Assembler Tutorial

1996 Edition
University of Guadalajara
Information Systems General Coordination.
Culture and Entertainment Web

June 12th 1995

This is an introduction for people who want to programming in assembler language.

Table of Contents

1. Introduction
2. Basic Concepts
3. Assembler programming
4. Assembler language instructions
5. Interruptions and file managing
6. Macros and procedures
7. Program examples

1. Introduction

Table of contents

1.1 What's new in the Assembler material
1.2 Presentation
1.3 Why learn Assembler language
1.4 We need your opinion

1.1 What's new in the Assembler material

After of one year that we've released the first Assembler material on-line. We've received a lot of e-mail where each people talk about different aspects about this material. We've tried to put these comments and suggestions in this update assembler material. We hope that this new Assembler material release reach to all people that they interest to learn the most important language for IBM PC.

In this new assembler release includes:

A complete chapter about how to use debug program
More example of the assembler material
Each section of this assembler material includes a link file to Free
On-line of Computing by Dennis Howe
Finally, a search engine to look for any topic or item related with this updated material.

1.2 Presentation

The document you are looking at, has the primordial function of introducing you to assembly language programming, and it has been thought for those people who have never worked with this language.

The tutorial is completely focused towards the computers that function with processors of the x86 family of Intel, and considering that the language bases its functioning on the internal resources of the processor, the described examples are not compatible with any other architecture.

The information was structured in units in order to allow easy access to each of the topics and facilitate the following of the tutorial.

In the introductory section some of the elemental concepts regarding computer systems are mentioned, along with the concepts of the assembly language itself, and continues with the tutorial itself.

1.3 Why learn assembler language

The first reason to work with assembler is that it provides the opportunity of knowing more the operation of your PC, which allows the development of software in a more consistent manner.

The second reason is the total control of the PC which you can have with the use of the assembler.

Another reason is that the assembly programs are quicker, smaller, and have larger capacities than ones created with other languages.

Lastly, the assembler allows an ideal optimization in programs, be it on their size or on their execution.

1.4 We need your opinion

Our goal is offers you easier way to learn yourself assembler language. You send us your comments or suggestions about this 96' edition. Any comment will be welcome.

2. Basic Concepts

Contents

2.1 Basic description of a computer system.
2.2 Assembler language Basic concepts
2.3 Using debug program

2.1 Basic description of a computer system.

This section has the purpose of giving a brief outline of the main components of a computer system at a basic level, which will allow the user a greater understanding of the concepts which will be dealt with throughout the tutorial.

Contents

2.1.1 Central Processor
2.1.2 Central Memory
2.1.3 Input and Output Units
2.1.4 Auxiliary Memory Units

Computer System.

We call computer system to the complete configuration of a computer, including the peripheral units and the system programming which make it a useful and functional machine for a determined task.

2.1.1 Central Processor.

This part is also known as central processing unit or CPU, which in turn is made by the control unit and the arithmetic and logic unit. Its functions consist in reading and writing the contents of the memory cells, to forward data between memory cells and special registers, and decode and execute the instructions of a program. The processor has a series of memory cells which are used very often and thus, are part of the CPU. These cells are known with the name of registers. A processor may have one or two dozen of these registers. The arithmetic and logic unit of the CPU realizes the operations related with numeric and symbolic calculations. Typically these units only have capacity of performing very elemental operations such as: the addition and subtraction of two whole numbers, whole number multiplication and division, handling of the registers' bits and the comparison of the content of two registers. Personal computers can be classified by what is known as word size, this is, the quantity of bits which the processor can handle at a time.

2.1.2 Central Memory.

It is a group of cells, now being fabricated with semi-conductors, used for general processes, such as the execution of programs and the storage of information for the operations.

Each one of these cells may contain a numeric value and they have the property of being addressable, this is, that they can distinguish one from another by means of a unique number or an address for each cell.

The generic name of these memories is Random Access Memory or RAM. The main disadvantage of this type of memory is that the integrated circuits lose the information they have stored when the electricity flow is interrupted. This was the reason for the creation of memories whose information is not lost when the system is turned off. These memories receive the name of Read Only Memory or ROM.

2.1.3 Input and Output Units.

In order for a computer to be useful to us it is necessary that the processor communicates with the exterior through interfaces which allow the input and output of information from the processor and the memory. Through the use of these communications it is possible to introduce information to be processed and to later visualize the processed data.

Some of the most common input units are keyboards and mice. The most common output units are screens and printers.

2.1.4 Auxiliary Memory Units.

Since the central memory of a computer is costly, and considering today's applications it is also very limited. Thus, the need to create practical and economical information storage systems arises. Besides, the central memory loses its content when the machine is turned off, therefore making it inconvenient for the permanent storage of data.

These and other inconvenience give place for the creation of peripheral units of memory which receive the name of auxiliary or secondary memory. Of these the most common are the tapes and magnetic discs.

The stored information on these magnetic media means receive the name of files. A file is made of a variable number of registers, generally of a fixed size; the registers may contain information or programs.

2.2 Assembler language Basic concepts

Contents

2.2.1 Information in the computers
2.2.2 Data representation methods

2.2.1 Information in the computer

Contents

2.2.1.1 Information units
2.2.1.2 Numeric systems
2.2.1.3 Converting binary numbers to decimal
2.2.1.4 Converting decimal numbers to binary
2.2.1.5 Hexadecimal system

2.2.1.1 Information Units

In order for the PC to process information, it is necessary that this information be in special cells called registers. The registers are groups of 8 or 16 flip-flops.

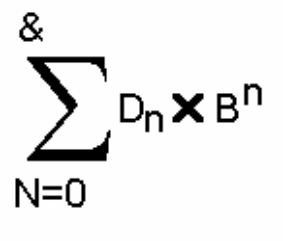
A flip-flop is a device capable of storing two levels of voltage, a low one, regularly 0.5 volts, and another one, commonly of 5 volts. The low level of energy in the flip-flop is interpreted as off or 0, and the high level as on or 1. These states are usually known as bits, which are the smallest information unit in a computer.

A group of 16 bits is known as word; a word can be divided in groups of 8 bits called bytes, and the groups of 4 bits are called nibbles.

2.2.1.2 Numeric systems

The numeric system we use daily is the decimal system, but this system is not convenient for machines since the information is handled codified in the shape of on or off bits; this way of codifying takes us to the necessity of knowing the positional calculation which will allow us to express a number in any base where we need it.

