COSC 204 LABWORK#1

Objectives:

To introduce the students to:

1. The general structure of computer register
2. Using a debugger to view the functionality of register during an instruction execution
3. Creating simple .COM program using debugger

To begin working with Debug, type the following prompt in your computer: C:/>Debug [Enter]

On the next line a dash will appear, this is the indicator of Debug, at this moment the instructions of Debug can be introduced using the following command:

to visualize the values of the internal registers of the CPU using the Debug program type -r[Enter]

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000  
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0100 NV EI PL NZ NA PO NC  
0D62:0100 2E CS:  
0D62:0101 803ED3DF00 CMP BYTE PTR [DFD3],00 CS:DFD3=03

To view a specific register the "r" command using as a parameter the name of the register whose value wants to be seen. For example:

-rbx  
BX 0000  
:

This instruction will only display the content of the BX register and the Debug indicator changes from "-" to ":" to change the value of the register type the new value and [Enter], or the old value can be left by pressing [Enter] without typing any other value.

**To create a program on the debug**

Type debug[Enter] on the operative system prompt.

To assemble a program on the Debug, the "a" (assemble) command is used; when this command is used, the address where you want the assembling to begin can be given as a parameter, if the parameter is omitted the assembling will be initiated at the locality specified by CS:IP, usually 0100h, which is the locality where programs with .COM extension must be initiated

C:\>debug  
-a 100  
0D62:0100 mov ax,0002  
0D62:0103 mov bx,0004  
0D62:0106 add ax,bx  
0D62:0108 nop  
0D62:0109

Type the command "t" (trace), to execute each instruction of this program,  
example:

-t

AX=0002 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000  
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0103 NV EI PL NZ NA PO NC  
0D62:0103 BB0400 MOV BX,0004

You see that the value 2 move to AX register. Type the command "t" (trace),  
again, and you see the second instruction is executed.

-t

AX=0002 BX=0004 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000  
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0106 NV EI PL NZ NA PO NC  
0D62:0106 01D8 ADD AX,BX

Type the command "t" (trace) to see the instruction add is executed, you will see the follow lines:

-t

AX=0006 BX=0004 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000  
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0108 NV EI PL NZ NA PE NC  
0D62:0108 90 NOP

The possibility that the registers contain different values exists, but AX and BX must be the same, since they are the ones we just modified.

To exit Debug use the "q" (quit) command.

**Storing and loading the programs**

The steps to save a program that it is already stored on memory are:

Obtain the length of the program subtracting the final address from the initial address, naturally in hexadecimal system.  
Give the program a name and extension. Put the length of the program on the CX register. Order Debug to write the program on the disk.

By using as an example the following program, we will have a clearer idea of how to take these steps:

When the program is finally assembled it would look like this:

0C1B:0100 mov ax,0002  
0C1B:0103 mov bx,0004  
0C1B:0106 add ax,bx  
0C1B:0108 int 20  
0C1B:010A

To obtain the length of a program the "h" command is used, since it will show us the addition and subtraction of two numbers in hexadecimal. To obtain the length of ours, we give it as parameters the value of our program's final address (10A), and the program's initial address (100). The first result the command shows us is the addition of the parameters and the second is the subtraction.

-h 10a 100  
020a 000a

The "n" command allows us to name the program.

-n test.com

The "rcx" command allows us to change the content of the CX register to the value we obtained from the size of the file with "h", in this case 000a, since the result of the subtraction of the final address from the initial address.

-rcx  
CX 0000  
:000a

Lastly, the "w" command writes our program on the disk, indicating how many bytes it wrote.

-w  
Writing 000A bytes

To load an already saved file two steps are necessary:

Give the name of the file to be loaded.  
Load it using the "l" (load) command.

To obtain the correct result of the following steps, it is necessary that the above program be already created.

Inside Debug we write the following:

-n test.com  
-l  
-u 100 109  
0C3D:0100 B80200 MOV AX,0002  
0C3D:0103 BB0400 MOV BX,0004  
0C3D:0106 01D8 ADD AX,BX  
0C3D:0108 CD20 INT 20

The last "u" command is used to verify that the program was loaded on memory. What it does is that it disassembles the code and shows it disassembled. The parameters indicate to Debug from where and to where to disassemble.

Debug always loads the programs on memory on the address 100H, otherwise indicated.

COSC 204 LABWORK#2

Objectives:

To introduce the students to:

1. The general structures of both exe-format and com-format programs.
2. Full and simplified segment directives.
3. Assembling and linking both exe-format and com-format programs using MASM version 5.00, TASM version 4.1, LINK version 5.6, TLINK version 7.1
4. Assembling and linking both exe-format and com-format programs using MASM version 6.14 in combination with LINK version 5.6
5. THE GENERAL STRUCTURE OF AN EXE-FORMAT PROGRAM

The general form of an exe-format program having a data segment is:

STACK SEGMENT STACK

DB 400H DUP(?)

STACK ENDS

DATA SEGMENT

; data definitions using DB, DW, DD, etc. come here

**. . .**

DATA ENDS

CODE SEGMENT

ASSUME CS : CODE , DS : DATA , SS : STACK

ENTRY: MOV AX , DATA ; Initialize DS

MOV DS , AX ;

**. . .**

; Return to DOS

MOV AX , 4C00H

INT 21H

CODE ENDS

END ENTRY

Example: The following exe-format program displays the string COSC 204 on the screen:

STACK\_SEG SEGMENT STACK

DB 200 DUP(?)

STACK\_SEG ENDS

DATA\_SEG SEGMENT

MESSAGE DB ‘COSC 204’ , ‘$’

DATA\_SEG ENDS

CODE\_SEG SEGMENT

ASSUME CS : CODE\_SEG , DS : DATA\_SEG , SS : STACK\_SEG

START: MOV AX , DATA\_SEG ; Initialize DS

MOV DS , AX ;

; Display the string

MOV AH , 09H

MOV DX , OFFSET MESSAGE

INT 21H

; Return to DOS

MOV AX , 4C00H

INT 21H

CODE\_SEG ENDS

END START

1. THE GENERAL STRUCTURE OF A COM-FORMAT PROGRAM

CODE SEGMENT

ASSUME CS : CODE , DS : CODE , SS : CODE

ORG 100H

ENTRY: JMP L1

; data definitions come here

**. . .**

L1:

**. . .**

; Return to DOS

MOV AX , 4C00H

INT 21H

CODE ENDS

END ENTRY

alternatively, the following general structure may be used:

CODE SEGMENT

ASSUME CS : CODE , DS : CODE , SS : CODE

ORG 100H

ENTRY:

**. . .**

; Return to DOS

MOV AX , 4C00H

INT 21H

; data definitions, if any, come here

**. . .**

CODE ENDS

END ENTRY

Example: The following com-format program displays the string COSC 204 on the screen:

CODE SEGMENT

ASSUME CS : CODE , DS : CODE , SS : CODE

ORG 100H

ENTRY**:** JMP START

MESSAGE DB ‘COSC 204’ , ‘$’

START**:** ; Display the string

MOV AH , 09H

MOV DX , OFFSET MESSAGE

INT 21H

; Return to DOS

MOV AX , 4C00H

INT 21H

CODE ENDS

END ENTRY

1. **SIMPLIFIED SEGMENT DIRECTIVES**

MASM version 5.0 and above, and TASM provide a simplified set of directives for declaring segments called simplified segment directives. To use these directives, you must initialize a memory model, using the **.**MODEL directive, before declaring any segment. The format of the **.**MODEL directive is:

**.**MODEL memory-model

The memory-model may be TINY, SMALL, MEDIUM, COMPACT, LARGE, or HUGE :

|  |  |
| --- | --- |
| memory-model | description |
| TINY | One segment. Thus code and data together may not be greater than 64K |
| SMALL | One code-segment. One data-segment. Thus neither code nor data may be greater than 64K |
| MEDIUM | More than one code-segment. One data-segment. Thus code may be greater than 64K |
| COMPACT | One code-segment. More than one data-segment. Thus data may be greater than 64K |
| LARGE | More than one code-segment. More than one data-segment. No array larger than 64K. Thus both code and data may be greater than 64K |
| HUGE | More than one code-segment. More than one data-segment. Arrays may be larger than 64K. Thus both code and data may be greater than 64K |

All of the program models except TINY result in the creation of exe-format programs. The TINY model creates com-format programs. **The .MODEL directive automatically generates the required ASSUME statement; thus programs with simplified segment directives do not have this directive.**

The simplified segment directives are: **.**CODE , **.**DATA , **.**STACK . The **.**CODE directive may be followed by the name of the code segment. The **.**STACK directive may be followed by the size of the stack segment, by default the size is 1K i.e., 1,024 bytes. The definition of a segment extends from a simplified segment directive up to another simplified segment directive or up to the END directive if the defined segment is the last one.

