trainer kit; check your program carefully before implementing it in order to avoid harming or damaging the kit ICs.
  - Don't touch or remove the kit ICs.
  - When you connect interfaces cards to the kit, make sure that you use the right connector in its right location.
  - If you face any problem or you have any question about the instruments operations please refer to the laboratory supervisor.
  - At the end of the lab turn-off all the instruments.

- **Lab Reports:**
  - Lab work is very important to you as an engineer, it gives you some practical experience that you will need in your future work. Also it increases your understanding of the theoretical material that you have taken at class.
  - Writing lab reports with high quality is also important to you, it develops your ability to express your ideas, work, and your observations to other people in the field.
  - You must follow the following sequence in organizing your lab report .

Lab Report Items

Cover Page
Objectives
Theoretical Background
Equipment
Procedure
Results
Conclusions

  - Be organized in your lab report, and be clear in expressing your theoretical and experimental data.
  - Make sure to include all the required and important data and items within your report to make it comprehensive and self contained.
  - Copies will take zero grades.
  - Report submission will be due to the next lab. Only one day late is allowed with 25% deduction.

Guidelines for Writing Lab Report

The most effective way to acquire the practical skills in engineering studies is usually by experimenting in a laboratory. The process of experimentation involves organization, observation, familiarization with various pieces of equipment, working with others, writing, and communicating ideas and information. These are the skills required of an engineer.

In a practical situation, such as that in the industry or university research, experiments are designed for the purpose of clarifying research questions or conflicting theories by means of collecting a series of data. The conclusions drawn from that data can be used to validate a theory or sometimes to develop a theory that explains the behavior of an engineering object. The report for this kind of experiments must includes an introduction to the topic and purpose of the experiment, the theory, method, procedure, equipment used in the experiment, the data presented in an organized manner, and the conclusions based on the data gathered.

In engineering education, lab experiments are usually designed to enhance the understanding in engineering topics. Students are supposed to "dirty their hand" in preparing the experiment setup, organize the experiment flow, and learn to observe the salient features as well as to spot any unexpected occurrence as part of the training to acquire the practical skill to become an engineer. Although the introduction and the procedure are usually given in the lab handouts, students should practice writing a proper lab report which includes all the necessary sections, targeting at a reader who does not have any prior knowledge about the experiment. This is to develop the skill in documenting the laboratory work and communicating that experience to others. This write-up gives some guidelines on what to write in each section in preparing laboratory reports for engineering curricula.

Title Page

The title page should contain the title of the experiment, the code and title of the course, the name of the writer, the date when the experimental work was performed, submission date of the lab report, and the name of lecturer for whom the report is prepared for.

Introduction or Objectives

An introduction is necessary to give an overview of the overall topic and the purpose of the report. The motivation to the initialization of the experimental work can be included. Its content should be general enough to orientate the reader gracefully into the subject materials.

Theoretical Background

This section is to discuss the theoretical aspects leading to the experiment. Typically, this involves the historical background of the theories published in the research literature and the questions or ambiguities arose in these theoretical work. Citations for the sources of information should be given in one of the standard bibliographic formats (for example, using square brackets with the corresponding number [2] that points to the List of References). Explore this background to prepare the readers to read the main body of the report. It should contain sufficient materials to enable the readers to understand why the set of data are collected, and what are the salient features to observe in the graph, charts and tables presented in the later sections.

Depending on the length and complexity of the report, the introduction and the theoretical background may be combined into one introductory section.

Experimental Method, Procedure and Equipment

This section describes the approach and the equipment used to conduct the experiment. It explains the function of each apparatus and how the configuration works to perform a

particular measurement. Students should not recopy the procedures of the experiment from the lab handout, but to summarize and explain the methodology in a few paragraphs.

Observations, Data, Findings, Results

The data should be organized and presented in the forms of graphs, charts, or tables in this section, without interpretive discussion. Raw data which may take up a few pages, and most probably won't interest any reader, could be placed in the appendices.

Calculations and Analysis

The interpretation of the data gathered can be discussed in this section. Sample calculations may be included to show the correlation between the theory and the measurement results. If there exists any discrepancy between the theoretical and experimental results, an analysis or discussion should follow to explain the possible sources of error.

The experimental data and the discussions may also be combined into one section, for example, under the heading called "Discussion of Experimental Results".

Conclusions

The conclusions section closes the report by providing a summary to the content in the report. It indicates what is shown by the experimental work, what is its significance, and what are the advantages and limitations of the information presented. The potential applications of the results and recommendations for future work may be included.

Appendices

The appendices are used to present derivations of formulae, computer program source codes, raw data, and other related information that supports the topic of the report.

List of References

The sources of information are usually arranged and numbered according to the order they are cited in the report. The reference materials may be entered in the following formats:

[1] Author, "Title of the book", 2nd edition, New York: Publisher, 1989.
[2] Author, "Title of the paper", Journal name, Vol. 2, No. 3, Jan 1990, pg. 456-458.
[3] Author, "Title of the paper", Proceedings of Conference 1991, pg. 5-6.
[4] Author, "Title of the thesis", Ph.D. thesis, RiceUniversity, Houston, May 1973.

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 1:
Review of the System Commands and Assembly Instruction Set

1.1 Objectives:
1. Be familiar with the software structure of MTS-8088 kit.
2. To utilize and apply the system commands of MTS-8088 system.
3. Calculate the physical address of storage locations in the memory address space.
4. Assemble instructions into the memory and Disassemble machine code stored in memory.

1.2 Pre-lab Preparation:
- Read the experiment thoroughly before coming to the lab.
- Download the Dosbox and try all the examples on the experiment.

1.3 Equipments:
- Personal computer.

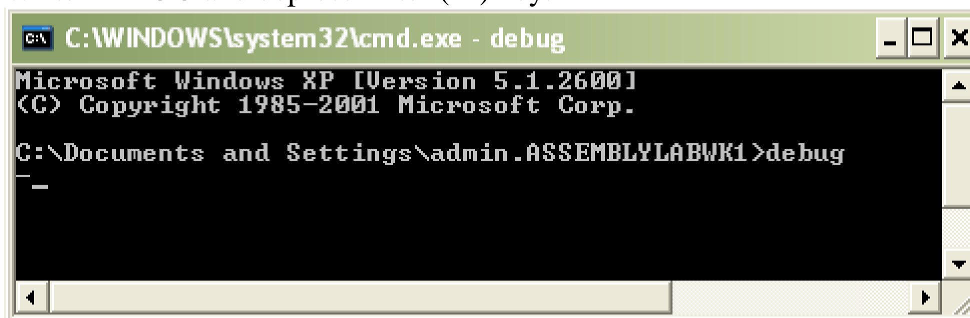
Introduction: How to enter Debug?

a) Working on 32 bit operating system:

To load DEBUG program from DOS:

Start ⟶ programs ⟶ Run ⟶ write cmd then depress Enter (↵) key.

Write DEBUG and depress Enter (↵) key:



Prompt (-) is then displayed.

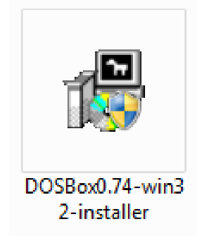
All we need to do is type in the command and then depress Enter (↵) key. These debug commands are the tools a programmer needs to enter, execute, and debug programs. At the completion of the operation, results are displayed on the screen and the DEBUG prompt (−) is redisplayed. The PC waits in this state for a new entry.

b) Working on 64 bit operating system:

You have to install the DOSBox 0.7 program.

DOSBox is a program that emulates the functions of MS-DOS, including sound, graphics, input, and networking.

Step 1:
Download the appropriate version of the program for your operating system, then Install the DOSBox 0.7setup on your computer.



Step 2:
Make a folder for you to use it when you need to run MS-DOS commands.
Create your folder on D:\ drive for example then name it 8086

Step 3:
For our lab you have to put all 8086 assembler commands inside this folder. (See figure 1)

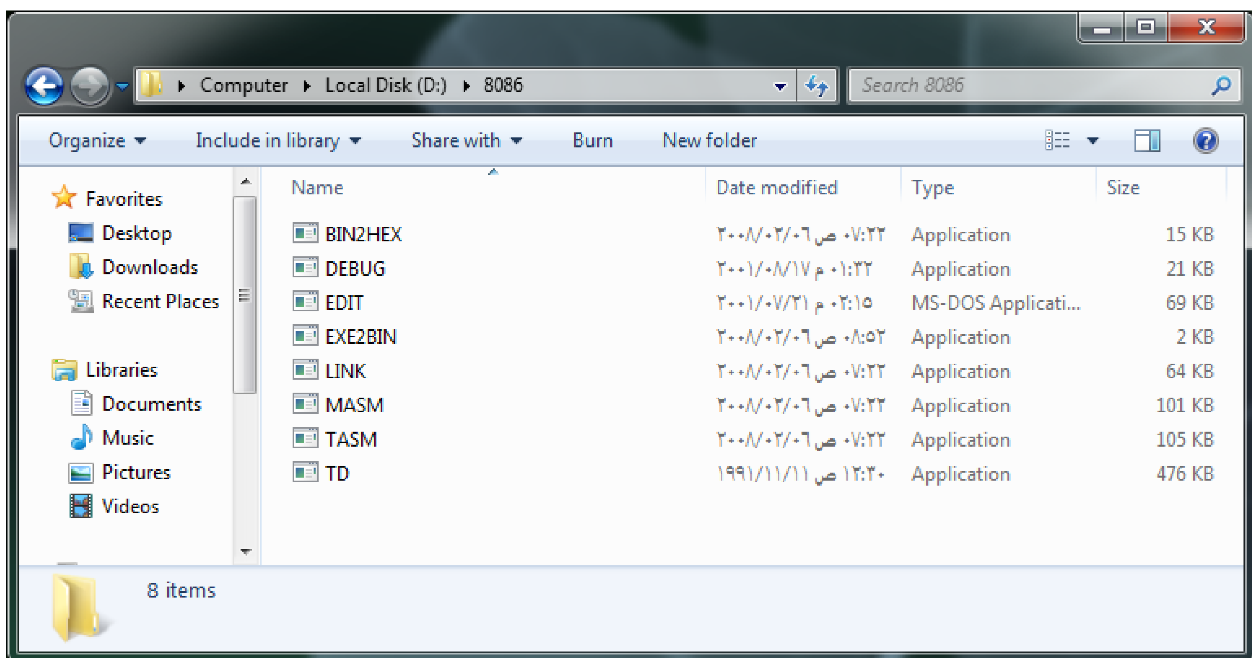


Figure 1

Step 4:
Click on DOSBox 0.74 to run the program.

Two black screens will be appeared, but we will use the below one. (Figure 2)

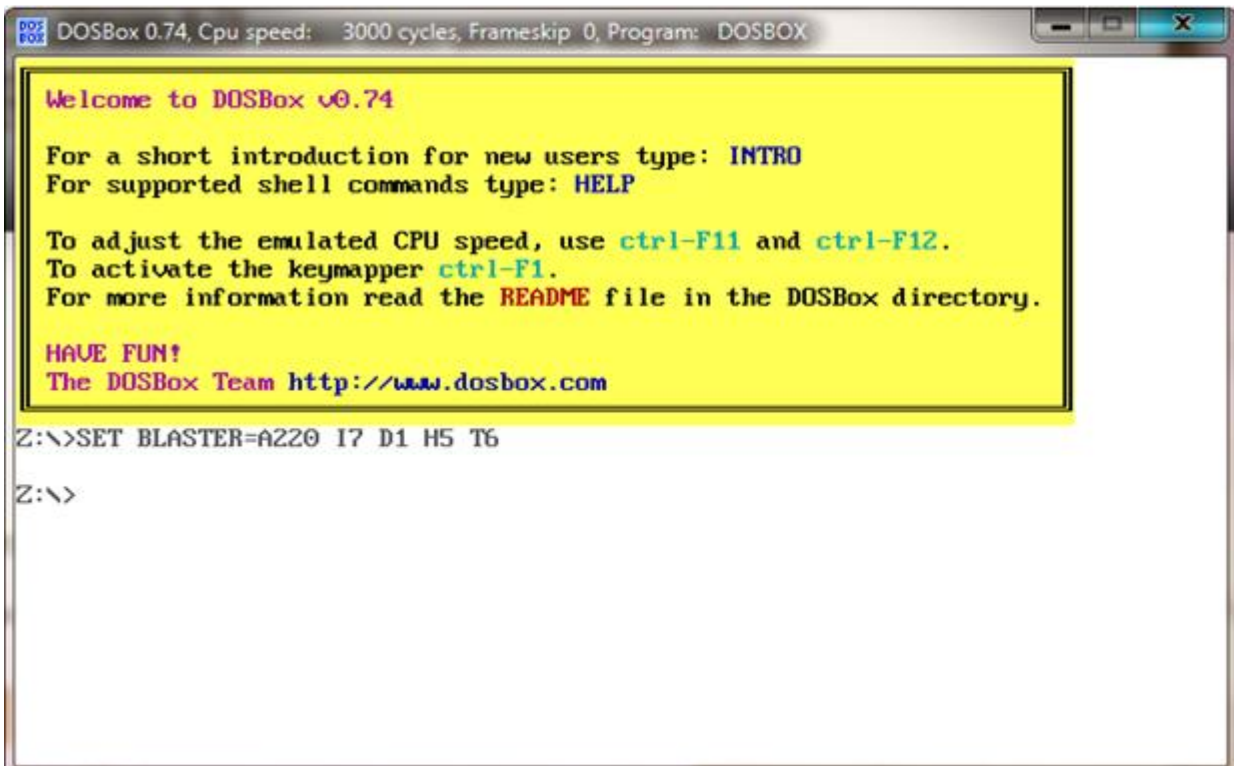


Figure 2

Step 5:
Mount the **D:\8086** directory, by writing the following commands. (See figure 3)