It is possible to represent a determined number in any base through the following formula:



Where n is the position of the digit beginning from right to left and numbering from zero. D is the digit on which we operate and B is the used numeric base.

2.2.1.3 converting binary numbers to decimals

When working with assembly language we come on the necessity of converting numbers from the binary system, which is used by computers, to the decimal system used by people.

The binary system is based on only two conditions or states, be it on(1) or off(0), thus its base is two.

For the conversion we can use the positional value formula:

For example, if we have the binary number of 10011, we take each digit from right to left and multiply it by the base, elevated to the new position they are:

```
Binary:          1          1        0        0        1

Decimal:       1*2^0  + 1*2^1 + 0*2^2 + 0*2^3 + 1*2^4

            =   1     +   2   +   0   +   0   +  16  = 19 decimal.
```

The ^ character is used in computation as an exponent symbol and the * character is used to represent multiplication.

2.2.1.4 Converting decimal numbers to binary

There are several methods to convert decimal numbers to binary; only one will be analyzed here. Naturally a conversion with a scientific calculator is much easier, but one cannot always count with one, so it is convenient to at least know one formula to do it.

The method that will be explained uses the successive division of two, keeping the residue as a binary digit and the result as the next number to divide.

Let us take for example the decimal number of 43.

```
43/2=21 and its residue is 1

21/2=10 and its residue is 1

10/2=5 and its residue is 0

5/2=2 and its residue is 1

2/2=1 and its residue is 0

1/2=0 and its residue is 1
```

Building the number from the bottom , we get that the binary result is
101011

2.2.1.5 Hexadecimal system

On the hexadecimal base we have 16 digits which go from 0 to 9 and from the letter A to the F, these letters represent the numbers from 10 to 15. Thus we count 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, and F.

The conversion between binary and hexadecimal numbers is easy. The first thing done to do a conversion of a binary number to a hexadecimal is to divide it in groups of 4 bits, beginning from the right to the left. In case the last group, the one most to the left, is under 4 bits, the missing places are filled with zeros.

Taking as an example the binary number of 101011, we divide it in 4 bits groups and we are left with:

10;1011

Filling the last group with zeros (the one from the left):

0010;1011

Afterwards we take each group as an independent number and we consider its decimal value:

0010=2;1011=11

But since we cannot represent this hexadecimal number as 211 because it would be an error, we have to substitute all the values greater than 9 by their respective representation in hexadecimal, with which we obtain:

2BH, where the H represents the hexadecimal base.

In order to convert a hexadecimal number to binary it is only necessary to invert the steps: the first hexadecimal digit is taken and converted to binary, and then the second, and so on.

2.2.2 Data representation methods in a computer.

Contents

2.2.2.1.ASCII code
2.2.2.2 BCD method
2.2.2.3 Floating point representation

2.2.2.1 ASCII code

ASCII is an acronym of American Standard Code for Information Interchange. This code assigns the letters of the alphabet, decimal digits from 0 to 9 and some additional symbols a binary number of 7 bits, putting the 8th bit in its off state or 0. This way each letter, digit or special character occupies one byte in the computer memory.

We can observe that this method of data representation is very inefficient on the numeric aspect, since in binary format one byte is not enough to represent numbers from 0 to 255, but on the other hand with the ASCII code one byte may represent only one digit. Due to this inefficiency, the ASCII code is mainly used in the memory to represent text.

2.2.2.2 BCD Method

BCD is an acronym of Binary Coded Decimal. In this notation groups of 4 bits are used to represent each decimal digit from 0 to 9. With this method we can represent two digits per byte of information.

Even when this method is much more practical for number representation in the memory compared to the ASCII code, it still less practical than the binary since with the BCD method we can only represent digits from 0 to 99. On the other hand in binary format we can represent all digits from 0 to 255.

This format is mainly used to represent very large numbers in mercantile applications since it facilitates operations avoiding mistakes.

2.2.2.3 Floating point representation

This representation is based on scientific notation, this is, to represent a number in two parts: its base and its exponent.

As an example, the number 1234000, can be represented as 1.123*10^6, in this last notation the exponent indicates to us the number of spaces that the decimal point must be moved to the right to obtain the original result.

In case the exponent was negative, it would be indicating to us the number of spaces that the decimal point must be moved to the left to obtain the original result.

2.3 Using Debug program

Contents

2.3.1 Program creation process
2.3.2 CPU registers
2.3.3 Debug program
2.3.4 Assembler structure
2.3.5 Creating basic assembler program
2.3.6 Storing and loading the programs
2.3.7 More debug program examples

2.31 Program creation process

For the creation of a program it is necessary to follow five steps:

Design of the algorithm, stage the problem to be solved is established and the best solution is proposed, creating squematic diagrams used for the better solution proposal.

Coding the algorithm, consists in writing the program in some programming language; assembly language in this specific case, taking as a base the proposed solution on the prior step.

Translation to machine language, is the creation of the object program, in other words, the written program as a sequence of zeros and ones that can be interpreted by the processor.

Test the program, after the translation the program into machine language, execute the program in the computer machine.

The last stage is the elimination of detected faults on the program on the test stage. The correction of a fault normally requires the repetition of all the steps from the first or second.

2.3.2 CPU Registers

The CPU has 4 internal registers, each one of 16 bits. The first four, AX, BX, CX, and DX are general use registers and can also be used as 8 bit registers, if used in such a way it is necessary to refer to them for example as: AH and AL, which are the high and low bytes of the AX register. This nomenclature is also applicable to the BX, CX, and DX registers.

The registers known by their specific names:

AX **A**ccumulator
BX **B**ase register
CX **C**ounting register
DX **D**ata register
DS **D**ata **S**egment register
ES **E**xtra **S**egment register
SS Battery segment register
CS **C**ode **S**egment register
BP **B**ase **P**ointers register
SI **S**ource **I**ndex register
DI **D**estiny **I**ndex register
SP Battery pointer register
IP Next **I**nstruction **P**ointer register
F **F**lag register

2.3.3 Debug program

To create a program in assembler two options exist, the first one is to use the TASM or Turbo Assembler, of Borland, and the second one is to use the debugger - on this first section we will use this last one since it is found in any PC with the MS-DOS, which makes it available to any user who has access to a machine with these characteristics.

Debug can only create files with a .COM extension, and because of the characteristics of these kinds of programs they cannot be larger that 64 kb, and they also must start with displacement, offset, or 0100H memory direction inside the specific segment.

