**Chapter Five**

**Assembly Instructions**

1. **Assembly Instructions**

To program in assembly, the programmer should first understand the architecture of the machine where he/she make the programming. In addition, the programmer should know the programming language (including assembler). The machine (Intel 8086 for this course) was discussed in detail at the beginning of the course. It was also stated that assembly statements are divided into three parts: Instructions, directives and pseudo-opcodes. In the previous chapters, a detail discussion of pseudo-opcodes and directives was given. In this chapter instructions are discussed in depth. Students should give time and read detail of this chapter because it is the core of the course. Instructions are categorized into instruction sets. Instruction sets contain related instruction together in to a package. There are several kinds of instruction sets, in this chapter, different instruction found in the common instruction sets will be discussed in detail.

* 1. **Keyboard Input and Video Output**

Most of the Input/Output operations in ELASS are performed using the input output functions of the special subroutine called INT $21 (Interrupt No. $21). The function number is specified in AH register and other parameters and/or return values are specified in other registers depending on the function type.

The three most common functions of Int $21 for character input and output are:

* Function $01 Keyboard Input with echo
* Function $02 Character Output
* Function $08 Keyboard Input without echo
  + 1. **Function $01 - Keyboard Input with echo**

This function provides the functionality of a character input from the keyboard and places the return value in AL register.

Syntax: AH = $01

AL = Return value (ASCII code of the character)

Example: MOV AH, $01

INT $21 this sample code makes the program execution to expect a character input from the user and places the typed character in AL register.

* + 1. **Function $02 - Character Output**

This function provides the functionality of a character output on the standard output device (Screen). The Character to be displayed should be specified in DL register.

Syntax: AH = $02

AL = Character to be displayed. (ASCII code)

Example: MOV AH, $02

MOV DL, 'A'

INT $21 this sample code displays the character "A" on the standard output device (Screen).

* 1. **Subroutines**

In ELASS subroutines are sub programs that are defined with PROC (PROCedure) and ENDP (END Procedure) clause. Subroutines are useful to avoid repetitive program lines so that they can be define as a procedure with PROC directive and can be called from anywhere within the program and any number of times.

* + 1. **Procedures**

A procedure is a section of a program whose beginning is defined with a PROC (PROCedure) directive and whose termination is defined by an ENDP (END Procedure) directive.

*Format:* proc\_name PROC

:

proc\_name ENDP

* + 1. **The CALL Instruction**

The CALL instruction is used to invoke the code in a subroutine.

*Format:* CALL ***procname***

Where ***procname*** is the name of the procedure

* + 1. **The RET Instruction**

The RET (Return) statement returns the control to the calling program or to the parent procedure.

*Example:*

Code Segment

Main PROC

-

CALL CR\_LF

-

Main ENDP

CR\_LF PROC

MOV AH, $02

MOV DL, $0D

INT $21

MOV AH, $02

MOV DL, $0D

INT $21

RET

CR\_LF ENDP

Code EndS

End

* 1. **INPUT and OUTPUT**

The input/output functionality of ELASS is not flexible to distinguish between characters and numerical values. All input/output information is treated as character. This is the programmer's job to handle numeric inputs and outputs.

* + 1. **Characters Input/Output**

The three most common functions of Int $21 for character input and output are:

* Function $01 Keyboard Input with echo
* Function $02 Character Output
* Function $08 Keyboard Input without echo
  + 1. **Number Input/Output**

In ELASS there is no special function, which displays or takes as input numerical values. The DOS function $02 takes the ASCII value of the character as a parameter in DL.

For example:

*MOV DL, 'A' MOV DL, $41 MOV DL, 65*

*MOV AH, $02 MOV AH, $02 MOV DL, $02*

*INT $21 INT $21 INT $21*

Have the same effect and all instructions move the binary value *'01000001'* in DL register. The program segment displays the character 'A" as an output. All input output functions treats the values as a character.

Two-digit numerical value can be contained in a byte size register but it should be displayed as 2 characters, which is 2 bytes of code.

For example the hexa-decimal value $47 is byte-sized information. If we display the value $47 with function 02, a character which is represented by the ASCII value of $47 (which is 'G') will be displayed. In order to display the value $47 there should be some program code, which converts each digit to its equivalent ASCII value and displays it. The value $47 should be displayed as two bytes of characters '4' and '7' or $34 (ASCII code of '4') and $37 (ASCII code of '7')

In binary format the value *01000111* should be converted *00110100 00110111*

The following sample program code displays the hexa-decimal value contained in BX register

Display a number in hexadecimal notation.

HEX $

Code SEGMENT

Set the BX register.

MOV BX, $1234

MOV CX, $0004

Next\_Char: MOV SI, $0004

Next\_Bit: SHL BX, 1

RCL DL, 1

DEC SI

JNZ Next\_Bit

AND DL, $0F

OR DL, $30

CMP DL, $3A

JB Disp\_Val

ADD DL, $07

Disp\_Val: MOV AH, $02

INT $21

LOOP Next\_Char

MOV AX, $4C00

INT $21

Code ENDS

END

By converting hexa-decimal values in their equivalent decimal notation, the program can be modified to display Decimal notations.

It is also necessary to have a code which converts the input information as numerical value for further arithmetic operation.