The DOSSEG directive, which is optional, tells the assembler to adopt the DOS segment-ordering conversion. For a SMALL model program, the order is code, data, stack. This directive should appear before any segment definitions.

**Note:** The instructions which initialize the DS register for an exe-format program with simplified segment directives are:

MOV AX , @DATA

MOV DS , AX

where AX may be replaced by any other 16-bit general purpose register.

Example:

DOSSEG

**.**MODEL SMALL

.STACK 200

**.**DATA

MESSAGE DB ‘COSC 204’ , ‘$’

**.**CODE

START: MOV AX , @DATA ; Initialize DS

MOV DS , AX ;

; Display the string

MOV AH , 09H

MOV DX , OFFSET MESSAGE

INT 21H

; Return to DOS

MOV AX , 4C00H

INT 21H

END START

For a com-format program simplified segment directives can only be used if the data is not at the end of the code segment after the Return to DOS instructions. If the data is at the end then full segment declarations using SEGMENT and ENDS directives must be used. Example:

**.**MODEL TINY

**.**CODE

ORG 100H

ENTRY: JMP START

MESSAGE DB ‘COSC 204’ , ‘$’

START: ; Display the string

MOV AH , 09H

MOV DX , OFFSET MESSAGE

INT 21H

; Return to DOS

MOV AX , 4C00H

INT 21H

END ENTRY

# ASSEMBLING, LINKING, AND EXECUTING 8086 ASSEMBLY LANGUAGE PROGRAMS

The assemblers and linkers have a number of options, which are of the form: **/option**. These options are not case-sensitive for MASM Version 5.00, TASM Version 4.1, LINK Version 5.3, and TLINK Version 7.1. However, they are case sensitive for ML.EXE (MASM Version 6.00 and above). In addition, the options for MASM (Version 5.00), TASM, LINK, TLINK may or may not be separated by blanks. The options for ML.EXE must be separated by blanks.

In what follows we only mention the options that we will use in this course. The elements within square brackets are optional:

Masm Version 5.00

Usage: MASM

or

MASM [options] source[.asm] , [object[.obj]] , [list[.lst]] , [cref[.crf]] [;]

/c Generate cross-reference in listing

/l Generate normal listing

/la Generate expanded listing

/n Suppress symbol tables in listing

/z Display source line for each error message

/Zi Generate symbolic information for CodeView

/Zd Generate line-number information

-------------------------------------------------------------------------------------------------------------------------------------------

Turbo Assembler Version 4.1

Usage: TASM [options] source[.asm] , [object[.obj]] , [list[.lst]] , [cref[.crf]] [;]

/c Generate cross-reference in listing

/l Generate normal listing

/la Generate expanded listing

/m# Allow # multiple passes to resolve forward references

/n Suppress symbol tables in listing

/z Display source line with error message

/zi, /zd, /zn Debug info: zi=full, zd=line numbers only, zn=none

Microsoft (R) Segmented Executable Linker Version 5.31.009

Usage: LINK

or

LINK [options] objfiles, runfile, mapfile, libfiles, deffile [;]

/CODEVIEW Full symbolic debug information for CodeView debugger

/CO Full symbolic debug information for CodeView debugger

/LINENUMBERS

/TINY Create COM file

/T Create COM file

-------------------------------------------------------------------------------------------------------------------------------------------

Turbo Link Version 7.1.26.1.

Syntax: TLINK [options] objfiles, runfile, mapfile, libfiles, deffile, resfiles [;]

/x No map

/v Full symbolic debug information

/t Create COM file (same as /Tdc)

/k Suppress "No stack" warning msg

-------------------------------------------------------------------------------------------------------------------------------------------

ML [options] filelist [/link linkOptions]

/AT Enable tiny model (.COM file)

/c Assemble without linking

/Sn Suppress symbol-table listing

/Fe<file> Name executable

/Fl[file] Generate listing

/Fo<file> Name object file

/Zd Add line number debug info

/Zi Add symbolic debug info

/Zm Enable MASM 5.10 compatibility

/link <linker options and libraries>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

LAB EXERCISES:

Download assemblers, linkers, text editors, prog1.asm, prog2.asm, prog3.asm and prog4.asm from the PC ics-said and then do the following exercises:

EXERCISE#1:

* 1. Assemble the exe-format program prog1.asm, with full segment definitions, using the command line:

masm /z prog1[.asm] , prog1[.obj] , prog1[.lst]

Where the items in square brackets are optional. This will generate the object file prog1.obj and the listing file prog1.lst. The assembler will prompt you for the name of the cross-reference file. Ignore this buy pressing the Enter key.

Note: If masm.exe is not in the same folder as prog1.asm, you must supply the full path in each of the three occurrences of prog1 in the above command line.

* 1. If the assembling in (a) was successful, link prog1.obj by using the command line:

link prog1[.obj] , prog1[.exe]

Note: If link.exe is not in the same folder as prog1.obj, you must supply the full path in each of the two occurrences of prog1 in the above command line. The linker will prompt you for the name of the library file, and then the name of the definitions file. Ignore each prompt by pressing the Enter key.

* 1. If the linking in (b) is successful, execute prog1.exe by either double clicking its icon or by using the command line:

prog1[.exe]

NOTE:

1. Steps similar to (a) , and (b) are used to assemble using TASM and TLINK.
2. It is possible to assemble by MASM 5.00 by double-clicking on its icon.
3. It is possible to link using LINK Version 5.31 by double-clicking on its icon.

EXERCISE#2:

(a) Assemble the exe-format program prog2.asm, with simplified segment definitions, using the command line:

masm /z prog2[.asm] , prog2[.obj] , prog2[.lst]

Where the items in square brackets are optional. This will generate the object file prog2.obj and the listing file prog2.lst. The assembler will prompt you for the name of the cross-reference file. Ignore this buy pressing the Enter key.

Note: If masm.exe is not in the same folder as prog2.asm, you must supply the full path in each of the three occurrences of prog2 in the above command line.

1. If the assembling in (a) was successful, link prog1.obj by using the command line:

link prog2[.obj] , prog2[.exe]

Note: If link.exe is not in the same folder as prog1.obj, you must supply the full path in each of the two occurrences of prog2 in the above command line. The linker will prompt you for the name of the library file, and then the name of the definitions file. Ignore each prompt by pressing the Enter key.

1. If the linking in (b) is successful, execute prog2.exe by either double clicking its icon or by using the command line:

prog2[.exe]

EXERCISE#3:

(a) Assemble the com-format program prog3.asm, with full segment definitions, using the command line:

masm /z prog3[.asm] , prog3[.obj] , prog3[.lst]

Where the items in square brackets are optional. This will generate the object file prog3.obj and the listing file prog3.lst. The assembler will prompt you for the name of the cross-reference file. Ignore this buy pressing the Enter key.

Note: If masm.exe is not in the same folder as prog3.asm, you must supply the full path in each of the three occurrences of prog3 in the above command line.

1. If the assembling in (a) was successful, link prog3.obj by using the command line:

link **/t** prog3[.obj] , prog3[.com]

Note: If link.exe is not in the same folder as prog3.obj, you must supply the full path in each of the two occurrences of prog3 in the above command line. The linker will prompt you for the name of the library file, and then the name of the definitions file. Ignore each prompt by pressing the Enter key.

1. If the linking in (b) is successful, execute prog3.com by either double clicking its icon or by using the command line:

prog3[.com]

NOTE: Similar steps can be used to assemble and link prog4.asm. This is a com-format program with simplified sgment directives.

EXERCISE#4:

(a) Assemble and link the exe-format program prog2.asm, with full segment definitions, using the command line:

ml /Fl prog1.asm

Note: **The extension .asm must be supplied**. This will generate the object file prog1.obj, the listing file prog1.lst and the executable file prog1.exe.

Note: ml.exe uses link.exe, therefore these files must be in the same folder.

1. If the assebling and linking in (a) is successful, execute prog1.exe by either double clicking its icon or by using the command line:

prog1[.exe]

EXERCISE#5:

(a) Assemble and link the com-format program prog3.asm, with full segment definitions, using the command line:

ml /Fl /AT prog3.asm

Note: **The extension .asm must be supplied**. This will generate the object file prog3.obj, the listing file prog3.lst and the executable file prog3.com.

Note: ml.exe uses link.exe, therefore these files must be in the same folder.

1. If the assebling and linking in (a) is successful, execute prog3.exe by either double clicking its icon or by using the command line:

prog3[.exe]

EXERCISE#6:

Assemble the exe-format program prog1.asm, and then try to link prog1.obj using the command line:

link /t prog1.obj , prog1.com

what happens?

EXERCISE#7:

Assemble the com-format program prog3.asm, link prog3.obj using the command line:

link prog3 , prog3

then execute the generated prog3.exe. What is the program output?