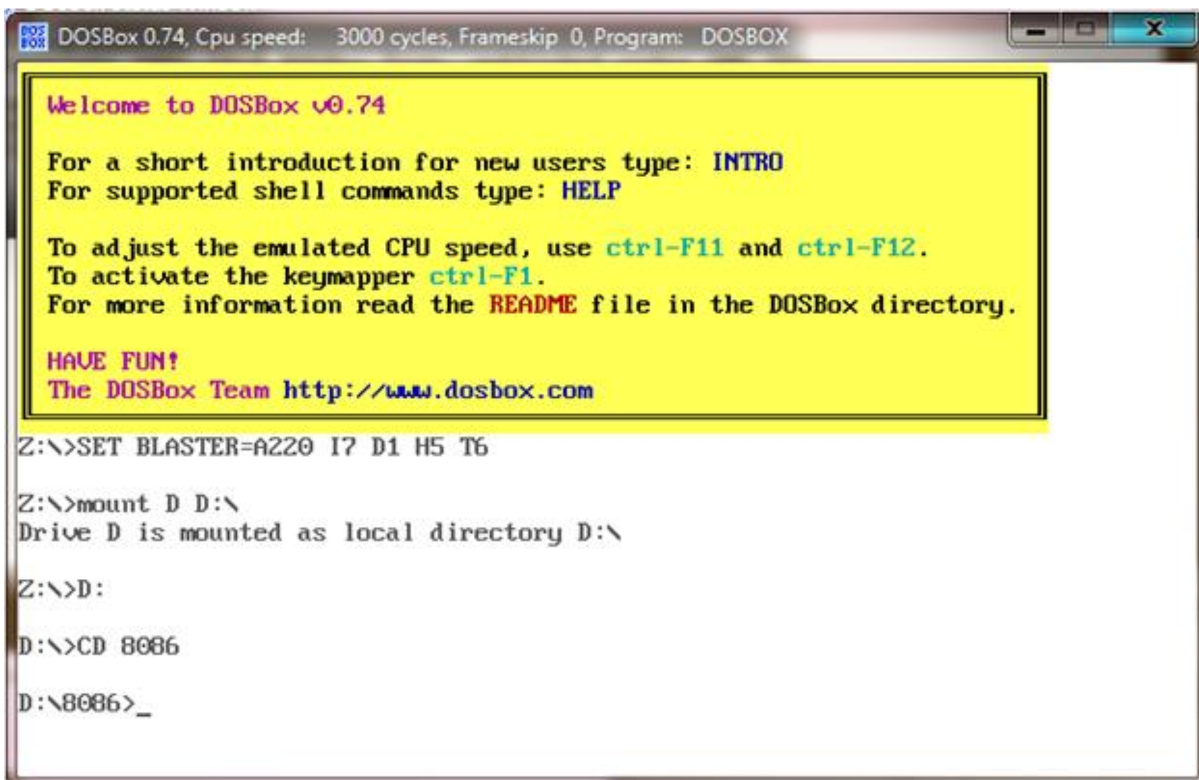


Figure 3

Step 6:
Now you can use all the command inside this folder just by typing it on this black window then press Enter.
For example to use the debug just write debug then press Enter. (See figure 4)

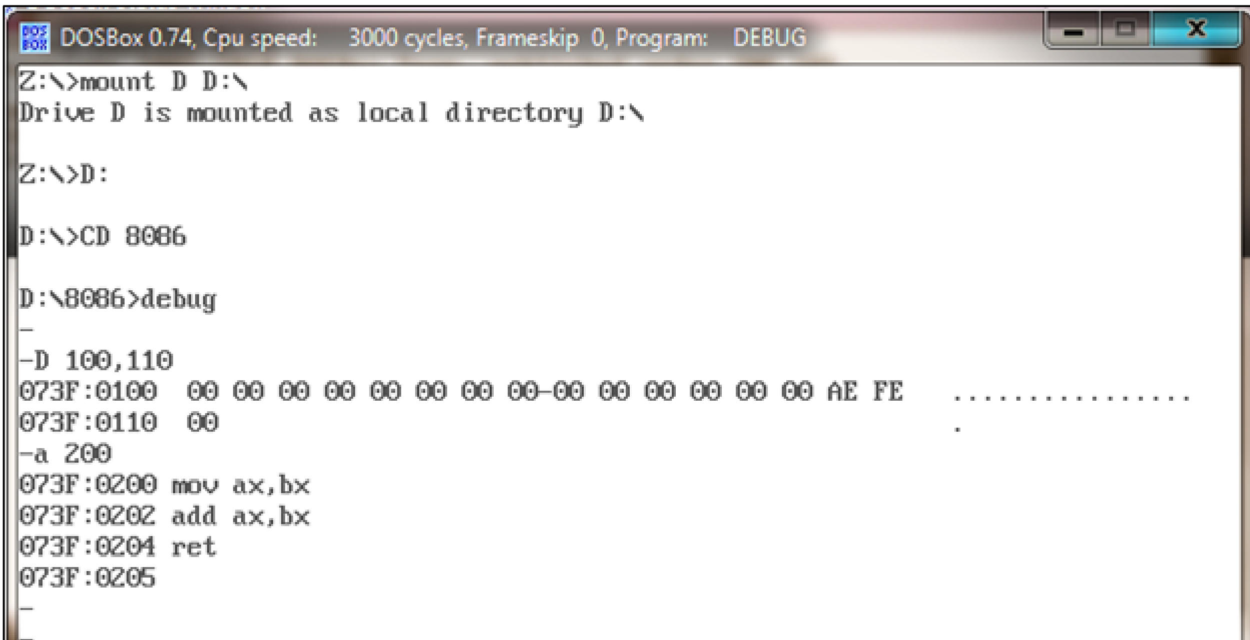


Figure4

First: System Commands

Table 1 lists the most of the system commands used in the MTS-8088. The 8088 System has two command groups. These are the system commands and the I/O Driver commands. The system commands provide ways to utilize the system's resources, and the I/O Driver Commands are used to control I/O devices.

Command	Description
	Memory Management Commands
D	Display the contents of Memory
C	Compare the contents of Memory
E	Edit/Modify the memory contents
F	Fill memory
M	Move the contents of memory
S	Search the memory
Assembler Commands	
A	Command A is used to write an assembly language program.
U	Disassemble the assembly language instructions into machine code
Program Control Commands	
G	Executing Programs
R	Display / modify the contents of registers.
T	Trace the program execution

Table 1: System commands.

Memory Management Commands:

D (Dump)

The D command displays memory on the screen as single bytes in both hexadecimal and ASCII.

Command formats:

D

D ***address***

D ***range***

If no address or range is given, the location begins where the last D command left off, or at location DS: 00 if the command is being typed for the first time. If *address* is specified, it consists of either a segment-offset address or just a 16-bit offset. *Range* consists of the beginning and ending addresses to dump.

Example	Description
D F000:0	Segment-offset
D ES:100	Segment register-offset
D 100	Offset

The default segment is **DS**, so the segment value may be left out unless you want to dump and offset from another segment location. A range may be given, telling Debug to dump all bytes within the range.

D 150, 15A (dump DS: 0150 through 015A, means 11 locations will be displayed)

Other segment registers or absolute addresses may be used, as following examples show:

Example	Description
D	Dump 128 bytes from the last referenced location
D SS:0,5	Dump the bytes at offset 0 to 5 from SS
D 915:0	Dump 128 bytes at offset zero from segment 0915H
D 0,200	Dump offsets 0-200 from DS

Memory Dump Example:

The following figure shows an example of a memory dump. The numbers at the left are the segment and offset address of the first byte in each line, the next 16 pairs of digits are the hexadecimal contents of each byte, the characters to the right are the ASCII representation of each byte**.**

```
-
-
-D 200
073F:0200  20 74 68 69 73 20 69 73-20 6D 61 6E 79 20 63 68    this is many ch
073F:0210  61 72 61 63 74 65 72 73-20 73 74 6F 72 65 64 20   aracters stored
073F:0220  69 6E 20 74 68 65 20 64-61 74 61 20 73 65 67 6D   in the data segm
073F:0230  65 6E 74 00 00 00 00 00-00 00 00 00 00 00 00 00   ent.............
073F:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
073F:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
073F:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
073F:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

E (Enter/Edit)

The **E** command places individual bytes in memory, you must supply a starting memory location where the values will be stored. If only an offset value is entered then the offset is assumed to be from **DS**, otherwise a 32 bit address may be entered or another segment register may be used. Command formats are:

E *address*	enter new byte value at *address*
E *address, list*	replace the contents of one or more bytes starting at the specified *address* with the values contained in the *list*.

Examples:

To begin entering hexadecimal or character data at DS: 100, type:

E 100

Press the space bar to advance to the next byte, and press Enter key to stop.

To enter a string into memory starting at location ES: 500, type:

E ES: 500,"this is a string values"

```
D:\8086>debug
-
-E 100
073F:0100  00.55   00.55   00.66   00.77   00.

-D 100,104
073F:0100  55 55 66 77 00                                    UUfw.
-
-E ES:500,"this is a string values"
-D ES:500,518
073F:0500  74 68 69 73 20 69 73 20-61 20 73 74 72 69 6E 67   this is a string
073F:0510  20 76 61 6C 75 65 73 00-00                         values..
-
-E 10,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
-
-D 10,1F
073F:0010  01 02 03 04 05 06 07 08-09 10 11 12 13 14 15 16   ................
-_
```

F (Fill)

The fill command fills a range of memory with a single value or list of values, the range must be specified as two offset addresses or segment-offset addresses. Command format:

F *range, list*

Here are some examples;

Example	Description
F 100,500,' '	Fill locations 100 through 500 with spaces
F ES:300,320,FF	Fill locations ES:300 through ES:320 with value FFh
F 20,30,BB	Fill locations DS:20 through DS:30 with value BBh
F 20,40,CC,DD,EE,11	Fill locations DS:20 through ds:40 with values CC,DD,EE,11

```
D:\8086>debug
-F ES:300,320,FF
-D ES:300 320
073F:0300  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
073F:0310  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
073F:0320  FF                                                .
-
-F 20 30 BB
-D 20 30
073F:0020  BB BB BB BB BB BB BB BB-BB BB BB BB BB BB BB BB   ................
073F:0030  BB                                                .
-F 20,40,CC,DD,EE,11
-D 20,40
073F:0020  CC DD EE 11 CC DD EE 11-CC DD EE 11 CC DD EE 11   ................
073F:0030  CC DD EE 11 CC DD EE 11-CC DD EE 11 CC DD EE 11   ................
073F:0040  CC                                                .
-
-
```

C (Compare)

The C command compares bytes between a specified ranges with the same number of bytes at a target address. Command format:

C ***range address***

For example, the bytes between DS: 0100 and DS: 0105 are compared to the bytes at DS: 0200:

C 100,105,200

The debug will display the **different values** between the two blocks.

```
-d 100,105
073F:0100  B4 02 B2 41 CD 21
-d 200,205
073F:0200  00 00 00 00 00 00
-c 100,105,200
073F:0100  B4  00  073F:0200
073F:0101  02  00  073F:0201
073F:0102  B2  00  073F:0202
073F:0103  41  00  073F:0203
073F:0104  CD  00  073F:0204
073F:0105  21  00  073F:0205
-
```

M (Move)

The M command copies a block of data from one memory location to another. The command format is:

M ***range address***

Range consists of the starting and ending addresses of the bytes to be copied. *Address* is the target location to which the data will be copied. All offsets are assumed to be from DS unless specified otherwise.
Examples:

Example	Description
M 100,105,110	Move bytes in the range DS:100-105 to location DS:110
M ES:100,105,110	Same as above, except that all offsets are relative to the segment value in ES

Sample string Move:
The following example uses the M command to copy the string 'ABCDEF' from offset 100h to 106h. First the string is stored at location 100h; then memory is dumped, showing the string, next we move (copy) the string to offset 106h and dump offsets 100h-10Bh:

```
-E 100,"ABCDEF"
-D 100,105
073F:0100  41 42 43 44 45 46                                  ABCDEF
-M 100,105,106
-D 100,10B
073F:0100  41 42 43 44 45 46 41 42-43 44 45 46                ABCDEFABCDEF
-
```

S (Search)

The S command searches a range of addresses for a sequence of one or more bytes the command format is:

S ***range list***

```
-
-E 100 "COPYCOPY"
-S 100,105,'C'
073F:0100
073F:0104
-S100,10A,"COP"
073F:0100
073F:0104
-
```

Here are some examples:

Example	Description
S 100,1000, 0D	Search DS:100 to DS:1000 for the value 0Dh
S 100,1000,CD,20	Search for the sequence CD,20
S 100,9FFF,"COPY"	Search for the word "COPY"

Assembler Commands

A (Assemble)
Assemble a program into machine language. Command formats:

A
A ***address***

If only the offset portion of*address* is supplied, it is assumed to be an offset from CS.

Here are examples:

Example	Description
A 100	Assemble at CS:100H
A	Assemble from current location
A DS:2000	Assemble at DS:2000H

When you press Enter at the end of each line, debug prompts you for the next line of inputs. Each input line starts with a segment –offset address. To terminate input, press the enter key on a blank line.

For example:

```
-a 100
073F:0100 mov ah,2
073F:0102 mov dl,41
073F:0104 int 21
073F:0106
-
```

U (Unassemble)

The U command translate memory into assembly language machine code. This is also called disassembling memory, if you don't supply an address debug disassembles from the location where the last U command left off. If the command is used for the first time after loading debug memory is unassembled from location CS: 100. Command formats are:

U
U *startaddr*
U *startaddrendaddr*

Where the *startaddr* is the starting point and *endaddr* is the ending address. Examples:

Example	Description
U	Disassemble the next 32 bytes.
U 0	Disassemble 32 bytes at CS:0
U 100,108	Disassemble the bytes from CS:100 to CS:108

Program Control Commands

G (Go)

Execute the program in memory.
Command formats:
G
G *breakpoint*
G= *startAddr breakpoint*
G= *startAddr breakpoint1 breakpoint2 …*

Breakpoint is a 16 or 32 bit address at which the processor should stop, and startAddr is an optional starting address for the processor, if no breakpoints are specified the program runs until it stop by itself and returns to debug, up to 10 breakpoints may be specified on the same command line. Examples:

Example	Description
G	Execute from the current location to the end of the program
G 50	Execute from the current location and stop before the instruction at offset CS:50
G= 10,50	Begin execution at CS:10 and stop before the instruction at offset CS:50

<u>T (Trace)</u>

The T command executes one or more instructions at either the current CS:IP location or at optional address. The contents of the registers are shown after each instruction is executed. The command formats are:

T
T ***count***
T ***= address, count***

Where count is the number of instructions to trace, and address is the starting address for the trace.

Examples:

Example	Description
T	Trace the next instruction
T 5	Trace the next five instructions.
T=105,10	Trace 16 instructions starting at CS:105

<u>R (Register)</u>

The **R** command may be used to do any of the following:

1. Display the contents of one register, and allowing it to be changed.
2. Display all registers, flags, and the next instruction about to be executed.
3. Display all eight flag setting, and allowing any or all of them to changed

Command formats:

R

R ***register***

Here are some examples:

Example	Description
R	Display the contents of all registers.
R IP	Display the contents of IP and prompt for a new value.
R CX	Display the contents of CX and prompt for a new value.
R F	Display all flags and prompt for a new flag value.

Once the **R F** command has displayed the flags, you can change any single flag by typing its new state. For example: we set the Zero flag by typing the following two command:

R F [Press Enter] ZR

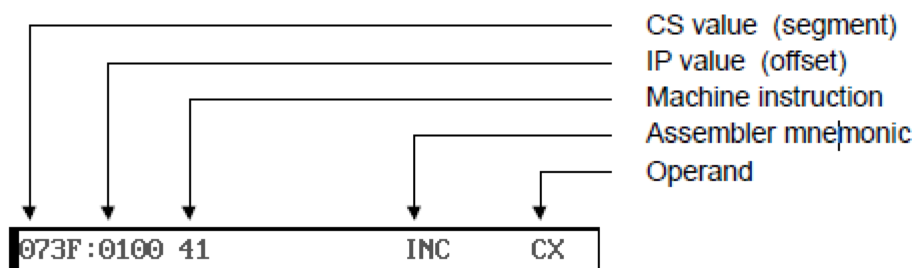
The following is a sample register display (all values in hexadecimal):

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 41            INC     CX
-
```

The complete set of possible flag mnemonics in Debug (ordered from left to right) are as follows:

Set	*Clear*
OV = Overflow	NV = No Overflow
DN = Direction Down	UP = Direction Up
EI = Interrupts Enabled	DI = Interrupts Disabled
NG = Sign Flag negative	PL = Sign Flag positive
ZR = Zero	NZ = Not Zero
AC = Auxiliary Carry	NA = No Auxiliary Carry
PO = Odd Parity	PE = Even Parity
CY = Carry	NC = No Carry

The **R** command also displays the next instruction to be executed:



Q (Quit)

The Q command quits Debug and returns to DOS.

Second: Addressing Modes;

The addressing modes are categorized into the categories:-

- Register Operand Addressing Mode.
- Immediate Operand Addressing Mode.
- Direct Addressing Modes.
- Register Indirect Addressing.
- Based-Index Addressing Mode.
- *Register Relative Addressing.*
- Based relative-plus-index addressing.

Third: Instruction Set:

Arithmetic Instructions

Instruction	Operation
ADD operand1,operand2	Operand1=operand1 + operand2
SUB operand1,operand2	Operand1=operand1- operand2
MUL operand Operand must be a REG or MEM	When operand is a **byte**: AX = AL * operand. when operand is a **word**: (DX AX) = AX * operand
DIV operand Operand must be a REG or MEM	when operand is a **byte**: AL = AX / operand AH = remainder (modulus) when operand is a **word**: AX = (DX AX) / operand DX = remainder (modulus)

Data Transfer Instructions:

Data movement instructions include:

1- General Purpose instructions group:	MOV
2- Stack instructions group:	PUSH, POP
4- String Instructions group:	LODS, STOS, MOVS

➔ General Purpose Instructions Group

MOV: Copy Data from source to destination. Source contents are not affected.
Format: MOV Destination, Source
Operation: (S) → (D)

Example:
MOV Ax, 00FE

➔ Stack Instructions Group

There are two typical instructions that are used with stack. They are:

A- **PUSH:** push word into stack
Format: PUSH (16-bit register)
Operation: 1) SP = SP – 2
2) (16-bit register) → (SS:SP)

B- **POP:** Pop word of the stack to destination register.
Format: POP (16-bit register)
Operation: 1) (SS:SP) → (16-bit register)
2) SP = SP + 2

Examples:

Before Executions	*Instructions*	*After Executions*

SP = B3F
SS = 0040
SS:B3A = 00
SS:B3B = 00
SS:B3C = 00
SS:B3D = 00
SS:B3E = 00
SS:B3F = 00
	Mov CX, 88AA
	Mov DX,BB99
	PUSH CX
	PUSH DX
		SP = B3B
		SS = 0040
		SS:B3A = 00
		SS:B3B = 99
		SS:B3C = BB
		SS:B3D = AA
		SS:B3E = 88
		SS:B3F = 00

SP = B3B SS = 0040 SS:B3A = 00 SS:B3B = 22 SS:B3C = 66 SS:B3D = CC SS:B3E = 88 SS:B3F = 00	POP BX POP wo[80]	SP = B3F SS = 0040 SS:B3A = 00 SS:B3B = 22 SS:B3C = 66 SS:B3D = CC SS:B3E = 88 SS:B3F = 00 BX = 6622 DS:80 = CC DS:81 = 88

➔ String Instructions Group

By String we mean a series of data words (or bytes) that reside in consecutive memory locations. The string instructions of the 8088 permit a programmer to implement operations such as moving data from one block of memory to another block elsewhere in memory. Associated with string operation, the direction flag that is explained below.

The Direction Flag

The Direction flag (D) (located in the flag register) selects the auto-increment (D = 0) or the auto decrement (D = 1) operation for the DI and SI registers during string operations. The Direction flag is used only with the String operations. The CLD instruction clears the D flag (D = 0), The STD instruction sets the D flag (D = 1)

A- **LODS**

The LODS instruction loads AL or AX with data stored at the data segment offset address indexed by the SI register. After loading AL with a byte or AX with a word, the contents of SI increment, if D = 0 or the contents of SI decrement, if D = 1.

Before Execution	*Instructions*	*After Execution*
Df = 0	LODSB	AL = DS:[SI] SI = SI + 1
Df = 1	LODSB	AL = DS:[SI] SI = SI – 1
Df = 0	LODSW	AL = DS:[SI] AH = DS:[SI+1] SI = SI + 2
Df = 1	LODSW	AL = DS:[SI] AH = DS:[SI+1] SI = SI - 2

B- **STOS**

The STOS instruction stores AL or AX at the extra segment memory locations addressed by the DI register.

Before Execution	*Instructions*	*After Execution*
Df = 0	STOSB	ES:[DI] = AL DI = DI + 1
Df = 1	STOSB	ES:[DI] = AL DI = DI – 1
Df = 0	STOSW	ES:[DI] = AL ES:[DI+1] = AH DI = DI + 2
Df = 1	STOSW	ES:[DI] = AL ES:[DI+1] = AH DI = DI - 2

C- **MOVS**

The instruction MOVS transfers data from one memory location to another. This is the only memory-to-memory transfer allowed in the 8088.

The MOVS instruction transfers a byte or word from the data segment location addressed by SI to the extra segment location addressed by DI.

The pointers (SI & DI) then increment or decrement the value as dictated by the direction flag.

Before Execution	*Instructions*	*After Execution*
Df = 0	MOVSB	ES:[DI] = DS:[SI] DI = DI + 1 SI = SI + 1
Df = 1	MOVSB	ES:[DI] = DS:[SI] DI = DI - 1

		SI = SI – 1
Df = 0	MOVSW	ES:[DI] = DS:[SI] ES:[DI+1] = DS:[SI+1] DI = DI + 2 SI = SI + 2
Df = 1	MOVSW	ES:[DI] = DS:[SI] ES:[DI+1] = DS:[SI+1] DI = DI – 2 SI = SI - 2

REPEAT

In most applications, the basic string operations must be repeated in order to process arrays of data. This is done by inserting a repeat prefix 'REP' before the instruction that is to be repeated. ***'REP'***, causes the basic operation to be repeated until the contents of register CX become equal to 0. Each time the instruction is executed, it causes CX to be tested for 0. if CX is found not to be 0, it is decremented by 1 and the basic string operation is repeated. ***'REP'*** is useful when used with the instructions: MOVS and STOS.

Examples

LODS

Before Execution	*Instructions*	*After Execution*
DS:7F = C2 DS:80 = 30 DS:81 = 24 DS:82 = 8B DS:83 = 71 DS:84 = FC DS:85 = 90 DS:86 = A0 DS:87 = F9	CLD MOV SI,80 LODSB	Df = 0 SI = 81 AL = 30
	CLD MOV SI,83 LODSW	Df = 0 SI = 85 AX = FC71

	STD
	MOV SI,83
	LODSW
	MOV DX,AX
	LODSW
		Df = 1
		SI = 7F
		DX = FC71
		AX = 8B24

STOS

Instructions	*After Execution*

CLD
MOV DI,53
MOV AL, 56
STOSB
	ES:53 = 56
	DI = 54
	Df = 0

CLD
MOV DI,70
MOV AX, 927C
STOSW
	ES:70 = 7C
	ES:71 = 92
	DI = 72
	Df = 0

STD
MOV DI, 94
MOV AL,33
STOSB
	ES:94 = 33
	DI = 93
	Df = 1

STD
MOV DI, 62
MOV AX,5D7F
STOSW
	ES:62 = 7F
	ES:63 = 5D
	DI = 60
	Df = 1

CLD
MOV DI,90
MOV AL,F3
MOV CX,4
REP STOSB
	ES:90 = F3
	ES:91 = F3
	ES:92 = F3
	ES:93 = F3
	DI = 94
	Df = 0

<table>
<tr><td>STD
MOV DI,88
MOV AX,749A
MOV CX,3
REP STOSW</td><td>ES:84 = 9A
ES:85 = 74
ES:86 = 9A
ES:87 = 74
ES:88 = 9A
ES:89 = 74
DI = 82
Df = 1</td></tr>
</table>

MOVS

<table>
<tr><th>Before Executions</th><th>Instructions</th><th>After Executions</th></tr>
<tr><td rowspan="5">DS:7E = 14
DS:7F = C2
DS:80 = 30
DS:81 = 24
DS:82 = 8B
DS:83 = 71
DS:84 = FC
DS:85 = 90
DS:86 = A0
DS:87 = F9
DS:88 = 8C</td><td>CLD
MOV SI,82
MOV DI,60
MOVSB</td><td>Df = 0
SI = 83
DI = 61
ES:60 = 8B</td></tr>
<tr><td>CLD
MOV SI,84
MOV DI,5B
MOVSB
MOVSB</td><td>Df = 0
SI = 86
DI = 5D
ES:5B = FC
ES:5C = 90</td></tr>
<tr><td>STD
MOV SI,86
MOV DI,9B
MOVSW</td><td>Df = 1
SI = 84
DI = 99
ES:9B = A0
ES:9C = F9</td></tr>
<tr><td>CLD
MOV SI,7F
MOV DI,6F
MOV CX,4
REP MOVSW</td><td>Df = 0
SI = 87
DI = 77
ES:6F = C2
ES:70 = 30
ES:71 = 24
ES:72 = 8B
ES:73 = 71
ES:74 = FC
ES:75 = 90
ES:76 = A0</td></tr>
<tr><td>STD
MOV SI,87
MOV DI,9B
MOV CX,3
REP MOVSW</td><td>Df = 1
SI = 81
DI = 95
ES:97 = 71
ES:98 = FC
ES:99 = 90
ES:9A = A0
ES:9B = F9
ES:9C = 8C</td></tr>
</table>

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 2
Developing Assembly Language Programs and Executing using Emu8086 and MASM

1.1 Objectives:

1. Assemble and execute instructions into the memory using Emu8086
2. Use MASM to identify then correct syntax errors and assemble a source program into object code.
3. Edit an existing source program.
4. Explore the listing file and identify its different parts.
5. Make a run module with the LINK program then load and execute it with the DEBUG program.
6. Learn how to write assembly programs using simplified and full segment definitions

1.2 Pre-lab Preparation:

- Read the experiment thoroughly.

1.3 Equipments:

- Personal computer with MASM 6.11 software installed on it.
- MTS-8088 kit.

1.4 Theoretical background:

The Microsoft Assembler package, MASM, is a programming environment that contains two major tools: the assembler/linker and the CodeView debugger. The assembler/linker translates x86 instructions to machine code and produces an ".exe" file that can be executed under DOS. The CodeView tool is an enhanced version of DEBUG with a graphical interface that also handles 32 bit instructions.