* 1. **Control Transfer**

When the program loader loads the executable program the segment address of the code segment is loaded in CS (Code Segment register) and the offset address of the first program code in IP (Instruction Pointer register). Then by incrementing the IP value the next program code will be executed. By this manner all instructions in the program executed sequentially. The only time the IP address changes its value to execute a program code different from the next instruction is, when there is a Subroutine call, when there is a JMP statement or where there is a Loop instruction.

* + 1. **Jumps**

The **Jump** instruction makes the program flow to the instruction at specific place in the program, which is addressed by a label. In ELASS there are two types of Jump instruction:

* Unconditional Jump and
* Conditional Jump.

The unconditional jump transfers the program flow to the instruction at specific place in the program, which is addressed by a label without considering any conditions. Unconditional Jump has the following syntax:

JMP ***label***

Where ***label*** is a program address identifier

A conditional jump instruction will test the state of some specified status flag or flags and direct program flow accordingly. The general syntax of conditional Jump is:

JXX ***label***

Where ***label*** is a program address identifier and ***XX*** is a condition or status to be tested

The following are some conditional jumps:

JZ - Jump if Zero Flag is set JNZ - Jump if Zero Flag is cleared

JC - Jump if Carry Flag is set JNC - Jump if Carry Flag is cleared

JS - Jump if Sign Flag is set JNS - Jump if Sign Flag is cleared

JE - Jump if Equal JNE - Jump if Not Equal

JGE - Jump if Greater or Equal JLE - Jump if Less or Equal

The following program displays characters A to Z by incrementing the content of DL

Example : MOV DL, $41

MOV CX, 26

Next: MOV AH, 02

INT $21

INC DL

DEC CX

JNZ Next

* + 1. **Loops**

In ELASS Program Loops can be constructed using conditional statements and/or using the **LOOP** instruction. The LOOP instruction with CX register executes the program lines with in the Loop block CX number of times. The instruction decrements the content of the CX register and, if the result is not zero, directs program flow to the instruction at the address referenced by label.

Syntax: LOOP label

Example MOV DL, $41

MOV CX, 26

Next: MOV AH, 02

INT $21

INC DL

LOOP Next

* 1. **Strings**

ELASS can Input and Output strings using the special subroutine called INT $21. The function number is specified in AH register and other parameters and/or return values are specified in other registers and/or Memory references.

* + 1. **String Input/Output**

In INT $21 there are two functions which are specially used for string Input and Output. These are:

* Function $09 String Output and
* Function $0A Buffered Keyboard Input

*Function $09 - String Output*

This function provides the functionality of string output on the standard output device (Screen). This function requires the string to be defined in the data segment and terminated with "$" character. The segment offset address should be expressed in DS:DX register.

Syntax: AH = $09

DS:DX = Segment offset address of the first character of the string

Example: MOV AX, Data

MOV DS, AX

MOV DX, Offset Text

MOV AH, $09

INT $21

Data Segment

Text DB 'Sample Text$'

Data EndS

This sample code displays the string "Sample Text" On the screen.

*Function $0A - Buffered Keyboard Input*

This function provides the functionality of a buffered keyboard input or string input from the keyboard. The string entered from the keyboarded is saved in the buffer defined in the data segment. The first byte of the data definition statement should contain a value, which shows the expected number of characters to be entered including *Carriage Return ($0D)*. The second byte of the definition contains the actual number of bytes entered excluding *Carriage Return ($0D)*. The characters can be accessed starting from the third byte of the definition.

Syntax: AH = $0A

DS:DX = Segment offset address of the buffer

Example: MOV AX, Data

MOV DS, AX

MOV DX, Offset Buffer

MOV AH, $0A

INT $21

Data Segment

Buffer DB 30, 31 DUP (' ')

Data EndS

This sample code expects a string of 30 characters to be entered from the keyboard.

* + 1. **String Manipulation (reading assignment)**

*Copying a string* with MOVSB instruction

The MOVSB (MOV String of Bytes) instruction copies the contents of the byte of memory at DS: [SI] in to the byte of memory at ES: [DI]. It also automatically increments the contents of SI and DI register. It copies only a byte of information.

Syntax: MOVSB

The REPeat instruction with CX register can be used to repeat the instruction CX number of times. REP instruction executes the string manipulation instruction and decrement the content of CX and if it is not zero it repeats again.

The REP instruction has another version, i.e. REPZ and REPNZ.

REPZ instruction repeats the instruction while Zero Flag is set and CX is different from Zero.

REPNZ instruction repeats the instruction while Zero Flag is cleared and CX is different from Zero.

Syntax: REP Instruction

Example: MOV AX, Data

MOV DS, AX

MOV ES, AX

MOV CX, Size Text

MOV SI, Offset Text

MOV DI, Offset Buffer

REP MOVSB

Data Segment

Text DB 'Sample Text'

Buffer DB 30 DUP(' ')

Data EndS

* + 1. **Copying a string with LODSB and STOSB instructions**

The LODSB (LOaD String of Bytes) moves the content of the byte at DS: [SI] into the AL register and increments the contents of the SI register. STOSB (STOre LOaD String of Bytes) moves the content of the AL register into the byte at ES: [DI] and increments the contents of the DI register. The LODSB and STOSB together have the same effect as MOVSB. The only difference between the two instructions is, character manipulation is impossible while copying in MOVSB. In LODSB and STOSB is possible while the character is in AL register.