Debug provides a set of commands that lets you perform a number of useful operations:

A Assemble symbolic instructions into machine code
D Display the contents of an area of memory
E Enter data into memory, beginning at a specific location
G Run the executable program in memory
N Name a program
P Proceed, or execute a set of related instructions
Q Quit the debug program
R Display the contents of one or more registers
T Trace the contents of one instruction
U Unassembled machine code into symbolic code
W Write a program onto disk

It is possible to visualize the values of the internal registers of the CPU using the Debug program. 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:

-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
```

All the contents of the internal registers of the CPU are displayed; an alternative of viewing them is to use 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 ":"

When the prompt is like this, it is possible to change the value of the register which was seen by typing the new value and [Enter], or the old value can be left by pressing [Enter] without typing any other value.

2.3.4 Assembler structure

In assembly language code lines have two parts, the first one is the name of the instruction which is to be executed, and the second one are the parameters of the command. For example:

add ah bh

Here "add" is the command to be executed, in this case an addition, and "ah" as well as "bh" are the parameters.

For example:

mov al, 25

In the above example, we are using the instruction mov, it means move the value 25 to al register.

The name of the instructions in this language is made of two, three or four letters. These instructions are also called mnemonic names or operation codes, since they represent a function the processor will perform.

Sometimes instructions are used as follows:

add al,[170]

The brackets in the second parameter indicate to us that we are going to work with the content of the memory cell number 170 and not with the 170 value, this is known as direct addressing.

2.3.5 Creating basic assembler program

The first step is to initiate the Debug, this step only consists of typing 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. And it will be the place we will use since only Debug can create this specific type of programs.

Even though at this moment it is not necessary to give the "a" command a parameter, it is recommendable to do so to avoid problems once the CS:IP registers are used, therefore we type:

```
a 100[enter]
mov ax,0002[enter]
mov bx,0004[enter]
add ax,bx[enter]
nop[enter][enter]
```

What does the program do?, move the value 0002 to the ax register, move the value 0004 to the bx register, add the contents of the ax and bx registers, the instruction, no operation, to finish the program.

In the debug program. After this is done, the screen will produce the following lines:

```
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.

2.3.6 Storing and loading the programs

It would not seem practical to type an entire program each time it is needed, and to avoid this it is possible to store a program on the disk, with the enormous advantage that by being already assembled it will not be necessary to run Debug again to execute it.

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 save an already loaded 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.

3 Assembler programming

Contents

3.1 Building Assembler programs
3.2 Assembly process
3.3 More assembler programs
3.4 Types of instructions

3.1 Building Assembler programs

Contents

3.1.1 Needed software
3.1.2 Assembler Programming

3.1.1 Needed software

In order to be able to create a program, several tools are needed:

First an editor to create the source program. Second a compiler, which is nothing more than a program that "translates" the source program into an object program. And third, a linker that generates the executable program from the object program.

The editor can be any text editor at hand, and as a compiler we will use the TASM macro assembler from Borland, and as a linker we will use the Tlink program.

The extension used so that TASM recognizes the source programs in assembler is .ASM; once translated the source program, the TASM creates a file with the .OBJ extension, this file contains an "intermediate format" of the program, called like this because it is not executable yet but it is not a program in source language either anymore. The linker generates, from a .OBJ or a combination of several of these files, an executable program, whose extension usually is .EXE though it can also be .COM, depending of the form it was assembled.

3.1.2 Assembler Programming

To build assembler programs using TASM programs is a different program structure than from using debug program.

It's important to include the following assembler directives:

.MODEL SMALL
Assembler directive that defines the memory model to use in the program

.CODE
Assembler directive that defines the program instructions

.STACK
Assembler directive that reserves a memory space for program instructions in the stack

END
Assembler directive that finishes the assembler program

Let's program

First step

use any editor program to create the source file. Type the following lines:

First example

```
; use ; to put comments in the assembler program
.MODEL SMALL; memory model
.STACK; memory space for program instructions in the stack
.CODE; the following lines are program instructions
mov ah,1h; moves the value 1h to register ah
mov cx,07h; moves the value 07h to register cx
int 10h;10h interruption
mov ah,4ch; moves the value 4 ch to register ah
int 21h; 21h interruption
END; finishes the program code
```

This assembler program changes the size of the computer cursor.

Second step

Save the file with the following name: examp1.asm
Don't forget to save this in ASCII format.

Third step

Use the TASM program to build the object program.

Example:

```
C:\>tasm exam1.asm
Turbo Assembler  Version 2.0  Copyright (c) 1988, 1990 Borland
International

Assembling file:   exam1.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  471k
```

The TASM can only create programs in .OBJ format, which are not executable by themselves, but rather it is necessary to have a linker which generates the executable code.

Fourth step

Use the TLINK program to build the executable program example:

```
C:\>tlink exam1.obj
Turbo Link   Version 3.0 Copyright (c) 1987, 1990 Borland
International

C:\>
```

Where exam1.obj is the name of the intermediate program, .OBJ. This generates a file directly with the name of the intermediate program and the .EXE extension.

Fifth step

Execute the executable program

`C:\>exam1[enter]`

Remember, this assembler program changes the size of the cursor.

Assembly process.

Segments
Table of symbols

SEGMENTS

The architecture of the x86 processors forces to the use of memory segments to manage the information, the size of these segments is of 64kb.

The reason of being of these segments is that, considering that the maximum size of a number that the processor can manage is given by a word of 16 bits or register, it would not be possible to access more than 65536 localities of memory using only one of these registers, but now, if the PC's memory is divided into groups or segments, each one of 65536 localities, and we use an address on an exclusive register to find each segment, and then we make each address of a specific slot with two registers, it is possible for us to access a quantity of 4294967296 bytes of memory, which is, in the present day, more memory than what we will see installed in a PC.

In order for the assembler to be able to manage the data, it is necessary that each piece of information or instruction be found in the area that corresponds to its respective segments. The assembler accesses this information taking into account the localization of the segment, given by the DS, ES, SS and CS registers and inside the register the address of the specified piece of information. It is because of this that when we create a program using the Debug on each line that we assemble, something like this appears:

1CB0:0102 MOV AX,BX

Where the first number, 1CB0, corresponds to the memory segment being used, the second one refers to the address inside this segment, and the instructions which will be stored from that address follow.

The way to indicate to the assembler with which of the segments we will work with is with the .CODE, .DATA and .STACK directives.