COSC 204 LABWORK#2

**Objectives:**

An introduction to:

1. DOS functions 01H , 02H, 08H, 09H, 40H, and 0AH
2. INT, MOV, INC, DEC, ADD, and SUB instructions.
3. Using turbo debugger version 5.0
4. Write an 8086 **COM-format** assembly language program which prompts for a lowercase letter x1 in the range **a** to **x**:

PLEASE, ENTER A LOWERCASE LETTER IN THE RANGE a TO x:

it then displays an output of the form:

X3 X2 X1

where X2 and X3 are the two uppercase characters which come after uppercase X1 in the alphabet.

For example, if the input is **c** then the output is:

E D C

**Note:** each two characters in the output are separated by one two blanks.

**Hint:** LowercaseLetter - 20H = UppercaseLetter

1. Write an 8086 **EXE-format** assembly language program that prompts for and reads a user’s name USERNAME (of maximum length 30 characters). It then displays a message of the form:

Mr. USERNAME studies 8086 Assembly language programming.

For example, if USERNAME is Said Abdallah then the output is:

Mr. Said Abdallah studies 8086 Assembly language programming.

------------------------------------------------------------------------------------------------------------------------------------

**EXERCIZE ON USING TURBO DEBBUGER**

In this exercise the program **add2.asm** is provided. To avoid the problem of supplying the full path in the command line, copy this file in the same folder containing TASM, TLINK, and the Turbo debugger files.

1. Assemble **add2.asm** using the command line:

tasm /m3/z/zi add2 , add2 , add2

1. Link add2.obj using the command line:

tlink /v/k add2 , add2

1. Invoke Turbo debugger by the command line:

td add2.exe

1. Execute the program one instruction at a time (i.e., trace the program) by right-clicking on the menu-option **“Trace**” or by pressing **F7**

Observe how the arrow indicating the current instruction moves.

1. When you are prompted to enter the first number, type **5**.
2. Select the menu-option: **Data**

Select: **add watch**

Type **VAR2** in the dialogue box that appears.

Select: **OK**

Observe the variable VAR2 and its current stored value 00H appears in the Watches window.

1. Select the menu-option: **View**

Select the option: **CPU**

Observe the Segment: offset addresses of instructions and data items, the machine language of the instructions, current register values, current flag values ((**c**)arry flag, (**z**)ero flag, (**s**)ign flag, (**o**)verflow flag, (**p**)arity flag, (**i**)interrupt enable flag, (**d**)irection flag), and finally observe what is in the stack.

1. While maintaining the CPU window open, trace the program by repeatedly pressing **F7**, observing how register values change. Trace the program up to when the second prompt appears.
2. Type: **3** .
3. Close the CPU window.

Notice that the current instruction is: **MOV BL , AL ; Save number in BL**

1. Click on the option **“Trace”** or press **F7** to continue tracing the program up to the next instruction: **MOV VAR2 , AL**.

Press **F7** to execute the instruction. Observe how the value of VAR2 changes to 33H in the watches window.

1. Place the cursor anywhere on the line containing the instruction: **INT 21H** following the comment: **; Display second number**

Press **F2**

This sets a breakpoint (i.e., a point at which program execution stops). Observe that a red line highlights the breakpoint.

1. Select the option: **“Run**” or press **F9**.

This will result in the execution of all the instructions from the current instruction up to; but not including the nearest break point.

1. Select the option: **Window**.

Select: **User Screen**.

Observe that the current output is:

**ENTER A POSITIVE INTEGER <= 9**

**ENTER A POSITIVE INTEGER <= 4**

1. **+**
2. Press any key to return to the debugger.
3. Execute the current **INT 21H** instruction by clicking on **“Trace”** or by pressing **F7**
4. Repeat the steps in (14.). Observe the current output is now:

# ENTER A POSITIVE INTEGER <= 9

**ENTER A POSITIVE INTEGER <= 4**

**5 + 3**

1. Press any key to return to the debugger.
2. Select the option: **“Run”** or press **F9** to execute the program to the end (since there are no further breakpoints).

A box with the message: **Terminated, exit code 0** appears. This indicates normal program termination.

Click on **OK** in this dialogue.

1. Repeat the steps in (14.). Observe the current output is now:

# ENTER A POSITIVE INTEGER <= 9

**ENTER A POSITIVE INTEGER <= 4**

**5 + 3 = 8**

1. Press any key to return to the debugger.
2. Place the cursor on the line containing the breakpoint (i.e., the line highlighted by red). Press **F2** to remove the break.
3. Select the option: **View**

Select: **Dump**

This will display the contents of the data segment.

**Right-click** on the dump window and select **Display as** to change the way the dump is displayed.

To view the contents of the stack segment, select **go to . . .** and then type **ss:sp** in the dialogue box which appears. Select **ok**.

1. Close the dump window.
2. Select option **File** , then **Quit** to return to the DOS prompt

COSC 204 LABWORK#3

Objectives:

1. Learning how to link separately assembled modules from the command line.
2. Learning how to use LIB.EXE and TLIB.EXE to create and modify object module libraries.
3. Learning how to use object module libraries.
4. Learning the difference between “Trace Into” and “Step Over” tracing modes of a debugger.

## LINKING SEPARATELY ASSEMBLED MODULES

Suppose PROG1.ASM, PROG2.ASM, PROG3.ASM are three modules to be executed as a single program. The steps are:

1. Assemble each of the files separately, in whatever order. Ignore any warning of a missing stack segment. This will create the object files:

PROG1.OBJ , PROG2.OBJ , and PROG3.OBJ

1. Link the three object files:

>LINK PROG1 + PROG2 + PROG3 , EXEFILE\_NAME (for exe-format modules)

or

>LINK /T PROG1 + PROG2 + PROG3 , COMFILE\_NAME (for com-format modules)

PROG1, PROG2, and PROG3 in the above command lines may appear in any order. If the name for the executable file is missing, the first object file name in the list is taken as the name of the executable file.

The same format is used if the linker is TLINK.

Note: If any of the object files is in a different directory from the directory in which the linker is located, then the full path of that file must be given.

1. Execute the generated executable file by the command:

>EXECUTABLE\_FILE\_NAME

A full path may be required.

Note: In an IDE such as The Programmers Work Bench, the concept of a project is used in assembling, linking, and executing multi-module programs.

------------------------------------------------------------------------------------------------------------------------------------------------

DOWNLOAD THE FILES LIB.EXE, TLIB.EXE, DSPLYMDL.ASM, READMDL.ASM, CALLER1.ASM, AND CALLER2.ASM FROM THE PC ICS-SAID AND THEN DO THE FOLLOWING TWO EXERCISES:

1. ASSEMBLE CALLER1.ASM AND THEN DSPLYMDL.ASM, LINK CALLER1.OBJ AND DSPLYMDL.OBJ, AND FINALLY EXECUTE THE GENERATED EXECUTABLE FILE.
2. ASSEMBLE CALLER2.ASM AND THEN READMDL.ASM, LINK CALLER2.OBJ, DSPLYMDL.OBJ, AND READMDL.OBJ, AND FINALLY EXECUTE THE GENERATED EXECUTABLE FILE.

Note: The contents of the downloaded **.**asm modules are given below:

------------------------------------------------------------------------------------------------------------------------------------------------

DsplyMdl.asm:

PUBLIC DISPLAY\_CHAR , DISPLAY\_STRING\_WITH\_FNCTN\_09H , DISPLAY\_STRING\_WITH\_FNCTN\_40H

CODE SEGMENT

ASSUME CS : CODE

DISPLAY\_CHAR PROC FAR

; Enter with DL the character to be displayed

PUSH AX ; save AX

MOV AH , 02H ; display character passed in DL

INT 21H ;

POP AX ; restore AX

RET

DISPLAY\_CHAR ENDP

DISPLAY\_STRING\_WITH\_FNCTN\_09H PROC FAR

; Enter with DX the offset of the string to be displayed and DS the segment number of the string.

PUSH AX ; save AX

MOV AH , 09H ; display a $-terminated string

INT 21H ;

POP AX ; restore AX

RET

DISPLAY\_STRING\_WITH\_FNCTN\_09H ENDP

DISPLAY\_STRING\_WITH\_FNCTN\_40H PROC FAR

; Enter with DX the offset of the string to be displayed, DS the segment number of the string, and CX the length of ; the string

PUSH AX ; save AX

PUSH BX ; save BX

MOV AH , 40H ; display string

MOV BX , 01H ; handle for the screen

INT 21H ;

POP BX ; restore BX

POP AX ; restore AX

RET

DISPLAY\_STRING\_WITH\_FNCTN\_40H ENDP

CODE ENDS

END

------------------------------------------------------------------------------------------------------------------------------------------------

ReadMdl.asm:

PUBLIC READ\_CHAR\_WITH\_ECHO , READ\_CHAR\_WITHOUT\_ECHO , READ\_STRING

DATA SEGMENT

BUFFER DB 81 , 82 DUP(?)