Syntax: LODSB

STOSB

Example: MOV AX, Data

MOV DS, AX

MOV ES, AX

MOV CX, Size Text

MOV SI, Offset Text

MOV DI, Offset Buffer

Next: LODSB

STOSB

Loop Next

Data Segment

Text DB 'Sample Text'

Buffer DB 30 DUP(' ')

Data EndS

* + 1. **Comparing two strings with CMPSB**

Two strings can be compared with CMPSB instruction and REPZ prefix. The CMPSB instruction compares the byte of information at DS:[SI] with the byte of information ES:[DI] and updates the flag register accordingly. The content of SI and DI register is also incremented automatically by 1.

Syntax: CMPSB

Example: MOV AX, Data

MOV DS, AX

MOV ES, AX

MOV CX, Size Text1

MOV SI, Offset Text1

MOV DI, Offset Text2

REPZ CMPSB

JZ Msg\_Equal

MOV DX, Offset Message2

MOV AH, $09

INT $21

JMP Exit

Msg\_Equal: MOV DX, Offset Message1

MOV AH, $09

INT $21

Exit: MOV AX $4C00

INT $21

Data Segment

Text1 DB 'Sample Text'

Text2 DB 'Sample TEXT'

Msg1 DB 'Texts are equal$'

Msg2 DB 'Texts are not Equal$'

Data EndS

* + 1. **Searching a character in the string**

A character stored in AL register can be searched in the string with SCASB instruction. The SCASB instruction compares the character in AL with the character referred by ES:[DI] and updates the status flag accordingly. The content of DI registers incremented by 1 automatically. With the help of REPNZ, the instruction can search the character specified in AL register in the string addressed by ES:[DI]. The REPNZ instruction repeats the SCASB instruction either CX number of times or while Zero flag is cleared.

Syntax: SCASB

Example: MOV AX, Data

MOV ES, AX

MOV CX, Size Text

MOV DI, Offset Text

MOV AL, 'T'

REPNZ SCASB

JNZ Msg\_NotFound

MOV DX, Offset Message1

MOV AH, $09

INT $21

JMP Exit

Msg\_NotFound: MOV DX, Offset Message2

MOV AH, $09

INT $21

Exit: MOV AX $4C00

INT $21

Data Segment

Text DB 'Sample Text'

Msg1 DB 'Character Found $'

Msg2 DB 'Character is not Found $'

Data EndS

* 1. **Arithmetic And Logical Operations**
     1. **Shift and Rotate instructions**

In general the shift and rotate instructions shift or rotate the content of a memory or a register to the left or to the right a number of times respectively. Both instructions has the following general syntax

XXX Destination, Count

Where XXX is either a Shift or Rotate instruction

Destination is either a register or memory reference and

Count is either 1 or CX register

* + - 1. **Shift Left**

Shift Left instruction shifts the content of the destination operand to the left and inserts a Zero bit to the right-most position. The left-most bit goes to the carry flag.

Syntax: SHL Destination, Count

Example1: MOV AL, $35 AL = 00110101

SHL AL, 1 AL = 01101010 CF = 0

Example2: MOV AL, $35 AL = 00110101

MOV CX, $03

SHL AL, CX AL = 10101000 CF = 1

* + - 1. **Shift Right**

Shift Right instruction shifts the content of the destination operand to the right and inserts a Zero bit to the left-most position. The right-most bit goes to the carry flag.

Syntax: SHR Destination, Count

Example1: MOV AL, $35 AL = 00110101

SHR AL, 1 AL = 00011010 CF = 1

Example2: MOV AL, $35 AL = 00110101

MOV CX, $03

SHR AL, CX AL = 00000110 CF = 1

* + - 1. **Rotate Left**

Rotate Left instruction shifts the content of the destination operand to the left and inserts the left-most bit to the right-most position. The left-most bit also goes to the Carry Flag.

Syntax: ROL Destination, Count

Example1: MOV AL, $35 AL = 00110101

ROL AL, 1 AL = 01101010 CF = 0

* + - 1. **Rotate Right**

Rotate Right instruction shifts the content of the destination operand to the right and inserts the right-most bit to the left-most position. The right-most bit also goes to the Carry Flag.

Syntax: ROL Destination, Count

Example1: MOV AL, $35 AL = 00110101

ROR AL, 1 AL = 10011010 CF = 1

* + - 1. **Rotate through Carry Left**

Rotate through Carry Left instruction shifts the content of the destination operand to the left and sets the carry flag with left-most bit. The previous value of the carry flag is inserted to the right-most position.

Syntax: RCL Destination, Count

Example1: MOV AL, $35 AL = 00110101 Assume CF = 1

RCR AL, 1 AL = 01101011 CF = 0

* + - 1. **Rotate through Carry Right**

Rotate through Carry Right instruction shifts the content of the destination operand to the right and sets the carry flag with right-most bit. The previous value of the carry flag is inserted to the left-most position.

Syntax: RCR Destination, Count

Example1: MOV AL, $35 AL = 00110101 Assume CF = 1

RCR AL, 1 AL = 10011010 CF = 1

* + 1. **Fixed Width Arithmetic** 
       1. **Addition**

In ELASS addition is performed using the **ADD** instruction. The ADD instruction has two parameters (Source and destination). This instruction adds the source and the destination values and put the result in the destination.

The destination can be 8-bit/16-bit register or memory reference. The destination also can be 8-bit/16-bit register, memory reference and also an immediate value.