<u>First: MASM 6.11:</u>

➔ **Assemble the Source File(.ASM file):**

After writing the source code in .asm file you can assemble it using the MASM software to do that; do the following steps:

1) Suppose we have exp2.asm file saved in the following directory: (D:\8086)
2) Open DOSboxthen go to the path (D:\8086).

```
Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>MOUNT D D:\
Drive D is mounted as local directory D:\

Z:\>D:

D:\>CD 8086

D:\8086>_
```

3) Use MASM/L command to assemble the source file **exp2.ASM** into a machine language object file **exp2.OBJ**.

```
D:\8086>MASM/L EXP2.ASM
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.

 Invoking: ML.EXE /I. /Zm /c /Fl /Ta EXP2.ASM

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: EXP2.ASM

D:\8086>
```

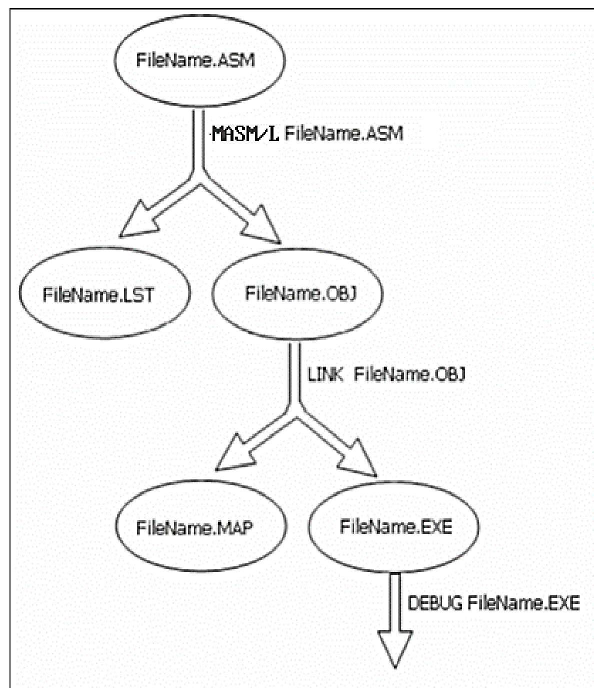
Note: MASM checks the source file for syntax errors. If it finds any, a short description of the errors will be displayed with the line number of each.
Now the files **exp2.OBJ** will be created.
Before feeding the ".OBJ" file into **LINK**, all syntax errors produced by the assembler must be corrected.
The ".lst" file which is optional is very useful to the programmer because it lists all the opcodes and offset addresses as well as errors that MASM detected.

4) LINK program with MASM version 6.11 take one or more object files and combine them into a single executable file (.EXE file).

```
D:\8086>LINK EXP2.OBJ

Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.

Run File [EXP2.exe]: EXP2.EXE
List File [nul.map]: EXP2.MAP
Libraries [.lib]:
Definitions File [nul.def]:
LINK : warning L4021: no stack segment

D:\8086>
```

Now the files **exp2.EXE** and **exp2.MAP** will be created.
The **".MAP"** file which is optional gives the name of each segment, where it starts, where it stops, and its size in bytes.
So when there are many segments for code or data there is a need to see where each is located and how many bytes are used by each.

We can summaries the above steps as in the beside figure:



Second: Emu8086

Emulator has a complete 8086 instruction set, press on HELP

• Complete 8086 instruction set

Third: Developing Assembly Language Programs

➔ **Data Definition**

Using assembly language the user can define different types of data with different sizes. Data allocation directives (DB, DW, DD…….) are used to declare variables as in the following examples:

Var1 DB 15 : Defines a variable called Var1 and initializes it with value 15
Var2 DB 12, 13, 15 : Defines an array of bytes called Var2, Var2 [0] =12, Var2 [1] =13

- An error occurs if the value assigned is greater than memory size allocated to it

VAR3 DB 270 : error
VAR4 DW 270 : correct

- Variables of type character can be defined by placing data between single quotations

STR1 DB 'A' ; STR1 ='A', the ASCII code of 'A'
STR2 DB '8' ; STR2='8', the ASCII code of '8'

- String variables can be defined by enclosing the string between single quotations

STR3 DB 'Hello to assembly language laboratory', STR3 [0] ='H' STR3 [1] = 'e'

➔ **Illegal instructions**

In assembly language there are some illegal operations that the user should avoid to make sure that the assembler will not generate errors:

1- Segment to Segment data transfer

MOV DS, ES; illegal
MOV DS, CS; illegal

2- Code segment (CS) can not be a destination operand

MOV CS, AX ; illegal

3- Write an immediate value to a segment

MOV ES, 1422h ; illegal

4- Instructions that have operands with different sizes

VAR DB 5
MOV AX, VAR ; error

➔ **Directives vs. Instructions.**

Directives:
1. Use Directives to tell assembler what to do
2. Commands that are recognized and acted upon by the assembler
   - Not part of the Intel instruction set
   - Used to declare code, data areas, select memory model, declare procedures, etc.
3. Different assemblers have different directives
   - NASM != MASM, for example.

Instructions:
1. Use Instructions to tell CPU what to do.
2. Assembled into machine code by assembler
3. Executed at runtime by the CPU
4. Member of the Intel IA-32,IA-16 instruction set

➔ **Segment Definition**

There are two ways to define segments in assembly

- Simplified segment Definition
- Full segment Definition

Simplified segment Definition

The following example shows the structure of a main module using simplified segment directives:

```
.MODEL memorymodel        ; It is required before you can use other simplified segment
                          directives
  .STACK                  ; Use default 1-kilobyte stack

  .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
```

➔ **.MODEL Directive**

The **.MODEL** directive defines the attributes that affect the entire module: memory model, default calling and naming conventions, operating system, and stack type.
You must place **.MODEL** in your source file before any other simplified segment directive. The syntax is: **.MODEL** *memorymodel*
The *memorymodel* field is required and must appear immediately after the **.MODEL** directive
The following list summarizes the *memorymodel* field.

Memory model	No. of code segments	No. of data segments	Data and code combined	Segment size
Tiny	1		Yes	64k
Compact	1	Many	No	64k
Small	1	1	No	64k
Medium	Many	1	No	64k
Large	Many	Many	No	64k
Flat	1		Yes	32 bit OS 4G

➔ Starting and Ending Code with .STARTUP and .EXIT

The easiest way to begin and end an MS-DOS program is to use the **.STARTUP** and **.EXIT** directives in the main module. The main module contains the starting point and usually the termination point.
These directives make MS-DOS programs easy to maintain. They automatically generate code appropriate to the stack distance specified with **.MODEL**. However, they do not apply to flat-model programs written for 32-bit operating systems.
To start a program, place the **.STARTUP** directive where you want execution to begin. Usually, this location immediately follows the **.CODE** directive:

```
.CODE
.STARTUP
            ;Place executable code here
.EXIT
END
```

Full Segment Definitions

If you need complete control over segments, you can fully define the segments in your program.

➔ Defining Segments with the SEGMENT Directive

A defined segment begins with the **SEGMENT** directive and ends with the **ENDS** directive:

Name **SEGMENT**

Name **ENDS**

➔ The ASSUME Directive

Recall that all memory addresses have two components: a segment address and an offset Address. Furthermore, every label in an assembly-language program *(with the single Exception of labels used before the SEGMENT directive*) represents some offset address from a segment address.
But *which* segment address?
Look at the data segment block named **MyData:**

```
; BEGIN DATA SEGMENT
MyData SEGMENT
Eatl DB "Eat at Joe's!"."$"       ; Strings are terminated by "$"
CRLF DB 0DH, 0AH,'$'                       ;for printing by DOS
MyData ENDS
; END DATA SEGMENT
```

Everything between the two directives SEGMENT and ENDS is the program's *data segment. There is nothing in this segment definition to tell the assembler that it is a data segment.* You can define variables in the code segment or in the stack segment if you want, even though it's customary and more correct programming practice to keep variables in the data segment.
Segment MyData could be just as easily considered a code segment, though not a stack segment. We have the problem of indicating to the assembler which segment is the data segment. This might seem like an easy one, but rather than a single problem it is actually two problems: one is that the assembler needs to know which segment address to put into the Data Segment (**DS**) register; and the other problem is which form of memory-addressing machine instructions to use.
The first problem is easily addressed:

```
MOV AX,MyData        ; Set up our own data segment address in DS
```

```
MOV DS,AX                ;Can't load segment reg. directly from memory
```
MyData,

If you recall, segment registers (ES, DS, SS), contain the segment address of a segment defined using the SEGMENT and ENDS directives. That address is first loaded into AX, and then from AX the address is loaded into DS. This roundabout path is necessary because *the DS register cannot be loaded with either immediate data or memory data; it must be loaded from one of the other registers*.

The end result is that the segment address represented by the label MyData is loaded into DS. This neatly solves the first problem of specifying the address of the data segment.

We simply load the data segment's address into DS. Now MyData can be considered a real data segment because its segment address is in the data segment register, DS.

That, however, doesn't solve the second problem. Although we wrote two instructions that moved the address of our data segment into DS, *the assembler doesn't "know" that this move took place.* Never forget that the assembler follows its orders without understanding them. It doesn't make inferences based on what you do to addresses or the segment registers. It must be *told* which segment is to be used as the data segment, the code segment, and the stack segment. Somewhere inside the assembler program is a little table where the assembler "remembers" that segment MyData is to be considered the data segment, and that segment MyCode is to be considered the code segment, and that segment MyStack is to be considered the stack segment. It can't remember these relationships, however, unless you first tell the assembler what they are somehow. This somehow (for the data, code, and extra segments, at least) is the ASSUME directive.

Why is this important? It has to do with the way the assembler creates the binary opcodes for a given instruction. When you write an instruction that addresses memory data like this:

MOV AX,Eat1

The assembler must put together the series of binary values that will direct the CPU to perform this action. What that series of binary values turns out to be depends on what segment the label Eat1 resides in. If Eat1 is in the *data segment*, the binary opcodes will be one thing, but if Eat1 resides in the *code segment, stack segment, or extra segment*, the binary opcodes will be something else again. The assembler must know whether any *label indicates an address* within the *data segment, code segment, stack segment, or extra segment*. The assembler knows that Eat1 indicates an address within the segment MyData, but you must tell the assembler that MyData is in fact the data segment.

The Syntax of the ASSUME Directive is:

ASSUME CS:MyCode, DS:MyData, SS:MyStack

Example :

The following code shows how to define the same program using the simplified and full segments definitions:

SIMPLIFIED SEGMENT DEFINITION	FULL SEGMENT DEFINITION

.MODEL SMALL
.STACK 64
 	
.DATA
N1 DW 1432H
N2 DW 4365H
SUM DW 0H
 	
.CODE
.STARTUP
 	
MOV AX,N1
ADD AX,N2
MOV SUM,AX
.EXIT
 	
END
	MyStack SEGMENT
	DB 64 DUP (?)
	MyStack ENDS
	MyData SEGMENT
	N1 DW 1432H
	N2 DW 4365H
	SUM DW 0H
	MyData ENDS
	MyCode SEGMENT
	ASSUME CS:MyCode, DS:MyData, SS:MyStack
	MOV AX,MyData
	MOV DS,AX
	MOV AX,N1
	ADD AX,N2
	MOV SUM,AX
	MOV AH,4C
	INT 21
	MyCode ENDS

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 3
Programming Techniques

1.1 Objectives:

1. Define subroutines .
2. Define macros by coding its definition directives, write assembly language instructions to invoke macros.

1.2 Equipments:

- Personal computer with MASM software installed on it.
- MTS-8088 kit.

1.3 Theoretical background:

First: Subroutines

A *subroutine (also known as procedure)* is a special segment of program that can be called for execution from any point in a program. The subroutine is written to provide a function that may be performed at various points in the main program. The process of transferring control from the main program to a subroutine and return control back to the main program is achieved by what is known as *subroutine handling instructions*.

➔ **Subroutine Handling Instruction:**

There are two basic instructions for subroutine handling: the call (**CALL**) and ret (**RET**) instructions. Together they provide the mechanism for calling a subroutine into operation and returning control back to the main program.

a- CALL
Format: CALL Operand
Operation: PUSH IP
IP = operand (16 bit)
Flags affected: None

b- RET
Format: RET or RET Operand
Operation: POP IP
SP = SP + Operand
Flags affected: None

```
.MODEL SMALL
.DATA
ARR  DB 0,1,1,2,3,5,8,13,21
SUM1 DB ?
.CODE
MOV AX,@DATA
MOV DS,AX
CALL SUM
MOV SUM1,BL
JMP FINISH
SUM PROC
LEA SI,ARR
MOV BX,00
MOV CX,9
L1:ADD BL,[SI]
   INC SI
   LOOP L1
RET
SUM ENDP

FINISH:
END
```

Second: Macros

A macro is a group of instructions that perform one task, just as a procedure performs one task. The difference is that a procedure is accessed via a CALL instruction, while a macro, and all the instructions defined in the macro, is inserted in the program at the point of usage. Macro sequences execute faster than procedures because there are no CALL and RET instructions to execute. The instructions of the macro are placed in your program by the assembler at the point they are invoked.

The **MACRO** and **ENDM** directives delineate a macro sequence. The first statement of a macro is the MACRO instruction, which contains the name of the macro and any parameters associated with it.

If a macro is expanded more than once in a program and there is a label in the label field of the body of the macro, these labels must be declared as LOCAL. Otherwise, an assembler error would be generatedwhen the same label was encountered in two or more places. The directive LOCAL is used to declare a local label. It must appear right after the MACRO directive, before comments and the body of the macro. The LOCAL directive can be used to declare all names and labels at once.

```
.MODEL SMALL
.DATA
N1 DB 12
N2 DB 5
.CODE
.STARTUP
MACRO MULT V1, V2
  LOCAL L1
  MOV AL,V1
MOV AH,V2
   MOV BX,00
   L1:ADD BL,AL
     DEC AH
     CMP AH,0
     JNE L1
MULT ENDM

MULT N1, N2
MULT  2, 3

END
```

Third: PUBLIC and EXTRN directives:

The **PUBLIC** and **EXTRN** directives are very important in modular programming. We use **PUBLIC** to declare that labels of code, data, or entire segments are available to other program modules. **EXTRN** (external) declares that labels are external to a module.