DATA ENDS

CODE SEGMENT

ASSUME CS : CODE , DS : DATA

READ\_CHAR\_WITH\_ECHO PROC FAR

PUSH CX ; save CX

MOV CH , AH ; save AH

MOV AH , 01H ; read character with echo

INT 21H ;

MOV AH , CH ; restore AH

POP CX ; restore CX

RET

READ\_CHAR\_WITH\_ECHO ENDP

READ\_CHAR\_WITHOUT\_ECHO PROC FAR

PUSH CX ; save CX

MOV CH , AH ; save AH

MOV AH , 08H ; read character without echo

INT 21H ;

MOV AH , CH ; restore AH

POP CX ; restore CX

RET

READ\_CHAR\_WITH\_ECHO ENDP

READ\_STRING PROC FAR

; returns BP (this module's DS value), SI (length of input string) , DI (offset of input string)

PUSH AX ; save AX

PUSH BX ; save BX

PUSH DX ; save DX

PUSH DS ; save DS of caller

MOV AX , DATA ; initialize DS

MOV DS , AX ;

MOV AH , 0AH ; read string into BUFFER

LEA DX , BUFFER ;

INT 21H ;

MOV BP , DS ; return DS value of this module's DATA segment

LEA SI , BUFFER[2] ; return the offset of the input string

MOV BH , 0 ; return length of the input string

MOV BL , BUFFER[1] ;

MOV DI , BX ;

POP DS ; restore the callers DS

POP DX ; restore DX

POP BX ; restore BX

POP AX ; restore AX

RET

READ\_STRING ENDP

CODE ENDS

END

------------------------------------------------------------------------------------------------------------------------

Caller1.asm:

EXTRN DISPLAY\_STRING\_WITH\_FNCTN\_40H : FAR , DISPLAY\_STRING\_WITH\_FNCTN\_09H : FAR

STACKSG SEGMENT STACK

DB 0400H DUP(?)

STACKSG ENDS

DATA SEGMENT

MSG1 DB 'I LIKE ASSEMBLY PROGRAMMING VERY MUCH', 0DH , 0AH

LEN\_MSG1 EQU $ - MSG1

MSG2 DB 'THAT IS WHY I HEREBY RESOLVE TO WORK HARD', '$'

DATA ENDS

CODE SEGMENT

ASSUME CS : CODE , DS : DATA , SS : STACKSG

ENTRY: MOV AX , DATA

MOV DS , AX

LEA DX , MSG1

MOV CX , LEN\_MSG1

CALL DISPLAY\_STRING\_WITH\_FNCTN\_40H

LEA DX , MSG2

CALL DISPLAY\_STRING\_WITH\_FNCTN\_09H

MOV AX , 4C00H

INT 21H

CODE ENDS

END ENTRY

------------------------------------------------------------------------------------------------------------------------------------------------

Caller2.asm:

EXTRN DISPLAY\_STRING\_WITH\_FNCTN\_40H : FAR , DISPLAY\_STRING\_WITH\_FNCTN\_09H : FAR

EXTRN READ\_STRING : FAR

STACKSG SEGMENT STACK

DB 0400H DUP(?)

STACKSG ENDS

DATA SEGMENT

MSG1 DB 'PLEASE ENTER YOUR NAME (MAX LENGTH 80 CHARACTERS):', 0DH , 0AH

LEN\_MSG1 EQU $ - MSG1

MSG2 DB 0DH , 0AH, 0DH, 0AH, 'WELCOME TO THE MINI-WORLD OF ASSEMBLY PROGRAMMING MR. ', '$'

DATA ENDS

CODE SEGMENT

ASSUME CS : CODE , DS : DATA , SS : STACKSG

ENTRY: MOV AX , DATA

MOV DS , AX

LEA DX , MSG1

MOV CX , LEN\_MSG1

CALL DISPLAY\_STRING\_WITH\_FNCTN\_40H

CALL READ\_STRING ; Returns the length of the string in DI,

; the offset address of the string in SI,

; and the segment address of the string in BP

LEA DX , MSG2

CALL DISPLAY\_STRING\_WITH\_FNCTN\_09H

MOV CX , DI ; Assign the length of the string to CX

MOV DX , SI ; Assign the offset address of the string to DX

MOV DS , BP ; Assign the segment address of the string to DS

CALL DISPLAY\_STRING\_WITH\_FNCTN\_40H

MOV AX , 4C00H

INT 21H

CODE ENDS

END ENTRY

# CREATING AND USING LIBRARIES OF OBJECT MODULES

The utility LIB.EXE (Microsoft Library Manager) or TLIB.EXE (Turbo Library Manager) can be used to create libraries of object modules. Such libraries form an important software tool for the assembly language programmer.

The usage of these two utilities is given below:

------------------------------------------------------------------------------------------------------------------------------------------------

LIB.EXE (Microsoft Library Manager) Version 3.20.010

Usage: LIB library\_name [options] [commands] [,listfile [,newlibrary]] [;]

Where:

1. the items in square brackets are optional.
2. **library\_name** identifies the library to be created or updated. It may include the path.
3. **newlibrary** identifies the output library file. This entry is ignored if the library library\_name is being created; otherwise if this field is empty, and the library library\_name does exist, LIB.EXE creates a backup copy of the library called library\_name.BAK
4. **listfile** identifies the output file for the library listing. This is a cross-reference file with information about all FAR procedures and modules in the library.
5. **Options:**

/? : display LIB options

/HELP : display help on LIB

/IGNORECASE : ignore case on names

/NOEXTDICTIONARY : do not build extended dictionary

/NOIGNORECASE : do not ignore case on names

/NOLOGO : do not display signon banner

/PAGESIZE:n : set library page size to n. The default is 16 bytes.

1. **Commands:**

A command is of the form:

command name

where name is the name of a single object file or a library name. The name may be separated from the command by zero or more blanks. The command is performed on a single object file if name is that of a single object file; otherwise if name is a library name then the command is performed on all object files contained in the library. The commands and their meanings are given in the following table:

|  |  |
| --- | --- |
| Command symbol | Meaning |
| + | Add to an object file or object files to the library. |
| - | Delete an object file or object files from the library. |
| \* | Copy (extract) an object file or object files from the library to an object file or object files with the same names. |
| -+ | Replace an object file or object files from the library by an object file with the same name or names as in the command. |
| -\* | Move (delete and extract) an object file or object files from the library to an object file or object files with same names. |

Note:

1. If name is more than 8 characters in length it is truncated to 8 characters by LIB.EXE. (TLIB.EXE accepts names that are longer then 8 characters).
2. If name is the name of a library and no extension is supplied then **.**LIB is the default extension.
3. When one double-clicks on LIB.EXE it displays prompts in the following sequence:
   1. **Library name:**
   2. If the library supplied in response to the above prompt does not exist, the following prompt is displayed:

**Library does not exist. Create? (y/n)**

* 1. If the library does exist or if the response to the above prompt was **y**, the following prompt is displayed:

**Operations:**

* 1. **List file:**

Note: Double-clicking on TLIB.EXE only result in the display of the usage of TLIB.EXE.

1. Both LIB.EXE and TLIB.EXE will reject the addition of an object file to a library if that object file contains a name in its PUBLIC directive that already appears in the PUBLIC directive of one of the modules in the library. LIB.EXE will issue a warning of the form:

Warning U415: ‘xxxxxxx’: symbol defined in module yyyyyy, redefinition ignored.

TLIB.EXE will issue a warning of the form:

Warning: ‘xxxxxxx’ already in LIB, not changed !

------------------------------------------------------------------------------------------------------------------------------------------------

TLIB (Turbo Library Manager) Version 4.00

Usage: TLIB library\_name [/C] [/E] [/P] [/0] commands [, listfile]

library\_name library file (may include the path)

commands sequence of operations to be performed (optional)

listfile file name for listing file (optional)

The commands are the same as those of LIB.EXE except that TLIB.EXE also supports the commands:

+- Replace an object file or object files from the library.

\*- Move (delete and extract) an object file or object files from the library.

Options:

/C case-sensitive library

/E create extended dictionary

/PSIZE set the library page size to SIZE. The default is 16 bytes.