The assembler adjusts the size of the segments taking as a base the number of bytes each assembled instruction needs, since it would be a waste of memory to use the whole segments. For example, if a program only needs 10kb to store data, the data segment will only be of 10kb and not the 64kb it can handle.

SYMBOLS CHART

Each one of the parts on code line in assembler is known as token, for example on the code line:

MOV AX,Var

we have three tokens, the MOV instruction, the AX operator, and the VAR operator. What the assembler does to generate the OBJ code is to read each one of the tokens and look for it on an internal "equivalence" chart known as the reserved words chart, which is where all the mnemonic meanings we use as instructions are found.

Following this process, the assembler reads MOV, looks for it on its chart and identifies it as a processor instruction. Likewise it reads AX and recognizes it as a register of the processor, but when it looks for the Var token on the reserved words chart, it does not find it, so then it looks for it on the symbols chart which is a table where the names of the variables, constants and labels used in the program where their addresses on memory are included and the sort of data it contains, are found.

Sometimes the assembler comes on a token which is not defined on the program, therefore what it does in these cased is to pass a second time by the source program to verify all references to that symbol and place it on the symbols chart.

There are symbols which the assembler will not find since they do not belong to that segment and the program does not know in what part of the memory it will find that segment, and at this time the linker comes into action, which will create the structure necessary for the loader so that the segment and the token be defined when the program is loaded and before it is executed.

3.3 More assembler programs

Another example

First step

Use any editor program to create the source file. Type the following lines:

```
;example11
.model small
.stack
.code
mov ah,2h  ;moves the value 2h to register ah
mov dl,2ah ;moves de value 2ah to register dl
   ;(Its the asterisk value in ASCII format)
int 21h    ;21h interruption
mov ah,4ch ;4ch function, goes to operating system
int 21h    ;21h interruption
end        ;finishes the program code
```

Second step

Save the file with the following name: exam2.asm
Don't forget to save this in ASCII format.

Third step

Use the TASM program to build the object program.

```
C:\>tasm exam2.asm
Turbo Assembler  Version 2.0  Copyright (c) 1988, 1990 Borland
International

Assembling file:   exam2.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  471k
```

Fourth step

Use the TLINK program to build the executable program

```
C:\>tlink exam2.obj
Turbo Link   Version 3.0 Copyright (c) 1987, 1990 Borland
International

C:\>
```

Fifth step

Execute the executable program

```
C:\>ejem11[enter]
*
C:\>
```

This assembler program shows the asterisk character on the computer screen

3.4 Types of instructions.

Contents

3.4.1 Data movement
3.4.2 Logic and arithmetic operations
3.4.3 Jumps, loops and procedures

3.4.1 Data movement

In any program it is necessary to move the data in the memory and in the CPU registers; there are several ways to do this: it can copy data in the memory to some register, from register to register, from a register to a stack, from a stack to a register, to transmit data to external devices as well as vice versa.

This movement of data is subject to rules and restrictions. The following are some of them:

*It is not possible to move data from a memory locality to another directly; it is necessary to first move the data of the origin locality to a register and then from the register to the destiny locality.

*It is not possible to move a constant directly to a segment register; it first must be moved to a register in the CPU.

It is possible to move data blocks by means of the movs instructions, which copies a chain of bytes or words; movsb which copies n bytes from a locality to another; and movsw copies n words from a locality to another. The last two instructions take the values from the defined addresses by DS:SI as a group of data to move and ES:DI as the new localization of the data.

To move data there are also structures called batteries, where the data is introduced with the push instruction and are extracted with the pop instruction. In a stack the first data to be introduced is the last one we can take, this is, if in our program we use these instructions:

```
PUSH AX
PUSH BX
PUSH CX
```

To return the correct values to each register at the moment of taking them from the stack it is necessary to do it in the following order:

```
POP CX
POP BX
POP AX
```

For the communication with external devices the out command is used to send information to a port and the in command to read the information received from a port.

The syntax of the out command is:

```
OUT DX,AX
```

Where DX contains the value of the port which will be used for the communication and AX contains the information which will be sent.

The syntax of the in command is:

IN AX,DX

Where AX is the register where the incoming information will be kept and DX contains the address of the port by which the information will arrive.

3.4.2 Logic and arithmetic operations

The instructions of the logic operations are: and, not, or and xor. These work on the bits of their operators. To verify the result of the operations we turn to the cmp and test instructions. The instructions used for the algebraic operations are: to add, to subtract sub, to multiply mul and to divide div.

Almost all the comparison instructions are based on the information contained in the flag register. Normally the flags of this register which can be directly handled by the programmer are the data direction flag DF, used to define the operations about chains.

Another one which can also be handled is the IF flag by means of the sti and cli instructions, to activate and deactivate the interruptions.

3.4.3 Jumps, loops and procedures

The unconditional jumps in a written program in assembler language are given by the jmp instruction; a jump is to moves the flow of the execution of a program by sending the control to the indicated address.

A loop, known also as iteration, is the repetition of a process a certain number of times until a condition is fulfilled. These loops are used (broken sentence).

4 Assembler language Instructions

Contents

4.1 Transfer instructions
4.2 Loading instructions
4.3 Stack instructions
4.4 Logic instructions
4.5 Arithmetic instructions
4.6 Jump instructions
4.7 Instructions for cycles: loop
4.8 Counting Instructions
4.9 Comparison Instructions
4.10 Flag Instructions

4.1 Transfer instructions

They are used to move the contents of the operators. Each instruction can be used with different modes of addressing.

MOV
MOVS (MOVSB) (MOVSW)

MOV INSTRUCTION

Purpose: Data transfer between memory cells, registers and the accumulator.

Syntax:

MOV Destiny, Source

Where **Destiny** is the place where the data will be moved and **Source** is the place where the data is.

The different movements of data allowed for this instruction are:

*Destiny: memory. Source: accumulator
*Destiny: accumulator. Source: memory
*Destiny: segment register. Source: memory/register
*Destiny: memory/register. Source: segment register
*Destiny: register. Source: register
*Destiny: register. Source: memory
*Destiny: memory. Source: register
*Destiny: register. Source: immediate data
*Destiny: memory. Source: immediate data

Example:

```
MOV AX,0006h
MOV BX,AX
MOV AX,4C00h
INT 21H
```

This small program moves the value of 0006H to the AX register, then it moves the content of AX (0006h) to the BX register, and lastly it moves the 4C00h value to the AX register to end the execution with the 4C option of the 21h interruption.