Syntax: ADD destination, source

Example: ADD AX, $25

Adds the contents of AX with the value $25 and puts the result in AX register (AX = AX + $25)

ADD AH, BH

Adds the contents of AH and BH register and puts the result in AH register (AH = AH + BH)

ADD CL, DS:[DI]

Adds the contents of CL register with the value referred by DS;[DI] and puts the result in CL register

(CL = CL + DS:[DI])

* + - 1. **Subtraction**

In ELASS Subtraction is performed using the **SUB** instruction. The SUB instruction has two parameters (Source and destination). This instruction subtracts the source value from the destination and put the result in the destination.

The destination can be 8-bit/16-bit register or memory reference. The destination also can be 8-bit/16-bit register, memory reference and also an immediate value.

Syntax: SUB destination, source

Example: SUB AX, $25

Subtracts the value $25 from AX and puts the result in AX register (AX = AX - $25)

SUB AH, BH

Adds the contents of BH from AH register and puts the result in AH register (AH = AH - BH)

SUB CL, DS:[DI]

Subtracts the value referred by DS;[DI] from CL register and puts the result in CL register (CL = CL - DS:[DI])

* + - 1. **Multiplication**

In ELASS multiplication is performed using the **MUL** instruction. The MUL instruction has one operand. This instruction multiplies the operand with AL or AX registers. If the operand is 8-bit, the MUL instruction multiplies the operand with **AL** register and puts the product in **AX** register. If the operand is 16-bit, the MUL instruction multiplies the operand with AX and puts the result in DX/AX register pair.

Syntax: MUL operand

Example: MOV AL, $12

MOV BL, $03

MUL BL

Multiplies the value in BL and AL and puts the result in AX register (AX = $12 \* $03)

Example: MOV AX, $1234

MOV BX, $0023

MUL BX

Multiplies the value in BX and AX and puts the result in DX/AX register (DX/AX = $1234 \* $0023)

* + - 1. **Division**

In ELASS division is performed using the **DIV** instruction. The DIV instruction has one operand. This instruction divides the operand in to AX or DX/AX registers pair. If the operand is 8-bit, the DIV instruction divides the operand in to AX register and puts the integer part of the quotient in AL register and the remainders in AH register. If the operand is 16-bit, the DIV instruction divides the operand in to DX/AX registers pair and puts the integer part of the quotient in AX register and the remainders in DX register.

Syntax: DIV operand

Example: MOV AL, $12

MOV BL, $03

DIV BL

Divides the value in BL into AX and puts the integer part in AL and the remainder in AH register (integer AL = AX/ BL, remainder AH = AX/BL)

Example: MOV DX, $0000

MOV AX, $1234

MOV BX, $0023

DIV BX

Divides the value in BX into DX/AX registers pair and puts the integer part in AX and the remainder in DX. (Integer AX = DX/AX/ BX, remainder DX = DX/AX/BX)

* 1. **Disk Input/Output**

Using INT $21 it is possible to create or open a disk file and to modify it's content. There are a number of functions which help to Create, Open, Read or Write on a disk File. When a file is on the disk, it is referred or accessed by the file *Descriptor* (Which contains Drive Name, Path and File Name). When the file is opened, the full or partial content of the file will be loaded in the buffer, which is reserved by the operating system. The operating system returns a 16-bit randomly generated value, which is called a *File Handle* to the program. Further the program can modify the content of the file using the File Handle. The most widely used file related functions in ELASS are:

* Function $3C Create and/or Open a Disk file
* Function $3D Open a Disk file
* Function $3E Close a file
* Function $3F Read from a file
* Function $40 Write to a file
* Function $5B Create a New file
  + 1. **Creating/Opening a file with function $3C**

Function $3C will create or open an existing file by truncating its size to zero. Function $3C requires the file descriptor in DS:[DX] with null terminating string and its attribute in CX register. Most general-purpose files are created with attribute $00. The file attribute is 8-bit and should be expressed in CL register. CH register is usually $00. If there is an error while Opening and/or creating the file, Carry Flag will be set and AX register returns the ERROR Code. In NO ERROR AX register contains the File Handle.

Syntax: AH = $3C

CX = File Attribute

DS:[DX] = ASCIIZ File Descriptor

*On NO ERROR* CF = Cleared *On ERROR* CF = Set

AX = File Handle AX = Error code

File Attributes in CL register:

Bit 0 - Read Only, Bit 1 - Hidden, Bit 2 - System, Bit 5 - Archive

Example: The following program opens a file named 'A:\MyFile.txt'. If it doesn't exist it will create a new one. If there is an error while opening/creating the file the program will display an error message.

Code SEGMENT

MOV AX, Data

MOV DS, AX

MOV DX, Offset File\_Name

MOV CX, $00

MOV AH, $3C

INT $21

JNC Exit\_Prg

MOV DX, Offset Err\_Msg

MOV AH, $09

INT $21

Exit\_Prg: MOV AX, $4C00

INT $21

Code ENDS

Data SEGMENT

File\_Name DB 'A:\MyFile.txt', $00

Err\_Msg DB 'Can not Open/Create a File $'

Data ENDS

* + 1. **Closing a file with function $3E**

Function $3E will close an open file. Function $3E requires the file handle in BX register.

Syntax: AH = $3E

BX = File Handle

*On NO ERROR* CF = Cleared *On ERROR* CF = Set

AX = Error code

* + 1. **Opening a file with function $3D**

Function $3D will open an existing file. Function $3D requires the file descriptor in DS:[DX] with null terminating string and access code in AL register.