/0 purge comment records

Note: If the library library\_name is modified, TLIB.EXE creates a backup copy of the library called library\_name.BAK

------------------------------------------------------------------------------------------------------------------------------------------------

EXERSICE 3 (CREATING A LIBRARY OF OBJECT MODULES)

Since DsplyMdl.obj, and ReadMdl.obj were created in Exercise 1 and 2, create a library by double clicking on LIB.EXE and then responding as shown:

Library name: **inout**

Library does not exist. Create? (y/n): **y**

Operations: **+ readmdl + dsplymdl**

List file: **inout.txt**

You can also create the library and generate a listing file by the command line:

>**LIB inout + readmdl + dsplymdl , inout.txt**

Note: **inout.txt** will have the contents:

------------------------------------------------------------------------------------------------------------------------------------------------

DISPLAY\_CHAR......dsplymdl DISPLAY\_STRING\_WITH\_FNCTN\_09H..dsplymdl

DISPLAY\_STRING\_WITH\_FNCTN\_40H..dsplymdl READ\_CHAR\_WITHOUT\_ECHO..readmdl

READ\_CHAR\_WITH\_ECHO..readmdl READ\_STRING.......readmdl

readmdl Offset: 00000010H Code and data size: 8dH

READ\_CHAR\_WITHOUT\_ECHO READ\_CHAR\_WITH\_ECHO

READ\_STRING

dsplymdl Offset: 00000110H Code and data size: 1aH

DISPLAY\_CHAR DISPLAY\_STRING\_WITH\_FNCTN\_09H DISPLAY\_STRING\_WITH\_FNCTN\_40H

------------------------------------------------------------------------------------------------------------------------------------------------

Note: If you use TLIB.EXE to generate the list file, its contents will be:

------------------------------------------------------------------------------------------------------------------------------------------------

Publics by module

dsplymdl size = 26

DISPLAY\_CHAR DISPLAY\_STRING\_WITH\_FNCTN\_09H

DISPLAY\_STRING\_WITH\_FNCTN\_40H

readmdl size = 141

READ\_CHAR\_WITHOUT\_ECHO READ\_CHAR\_WITH\_ECHO

READ\_STRING

------------------------------------------------------------------------------------------------------------------------------------------------

EXERCISE 4 (USING A LIBRARY OF OBJECT MODULES)

1. Delete the executable files that were generated in Exercise 1 and 2.
2. Link caller1.obj to in\_out.lib using the command line:

>**LINK caller1.obj + inout.lib**

this will generate the executable file caller1.exe

1. Link caller2.obj to in\_out.lib by the command line:

>**LINK caller2.obj + inout.lib**

this will generate the executable file caller2.exe

----------------------------------------------------------------------------------------------------------------------------------------------EXERCISE 5 (PROGRAMMING ASSIGNMENT)

(a) Write an exe-format, 8086 assembly language program, which has full segment directives, to prompt for and read a character (DON’T USE READ\_STRING). The program must then display an output of the form:

THE INPUT CHARACTER IS:

followed by the character that was read. Use DSPLY\_CHAR to display the character.

Your program must use the FAR procedures defined in **ReadMdl.asm** and **DsplyMdl.asm** for:

1. Displaying the prompt.
2. Reading the character.
3. Displaying the message: THE INPUT CHARACTER IS:
4. Displaying the character.

(b) Assemble the program, then link the object file to **ReadMdl.obj** and **DsplyMdl.obj**, and finally execute the generated executable file.

(c) Link the object file to **in\_out.lib**, and then execute the generated executable file.

------------------------------------------------------------------------------------------------------------------------------------------------

EXERCISE 6 (PROGRAMMING ASSIGNMENT)

1. Write an 8086 **EXE-format** assembly language program that prompts for and reads a user’s name USERNAME (of maximum length 50 characters). It then prompts for and reads the name of the user’s town USERTOWN (of maximum length 30 characters). It then displays a message of the form:

Mr. USERNAME lives in

USERTOWN town.

Your program **must** contain appropriate NEAR procedures to read a string and to display a string.

1. Assemble your program, adding debug information to the object file, by a command line of the form:

**TASM /zi prog , prog , prog**

1. Link the program, adding debug information to the executable file, by a command line of the form:

**TLINK /v prog , prog**

1. After executing your program, invoke Turbo debugger by a command line of the form:

**TD prog**

1. Notice the difference in tracing the program by F7 (Trace Into) and F8 (Step over).
2. Observe the values pushed on top of the stack when your execute procedure calls.
3. Observe the values popped from the stack when you execute RET instructions.

Note:

* 1. To reset a program after normal termination, use **Ctrl-F2** or select the ***Run*** option and then select ***Program reset***.
  2. To view the top of the stack, select the ***View*** option, select ***Dump***, right click on the dump window, select ***Goto . . .*** , type ***ss : sp*** in the dialogue box which appears, and then click on the ***OK*** button.

COSC 204 LABWORK#4

Objectives:

Introducing students to:

1. MACROS

(a) macros calling other macros, (b) Local and global identifiers, (c) **.**LALL and **.**SALL directives, (d) macro include-files.

1. PROCUDURES WHICH PASS AND RETURN VALUES THROUGH THE STACK.
2. USING BATCH FILES IN ASSEMBLING AND LINKING ASSEMBLY LANGUAGE PROGRAMS.

Download macro1.asm, macro2.asm, alem.bat, alet.bat, mlb.bat, and the folder IrvineASMEditor from PC ics-said

Note: The contents of macro1.asm and macro2.asm are given below:

macro1.asm:

-------------------------------------------------------------------------------------------------------------------------------------------

DISPLAY\_CHAR MACRO CHAR

PUSH AX

PUSH DX

MOV AH , 02H

MOV DL , CHAR

INT 21H

POP DX

POP AX

ENDM

CRLF MACRO

DISPLAY\_CHAR 0DH

DISPLAY\_CHAR 0AH

ENDM

STACK SEGMENT STACK

DB 400 DUP(?)

STACK ENDS

CODE SEGMENT

ASSUME CS : CODE , SS : STACK

ENTRY:

DISPLAY\_CHAR 'A'

CRLF

DISPLAY\_CHAR 'B'

MOV AX , 4C00H

INT 21H

CODE ENDS

END ENTRY

-------------------------------------------------------------------------------------------------------------------------------------------

macro2.asm:

-------------------------------------------------------------------------------------------------------------------------------------------

TOWNS MACRO

LOCAL TOWN

DATA SEGMENT

TOWN DB 'DHAHRAN', 0DH , 0AH , '$'

DATA ENDS

PUSH DX

MOV DX , OFFSET TOWN ; local TOWN

CALL DISPLAY\_STRING ; DISPLAY\_STRING is a global identifier

MOV DX , OFFSET COUNTRY ; COUNTRY is a global identifier

CALL DISPLAY\_STRING

POP DX

ENDM

STACK SEGMENT STACK

DB 0400H DUP(?)

STACK ENDS

DATA SEGMENT

TOWN DB 'RIYADH', 0DH , 0AH , '$'

COUNTRY DB 'SAUDI ARABIA', 0DH , 0AH , '$'

DATA ENDS

CODE SEGMENT

ASSUME CS : CODE , DS : DATA , SS : STACK

MAIN PROC

MOV AX, DATA

MOV DS , AX

MOV DX , OFFSET TOWN ;global TOWN

CALL DISPLAY\_STRING

TOWNS ; a macro call

MOV AX , 4C00H

INT 21H

MAIN ENDP

DISPLAY\_STRING PROC

PUSH AX

MOV AH , 09H

INT 21H

POP AX

RET

DISPLAY\_STRING ENDP

CODE ENDS

END MAIN

-------------------------------------------------------------------------------------------------------------------------------------------

**.**SALL (Suppress ALL) and **.**LALL (List ALL) directives

The **.**SALL directive suppresses macro expansions in the listing file. The directive can be placed anywhere in the **.**asm file before the calls to the macro or macros whose expansions are to be suppressed. To re-enable macro expansions, use the **.**LALL directive.

-------------------------------------------------------------------------------------------------------------------------------------------

TASK#1

Generate macro1.lst and macro2.lst by:

1. Not using .SALL in the .asm files, and
2. Using .SALL in the .asm files.

in both cases observe the listing files.

-------------------------------------------------------------------------------------------------------------------------------------------

TASK#2

1. Copy the macros: DISPLAY\_CHAR and CRLF of macro1.asm to a text file mymacros.txt or mymacros.mac and then delete them from macro1.asm.
2. include the text file containing the macros in the modified macros1.asm by a statement of the form:

include c:\workarea\mymacros.txt

(Assuming that mymacros.txt is located in the specified folder)

1. Assemble macro1.asm, link macro1.obj, and then execute macro1.exe.

-------------------------------------------------------------------------------------------------------------------------------------------

PROGRAMMING ASSIGNMENTS:

1. Write an 8086 Assembly language program that prompts for and reads a student’s name, it then prompts for and reads a student’s major. It finally displays an output of the form:

Mr. STUDENTNAME majors in STUDENTMAJOR

The reading must be done using a macro READ\_STRING that has one parameter: the name of the buffer to read into. All string displays, including the buffer displays, must be done in a macro STRING\_DISPLAY that has the header:

STRING\_DISPLAY MACRO STRINGNAME , LENSTRING

Where the actual parameter corresponding to STRINGNAME is the name of the string to be displayed, and LENSTRING corresponds to a ***16-bit*** length of the string to be displayed. Your macro must use DOS function 40H.