MOVS (MOVSB) (MOVSW) Instruction

Purpose: To move byte or word chains from the source, addressed by SI, to the destiny addressed by DI.

Syntax:

MOVS

This command does not need parameters since it takes as source address the content of the SI register and as destination the content of DI. The following sequence of instructions illustrates this:

```
MOV SI, OFFSET VAR1
MOV DI, OFFSET VAR2
MOVS
```

First we initialize the values of SI and DI with the addresses of the VAR1 and VAR2 variables respectively, then after executing MOVS the content of VAR1 is copied onto VAR2.

The MOVSB and MOVSW are used in the same way as MOVS, the first one moves one byte and the second one moves a word.

4.2 Loading instructions

They are specific register instructions. They are used to load bytes or chains of bytes onto a register.

LODS (LODSB) (LODSW)
LAHF
LDS
LEA
LES

LODS (LODSB) (LODSW) INSTRUCTION

Purpose: To load chains of a byte or a word into the accumulator.

Syntax:

LODS

This instruction takes the chain found on the address specified by SI, loads it to the AL (or AX) register and adds or subtracts , depending on the state of DF, to SI if it is a bytes transfer or if it is a words transfer.

```
MOV SI, OFFSET VAR1
LODS
```

The first line loads the VAR1 address on SI and the second line takes the content of that locality to the AL register.

The **LODSB** and **LODSW** commands are used in the same way, the first one loads a byte and the second one a word (it uses the complete AX register).

LAHF INSTRUCTION

Purpose: It transfers the content of the flags to the AH register.

Syntax:

LAHF

This instruction is useful to verify the state of the flags during the execution of our program.

The flags are left in the following order inside the register:

SF ZF ?? AF ?? PF ?? CF

The "**??**" means that there will be an undefined value in those bits.

LDS INSTRUCTION

Purpose: To load the register of the data segment

Syntax:

LDS destiny, source

The source operator must be a double word in memory. The word associated with the largest address is transferred to DS, in other words it is taken as the segment address. The word associated with the smaller address is the displacement address and it is deposited in the register indicated as destiny.

LEA INSTRUCTION

Purpose: To load the address of the source operator

Syntax:

LEA destiny, source

The source operator must be located in memory, and its displacement is placed on the index register or specified pointer in destiny.

To illustrate one of the facilities we have with this command let us write an equivalence:

MOV SI,OFFSET VAR1

Is equivalent to:

LEA SI,VAR1

It is very probable that for the programmer it is much easier to create extensive programs by using this last format.

LES INSTRUCTION

Purpose: To load the register of the extra segment

Syntax:

LES destiny, source

The source operator must be a double word operator in memory. The content of the word with the larger address is interpreted as the segment address and it is placed in ES. The word with the smaller address is the displacement address and it is placed in the specified register on the destiny parameter.

4.3 Stack instructions

These instructions allow the use of the stack to store or retrieve data.

POP
POPF
PUSH
PUSHF

POP INSTRUCTION

Purpose: It recovers a piece of information from the stack

Syntax:

POP destiny

This instruction transfers the last value stored on the stack to the destiny operator, it then increases by 2 the SP register.

This increase is due to the fact that the stack grows from the highest memory segment address to the lowest, and the stack only works with words, 2 bytes, so then by increasing by two the SP register, in reality two are being subtracted from the real size of the stack.

POPF INSTRUCTION

Purpose: It extracts the flags stored on the stack

Syntax:

POPF

This command transfers bits of the word stored on the higher part of the stack to the flag register.

The way of transference is as follows:

BIT	FLAG
0	CF
2	PF
4	AF
6	ZF
7	SF
8	TF
9	IF
10	DF
11	OF

These localities are the same for the PUSHF command.

Once the transference is done, the SP register is increased by 2, diminishing the size of the stack.

PUSH INSTRUCTION

Purpose: It places a word on the stack.

Syntax:

PUSH source

The PUSH instruction decreases by two the value of SP and then transfers the content of the source operator to the new resulting address on the recently modified register.

The decrease on the address is due to the fact that when adding values to the stack, this one grows from the greater to the smaller segment address, therefore by subtracting 2 from the SP register what we do is to increase the size of the stack by two bytes, which is the only quantity of information the stack can handle on each input and output of information.

PUSHF INSTRUCTION

Purpose: It places the value of the flags on the stack.

Syntax:

PUSHF

This command decreases by 2 the value of the SP register and then the content of the flag register is transferred to the stack, on the address indicated by SP.

The flags are left stored in memory on the same bits indicated on the POPF command.

4.4 Logic instructions

They are used to perform logic operations on the operators.

AND
NEG
NOT
OR
TEST
XOR

AND INSTRUCTION

Purpose: It performs the conjunction of the operators bit by bit.

Syntax:

AND destiny, source

With this instruction the "y" logic operation for both operators is carried out:

Source	Destiny	Destiny
1	1	1
1	0	0
0	1	0
0	0	0

The result of this operation is stored on the destiny operator.

NEG INSTRUCTION

Purpose: It generates the complement to 2.

Syntax:

NEG destiny

This instruction generates the complement to 2 of the destiny operator and stores it on the same operator.

For example, if AX stores the value of 1234H, then:

NEG AX

This would leave the EDCCH value stored on the AX register.

NOT INSTRUCTION

Purpose: It carries out the negation of the destiny operator bit by bit.

Syntax:

NOT destiny

The result is stored on the same destiny operator.

OR INSTRUCTION

Purpose: Logic inclusive OR

Syntax:

OR destiny, source

The OR instruction carries out, bit by bit, the logic inclusive disjunction of the two operators:

Source	Destiny	Destiny
1	1	1
1	0	1
0	1	1
0	0	0

TEST INSTRUCTION

Purpose: It logically compares the operators

Syntax:

TEST destiny, source

It performs a conjunction, bit by bit, of the operators, but differing from AND, this instruction does not place the result on the destiny operator, it only has effect on the state of the flags.

XOR INSTRUCTION

Purpose: OR exclusive

Syntax:

XOR destiny, source

Its function is to perform the logic exclusive disjunction of the two operators bit by bit.

Source	Destiny	Destiny
1	1	0
0	0	1
0	1	1
0	0	0

4.5 Arithmetic instructions

They are used to perform arithmetic operations on the operators.