Syntax: AH = $3D

AL = Access code

DS:[DX] = ASCIIZ File Descriptor

*On NO ERROR* CF = Cleared *On ERROR* CF = Set

AX = File Handle AX = Error code

Access Code in AL register:

00 - Read Only, 01 - Write Only, 02 - Read/Write

Example: The following program opens a file named 'A:\MyFile.txt'. If it doesn't exist it will generate an error and the error message will be displayed.

Code SEGMENT

MOV AX, Data

MOV DS, AX

MOV DX, Offset File\_Name

MOV CX, $00

MOV AH, $3D

INT $21

JNC Exit\_Prg

MOV DX, Offset Err\_Msg

MOV AH, $09

INT $21

Exit\_Prg: MOV AX, $4C00

INT $21

Code ENDS

Data SEGMENT

File\_Name DB 'A:\MyFile.txt', $00

Err\_Msg DB 'Can not Open a File $'

Data ENDS

* + 1. **Creating a New File with function $5B**

Function $5B is the same as function $3C, except that it will generate an error message if a file exist with the same name.

Syntax: AH = $3B

CX = File Attribute

DS:[DX] = ASCIIZ File Descriptor

*On NO ERROR* CF = Cleared *On ERROR* CF = Set

AX = File Handle AX = Error code

File Attributes in CL register:

Bit 0 - Read Only, Bit 1 - Hidden, Bit 2 - System, Bit 5 - Archive

Example: The following program creates a new file named 'A:\MyFile.txt'. If it exists it will generate an error and the program displays an error message.

Code SEGMENT

MOV AX, Data

MOV DS, AX

MOV DX, Offset File\_Name

MOV CX, $00

MOV AH, $5B

INT $21

JNC Exit\_Prg

MOV DX, Offset Err\_Msg

MOV AH, $09

INT $21

Exit\_Prg: MOV AX, $4C00

INT $21

Code ENDS

Data SEGMENT

File\_Name DB 'A:\MyFile.txt', $00

Err\_Msg DB 'Can not Open/Create a File $'

Data ENDS

* + 1. **Reading from a File with function $3F**

Function $3F will read the content of a file in to the program buffer. This function requires the file handle in BX, the buffer address in DS:[DX] and the number of bytes to be read in CX register. If the size of the file is less than the value specified in CX register, the actual file size will be read and the actual number of bytes be returned in AX register.

Syntax: AH = $3F

BX = File Handle

CX = Number of Bytes to be read

DS:[DX] = Buffer address

*On NO ERROR* CF = Cleared *On ERROR* CF = Set

AX = Actual number of bytes read AX = Error code

Example: The following program opens a file named 'A:\MyFile.txt' and reads it's content to the program buffer.

Code SEGMENT

MOV AX, Data

MOV DS, AX

MOV DX, Offset File\_Name

MOV AH, $3D

MOV AL, $00

INT $21

JC Disp\_ErrMsg

MOV BX, AX

MOV CX, Size Buffer

MOV DX, Offset Buffer

MOV AH, $3F

INT $21

JC Disp\_ErrMsg

MOV AH, 3E

INT $21

JMP Exit\_Prg

Disp\_ErrMsg: MOV DX, Offset Err\_Msg

MOV AH, $09

INT $21

Exit\_Prg: MOV AX, $4C00

INT $21

Code ENDS

Data SEGMENT

File\_Name DB 'A:\MyFile.txt', $00

Buffer DB 100 DUP(' ')

Err\_Msg DB 'Can not Open/Read a File $'

Data ENDS

* + 1. **Writing to a File with function $40**

Function $40 will write the content of a program buffer to a file. This function requires the file handle in BX, the buffer address in DS:[DX] and the number of bytes to be written in CX register.

Syntax: AH = $40

BX = File Handle

CX = Number of Bytes to be written

DS:[DX] = Buffer address

*On NO ERROR* CF = Cleared *On ERROR* CF = Set

AX = Actual number of bytes written AX = Error code

Example: The following program opens a file named 'A:\MyFile.txt' and writes the content of the program buffer (100 'A's) in to the file.

Code SEGMENT

MOV AX, Data

MOV DS, AX

MOV DX, Offset File\_Name

MOV AH, $3D

MOV AL, $01

INT $21

JC Disp\_ErrMsg

MOV BX, AX

MOV CX, Size Buffer

MOV DX, Offset Buffer

MOV AH, $40

INT $21

JC Disp\_ErrMsg

MOV AH, 3E

INT $21

JMP Exit\_Prg

Disp\_ErrMsg: MOV DX, Offset Err\_Msg

MOV AH, $09

INT $21

Exit\_Prg: MOV AX, $4C00

INT $21

Code ENDS

Data SEGMENT

File\_Name DB 'A:\MyFile.txt', $00

Buffer DB 100 DUP('A')

Err\_Msg DB 'Can not Open/Write a File $'

Data ENDS

* + 1. **More About Disk I/O**

Starting with DOS v2.0, Microsoft introduced a set of file handling procedures which (finally) made disk file access bearable under MS-DOS. Not only bearable, but actually easy to use!. File commands which deal with filenames (Create, Open, Delete, Rename, and others) are passed the address of a zero-terminated pathname. Those that actually open a file (Create and Open) return a file handle as the result (assuming, of course, that there wasn’t an error). This file handle is used with other calls (read, write, seek, close, etc.) to gain access to the file you have opened. In this respect, a file handle is not unlike a file variable in Pascal.