1. Write an 8086 Assembly language module that contains the following definitions in its data segment:

MESSAGE DB “I LIKE ICS-232 VERY MUCH”

MESSAGELEN EQU $ - MESSAGE

The module calls a separately assembled 8086 Assembly language module, passing to it the segment address of the data segment in the DS register, the offset address of MESSAGE in the stack, and the length of MESSAGE in the stack. The called module then uses DOS function 40H to display the string MESSAGE.

COSC 204 LABWORK#5

Objectives:

Exercises on:

1. JMP, Conditional jump, LOOP, conditional loop instructions.

2. One-dimensional array manipulation.

1. Write a **com-format**, 8086 Assembly language program that prompts for and reads a character. It then displays one of the massages: THE CHARACTER IS A HEXADECIMAL DIGIT or THE CHARACTER IS NOT A HEXADECIMAL DIGIT accordingly.

Note: The reading of the character, the display of the prompt, and the display of the appropriate message must be done using **two** appropriate macros defined in a macro include-file.

1. Write an **exe-format**, 8086 Assembly language program which:
   1. prompts for and reads a **maximum of** 10 ASCII digits into a one-dimensional array of size 10.
   2. prompts for and reads a search character.
   3. determines whether the search character is a member of the one-dimensional array or not.
   4. displays one of the messages: THE CHARACTER IS FOUND or THE CHARACTER IS NOT FOUND accordingly.

Your program must:

1. use appropriate procedures.

(ii) check for input validity. In case of invalid input, the program should beep (the bell character is 07H), return the cursor one column back (the backspace character is 08H), and give the user a chance to enter another value.

(iii) be able to process less than 10 elements for task (a), that is, if the user presses the enter key (0DH) before 10 values are read, the reading loop must terminate.

(iii) use an appropriate conditional loop instruction in performing task (c)

1. Write a **com\_format**, 8086 Assembly language program which, starting with the characters ‘A’, ‘9’ and the blank character (‘ ’) only, will generate the following output:

ABCD

99999999

EFGH

88888888

IJKL

77777777

MNOP

66666666

QRST

55555555

Your program must :

* 1. contain no data definitions.
  2. use a minimal number of loops.
  3. use simplified segment directives.

1. Write a program that prompts for and reads a string of maximum length 80 using DOS function 0AH. The program then converts each lowercase letter in the string, if any, to uppercase and then displays an output of the form:

THE OUTPUT STRING IS: ***converted\_string***

Where ***converted\_string*** is the input string in which all lowercase letters, if any, have been converted to uppercase.

Use appropriate macros or procedures in your solution.

COSC 204 LABWORK#6

Objectives:

1. Command-line text processing.
2. Two-dimensional array manipulation (BasedIndexed addressing mode).
3. Structures, Nested structures, arrays of structures.

Download cmndln2.asm, struct5.asm , and struct6.asm from the PC ics-said

TASK#1:

1. Assemble and link , using **ml.exe**, both struct5.asm and struct6.asm

Use a command line of the form:

ml /Fl progname.asm

1. Execute each of the generated executable files.

TASK#2

1. Assemble and then link cmndln2.asm
2. Execute cmndln2.exe using the command line:

cmndln2 assembly language programming

PROGRAMMING ASSIGNMENTS

1. Write an exe-format 8086 Assembly language program that accepts a command line of the form:

x+y

or

x-y

Where x and y are digits such that if the operator is + then x + y ≤ 9, and if the operator is **-** , then x ≥y .

Your program should display the result of the operation on the two digits. Don’t test for input validity; but make sure your command-line satisfies the given conditions.

1. Modify the given struct5.asm such that it also displays the day and the month of birth of each student.
2. A two-dimensional array is defined as:

ARRAY DB 1 , 0 , 2 , 1

DB 2 , 1 , 0 , 0

DB 0 , 1 , 1 , 0

Write an 8086 Assembly language program to find the sum of the array. Your program must use BasedIndexed addressing mode. The program must be able to find the sum of any two-dimensional byte array of size 3 \* 4 and whose sum is a value in the range 0 to 9.

COSC 204 LABWORK#7

Objectives:

1. RECURSIVE PROCEDURES.

Terminating gracefully from a recursive procedure in which an error condition is detected.

Preventing the stack overflow condition in recursive procedures.

1. STAND ALONE PROCEDURES.

Using macros to write stand-alone procedures.

Using Nested Procedures to write stand -alone procedures.

1. HEXADECIMAL I/O ROUTINES.
2. DETECTING SIGNED AND UNSIGNED OVERFLOW.

DETECTING UNSIGNED OVERFLOW

* The Carry Flag (CF) is set if there is unsigned overflow. Unsigned overflow occurs if the result is too large to fit into the destination operand. In a program unsigned overflow can be detected by a construct of the form:

Flag modifying instruction

JC UNSIGNED\_OVERFLOW

**.**

**. ; action if there is no unsigned overflow**

**.**

JMP DONE

UNSIGNED\_OVERFLOW**:**

**.**

**. ; action if there is unsigned overflow**

**.**

DONE**:**

DETECTING SIGNED OVERFLOW

* The Overflow Flag (OF) is set if there is signed overflow. The Carry Flag is irrelevant for signed operations. Signed overflow occurs if the result is too large to fit into the destination operand or if the destination sign bit changes. In a program unsigned overflow can be detected by a construct of the form:

Flag modifying instruction

JO SIGNED\_OVERFLOW

**.**

**. ; action if there is no signed overflow**

**.**

JMP DONE

SIGNED\_OVERFLOW**:**

**.**

**. ; action if there is signed overflow**

**.**

DONE**:**

HEXADECIMAL I/O ROUTINES

1. Hexadecimal output.

Write a procedure to display the contents of BX in hexadecimal.

The algorithm uses the conversions:

DecimalNumber(0, 1, 2, **. . .** ,9) + 30H HexadecimalDigit('0', '1', '2', **. . .** ,'9')

DecimalNumber(10, 11, 12, 13, 14, 15) + 37H HexadecimalDigit('A', 'B', 'C', 'D', 'E', 'F')

There are 4 hexadecimal digits in a 16-bit operand. The algorithm displays one hexadecimal digit of BX at a time:

for 4 times do

MOV DL , BH ; put the current two highest hex digits of BX in DL

Shift DL 4 times to the right ; leave the current highest hex digit of BX in DL

if((DL) ≤ 9)then

(DL) := (DL) + 30H ; convert to hex digit in {'0', '1', '2', **. . .** , '9'}

else

(DL) := (DL) + 37H ; convert to hex digit in {'A', 'B', 'C', 'D', 'E', 'F'}

endif

output: (DL)

Rotate BX left 4 times ; put the next hex digit in the highest 4 bits of BX

endfor

output: 'H'

HEX\_DISPLAY PROC

; Displays the contents of BX in hexadecimal

PUSH AX

PUSH CX

PUSH DX

MOV AH , 02H

MOV CX , 4

FOR: MOV DL , BH

PUSH CX

MOV CL , 4

SHR DL , CL

CMP DL , 9

JA LETTER

ADD DL , 30H

JMP DISPLAY

LETTER: ADD DL , 37H

DISPLAY: INT 21H

ROL BX , CL

POP CX

LOOP FOR

MOV DL , 'H'

INT 21H

POP DX

POP CX

POP AX

RET

HEX\_DISPLAY ENDP

1. Hexadecimal input

Write a procedure to input a hexadecimal value in the BX register. The reading should stop if either 4 hexadecimal digits have been read or if the "Enter" key (ASCII code: 0DH) is pressed. For any invalid input, the procedure should beep and give the user a chance of entering another value. If only 0DH is entered the procedure should set the Carry Flag.

The algorithm uses the following conversions:

character in {'0' , '1' , '2' , . . . , '9'} -30H digit in {0 , 1 , 2 , . . . , 9}

character in {'A' , 'B' , 'C', . . . , 'F'} -37H digit in {10 , 11 , 12 , . . . , 15}

character in {'a' , 'b' , 'c', . . . , 'f'} -57H digit in {10 , 11 , 12 , . . . , 15}

The pseudo-code algorithm is:

MOV BX , 0

MOV CL , 4

MOV CH , 0 ; counts the number of hexadecimal digits read

do

{

Read: character

if(character is a hexadecimal digit)then

{

convert character to binary

INC CH

SHL BX , CL ; shift zeroes in the lowest 4 bits of BX

Insert character in the lowest 4 bits of BX

}

else if( character ≠ 0DH )then

{

Beep ;

Move the cursor back ;

}

else

break ;

endif

} while ( (CH) < 4 )

if( (CH) = 0 )then

STC

else

CLC

endif

HEX\_INPUT PROC

; Inputs a 16-bit hexadecimal value in the BX register

PUSH AX

PUSH CX

PUSH DX

MOV BX , 0

MOV CL , 4

MOV CH , 0

START: MOV AH , 01H

INT 21H

CMP AL , '0'

JB INVALID?