The **PUBLIC** directive is placed in the opcode field of an assembly language statement to define a label as public, so that the label can be used by other modules. The label declared as public can be a jump address, a data address, or an entire segment. When segments are made public, they are combined with other public segments that contain data with the same segment name. See Example 1

```
;EXAMPLE 1:
.MODEL SMALL
.DATA
            PUBLIC Table1
            PUBLIC Table2
            PUBLIC Table3
            PUBLIC Datax
Table1      DW
01H,02H,3FH,42H,05H,16H
Table2      DW 10DUP(?)
Table3      DB 10DUP(?)
Datax DD 4DUP(?)

.CODE
.STARTUP
PUBLIC      Main
Main  PROC FAR
      MOV SI, OFFSET Table1
      MOV DI, OFFSET Table2
      MOV CX, 6H
LO:
      MOVSW
      LOOP LO
Main ENDP
END
```

The **EXTRN** statement appears in both data and code segments to define labels as external to the segment. If data are defined as external, their sizes must be defined as BYTE, WORD, or DWORD. If a jump or call address is external, it must be defined as NEAR or FAR. Example 2 shows how the external statement is used to indicate that several labels are external to the program listed.

```
;EXAMPLE 2:
.MODEL SMALL
.DATA
       EXTRN Table1:WORD
       EXTRN Table2:WORD
       EXTRN Table3:BYTE
       EXTRN DATAx :DWORD
.CODE
       EXTRN Main:FAR
.STARTUP
       MOV DX, OFFSET Table3
       MOV AX,05H
       MOV [DX],AX
.EXIT
END
```

Fourth: Libraries

Library files are collections of procedures that are used by many different programs. These procedures are assembled and compiled into a library file by the **LIB** program that accompanies the MASM assembler program. Libraries allow common procedures to be collected into one place so they can be used by many different applications. The library file (FILENAME.LIB) is invoked when a program is linked with the linker program.

When the library file is linked with a program, only the procedures required by the program are removed from the library file and added to the program- If any amount of assembly language programming is to be accomplished efficiently, a good set of library files is essential and saves many hours in receding common functions.

A library file is created with the LIB command, typed at the DOS prompt. A library file is a collection of assembled .OBJ files that each performs one procedure or task. The name of the procedure must be declared PUBLIC in a library file and does not necessarily need to match the file name. Each procedure is defined as a FAR procedure, so that the linker can place the procedures in a code segment separate from the main program.

Example: Library creation:

Assume we have three different subroutines each is written in assembly source file as shown in figure1 (trianglearea,squarearea,display) and you want to create a library from these three subroutines, you need to follow the following steps:

sub1.asm

```
.model small
.data
extrn len:byte
extrn wid:byte
extrn result:byte
.code
mov ax,@data
mov ds,ax
public trianglearea
trianglearea proc far

    mov ah,0
    mov al,len
    mov bl,wid
    mul bl
    mov bl,2
    div bl
    mov result,al
    ret
trianglearea endp
end
```

sub2.asm

```
.model small
.data
extrn len:byte
extrn result:byte
.code
mov ax,@data
mov ds,ax
 public squarearea
squarearea proc far

  mov ah,0
  mov al,len
  mov bl,len
  mul bl
  mov result,al
   ret
squarearea endp

end
```

sub3.asm

```
.model small
.data
extrn result:byte

.code

mov ax,@data
mov ds,ax
 public display
display proc far

mov ah,0
mov al,result
mov bl,0ah
div bl
mov bl,al
mov bh,ah
add bl,30h
add bh,30h
mov ah,0eh
mov al,bl
int 10h
mov al,bh
int 10h

 ret

display endp


end
```

test.asm

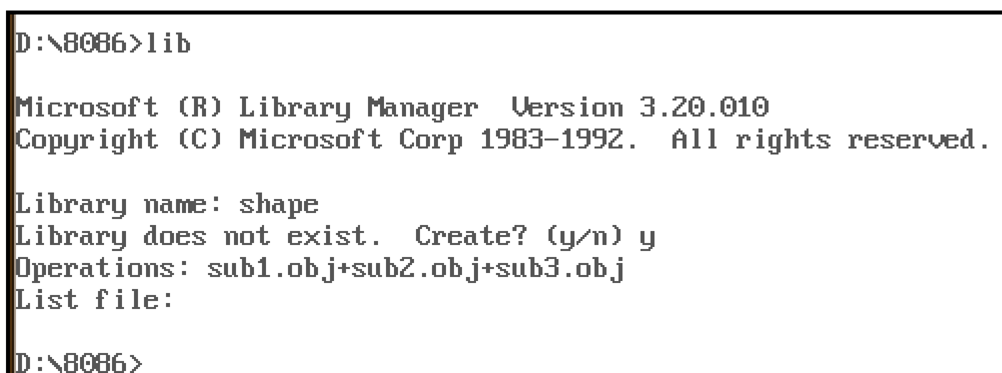
```
.model small
.data
public len
public wid
public result
len db 6
wid db 4
result db ?
msg db "the shape area= $"
.code

mov ax,@data
mov ds,ax
extrn trianglearea:far
extrn squarearea:far
extrn display:far


call trianglearea
mov ah,9
mov dx,offset msg
int 21h

call display
call squarearea
mov ah,9
mov dx,offset msg
int 21h
call display

ret

end
```

Figure 1

Step#1 **:** masm each **asm** file

```
DOSBox 0.74, Cpu speed:    3000 cycles, Frameskip  0, Program:   DOSBOX
 Assembling: sub2.asm

D:\8086>masm sub1.asm
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.

 Invoking: ML.EXE /I. /Zm /c /Ta sub1.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: sub1.asm

D:\8086>masm sub2.asm
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.

 Invoking: ML.EXE /I. /Zm /c /Ta sub2.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: sub2.asm

D:\8086>masm sub3.asm
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.

 Invoking: ML.EXE /I. /Zm /c /Ta sub3.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: sub3.asm

D:\8086>_
```

Step#2 **:** Use **lib** command to create the library from the object files created from step#1 **(name it shape)**



```
D:\8086>lib

Microsoft (R) Library Manager  Version 3.20.010
Copyright (C) Microsoft Corp 1983-1992.  All rights reserved.

Library name: shape
Library does not exist.  Create? (y/n) y
Operations: sub1.obj+sub2.obj+sub3.obj
List file:

D:\8086>_
```

The LIB program begins with the copyright message from Microsoft, followed by the prompt Library name. The library name chosen is name for the name.LIB file. Because this is a new

file, the library program asks if we wish to create the library file. The Operations: prompt is where the library module names are typed. In this case, we create a library by using three procedure files (SUB1, SUB2, and SUB3). The list file shows the contents of the library, the list file shows the size and names of the files used to create the library, and the public label (procedure name) that is used in the library file.

Step#3**:** write a test program that call all the subroutines in the library and **masm** it.

```
D:\8086>masm test.asm
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.

 Invoking: ML.EXE /I. /Zm /c /Ta test.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: test.asm

D:\8086>
```

Step#4**:** link **test.obj** with the library (**shape.lib**)

```
D:\8086>link test + shape.lib

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [TEST.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment
```

Once the library file is linked to your program file, only the library procedures actually used by your program are placed in the execution file. Don't forget to use the label EXTRN when specifying library calls from your program module.

Step#5**:** run the test program **(test.exe)** to see the output

```
D:\8086>test.exe
the shape area= 12the shape area= 36
```

*change the values in the test program (len & wid) and repeat the steps 3, 4 &5 to see different result**.**

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 4
BIOS Interrupts Programming

1.1 Objectives:

1. To become familiar with the BIOS interrupts of the 8086/88 processor.

1.2 Pre-lab Preparation:

- Read the experiment thoroughly and make a real effort to run the code examples.

1.3 Equipments:

- Personal computer with MASM software installed on it.
- MTS-8088 kit.

1.4 Theoretical background:

There are some extremely useful subroutines within BIOS and DOS that are available to the user through the INT instruction. The INT instruction is somewhat like a FAR call. When it is invoked, it saves CS:IP and then flags on the stack and goes to the subroutine associated with that interrupt. The INT instruction has the following format:

```
INT xx          ; the interrupt number xx can be 00 – FF
```

Since interrupts are numbered 00 to FF, this gives a total of 256interrupts in 80x86 microprocessors. Of these 256 interrupts, two are the most widely used: INT10 and INT 21. Each one can perform many functions. You can find a list of these interrupts in textbooks or on the web. Before the service of INT 10H or INT 21H is requested, certain registers must have specific values in them, depending on the function being requested. Various functions of INT 21H and INT 10H are selected by the value put in the AH register.

BIOS Interrupts

Interrupt types 0-1FH are known as BIOS interrupts. This is because most of these service routines are BIOS routines residing in the ROM segment F000h.

➔ **Interrupt Types 0 – 7**

Interrupt types 0 – 7 are reserved by Intel, with types 0 – 4 being predefined. IBM uses type 5 for print screen. Types 6 and 7 are not used.

➔ **Interrupt Types 8h – Fh**

The 8086 has only one terminal for hardware interrupt signals. To allow more devices to interrupt the 8086, IBM uses an interrupt controller, the Intel 8259 chip which can interface up to eight devices. Interrupt types 8 – Fh are generated by hardware devices connected to 8259.

➔ **Interrupt Types 10h – 1Fh**

The interrupt routines 10h – 1Fh can be called by application programs to perform various I/O operations and status checking.

Text Display Programming

One of the most interested and useful applications of assembly language are in controlling the monitor display

.

➔ ***Display Modes***

We commonly see both text and picture images displayed on the monitor. The computer has different techniques and memory requirements for displaying text and picture graphics. So the adapters have two display modes: text and graphics. In the **text mode**, the screen is divided into columns and rows, typically 80 columns by 25 rows, and the character is displayed at each screen position. In **graphics mode**, the screen is divided into columns and rows, and each screen position is called a pixel. A picture can be displayed by specifying the color of each pixel on the screen.

➔ ***Kinds of video adapters***

The video adapters for the IBM PC differ in resolution and the number of colors that can be displayed. IBM introduced two adapters with the original PC, the **MDA** (Monochrome Display Adapter) and **CGA** (Color Graphics Adapter). The MDA can only display text and was intended for business software, which at that time did not use graphics. The CGA can display in color both text and graphics, but it has lower resolution. In text mode, each character cell is only 8×8 dots. In 1984 IBM introduced **EGA** (Enhanced Graphics Adapter), which has good resolution and color graphics. The character cell is 8×14 dots. In 1988 IBM introduced the PS/2 models which are equipped with the **VGA** (Video Graphics Array) and **MCGA** (Multi-color Graphics Array) adapters. These adapters have better resolution and can display more colors in graphics mode than EGA. The character cell is 8×19 dots.

➔ ***Mode Numbers***

Depending on the type of adapter present, a program can select text or graphics modes. Each mode is identified by a mode number; table 4.1 lists the text modes for the different kinds of adapters.

Table 4.1: Video adapter text modes

Mode Number	Description	Adapters
0	40×25 16-color text (color burst off)	CGA,EGA,MCGA,VGA
1	40×25 16-color text	CGA,EGA,MCGA,VGA
2	80×25 16-color text (color burst off)	CGA,EGA,MCGA,VGA
3	80×25 16-color text	CGA,EGA,MCGA,VGA
7	80×25 Monochrome text	MDA,EGA,VGA

➔ ***Display Pages***

For the DMA, the display memory can hold one screenful of data. The graphics adapter, however, can store several screens of text data. This is because graphics display requires more memory, so the memory unit in a graphics adapter is bigger. To fully use the display memory, a graphics adapter divides its display memory into **display pages**. One page can hold the data for one screen. The pages are numbered, starting with 0; the number of pages available depends on the adapter and the mode selected. If more than one page is available, the program can display one page while updating another one. Table 4.2 shows the number of display pages for the MDA, CGA, EGA, and VGA in text mode. In the 80×25 text mode, each display page is 4KB. The MDA has only one page, page0; it starts at location B000:0000h. The CGA has four pages, starting at address B800:0000h. In text mode, the EGA and VGA can emulate either the MDA or CGA.

Table 4.2: Number of text mode display pages

Modes	Maximum Number of Pages		
	CGA	**EGA**	**VGA**
0-1	8	8	8
2-3	4	8	8
7	NA	8	8

➔ ***The Attribute Byte***

In a display page, the high byte of the word that specifies a display character is called the **attribute byte**. It describes the color and intensity of the character, the background color, and whether the character is blinking and/or underlined.

➔ ***16 – Color Display***

The attribute byte for 16 – color text display (modes 0 – 3) has the format shown below. A 1 in a bit position selects an attribute characteristic. Bits 0 – 2 specify the color of the character (foreground color) and bits 4 – 6 give the color background at the character's position. Bit 3 specifies the intensity of the character and bit 7 specifies whether the character is blinking or not.

Bit	7	6	5	4	3	2	1	0
Attribute	Blinking	Background color			Intensity	Foreground color		
		Red	Green	Blue		Red	Green	Blue

➔ ***Monochrome Display***

For monochrome display, the possible colors are white and black. For white, the RGB bits are all 1; for black, they are all 0. **Normal video** is a white character on a black background; the attribute byte is 0000 0111 = 7h. **Reverse video** is a black character on a white background, so the attribute is 0111 0000 = 70h.