All the following DOS filing functions (all of which are the sub functions of the Interrupt $21) return an error status in the carry flag. If the carry flag is clear when DOS returns to your program, then the operation was completed successfully. If the carry flag is set upon return, then some sort of error has occurred and the AX register contains the error number. The actual error return values will be discussed along with each function in the following sections.

***1. Open File***

**Function (AH)**: **3Dh**

**Entry parameters**:

**AL-** File Access Value

0- File opened for reading

1- File opened for writing

2- File opened for reading and writing

**DS:DX**- Point at a zero terminated string containing the filename.

**Exit Parameters**: If the **carry is set**, **AX** contains one of the following error codes:

2- File not found

4- Too many open files

5- Access denied

12- Invalid access

If the carry is clear, **AX** contains the file handle value assigned by DOS.

***2. Create File***

**Function (AH)**: **3Ch**

**Entry parameters**:

**DS:DX**- Point at a zero terminated string containing the filename.

**CX**- File attribute

**Exit Parameters**: If the **carry is set**, **AX** contains one of the following error codes:

3- Path not found

4- Too many open files

5- Access denied

If the carry is clear, AX is returned containing the file handle.

***3. Close File***

**Function (AH)**: **3Eh**

**Entry parameters**:

**BX**- File Handle.

**CX**- File attribute

**Exit Parameters**: If the carry flag is set, ax contains 6, the only possible error, which is an invalid handle error.

This call is used to close a file opened with the Open or Create commands above. It is passed the file handle in the BX register and, assuming the file handle is valid, closes the specified file.

***4. Read from a File***

**Function (AH)**: **3Fh**

**Entry parameters**:

BX- File handle

CX- Number of bytes to read

DS:DX- Array large enough to hold bytes read

**Exit Parameters**: If the carry flag is set, AX contains one of the following error codes

5- Access denied

6- Invalid handle

If the carry flag is clear, AX contains the number of bytes actually read from the file.

The read function is used to read some number of bytes from a file. The actual number of bytes is specified by the CX register upon entry into DOS. The file handle, which specifies the file from which the bytes are to be read, is passed in the BX register. The DS:DX register contains the address of a buffer into which the bytes read from the file are to be stored. On return, if there wasn’t an error, the AX register contains the number of bytes actually read. Unless the end of file (EOF) was reached, this number will match the value passed to DOS in the CX register. If the end of file has been reached, the value return in AX will be somewhere between zero and the value passed to DOS in the CX register. *This is the only test for the EOF condition.*

***5. Write to a File***

**Function (AH)**: **40h**

**Entry parameters**:

BX- File handle

CX- Number of bytes to write

DS:DX- Address of buffer containing data to write

**Exit Parameters**: If the carry is set, ax contains one of the following error codes

5- Accessed denied

6- Invalid handle

If the carry is clear on return, ax contains the number of bytes actually written to the file.

This call is almost the converse of the read command presented earlier. It writes the specified number of bytes at DS:DX to the file rather than reading them. On return, if the number of bytes written to the file is not equal to the number originally specified in the CX register, the disk is full and this should be treated as an error.

***6. Seek (Move file pointer)***

**Function (AH)**: **42h**

**Entry parameters**:

AL- Method of moving

0- Offset specified is from the beginning of the file.

1- Offset specified is distance from the current file pointer.

2- The pointer is moved to the end of the file minus the specified offset.

BX- File handle.

CX:DX- Distance to move, in bytes.

**Exit Parameters**: If the carry is set, AX contains one of the following error codes

1- Invalid function

6- Invalid handle

If the carry is clear, DX:AX contains the new file position

This command is used to move the file pointer around in a random access file. There are three methods of moving the file pointer, an absolute distance within the file (if al=0), some positive distance from the current file position (if AL=1), or some distance from the end of the file (if AL=2). If AL doesn’t contain 0, 1, or 2, DOS will return an invalid function error. If this call is successfully completed, the next byte read or written will occur at the specified location.

***7. Delete File***

**Function (AH)**: **41h**

**Entry parameters**:

DS:DX- Address of pathname to delete

**Exit Parameters**: If carry set, AX contains one of the following error codes

2- File not found

5- Access denied

This function will delete the specified file from the directory. The filename must be an unambiguous filename (i.e., it cannot contain any wildcard characters).

***8. Rename File***

**Function (AH)**: **56h**

**Entry parameters**:

DS:DX- Pointer to pathname of existing file

ES:DI- Pointer to new pathname

**Exit Parameters**: If carry set, ax contains one of the following error codes

2- File not found

5- Access denied

17- Not the same device

This command serves two purposes: it allows you to rename one file to another and it allows you to move a file from one directory to another (as long as the two subdirectories are on the same disk).

***9. Change/Get File Attributes***

**Function (AH)**: **43h**

**Entry parameters**:

AL- Sub function code

0- Return file attributes in cx

1- Set file attributes to those in cx

CX- Attribute to be set if AL=01

DS:DX- address of pathname

**Exit Parameters**: If carry set, AX contains one of the following error codes:

1- Invalid function

3- Pathname not found

5- Access denied

If the carry is clear and the sub function was zero, then CX will contain the file’s attributes.