ADC
ADD
DIV
IDIV
MUL
IMUL
SBB
SUB

ADC INSTRUCTION

Purpose: Cartage addition

Syntax:

ADC destiny, source

It carries out the addition of two operators and adds one to the result in case the CF flag is activated, this is in case there is carried.

The result is stored on the destiny operator.

ADD INSTRUCTION

Purpose: Addition of the operators.

Syntax:

ADD destiny, source

It adds the two operators and stores the result on the destiny operator.

DIV INSTRUCTION

Purpose: Division without sign.

Syntax:

DIV source

The divider can be a byte or a word and it is the operator which is given the instruction.

If the divider is 8 bits, the 16 bits AX register is taken as dividend and if the divider is 16 bits the even DX:AX register will be taken as dividend, taking the DX high word and AX as the low.

If the divider was a byte then the quotient will be stored on the AL register and the residue on AH, if it was a word then the quotient is stored on AX and the residue on DX.

IDIV INSTRUCTION

Purpose: Division with sign.

Syntax:

IDIV source

It basically consists on the same as the DIV instruction, and the only difference is that this one performs the operation with sign.

For its results it used the same registers as the DIV instruction.

MUL INSTRUCTION

Purpose: Multiplication with sign.

Syntax:

MUL source

The assembler assumes that the multiplicand will be of the same size as the multiplier, therefore it multiplies the value stored on the register given as operator by the one found to be contained in AH if the multiplier is 8 bits or by AX if the multiplier is 16 bits.

When a multiplication is done with 8 bit values, the result is stored on the AX register and when the multiplication is with 16 bit values the result is stored on the even DX:AX register.

IMUL INSTRUCTION

Purpose: Multiplication of two whole numbers with sign.

Syntax:

IMUL source

This command does the same as the one before, only that this one does take into account the signs of the numbers being multiplied.

The results are kept in the same registers that the MOV instruction uses.

SBB INSTRUCTION

Purpose: Subtraction with cartage.

Syntax:

SBB destiny, source

This instruction subtracts the operators and subtracts one to the result if CF is activated. The source operator is always subtracted from the destiny.

This kind of subtraction is used when one is working with 32 bits quantities.

SUB INSTRUCTION

Purpose: Subtraction.

Syntax:

SUB destiny, source

It subtracts the source operator from the destiny.

4.6 Jump instructions

They are used to transfer the flow of the process to the indicated operator.

JMP
JA (JNBE)
JAE (JNBE)

JB (JNAE)
JBE (JNA)
JE (JZ)
JNE (JNZ)
JG (JNLE)
JGE (JNL)
JL (JNGE)
JLE (JNG)
JC
JNC
JNO
JNP (JPO)
JNS
JO
JP (JPE)
JS

JMP INSTRUCTION

Purpose: Unconditional jump.

Syntax:

JMP destiny

This instruction is used to deviate the flow of a program without taking into account the actual conditions of the flags or of the data.

JA (JNBE) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JA Label

After a comparison this command jumps if it is or jumps if it is not down or if not it is the equal.

This means that the jump is only done if the CF flag is deactivated or if the ZF flag is deactivated, that is that one of the two be equal to zero.

JAE (JNB) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JAE label

It jumps if it is or it is the equal or if it is not down.

The jump is done if CF is deactivated.

JB (JNAE) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JB label

It jumps if it is down, if it is not , or if it is the equal.

The jump is done if CF is activated.

JBE (JNA) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JBE label

It jumps if it is down, the equal, or if it is not .

The jump is done if CF is activated or if ZF is activated, that any of them be equal to 1.

JE (JZ) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JE label

It jumps if it is the equal or if it is zero.

The jump is done if ZF is activated.

JNE (JNZ) INSTRUCTION

Purpose: Conditional jump.

Syntax:

JNE label

It jumps if it is not equal or zero.

The jump will be done if ZF is deactivated.

JG (JNLE) INSTRUCTION

Purpose: Conditional jump, and the sign is taken into account.

Syntax:

JG label

It jumps if it is larger, if it is not larger or equal.

The jump occurs if ZF = 0 or if OF = SF.

JGE (JNL) INSTRUCTION

Purpose: Conditional jump, and the sign is taken into account.

Syntax:

JGE label

It jumps if it is larger or less than, or equal to.

The jump is done if SF = OF

JL (JNGE) INSTRUCTION

Purpose: Conditional jump, and the sign is taken into account.

Syntax:

JL label

It jumps if it is less than or if it is not larger than or equal to.

The jump is done if SF is different than OF.

JLE (JNG) INSTRUCTION

Purpose: Conditional jump, and the sign is taken into account.

Syntax:

JLE label

It jumps if it is less than or equal to, or if it is not larger.

The jump is done if ZF = 1 or if SF is defferent than OF.

JC INSTRUCTION

Purpose: Conditional jump, and the flags are taken into account.

Syntax:

JC label

It jumps if there is cartage.

The jump is done if CF = 1

JNC INSTRUCTION

Purpose: Conditional jump, and the state of the flags is taken into account.

Syntax:

JNC label

It jumps if there is no cartage.

The jump is done if CF = 0.

JNO INSTRUCTION

Purpose: Conditional jump, and the state of the flags is taken into account.

Syntax:

JNO label

It jumps if there is no overflow.

The jump is done if OF = 0.

JNP (JPO) INSTRUCTION

Purpose: Conditional jump, and the state of the flags is taken into account.

Syntax:

JNP label

It jumps if there is no parity or if the parity is uneven.

The jump is done if PF = 0.

JNS INSTRUCTION

Purpose: Conditional jump, and the state of the flags is taken into account.

Syntax:

JNP label

It jumps if the sign is deactivated.

The jump is done if SF = 0.

JO INSTRUCTION

Purpose: Conditional jump, and the state of the flags is taken into account.

Syntax:

JO label

It jumps if there is overflow.

The jump is done if OF = 1.

JP (JPE) INSTRUCTION

Purpose: Conditional jump, the state of the flags is taken into account.

Syntax:

JP label

It jumps if there is parity or if the parity is even.

The jump is done if PF = 1.

JS INSTRUCTION

Purpose: Conditional jump, and the state of the flags is taken into account.

Syntax:

JS label

It jumps if the sign is on.

The jump is done if SF = 1.

4.7 Instructions for cycles:loop

They transfer the process flow, conditionally or unconditionally, to a destiny, repeating this action until the counter is zero.