➔ ***INT 10h***

Even though we can display data by moving them directly into the active display page, this is a very tedious way to control the screen. Instead we use the BIOS video screen routines which is invoked by the **INT 10h** instructions; a video functions is selected by putting a function number in the AH register.

Function 0: **Select Display Mode** Input: AH = 0 AL = mode number (see table 7.1) Output: none
Function 1: **Change Cursor Size** Input: AH = 1 CH = Starting scan line CL = ending scan line Output: none

In text mode, the cursor is displayed as a small dot array at a screen position (in graphics mode, there is no cursor). For the MDA and EGA, the dot array has 14 rows (0 – 13) and for the CGA, there are 8 rows (0 – 7). Normally only rows 6 and 7 are lit for the CGA cursor, and rows 11 and 12 for the MDA and EGA cursor. To change the cursor size, put the starting and ending numbers of the rows to be lit in CH and CL, respectively.

Function 2: **Move Cursor** Input: AH = 2 DH = new cursor row (0 – 24) DL = new cursor column. 0 – 79 for 80×25 display, 0 – 39 for 40×25 display. BH = Page number Output: none
Function 3: **Get Cursor Position and Size** Input: AH = 3 BH = page number Output: DH = cursor row DL = cursor column CH = cursor starting scan line CL = cursor ending scan line
Function 5: **Select Active Display Page** Input: AH = 5 AL = active display page 0 – 7 for modes 0,1 0 – 3 for CGA modes 2,3 0 – 7 for EGA, MCGA, VGA modes 2,3 0 – 7 for EGA, VGA mode 7 Output: none
Function 6: **Scroll the Screen or a Window UP** Input: AH = 6 AL = number of lines to scroll (Al = 0 means scroll the whole screen or window) BH = attribute for blank lines CH,CL = row, column for upper left corner of window (see table 4.3) DH,DL = row, column for lower right corner of window (see table 4.3) Output: none
Function 7: **Scroll the screen or a Window Down** Input: AH = 7 AL = number of lines to scroll (Al = 0 means scroll the whole screen or window) BH = attribute for blank lines CH,CL = row, column for upper left corner of window (see table 4.3) DH,DL = row, column for lower right corner of window (see table 4.3) Output: none
Function 8: **Read Character at the Cursor** Input: AH= 8 BH = page number Output: AH = attribute of character AL = ASCII code of character

Table 4.3: some 80×25 screen positions

Position	Decimal		Hex	
	Column	Row	Column	Row
Upper left corner	0	0	0	0
Lower left corner	0	24	0	18
Upper right corner	79	0	4F	0
Lower right corner	79	24	4F	18
Center of the screen	39	12	27	C

Function 9:
Display Character at the Cursor with Any Attribute
Input: AH = 9
BH = page number
AL = ASCII code of character
CX = number of times to write character
BL = attribute of character
output: none

Function Ah:
Display Character at the Cursor with Current Attribute
Input: AH = 0Ah
BH = page number
AL = ASCII code of character
CX = number of times to write character
Output: none

Function Eh:
Display Character and Advance Cursor
Input: AH = 0Eh
AL = ASCII code of character
BH = page number
BL = foreground color (graphics mode only)
Output: none

This is the BIOS function used by INT 21h, function 2, to display a character. The control characters bell (07h), backspace (08h), line feed (0Ah), and carriage return (0Dh) cause control functions to be performed.

Function Fh:
Get Video Mode
Input: AH = 0Fh
Output: AH = number of screen columns
AL = display mode (see table 4.1)
BH = active display page

This function can be used with function 5 to switch between pages being displayed.

➔ ***Graphics Modes***

In graphics mode operation, the screen display is divided into columns and rows; and each screen position, given by a column number and row number, is called a **pixel** (picture element), the number of columns and rows gives the resolution of the graphics mode. The columns are numbered from left to right starting with 0, and rows are numbered from top to bottom starting with 0, depending on mapping of rows and columns into the scan lines and dot positions, a pixel may contain one or more dots. For example, in the low-resolution mode of the CGA, there are 160 columns by 100 rows, but the CGA generates 320 dots and 200 scan lines; so a pixel is formed of 2×2 set of dots. A graphics mode is called **APA (all points addressable)** if it

maps a pixel into a single dot. Table 4.4 shows the APA graphics modes of the CGA, EGA, and VGA. To maintain compatibility, the EGA is designed to display all CGA modes and the VGA can display all the EGA modes.

Table 4.4: Video adapter graphics display modes

Mode Number (Hex)	**CGA Graphics**
4	320×200 4 color
5	320×200 4 color (color burst off)
6	640×200 2 color
	EGA Graphics
D	320×200 16 color
E	640×200 16 color
F	640×350 Monochrome
10	640×350 16 color
	VGA Graphics
11	640×480 2 color
12	640×480 16 color
13	320×200 256 color

Note: the screen mode is normally set to text mode; hence the first operation to begin graphics display is to set the display mode using function 0, INT 10h.

CGA Graphics

The CGA has three graphics resolutions: a low resolution of 160×100, a medium resolution of 320×200, and a high resolution of 640×200. only the medium-resolution and high-resolution modes are supported by the BIOS INT 10h routine.

Medium-Resolution Mode

The CGA can display 16 colors; Table 4.5 shows the 16 colors of the CGA. In medium resolution, four colors can be displayed at one time. This is due to the limited size of the display memory.

Table 4.5: Sixteen standard CGA colors

I R G B	**Color**
0 0 0 0	Black
0 0 0 1	Blue
0 0 1 0	Green
0 0 1 1	Cyan
0 1 0 0	Red
0 1 0 1	Magenta
0 1 1 0	Brown
0 1 1 1	White
1 0 0 0	Gray
1 0 0 1	Light Blue
1 0 1 0	Light Green
1 0 1 1	Light Cyan
1 1 0 0	Light Red
1 1 0 1	Light Magenta
1 1 1 0	Yellow
1 1 1 1	Intense White

To read or write a pixel, we must identify the pixel by its column and row numbers. These functions 0Dh and 0Ch are for read and write respectively.

Function 0Ch: **Write Graphics Pixel** Input: AH = 0Ch AL = pixel value BH = page (for the CGA, this value is ignored) CX = column number DX = row number Output: none
Function 0Dh: **Read Graphics Pixel** Input: AH = 0Dh BH = page (for the CGA, this value is ignored) CX = column number DX = row number Output: AL = pixel value

➔ ***Examples:***

Below are two examples that use **BIOS interrupts:**

a) Setting the cursor position to the middle of the screen then display character A five times with a blue color and a white background.

```
MOV AH, 02          ; set cursor option
MOV BH, 00          ; page 0
MOV DL, 40          ; column position
MOV DH, 13          ; row position
INT 10H
MOV AH,9            ;Display a character with attribute
MOB BL,72H          ; attribute of character
MOV CX,5            ; number of times to write character
MOV AL,'A'          ;character to display
INT 10H
```

b) Draw the following line :

```
MOV AH,0            ;change mode
MOV AL,12H          ;select graphic mode
INT 10H

MOV AH,0CH          ;write pixel value
MOV AL,07           ;pixel value(color)
MOV CX,100          ;column number
MOV DX,150          ;row number
L1:INT 10H
    INC CX          ;move to the next column
    CMP CX,150
    JNZ LQ1
```

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 5
DOS & Mouse Interrupts Programming

1.1 Objectives:

1. To become familiar with the DOS interrupts of the 8086/88 processor.
2. To become familiar with the mouse interrupts.

1.2 Pre-lab Preparation:

- Read the experiment thoroughly and make a real effort to run the code examples.

1.3 Equipments:

- Personal computer with MASM software installed on it.
- MTS-8088 kit.

1.4 Theoretical background:

The interrupt types 20-3FH are serviced by DOS routines that provide high level service to hardware as well as system resources such files and directories. The most useful is INT 21H, which provides many functions for doing keyboard, video, and file operations.

Interrupt 20h – Program terminate Interrupt 20h can be used by a program to return control to DOS. It is not widely used because CS must be set to the program segment prefix before using INT 20h. It is more convenient to exit a program with INT 21h, function 4Ch.

Interrupt 21h – Function Request Interrupt 21h may be use for various functions, these functions may be classified as character I/O, file processing, memory management, disk access, and miscellaneous.

Interrupt 22h – 26h Interrupt routines 22h – 26h handle Ctrl-Break, critical errors, and direct disk access.

Interrupt 27h–Terminate but Stay Resident Interrupt 27h allows programs to stay in memory after termination

➔ ***INT 21h***

INT 21h may be used to invoke a large number of DOS functions, a particular function is requested by placing a function number in the AH register and invoking INT 21h. Here we are interested in the following functions.

Character I/O Functions:

Function 1:
Single Key Input
Input: AH = 1
Output: AL = ASCII code if character Key is pressed
= 0 if non character key is pressed

The processor will wait for the user to hit a key if necessary. If a character key is pressed, AL gets its ASCII code; the character is also displayed on the screen. Because INT 21h doesn't prompt the user for input, he might not know whether the computer is waiting for input or is occupied by some computation. The next function can be used to generate an input prompt.

Function 2: **Display a character or execute a control function** Input: AH = 2 DL = ASCII code of the display character or control character Output: AL = ASCII code of the display character or control character

Table 5.7 shows some of the control characters and their corresponding control functions.

Table 5.7: control characters and functions

Function	Symbol	ASCII Code
Beep (sounds a tone)	BEL	7
backspace	BS	8
tab	HT	9
Line feed (new line)	LF	A
Carriage return (start of current line)	CR	D

In case the programmer wants to prompt user for input by a meaningful message function 9 could be used.

Function 9: **Display a String** Input: AH = 9 DX = offset address of string. The string must end with a '$' character.

See also AH = 6h, AH = 7h, and AH = 0Ah

➔ File Processing Functions

INT 21h provides a group of functions called **file handle functions**. These functions make file operations much easier than the file control block method used before. In the latter, the programmer was responsible for setting a table that contained information about open files. With file handle functions, DOS keeps track of open file data in its own internal tables, thus relieving the programmer of this responsibility.

File Handle

When a file is opened or created in a program, DOS assign it a unique number called the **file handle**. This number is used to identify the file, so program must save it.
There are five predefined file handles. They are:

0 Keyboard
1 Screen
2 Error output – screen
3 Auxiliary device
4 Printer

File Errors

There is many opportunities for errors in INT 21h file handling; DOS identifies each error by a code number. In the functions described here, if an error occurred then CF is set and the code number appears in AX. Table 5.8 shows the most common file handling errors.

Table 5.8: File handling errors

Hex Error Code	Meaning
1	Invalid function number
2	File not found
3	Path not found
4	All available handles in use
5	Access denied
6	Invalid file handle
C	Invalid access code
F	Invalid drive specified
10	Attempt to remove current directory
11	Not the same device
12	No more files to be found

Opening and closing a file:

Before a file can be used, it must be opened. To create a new file or rewrite an existing file, the user provides a file name and an attribute; DOS return a file handle.

Function 3Ch
Create a New File/Rewrite a File
Input: AH= 3Ch
DS:DX = address of file name which is an ASCIIZ string
CL = attribute byte
Output: If successful, AX = file handle
Error if CF = 1, error code in AX (3,4, or 5)

Attribute byte is a byte in which each bit specifies file attribute.

0	1	2	3	4	5	6	7	**Bit**
Read-Only file	Hidden file	DOS system file	Volume label	Subdirectory	Archive bit	Not used	Not used	**Attribute**

To open an existing file, there is another function.

Function 3Dh
Open an Existing File
Input: AH = 3Dh
DS:DX = address of a file name which is an ASCIIZ string
AL = access code: 0 means open for reading
1 means open for writing
2 means open for both
output: if successful, AX = file handle
Error if CF = 1, error code in AX (2, 4, 5, 12)

After a file has been processed, it should be closed. This frees the file handle for use with another file. If the file is being written, closing causes any data remaining in memory to be written to the file, and the file's time, date, and size to be updated in the directory.

Function 3Eh
Close a File
Input: BX = file handle
Output: if CF = 1, error code in AX (6)

Reading a file

The following function reads a specified number of bytes from a file and stores them in memory.

Function 3Fh **Read a File** Input: AH = 3Fh BX = file handle CX = number of bytes to read DS:DX = memory buffer address Output: AX = number of bytes actually read If AX = 0 or AX < CX, end of file encountered If CF = 1, error code in AX (5, 6)

Writing a file

Function 40h writes a specified number of bytes to a file or device.

Function 40h **Write File** Input: AH = 40h BX = file handle CX = number of bytes to write DS:DX = data address Output: AX = bytes written. If AX < CX, error (full disk) If CF = 1, error code in AX (5, 6)

➔ **INT 33h : Mouse driver interrupts**

function 00h **Mouse initialization** Input: AX=00 Output: if successful: AX=0FFFFh and BX=number of mouse buttons. if failed: AX=0.
Function 01h **Show mouse pointer** Input: AX=01 Output : display mouse pointer
Function 02h **Hide visible mouse pointer.** Input: AX=02 Output : hide mouse pointer
Function 03h **Get mouse position and status of its buttons** Input: AX=03 Output: if left button is down: BX=1 if right button is down: BX=2 if both buttons are down: BX=3 CX = x column number DX = y row number ; Note: the value of CX is doubled.