***Examples***

; This Program will Create a new file called NameList.txt

; and writes a list of names from the program buffer to the file

; The file name is defined in the program buffer

Hex $

Code segment

Mov BX, Data

Mov DS, BX

;Creating/Open a File

Mov AH, $3C

Mov CX, $00

Mov DX, offset File\_Name

Int $21

;Write list of names to the File

Mov BX, AX

Mov AH, $40

Mov CX, Size Name\_list

Mov DX, Offset Name\_List

Int $21

;Close the File

Mov AH, $3E

Int $21

Mov AX, $4C00

Int $21

Code EndS

Data Segment

File\_Name db 'NameList.txt', $00

Name\_List db 'Abebe, Kebede, Tesfaye.'

Data EndS

End

; This Program will open a file called NameList.txt

; and display the content on the screen

Hex $

Code segment

Mov BX, Data

Mov DS, BX

;Opening a File

Mov AH, $3D

Mov AL, $00

Mov DX, offset File\_Name

Int $21

;Read the content of a File

Mov BX, AX

Mov AH, $3F

Mov CX, Size Name\_list

Mov DX, Offset Name\_List

Int $21

;Close the File

Mov AH, $3E

Int $21

;Display the Content of the Buffer

Mov AH, $09

Mov DX, Offset Name\_List

Int $21

Mov AX, $4C00

Int $21

Code EndS

Data Segment

File\_Name db 'NameList.txt', $00

Name\_List db 100 dup ('$')

Data EndS

End

* 1. **Example Code**

The following program sequence accepts up to 100 characters from the user via keyboard without displaying the characters. A user may press the enter key to terminate entry before 100 characters. Then the program:

* *Will display the characters typed on a new line*
* *Will display an encrypted form of each character which is a result of XORing each character’s ASCII by $OF.*
* *Will display the decrypted form of each characters, which again is a result of XORing back each character ASCII with the same value (Note: X⊕Y⊕Y=X⊕0=X)*

*Hex $*

*Code Segment*

*;initialize the data Segment*

*Mov BX,Data*

*Mov DS,BX*

*;Prompt the user to type characters*

*Mov DX,offset prompt*

*Mov AH,$09*

*Int $21*

*;store characters data in a predefined memory buffer*

*Mov CX,size Chars*

*Mov SI,0*

*C: Mov AH,$08 ; keyboard input (Echo off)*

*Int $21*

*CMP AL,$OD ; Is Enter Key Pressed?*

*JE L*

*Mov Chars[SI], AL*

*Inc SI*

*Loop C*

*;Display all of the typed characters*

*L: Mov CX,size Chars*

*Mov SI,0*

*Call newline*

*U: Mov DL,Chars[SI]*

*CMP DL,$00*

*JE M*

*Mov AH,$02*

*Int $21*

*Inc SI*

*Loop U*

*;Encrypt and Display the Result, and also store encrypted data for future ;processing*

*M: Mov CX,size Chars*

*Mov SI,0*

*Call newline*

*Mov DX,offset prompt1*

*Mov AH,$09*

*Int $21*

*K: CMP Chars[SI],$00*

*JE S*

*Mov AL,Chars[SI]*

*XOR AL,$0F*

*Mov Chars[SI],AL*

*Mov DL,AL*

*Mov Ah,$02*

*Int $21*

*Inc SI*

*Loop K*

*;Decrypt and Display the Result*

*S: Mov CX,size Chars*

*Mov SI,0*

*Call newline*

*Mov DX,offset prompt2*

*Mov AH,$09*

*Int $21*

*K1: CMP Chars[SI],$00*

*JE Exit*

*Mov AL,Chars[SI]*

*XOR AL,$0F*

*Mov DL,AL*

*Mov Ah,$02*

*Int $21*

*Inc SI*

*Loop K1*

*Exit: Mov Ah,$4C*

*Int $21*

*;declares procedure called newline*

*Newline Proc*

*Mov DL,10*

*Mov AH,2*

*Int $21*

*Mov DL,13*

*Mov AH,2*

*Int $21*

*Newline endp*

*Code ends*

*;Data Segment*

*Data Segment*

*Prompt db “Enter Characters (Enter Key to Terminate): $”*

*Prompt1 db “ Encrypted Form: $”*

*Prompt2 db “Decrypted Form: $”*

*Chars db 100 dup ($00)*

*Data ends*

***Exercise:***

*Write a program sequence to display an array of characters declared (as a variable) in the program in reverse order. For example if the characters “Study Hard” defined in the program, then the output shall be “draH ydutS”.*

**Note: motivated students may read Chapter Six of the Text Book for More Detail on the Topics Covered In This Chapter.**

* 1. **Video Output (reading assignment)**

Assembly program can direct video output at any of three separate levels:

* Using DOS function calls
* Using BIOS interrupt (INT $10)
* By directly accessing the display-generating hardware.

At the highest level, DOS functions $02 for character display and $09 for string display are very easy to use, and they are completely machine independent, but they limit output to sequences of ASCII characters. At the lowest level, direct video access requires a great deal of programming effort and must be tailored to match the host computer, but it can drive display device to its physical limits.

The intermediate level of BIOS interrupts provides almost as much power as direct access with almost as much convenience and as much portability as the DOS functions.

*Video Display Hardware*

Originally IBM PC was designed to support either of two types of display hardware – a Color Graphics Adapter (CGA) or a Monochrome Display Adapter (MDA).