CMP AL , '9'

JBE VALID1

CMP AL , 'A'

JB INVALID

CMP AL , 'F'

JBE VALID2

CMP AL , 'a'

JB INVALID

CMP AL , 'f'

JBE VALID3

JMP INVALID

VALID1: SUB AL , 30H

JMP INSERT

VALID2: SUB AL , 37H

JMP INSERT

VALID3: SUB AL , 57H

INSERT: INC CH

SHL BX , CL

OR BL , AL

JMP NEXT

INVALID?: CMP AL , 0DH

JE END\_DO\_WHILE

INVALID: MOV AH , 02H ; beep

MOV DL , 07H ;

INT 21H ;

MOV DL , 08H ; backspace

INT 21H ;

NEXT: CMP CH , 4

JB START

END\_DO\_WHILE:

CMP CH , 0

JE NO\_INPUT

CLC

JMP EXIT

NO\_INPUT: STC

EXIT: POP DX

POP CX

POP AX

RET

HEX\_INPUT ENDP

DOWNLOAD THE FOLDER LABWORK7 FROM THE PC ics-said

---------------------------------------------------------------------------------------------------------------------------------------

TASK#1 (NESTED PROCEDURES)

* 1. Assemble nstdproc.asm, link nstdproc.obj, and execute nsrdproc.com
  2. Modify nstdproc.asm such that the procedure crlf is still within the procedure dsply\_char but at the beginning. Assemble the modified nstdproc.asm, link nstdproc.obj, and execute nsrdproc.com. WHAT IS THE OUTPUT?
  3. Assemble nstdprc2.asm, link nstdprc2.obj, and execute nstdprc2.com
  4. Modify nstdprc2.asm such that the procedure crlf and string\_display2 are still within the procedure dsply\_string but at the beginning. Assemble the modified nstdprc2.asm, link nstdprc2.obj, and execute nsrdprc2.com. WHAT IS THE OUTPUT?
  5. From (b) and (d) what can you conclude about the placement of nested procedures?

NOTE: MASM VERSION 4.00 AND LOWER VERSIONS DO NOT SUPPORT NESTED PROCEDURE DEFINITIONS. TASM VERSION 3.2 AND HIGHER VERSIONS, MASM VERSION 5.00 AND HIGHER VERSIONS DO SUPPORT NESTED PROCEDURE DEFINITIONS.

---------------------------------------------------------------------------------------------------------------------------------------

TASK#2 (NESTED MACROS)

1. Study the program nstdmac.asm. This is a demonstration of the fact that macro definitions can appear inside other macros.
2. Assemble the program, link, and then execute it.

---------------------------------------------------------------------------------------------------------------------------------------

TASK#3 (RECURSIVE PROCEDURES, USING MACROS TO WRITE STAND-ALONE PROCEDURES)

1. Study the programs rcsvstrg.asm and rcsvsum2.asm. Specifically note how macros are used to write stand-alone procedures. Also note how rcsvsum2.asm terminates gracefully if an error condition is detected within the recursion chain.
2. Assemble rcsvstrg.asm, link rcrsvstrg.obj, and execute rcsvstrg.exe.
3. Assemble rcsvsum2.asm, link rcsvsum2.obj, and execute rcsvsum2.exe.

---------------------------------------------------------------------------------------------------------------------------------------

PROGRAMMING ASSIGNMENT:

1. Using the procedures HEX\_INPUT and HEX\_DISPLAY given in the file **hex.asm**, write a complete exe-format, 8086 Assembly language program that prompts for and reads two **signed**, 16-bit hexadecimal numbers X and Y. It then displays an output of the form:

X + Y = Z

Where Z is the sum.

If there is signed overflow the message SIGNED OVERFLOW must be displayed on the line following the above output (You may find it useful to use PUSHF and then POPF).

Note: The procedure HEX\_INPUT takes care of invalid input; specifically it sets the carry-flag if the user does not enter any input. Your main procedure must terminate if any error condition is detected by HEX\_INPUT.

**Test your program for values given below.**

r’s complement addition

Addition is performed in the usual manner, irrespective of the signs of the addends. **Any final carry is discarded**.

# Signed addition overflow

**The addition of two signed binary or hexadecimal numbers will cause an overflow if the sign of the addends are similar; but they are different from the sign of the result.**

For decimal signed addition, overflow is simply determined if the result is outside the range -128 to +127 for 8-bit additions, and outside the range -32768 to +32767 for 16-bit additions.

Example: Perform each of the following signed additions and determine whether overflow will occur or not: (a) 483FH + 645AH (b) FFE7H + FFF6H (c) E9FFH + 8CF0H (d) 0206H + FFB0H

Solution:

1. 483FH (+ve) (b) FFE7H (-ve) (c) E9FFH (-ve)

+ 645AH (+ve) + FFF6H (-ve) + 8CF0H (-ve)

AC99H (-ve) (1)FFDDH (-ve) (1)76EFH (+ve)

SIGNED OVERFLOW ↑ ↑

discard discard

NO SIGNED OVERFLOW SIGNED OVERFLOW

(d) 0206H (+ve)

+ FFB0H (-ve)

(1)01B6H (+ve)

↑

discard

NO SIGNED OVERFLOW; addends have opposite signs

1. Modify the program you wrote in (1.) such that the program prompts for and reads two **unsigned** hexadecimal numbers. If there is unsigned overflow the message to be displayed is: UNSIGNED OVERFLOW.

Unsigned addition overflow

An n-bit, unsigned addition overflow occurs if the sum is outside the allowed range of values for an n-bit number. Overflow condition is detected, computationally, for binary and hexadecimal addition by the existence of a final carry. For decimal addition overflow is detected, computationally, by the sum being outside the range. Since the CPU performs all additions in binary, decimal overflow also result in a final carry.

In case of unsigned addition overflow, the value in the destination operand is the unsigned number obtained by discarding the final carry. This value will not be the correct sum.

Example:

EA9BH

+ FFF6H

1EA91H

There is unsigned overflow.

Value in destination is EA91H

COSC 204 LABWORK#9

MOV AH , 0FH ; Get current video mode

INT 10H

A video mode is selected by the BIOS function:

MOV AH , 00H

MOV AL , video\_mode\_number

INT 10H

INT 10H subfunction 05H : Set active video-page

For text modes 0 - 3 , this subfunction enable the selection of the page that is to display. Pages may be 0 - 3 in 80-column mode or 0-7 in 40-column mode. Text written in one page is kept intact while another page is being displayed.

MOV AH , 05H

MOV AL , video-page#

INT 10H

INT 10H subfunction 09H: Display character and its attribute at current cursor position

The subfunction displays a character together with its attribute, in text- or graphics-mode, starting at the current cursor position, a number of times specified in the CX register. The subfunction does not advance the cursor beyond the last character displayed.

MOV AH , 09H

MOV AL , character

MOV BH , video-page#

MOV BL , attribute ; Attribute (text) or color (graphics)

MOV CX , repetition-factor

INT 10H

INT 10H subfunction 13H : Display character string or character

The subfunction displays a string or a character with options of setting attributes and moving the cursor.

MOV AX, DATA

MOV DS , AX

MOV ES , AX

**. . .**

MOV AH , 13H

MOV AL , function# ; 0, 1, 2, or 3

MOV BH , video-page#

MOV BL , attribute

MOV BP , OFFSET string ; Address of string in ES:BP

MOV CX , length-of-string

MOV DH , row#

MOV DL , column#

INT 10H

The four functions in AL are:

1. Display attribute and string; do not advance cursor.
2. Display attribute and string; advance cursor.

2 Display character then attribute; do not advance cursor.

3 Display character then attribute; advance cursor.

. INT 10H subfunction 02H : Set cursor position

MOV AH , 02H

MOV BH , current video-page#

MOV DH , row#

MOV DL , column#

INT 10H

To make the cursor invisible:

Set the starting scan line to an illegal value by turning on bit 5 in CH:

MOV AH , 01H

OR CH , 00100000B

INT 10H

INT 10H subfunction 06H : Scroll window up

MOV AH , 06H

MOV AL , number-of-lines-to-be-scrolled ; (0 means **all** window lines)

MOV CH , upper-row#

MOV CL , left-column#

MOV DH , lower-row#

MOV DL , right-column#

MOV BH , video-attribute for each blank line

INT 10H

INT 10H subfunction 08H : Read character and its attribute at current cursor position

The subfunction returns the character at the current cursor position, on the selected video-page, in either text or graphics mode. The character is returned in AL and its attribute in AH.