Example:

Write a code to check the mouse click, if the right button is clicked draw a yellow pixel in the mouse location, else if the left button is clicked draw a red pixel in the mouse location.

```
.MODEL SMALL
.CODE
MOV AH,0
MOV AL,12H
INT 10H
MOV AX,00
INT 33H
MOV AX,1
INT 33H
AGAIN:
MOV AX,3
INT 33H
CMP BX,1
JE LEFT
CMP BX,2
JE RIGHT
JMP AGAIN
LEFT:
 SHR CX,1
 MOV AH,0CH
 MOV AL,0EH
 INT 10H
 JMP AGAIN
RIGHT:
  SHR CX,1
 MOV AH,0CH
 MOV AL,04H
 INT 10H
 JMP AGAIN
END
```

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 6
Parallel Data Input/Output

1.1 Objectives:

1) To understand the decoding circuit that is implemented for the 8088 input and output subsystems.
2) To know how to download assembly program from PC to the MTS-8088 kit, using serial communication port RS232.
3) Programming the 8255 Programmable Peripheral Interface.

1.2 Pre-lab Preparation:

- Read the experiment thoroughly.
- Read ***Appendix A: Communicating with MTS-8088 kit*** (Very Important).

1.3 Equipments:

- Personal computer with MASM software installed on it.
- MTS-8088 kit.
- EDS-8809.
- 50 pins IDC flat cable
- RS232 serial cable.

1.4 Theoretical background:

Input and output Units provide the microprocessor with the means for communicating with the outside world. Examples for the I/O interfaces are: PC keyboard, display and serial communication interface. For the 8088 microprocessor, 16- address lines are used to assign up to 2^{16} I/O port. Figure 1 is an example of an 8-port output circuit for the 8088 microprocessor.

First: 8255 Programmable Peripheral Interface (PPI):

The 8255 is an example of I/O port. It is a programmable peripheral interface device designed for use with Intel microcomputer systems. The programmable peripheral interface (PPI) 8255 is used with 8088 microprocessor to permit easy implementation of parallel I/O. It has three ports that are used as I/O ports, And a control register. Table 1 shows the addresses of them.

Address	Address allocated to
FF10 H	Port A
FF11 H	Port B
FF12 H	Port C
FF13 H	Control Register

Table 1

Figure1: shows the pin layout of the programmable 8255 interface.

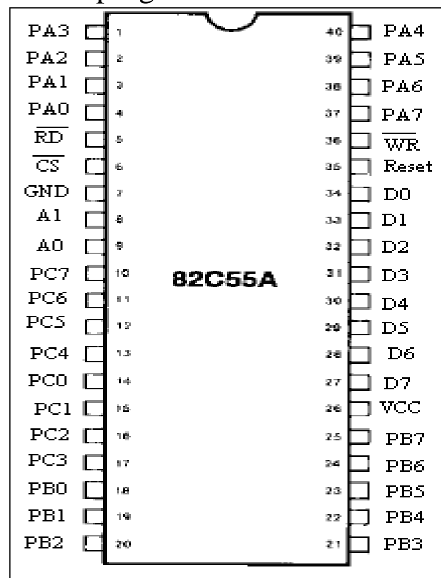


Figure1

➔ **To Write a Program:**

1. **Send Control Byte to Control Register.**

Control Register (8 bits):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	Port A	Upper Port C	0	Port B	lower Port C

To decide the value of bits (0, 1, 3, and 4):

If the port is connected to Input Device, the value will be 1.

If the port is connected to Output Device, the value will be 0.

Example:

If we connect 7-segment display to Port A (Output), 8 Push buttons to Port B (Input), and 8 LEDs to Port C (Output), the value of Control Register should be 10000010B= 82H.

MOV DX, …….. ; put the address of Control Register

MOV AL, ……… ; put the value of Control Byte. It is 82h in the example above. OUT DX, AL ; after executing OUT instruction, the Control Register is initialized.

2. **To Read from any Port.**

MOV DX, …….. ; put the address of the Port you want to read a Byte from.

IN AL, DX ;after executing IN instruction, AL contains the Byte you have just read

3. **To Write (Send) on Any Port.**

MOV DX, …….. ; put the address of the Port you want to send a byte to it.

MOV AL, ……… ; put the value of the Byte (Data).

OUT DX, AL ; after executing OUT instruction, the date is transferred to the Port.

Second: 7-segment display:

The 7-segment LED display can be found in many applications that range from the simple toaster ovens to more sophisticated industrial control rooms. It is composed of 7 LEDs that are fabricated in one case to make it more convenient for displaying numbers and some letters, figure 2 shows the connection way of six 7-segments at port A, port B, and port C.

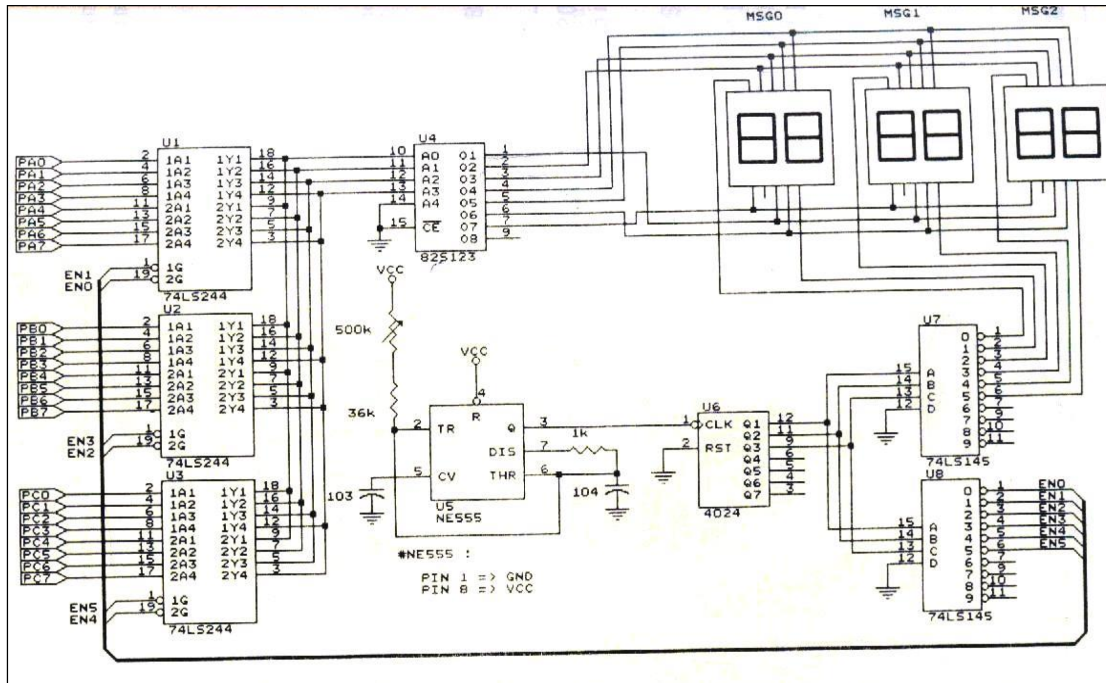


Figure 2: "Connection of 7-segments display"

To avoid displaying the consecutive hexadecimal digits too quickly, an appropriate delay period needs to be inserted before sending the subsequent digit to the 7-segment display. This can be accomplished by writing a time delay loop as follows:

Example:

To display the following sequence on the 7-segment display connected to PORTA (1, 2, … , 9)

```
.MODEL SMALL
.CODE
MOV DX, 0FF13H
MOV AL, 80H
OUT DX, AL
MOV DX, 0FF10H
MOV AL, 1
DISPLAY:
        OUT DX, AL
        CALL DELAY
        INC AL
        CMP AL, 10
JNE DISPLAY
JMP FINISH
DELAY PROC
MOV CX, 0FFFFH
L1: LOOP L1
RET
DELAY ENDP
FINISH:
END
```

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 7
I/O Applications: Dynamic Display

1.1 Objectives:

1. Implement wider applications on the I/O ports of the Microprocessor.
2. Understand the display principle of dot matrix LEDs module of EDS-8809.
3. Understand the principle of data's shift.

- .

1.3 Equipments:

- Personal computer with MASM software installed on it.
- MTS-8088 kit.
- EDS-8809.
- 50 pins IDC flat cable
- RS232 serial cable.

1.4 Theoretical background:

First: Dynamic display

We see the advertisement signs in streets, which display moving photos and text messages with animation. In this experiment we will implement a similar idea in which we will display a string of characters on the EDS-8809 dot matrix LEDs. This practical application will make use of I/O ports of the microprocessor.

We have the architecture of the dot matrix display is shown below in Figure1

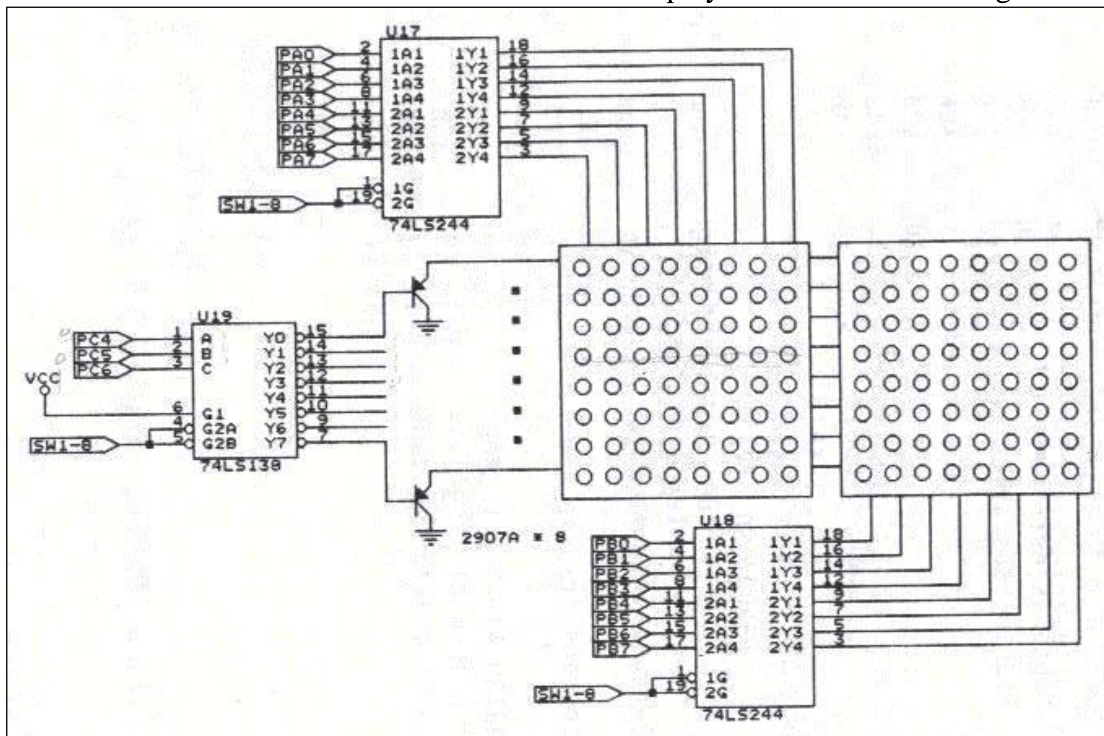


Figure1: The architecture of the dot matrix display

7.4 How to output the data?

First we have to change the dip switch to be enabled to output data to the dot matrix display, as shown in Figure2.

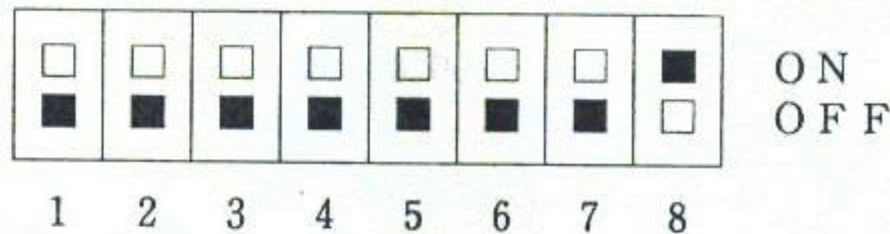


Figure2: The dip switch

The following example shows how to set the control register to make the ports behave as output ports:

Example :

Write instructions to enable the first row with 8 LEDs ON port A

```
MOV DX, FF12     ; select port C
MOV AL, 10h       ; select row1 (second row)
OUT DX, AL
MOV DX, FF10     ; select port A
MOV AL, FF        ; write value FF to port A
OUT DX, AL
```

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 8
LCD & Keypad Interrupts

1.1 Objectives:

- To study and practice the use of various types of software interrupts.
- To use interrupts to write a simple and real application.

1.2 Equipments and Materials:

- MTS-8088 Kit.

1.3 Overview

MTS-8088 provides several I/O drivers to handle I/O devices; these programs can be called by application programs to perform I/O. The I/O drivers provides by the EGC-8088 are accessed through interrupt requests.

The routines we need to use in our lab are:

- INT81H The keyboard driver.
- INT84H The LCD driver.
- INT85H Return to the system monitor.

Each of these software interrupt instructions require certain parameters to function properly.

The functions provided by each I/O driver and their parameters are described in detail in the following sections.

1.3.1 The Keyboard Driver: INT81H

This software provides the following functions:

AH=0: Read a character from the key board.

AH=1: Read a command line from the keyboard

AH=2: Read the keyboard status.

AH=0: Read a character from the key board.

The driver only returns the scan code of the keystroke and its ASCII code to the calling program when a key is pressed.

AL: The ASCII code of the keystroke.

AH: The scan code of the keystroke.

AH=1: Read a command line from the keyboard.

When AH=1, the CPU reads the keystrokes until <CR> is pressed. If the backspace key is pressed, the last input character will be erased from the input data buffer. If a control character is typed, it is ignored. The maximum number of keystrokes is 40 characters.

ES:DI points to the starting address of the data input buffer.

AH=2: Read the keyboard status.

When AH=2, the CPU reads the keyboard status. this means that the driver only indicates whether there is a keystroke; it doesn't read the key. The value of AL holds the keyboard status:

AL=0: key is pressed

AL=FF: No key is pressed.