*Video Modes*

CGA supports seven video modes:

Mode Number Description

0, 1 40 – column text

2, 3 80 – column text

4, 5 40 – column text / medium-resolution graphics

6 80 – column text / high-resolution graphics

Mode 1, 3, 5 generates color burst signal which is required by composite monitors (like home television sets used as computer display devises)

*Video RAM*

At the heart of each display adapter is a section of random-access memory called Video RAM. Video RAM is functionally positioned midway between the CPU and image-generating circuitry. The CPU can write to and read from video RAM just as it can from any other part of memory. The unique aspect of Video RAM is that can also be read by the video image generator.

The CGA contains 16K of Video RAM mapped to absolute address $B8000 through $BBFFF.

Image generator treats the video display as if it were a mosaic of picture elements called pixels arrayed in a rectangular matrix.

The dimensions of the pixel matrix vary from adapter to adapter and within adapters from one mode of operation to another.

The CGA can support matrices of either:

200 rows by 640 columns (modes 0, 1, 4 and 5) or

200 rows by 320 columns (modes 2, 3 and 6)

In purely text modes (0, 1, 2, and 3) pixels are logically organized in to a serious of character cells. Each word of a video RAM is mapped to one of the character cell (8X8) on screen.

First byte - character code

Second byte - Attribute

The image generating hardware translates the contents of each word of video RAM in to the pixel image of a character for display.

In 40 - column text modes there are 25 rows of 40 characters on screen for a total of 1000 characters that take 2000 bytes of video RAM. In the 80 - column text modes 25 rows of 80 characters are displayed on the screen and require 4000 bytes of video RAM.

In medium-resolution graphics (modes 4 and 5) the CGA recognizes the screen in to a matrix of 200 rows by 320 columns (a total of 64,000 pixels). Each pixel is represented by 2 bits of video RAM. The two bits of memory assigned to each pixel can take a total of four states, so each pixel can be displayed in any one of four colors.

At 4 pixels to a byte, the 64, 000 pixels can occupy almost the entire 16K of video RAM.

In high-resolution graphics (mode 6), the screen is organized into a matrix of 200 rows of 640 columns of pixels for a total of 128,000 pixels. Each pixel is represented by a single bit of video RAM. Each pixel in a high-resolution display can either be on or off.

At 8 pixels to a byte, the 128,000 pixels in a high-resolution display also occupy almost the entire 16K of the video RAM.

*Video management through BIOS interrupts. (*INT $10)

The DOS function calls for video management are limited in some important respects. They can be used to display only a fixed character set. And also have the most inconvenient facility for positioning text on screen.

The BIOS interrupts overcome the limitations imposed by the DOS function calls. The principal BIOS interrupt for video management is INTerrupt $10. This interrupt has a number of functions that are generally applicable to video modes 0 through 6.

The various functions of interrupt $10 can be used to generate alphanumeric displays consisting of character-oriented image or graphic display consisting of pixel-oriented-image.

*Function $00 - Set video mode*

AH = $00

AL = Video Mode

- 0, 1 40 column text

- 2, 3 80 column text

- 4, 5 40 column text/medium resolution graphics

- 6 80 column text/high resolution graphics

*Function $01 - Set cursor size*

AH = $01

CH = Top line of cursor

CL = Bottom line of cursor

CX = $2000 cursor off.

*Function $02 - Set cursor position*

AH = $02

DH = Row

DL = Column

*Function $03 - Read cursor position*

AH = $03

Return - DH = Row

DL = Column

CX = Cursor size

*Function $09 - Display character with attribute*

AH = $09

AL = Character to display

BL = Character attribute

CX = Count

This function doesn't adjust the cursor position

Character Attribute

---- Foreground ------

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| BL | *R* | *G* | *B* | IN | R | G | B |

- Background -

BL - Blink IN - Intensity

R – Red G – Green B – Blue

***R G B IN R G B***

0 0 0 Black 1 0 0 0 Black

0 0 1 Blue 1 0 0 1 Light Blue

0 1 0 Green 1 0 1 0 Light Green

0 1 1 Cyan 1 0 1 1 Light Cyan

1 0 0 Red 1 1 0 0 Light Red

1 0 1 Magenta 1 1 0 1 Light Magenta

1 1 0 Brown 1 1 1 0 Yellow

1 1 1 White 1 1 1 1 Bright White

*Function $0A - The same as $09 except existing character attribute maintained.*

*Function $0B - Select color*

Service $00 select border

AH = $0B

BH = $00

BL = Border color (mode 0 - 3)

BL = Background color (mode 4 - 5)

BL = Foreground color (mode 6)

Service $01 select graphics palette

AH = $0B

BH = $01

BL = Palette

Palette $00 Green, Red, Yellow

Palette $01 Cyan, Magenta, White

*Function $0C - Set pixel*

Service $00 select border

AH = $0C

AL = color

CX = Column

DX = Row

In video mode 4 and 5 AL register may contain a value in the range $00 through $03; which will specify a color depending on the currently active palette.

AL Palette $00 Palette $01

00 Background Background

01 Green Cyan

02 Red Magenta

03 Yellow White

In video mode 6 AL register may contain $00 (off) or $01 (on).

*Function $0f - Read video status*

AH = $0F

Return - AH = Screen width in text columns

AL = Video mode