LOOP
LOOPE
LOOPNE

LOOP INSTRUCTION

Purpose: To generate a cycle in the program.

Syntax:

LOOP label

The loop instruction decreases CX on 1, and transfers the flow of the program to the label given as operator if CX is different than 1.

LOOPE INSTRUCTION

Purpose: To generate a cycle in the program considering the state of ZF.

Syntax:

LOOPE label

This instruction decreases CX by 1. If CX is different to zero and ZF is equal to 1, then the flow of the program is transferred to the label indicated as operator.

LOOPNE INSTRUCTION

Purpose: To generate a cycle in the program, considering the state of ZF.

Syntax:

LOOPNE label

This instruction decreases one from CX and transfers the flow of the program only if ZF is different to 0.

4.8 Counting instructions

They are used to decrease or increase the content of the counters.

DEC
INC

DEC INSTRUCTION

Purpose: To decrease the operator.

Syntax:

DEC destiny

This operation subtracts 1 from the destiny operator and stores the new value in the same operator.

INC INSTRUCTION

Purpose: To increase the operator.

Syntax:

INC destiny The instruction adds 1 to the destiny operator and keeps the result in the same destiny operator.

4.9 Comparison instructions

They are used to compare operators, and they affect the content of the flags.

CMP
CMPS (CMPSB) (CMPSW)

CMP INSTRUCTION

Purpose: To compare the operators.

Syntax:

CMP destiny, source

This instruction subtracts the source operator from the destiny operator but without this one storing the result of the operation, and it only affects the state of the flags.

CMPS (CMPSB) (CMPSW) INSTRUCTION

Purpose: To compare chains of a byte or a word.

Syntax:

CMP destiny, source

With this instruction the chain of source characters is subtracted from the destiny chain.

DI is used as an index for the extra segment of the source chain, and SI as an index of the destiny chain.

It only affects the content of the flags and DI as well as SI are incremented.

4.10 Flag instructions

They directly affect the content of the flags.

CLC
CLD
CLI
CMC
STC
STD
STI

CLC INSTRUCTION

Purpose: To clean the cartage flag.

Syntax:

CLC

This instruction turns off the bit corresponding to the cartage flag, or in other words it puts it on zero.

CLD INSTRUCTION

Purpose: To clean the address flag.

Syntax:

CLD

This instruction turns off the corresponding bit to the address flag.

CLI INSTRUCTION

Purpose: To clean the interruption flag.

Syntax:

CLI

This instruction turns off the interruptions flag, disabling this way those maskarable interruptions.

A maskarable interruptions is that one whose functions are deactivated when IF=0.

CMC INSTRUCTION

Purpose: To complement the cartage flag.

Syntax:

CMC

This instruction complements the state of the CF flag, if CF = 0 the instructions equals it to 1, and if the instruction is 1 it equals it to 0.

We could say that it only "inverts" the value of the flag.

STC INSTRUCTION

Purpose: To activate the cartage flag.

Syntax:

STC

This instruction puts the CF flag in 1.

STD INSTRUCTION

Purpose: To activate the address flag.

Syntax:

STD

The STD instruction puts the DF flag in 1.

STI INSTRUCTION

Purpose: To activate the interruption flag.

Syntax:

STI

The instruction activates the IF flag, and this enables the maskarable external interruptions (the ones which only function when IF = 1).

5 Interruptions and file managing

Contents

5.1 Internal hardware interruptions
5.2 External hardware interruptions
5.3 Software interruptions
5.4 Most Common interruptions

5.1 Internal hardware interruptions

Internal interruptions are generated by certain events which come during the execution of a program.

This type of interruptions are managed on their totality by the hardware and it is not possible to modify them.

A clear example of this type of interruptions is the one which actualizes the counter of the computer internal clock, the hardware makes the call to this interruption several times during a second in order to maintain the time to date.

Even though we cannot directly manage this interruption, since we cannot control the time dating by means of software, it is possible to use its effects on the computer to our benefit, for example to create a "virtual clock" dated continuously thanks to the clock's internal counter. We only have to write a program which reads the actual value of the counter and to translates it into an understandable format for the user.

5.2 External hardware interruptions

External interruptions are generated by peripheral devices, such as keyboards, printers, communication cards, etc. They are also generated by coprocessors. It is not possible to deactivate external interruptions.

These interruptions are not sent directly to the CPU, but rather they are sent to an integrated circuit whose function is to exclusively handle this type of interruptions. The circuit, called PIC8259A, is controlled by the CPU using for this control a series of communication ways called paths.

5.3 Software interruptions

Software interruptions can be directly activated by the assembler invoking the number of the desired interruption with the INT instruction.

The use of interruptions helps us in the creation of programs, and by using them our programs are shorter, it is easier to understand them and they usually have a better performance mostly due to their smaller size.

This type of interruptions can be separated in two categories: the operative system DOS interruptions and the BIOS interruptions.

The difference between the two is that the operative system interruptions are easier to use but they are also slower since these interruptions make use of the BIOS to achieve their goal, on the other hand the BIOS interruptions are much faster but they have the disadvantage that since they are part of the hardware, they are very specific and can vary depending even on the brand of the maker of the circuit.

The election of the type of interruption to use will depend solely on the characteristics you want to give your program: speed, using the BIOS ones, or portability, using the ones from the DOS.

5.4 Most common interruptions

Contents

5.4.1 Int 21H (DOS interruption)
Multiple calls to DOS functions.

5.4.2 Int 10H (BIOS interruption)
Video input/output.

5.4.3 Int 16H (BIOS interruption)
Keyboard input/output.

5.4.4 Int 17H (BIOS interruption)
Printer input/output.

5.41 21H Interruption

Purpose: To call on diverse DOS functions.

Syntax:

Int 21H

Note: When we work in TASM program is necessary to specify that the value we are using is hexadecimal.

This interruption has several functions, to access each one of them it is necessary that the function number which is required at the moment of calling the interruption is in the AH register.

Functions to display information to the video.

02H Exhibits output
09H Chain Impression (video)
40H Writing in device/file

Functions to read information from the keyboard.

01H Input from the keyboard
0AH Input from the keyboard using buffer
3FH Reading from device/file

Functions to work with files.

In this section only the specific task of each function is exposed, for a reference about the concepts used, refer to unit 7, titled : "Introduction to file handling".