1.3.2: The LCD driver: INT84H.

The LCD is the main display device for the MTS=8088. The LCD driver writes a character to the current cursor position and advances the cursor forward one column.

AL: character ASCII code to be displayed.

1.3.3: Return to the system BIOS:INT85H

This driver is used to return the program to the system BIOS. The user can use by executing INT85H.

Hashemite University
Faculty of Engineering and Technology
Computer Engineering Department

Experiment 9
Design and conduct an experiment

1. Objectives:

- Design and Conduct an Experiment to design a project using software or hardware tools existing on the lab.
- The ability to interpret and analyze given situation and plan solutions for it.
- The ability to choose the appropriate tools and instruments that suit the desired job.
- The ability to use the chosen tools and instruments to achieve the goals of the experiment.

2. Student proposal

You have to submit a proposal before starting your experiment. Your proposal will describe the objective or the goal of the experiment in addition to the proposed experimental set-up, instruments, tools, governing equations, etc.

Remember that your work does not based on the existing well-defined experimental procedure (manual).

You have to develop a new technique in the lab, so that you can conduct a new experiment to achieve the goal. In addition, you have to report the results. Students can use any of lab equipment, tools, instrument to setup their own experiments.

The design of an experiment can be integrated in the lab through different approaches:

- Design-build-test approach.
- Modifying an existing experimental setup.
- Utilizing instrumentations from other experiment to study a certain phenomenon.

3. Deliverables:

Each group must deliver a report written using a word processor. The final report should include:

- **Objectives:** Statement of what you are going to achieve.
- **Designing experiments**: develop a methodology which will produce high quality data that can be used to evaluate specific process or parameter.
- **Experimental setup:** the apparatus, devices, and instruments used to conduct the experiment should be clearly specified [figures or photos may be added].
- **Theoretical background:** the theory related to the experiment, including all assumptions and equations.

- **Conduct the experiment:** clear procedure should be specified.
- **Experimental results:** represent all data.
- **Analyze and interpret data:** develop, if you can, a mathematical model or computer simulation to correlate or interpret experimental results.
- **Discussion and conclusions:** list and discuss several possible reasons for deviations between predicted and measured results from an experiment.

Appendix A
Communicating with MTS-8088 KIT

Objective

This Appendix will help you to know how to communicate with the 8088 MTS kit, how to download assembly programs from PC to KIT and how to upload it.

Introduction

Although the BGC-8088 MICROENGINEER is small and simple device, it contains all the hardware elements of a computer. In particular, it provides RS-232 interface hardware circuit and software driver for communication with the outside world. Here we present 2 built-in commands for communication with the BGC-8088.

Procedure

*Step 1:*Prepare the assembly code

a) Write the assembly code in a text document then save it as **.ASM** file.
b) Assemble the source code using MASM/L command as follow:
   i) Suppose we have **Try.asm** file saved in the following directory: (E:\8086).
   ii) Open DOS from Run ⟶ cmd, and then go to the path E:\8086. (See figure 1).
   iii) Use MASM/L instruction to assemble the source file **Try.ASM** into a machine language object file **Try.OBJ**

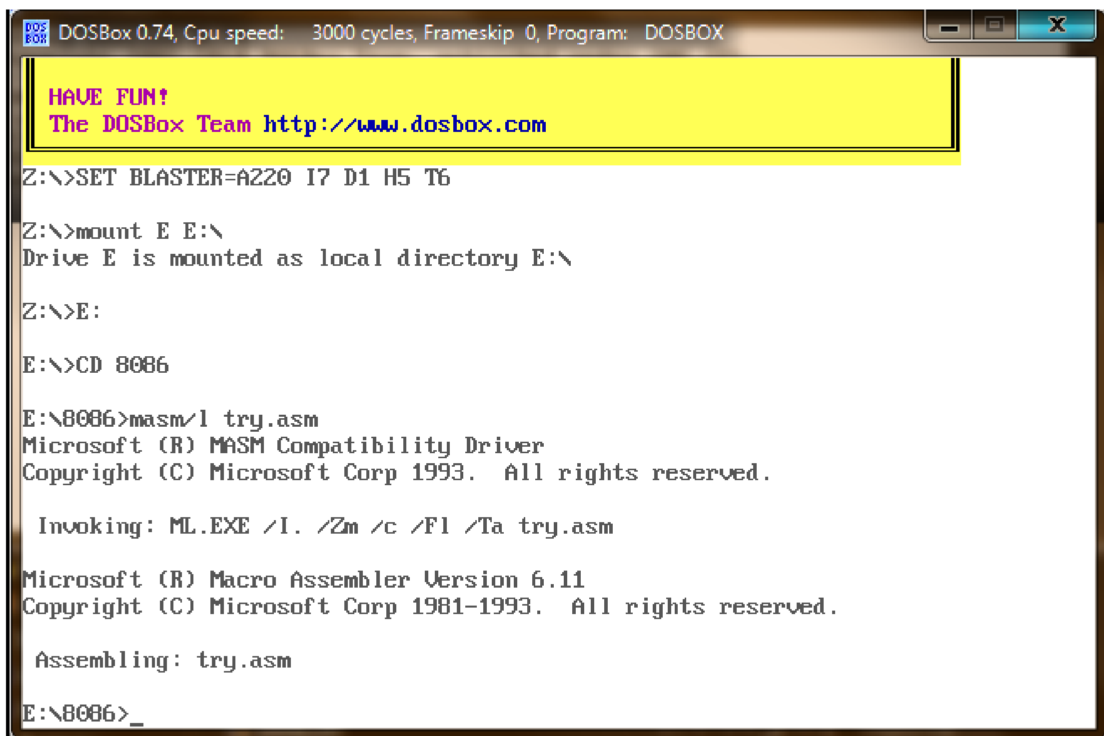


Figure 1: change directory & assemble a source file

Note: MASM checks the source file for syntax errors. If it finds any, a short description of the errors will be displayed with the line number of each.

Now the files **Try.OBJ** and **Try.LST** will be created.

Before feeding the ".OBJ" file into LINK, all syntax errors produced by the assembler must be corrected.

The ".lst" file which is optional is very useful to the programmer because it lists all the opcodes and offset addresses as well as errors that MASM detected.

c) LINK program with MASM version 6.11 take one or more object files and combine them into a single executable file (.EXE file). (See figure 2)

```
E:\8086>link try.obj

Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.

Run File [try.exe]:
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:
LINK : warning L4021: no stack segment
```

Figure 2: Link .OBJ file

Now the files **Try.EXE** and **Try.MAP** will be created.

The ".MAP" file which is optional gives the name of each segment, where it starts, where it stops, and its size in bytes.

d) Now convert the Execution file **Try.EXE** into binary file **Try.BIN**. (See figure 3)

```
E:\8086>exe2bin try.exe
```

Figure 3: EXE2BIN command

Step 2: Establish the connections

a) You need to connect a serial wire with COM1 or COM2 ports of the PC and the other end with the MTS-8088 KIT.
b) Now power ON the KIT.
c) Now open the data transfer program. (See figure 4).

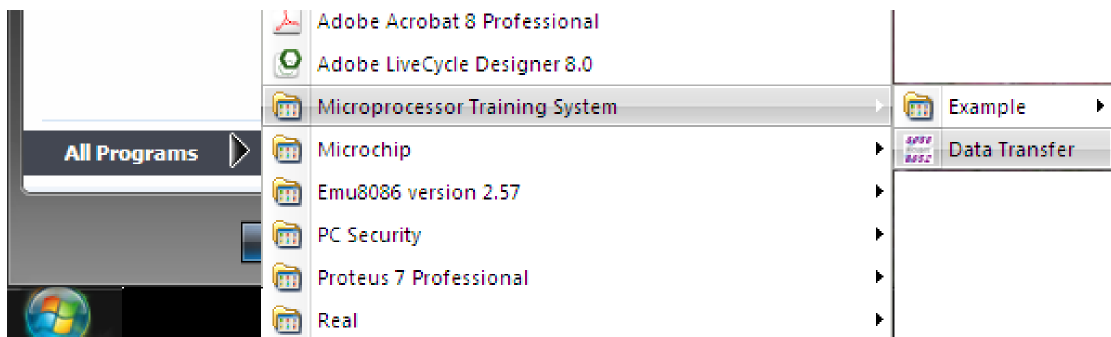


Figure 4

d) The welcome Screen will appear for a small period.

e) Then the settings screen will appear, you have to choose the type of the kit and the COM port you are connected with. (See figure 5)

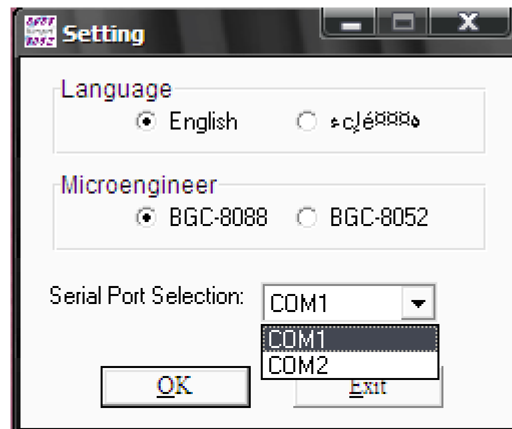


Figure 5: Settings

f) Now the following screen will appear to choose the .BIN file which you want to download it. (see figure 6)

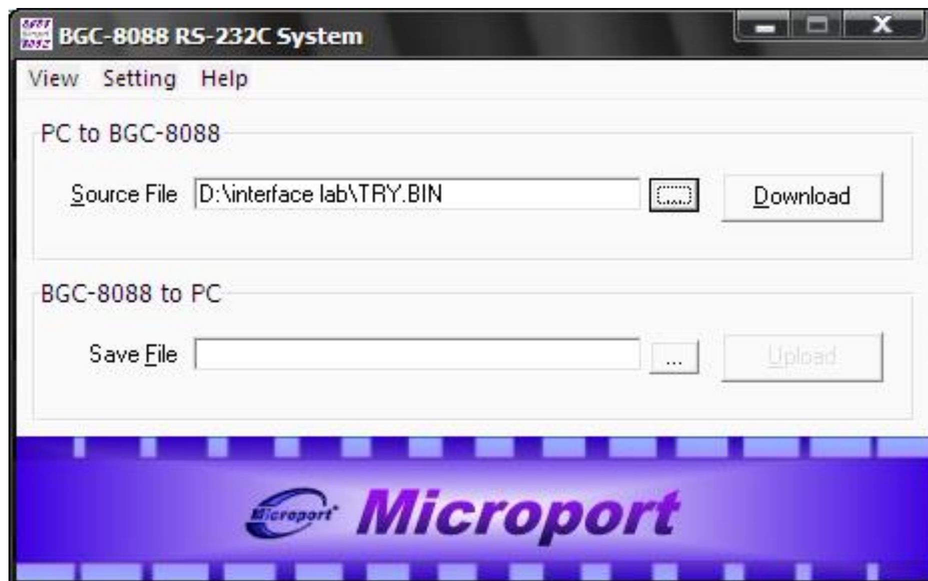


Figure 6

g) Now in the kit use the keyboard to enter the L command.
If you want to store data at a memory location other than 0100:0000, the starting address must be indicated, <enter> must be pressed to get the BGC-8088 ready for receiving data.
When it is ready for receiving data the following information is displayed.

```
DOWNLOAD....
```

h) Now press the Download button in the other side (PC), the transmition is started, after completion the total number of bytes transferred is displayed on the LCD of the KIT and on the PC. (see figure 7)

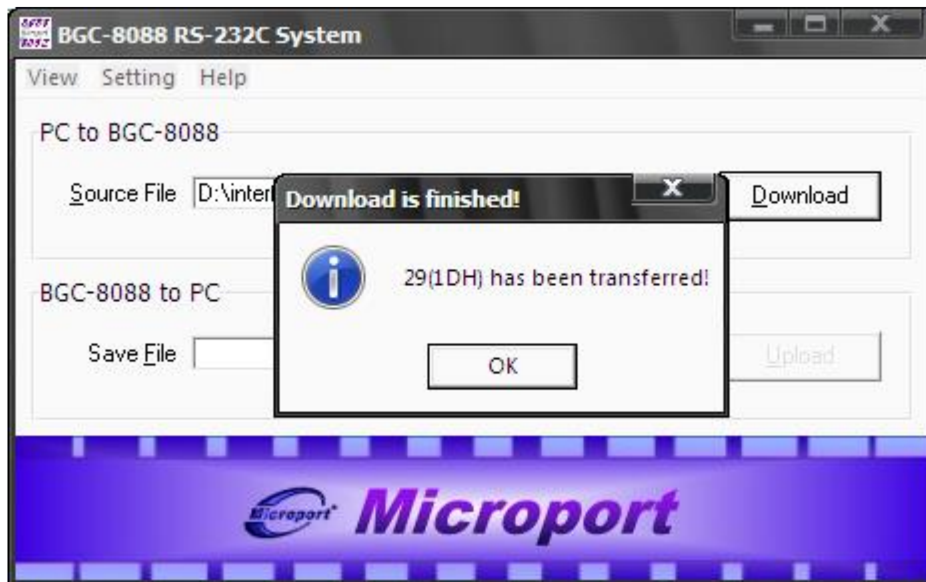


Figure 7

If these massages displayed, then the download was successful.

i) After downloading a program the user may use the G or T command to execute the program.
If the program results are not as expected, the program can be modified easily on the PC and downloaded a gain.

This interface allows the user to program in assembly on the PC, and to test that program easily on the BGC-8088 MICROENGINEER, this is much easier than using the A command to write programs for the BGC-8088 Kit.