MOV AH , 08H

MOV BH , video-page#

INT 10H

In graphics mode the operation returns 00H for a non-ASCII character. Since this operation reads one character at a time, you have to code a loop using subfunction 02H (Set cursor position) to read successive characters.

# CGA GRAPHICS MODES

MOV AH , 0BH ; Select graphics palette

MOV BH , 01H ;

MOV BL , palette ; (00H or 01H)

INT 10H

-------------------------------------------------------------------------------------------------------------------------------------------

MOV AH , 0BH ; Select background color

MOV BH , 00H ;

MOV BL , color-code ; background color: Video modes 4 and 5

INT 10H

-------------------------------------------------------------------------------------------------------------------------------------------

Any given pixel can be independently assigned a color by a call to subfunction 0CH of INT 10H (The function is valid only in video modes 4 , 5, and 6):

MOV AH , 0CH

MOV AL , color-code

MOV CX , column# ; Pixel,

MOV DX , row# ; coordinates

INT 10H

In video modes 4 and 5, the AL register may contain a value in the range 00H through 03H, which will specify a color depending on the currently active palette and background:

|  |  |  |
| --- | --- | --- |
| AL | Palette 0 | Palette 1 |
| 00H | background | background |
| 01H | green | Cyan |
| 02H | red | Magenta |
| 03H | yellow | White |

------------------------------------------------------------------------------------------------------------------------------------------

COSC 204 LABWORK#10

Objectives:

1. User defined TSRs.
2. Background processing (The interception of INT 1CH)
3. Hot-key TSRs (The interception of both INT 1CH and INT 09H)

NOTE:

RUN ALL THE PROGRAMS IN THIS LABWORK FROM A DOS WINDOW AND NOT BY DOUBLE-CLICKING ON THE .COM OR .EXE ICON.

EXITING THE DOS WINDOW IN A WINDOWS 98 OR WINDOWS NT ENVIRONMENT WILL REMOVE FROM MEMORY ALL THE TSRs YOU HAVE INSTALLED.

-----------------------------------------------------------------------------------------------------------------------------------

STUDY ASSIGNMENT#1 (USER DEFINED TSRs)

The com-format program **isr62h.asm** installs a TSR which when invoked by the call:

INT 62H

displays, on the screen, the message: WELCOME TO THE INTERESTING WORLD OF TSR ROUTINES

The com-format program **i62h\_usr.asm** invokes INT 62H five times, and hence displays the above message five times.

1. Assemble, link, and then execute isr62h. This will install the TSR and return control to DOS.
2. Assemble, link, and then execute i62h\_usr.
3. Study the structures of both **isr62h.asm** and **i62h\_usr.asm**.

STUDY ASSIGNMENT#2 (USER DEFINED TSRs)

The com-format program **isr3.asm** installs a TSR which when invoked by the call sequence:

MOV AH , function\_number (00H, 01H, 02H)

MOV DX , Number

INT 61H

will display an unsigned number in either hexadecimal, decimal, or binary, depending on the function number in AH. The TSR displays an error message if an invalid function number is passed in AH. The caller to the TSR must ensure that the number passed in DX is valid.

The exe-format program **isr3user.asm** prompts for and reads an unsigned decimal number, and a function number. If the number is valid it passes these two values to **isr3** TSR.

1. Assemble, link, and then execute isr3. This will install the TSR and return control to DOS.
2. Assemble, link, and then execute isr3user.
3. Study the structures of both **isr3.asm** and **isr3user.asm**.

--------------------------------------------------------------------------------------------------------------------------------------

STUDY ASSIGNMENT#3 (BACKGROUND PROCESSING – THE INTERCEPTION OF INT 1CH)

NOTE: IN THIS ASSIGNMENT, MAKE SURE YOU DON’T INSTALL A BACKGROUND PROCESS MORE THAN ONCE. ALSO MAKE SURE YOU INSTALL ONLY ONE BACKGROUND PROCESS AT A TIME. EXIT THE DOS WINDOW IF YOU WANT TO INSTALL ANOTHER BACKGROUND PROCESS.

1. **The com-format program** clr7disr.asm **installs a background process that displays a window, at the upper right corner of the screen, with a different color every five seconds. The background process is deactivated after seven colors have been displayed.**
2. **Assemble, link, and then execute** clr7disr.asm**.**
3. **Study the structure of** clr7disr.asm.

**Exit the DOS window before doing part (b)**

1. **The com-format program** bkisr2.asm **installs a background process that beeps every fifteen seconds.**
2. **Assemble, link, and then execute** bkisr2.asm**.**

**(ii) Study the structure of** bkisr2.asm**.**

**---------------------------------------------------------------------------------------------------------------------------------------**

**STUDY ASSIGNMENT#4 (HOT-KEY TSRs)**

NOTE: IN THIS ASSIGNMENT, MAKE SURE YOU DON’T INSTALL A HOT-KEY TSR MORE THAN ONCE. ALSO MAKE SURE YOU INSTALL ONLY ONE HOT-KEY TSR AT A TIME. EXIT THE DOS WINDOW IF YOU WANT TO INSTALL ANOTHER HOT-KEY TSR.

* 1. **The com-format program htkyTSR2.asm installs a hot-key TSR that displays a blue window at the upper right corner of the screen. After this, whenever the key combination Ctrl-RightShift is pressed the color of the window toggles between blue and green.**

1. **Assemble, link, and then execute** htkyTSR2**.**

**(ii) Study the structure of** htkyTSR2.asm

Exit the DOS window before doing part (b)

* 1. **The com-format program htkytsr3.asm installs a hot-key TSR beeps whenever the key combination Ctrl-RightShift is pressed.**

1. **Assemble, link, and then execute** htkytsr3**.**

**(ii) Study the structure of** htkytsr3.asm

**-------------------------------------------------------------------------------------------------------------------------------------------**

**PROGRAMMING ASSIGNMENT#1 (USER DEFINED TSR)**

**(a) Modify isr3.asm such that:**

1. **your name is defined as a string (Note: Don’t use NAME for the string name because it is a reserved word)**
2. **the length of your name is computed.**
3. **The DS and ES registers are initialized as:**

**MOV BX , CS**

**MOV DS , BX**

**MOV ES , BX**

1. **The screen is cleared and given the attribute “blinking yellow on blue”.**
2. **Your name is displayed centered on the screen, using BIOS function 13H.**
3. **After the display, the TSR waits for a keystroke (Use DOS function 08H).**
4. **Control is returned to the caller by IRET**

**(b) Assemble, link, and then execute your modified isr3 to install it in memory.**

**(c) Write, assemble, link, and execute a program that invokes your modified isr3 TSR.**

**PROGRAMMING ASSIGNMENT#2 (HOT-KEY TSRs)**

**The program** htkytsr1.asm **installs a hot-key TSR that displays a blue window at the upper right corner of the screen, when the key combination Ctrl-RightShift is pressed. Modify this program such that when Ctrl-RightShift is pressed the TSR:**

1. **clears the whole screen, giving it the attribute “blinking yellow on a blue background”**
2. **places the cursor at the center of the screen. (Use BIOS function 02H).**
3. **displays the character T three times (Use BIOS function 09H).**

COSC 204 LABWORK#11

Objectives:

1. FILE I/O using File handles. (DOS functions: 3CH, 5BC, 3DH, 3EH, 41H, 42H, 3FH, 40H)
2. Searching for a text file in a directory (excluding its subdirectories). (DOS functions: 4EH, 1AH, 4FH)
3. Moving the file pointer using negative offsets.

# TASK#1

The given program **fldsply4.asm** prompts for and reads the path to a text file. If there is no I/O error, the contents of the text file are displayed on the screen, 24 lines at a time. Assemble, link, and then execute the program.

# TASK#2

The given program **fileptr2.asm** displays the last 20 bytes of the text file given in the file descriptor in its data segment. Modify the file descriptor such that it becomes that of a particular text file in your system. Assemble, link, and then execute the program.

# TASK#3

The given program **filefnd.asm** prompts for and reads the path to a text file. The file name may contain wild characters ‘\*’ or/and ‘?’. If there is no I/O error, the names of all files, in the subdirectory, matching the given file name are displayed on the screen. Assemble, link, and then execute the program.

# PROGRAMMING ASSIGNMENT#1

Write an 8086 Assembly language program that prompts for and reads the path, of maximum length 80, to a text file. If the file already exists the user should be prompted with a Y/N (Yes or No) question as to whether he wants to overwrite the existing file or not. If the answer is “N”, the program should terminate. If the answer is “Y” or if the text file does not exist, the program should prompt and then read from the user some text of maximum size 300 bytes. The text should then be written to the existing text file or to a new created text file respectively.

# PROGRAMMING ASSIGNMENT#2

Using a text editor create two text files. Write some text, not exceeding 500 characters, in each of the files. Write an 8086 Assembly language program that will append the text of one of the text files to the other text file. Your program must then display the contents of the resulting file on the screen.