FCB Method

0FH Open file
14H Sequential reading
15H Sequential writing
16H Create file
21H Random reading
22H Random writing

Handles

3CH Create file
3DH Open file
3EH Close file driver
3FH Reading from file/device
40H Writing in file/device
42H Move pointer of reading/writing in file

VIDEO DISPLAY FUNCTIONS

02H FUNCTION

Use:

It displays one character to the screen.

Calling registers:

AH = 02H
DL = Value of the character to display.

Return registers:

None.

This function displays the character whose hexadecimal code corresponds to the value stored in the DL register, and no register is modified by using this command.

The use of the 40H function is recommended instead of this function.

09H FUNCTION

Use:

It displays a chain of characters on the screen.

Call registers:

AH = 09H
DS:DX = Address of the beginning of a chain of characters.

Return registers:

None.

This function displays the characters, one by one, from the indicated address in the DS:DX register until finding a $ character, which is interpreted as the end of the chain.

It is recommended to use the 40H function instead of this one.

40H FUNCTION

Use:

To write to a device or a file.

Call registers:

AH = 40H
BX = Path of communication
CX = Quantity of bytes to write
DS:DX = Address of the beginning of the data to write

Return registers:

CF = 0 if there was no mistake

AX = Number of bytes written

CF = 1 if there was a mistake

AX = Error code

The use of this function to display information on the screen is done by giving the BX register the value of 1 which is the preassigned value to the video by the operative system MS-DOS.

KEYBOARD INFORMATION FUNCTIONS

01H FUNCTION

Use:

To read a keyboard character and to display it.

Call registers

AH = 01H

Return registers:

AL = Read character

It is very easy to read a character from the keyboard with this function, the hexadecimal code of the read character is stored in the AL register. In case it is an extended register the AL register will contain the value of 0 and it will be necessary to call on the function again to obtain the code of that character.

0AH FUNCTION

Use:

To read keyboard characters and store them on the buffer.

Call registers:

AH = 0AH
DS:DX = Area of storage address
BYTE 0 = Quantity of bytes in the area
BYTE 1 = Quantity of bytes read
from BYTE 2 till BYTE 0 + 2 = read characters

Return characters:

None.

The characters are read and stored in a predefined space on memory. The structure of this space indicate that in the first byte are indicated how many characters will be read. On the second byte the number of characters already read are stored, and from the third byte on the read characters are written.

When all the indicated characters have been stored the speaker sounds and any additional character is ignored. To end the capture of the chain it is necessary to hit [ENTER].

3FH FUNCTION

Use:

To read information from a device or file.

Call registers:

AH = 3FH
BX = Number assigned to the device
CX = Number of bytes to process
DS:DX = Address of the storage area

Return registers:

CF = 0 if there is no error and AX = number of read bytes.
CF = 1 if there is an error and AX will contain the error code.

FILE WORKING FUNCTIONS:

FCB FUNCTIONS:

0FH FUNCTION

Use:

To open an FCB file

Call registers:

AH = 0FH
DS:DX = Pointer to an FCB

Return registers:

AL = 00H if there was no problem, otherwise it returns to 0FFH

14H FUNCTION

Use:

To sequentially read an FCB file.

Call registers:

AH = 14H
DS:DX = Pointer to an FCB already opened.

Return registers:

AL = 0 if there were no errors, otherwise the corresponding error code will be returned: 1 error at the end of the file, 2 error on the FCB structure and 3 partial reading error.

What this function does is that it reads the next block of information from the address given by DS:DX,

What this function does is that it reads the next block of information from the address given by DS:DX, and dates this register.

15H FUNCTION

Use:

To sequentially write and FCB file.

Call registers:

AH = 15H
DS:DX = Pointer to an FCB already opened.

Return registers:

AL = 00H if there were no errors, otherwise it will contain the error code: 1 full disk or read-only file, 2 error on the formation or on the specification of the FCB.

The 15H function dates the FCB after writing the register to the present block.

16H FUNCTION

Use:

To create an FCB file.

Call registers:

AH = 16H
DS:DX = Pointer to an already opened FCB.

Return registers:

AL = 00H if there were no errors, otherwise it will contain the 0FFH value.

It is based on the information which comes on an FCB to create a file on a disk.

21H FUNCTION

Use:

To read in an random manner an FCB file.

Call registers:

AH = 21H
DS:DX = Pointer to and opened FCB.

Return registers:

A = 00H if there was no error, otherwise AH will contain the code of the error: 1 if it is the end of file, 2 if there is an FCB specification error and 3 if a partial register was read or the file pointer is at the end of the same.

This function reads the specified register by the fields of the actual block and register of an opened FCB and places the information on the DTA, Disk Transfer Area.

22H FUNCTION

Use:

To write in an random manner an FCB file.

Call registers:

AH = 22H
DS:DX = Pointer to an opened FCB.

Return registers:

AL = 00H if there was no error, otherwise it will contain the error code: 1 if the disk is full or the file is an only read and 2 if there is an error on the

It writes the register specified by the fields of the actual block and register of an opened FCB. It writes this information from the content of the DTA.

FILE WORKING FUNCTIONS:

HANDLES:

3CH FUNCTION

Use:

To create a file if it does not exist or leave it on 0 length if it exists,
Handle.

Call registers:

AH = 3CH
CH = File attribute
DS:DX = Pointer to an ASCII specification.

Return registers:
CF = 0 and AX the assigned number to handle if there is no error, in case there is, CF will be 1 and AX will contain the error code: 3 path not found, 4 there
CF will be 1 and AX will contain the error code: 3 path not found, 4 there are no handles available to assign and 5 access denied.

This function substitutes the 16H function. The name of the file is specified on an ASCII chain, which has as a characteristic being a conventional chain of bytes ended with a 0 character.

The file created will contain the attributes defined on the CX register in the following manner:

Value	Attributes
00H	Normal
02H	Hidden
04H	System
06H	Hidden and of system

The file is created with the reading and writing permissions. It is not possible to create directories using this function.

3DH FUNCTION

Use:

It opens a file and returns a handle.

Call registers:

AH = 3DH
AL = manner of access
DS:DX = Pointer to an ASCII specification

Return registers:

CF = 0 and AX = handle number if there are no errors, otherwise CF = 1 and AX = error code: 01H if the function is not valid, 02H if the file was not found, 03H if the path was not found, 04H if there are no available handles, 05H in case access is denied, and 0CH if the access code is not valid.

The returned handled is 16 bits.

The access code is specified in the following way:

BITS

```
       7  6  5  4  3  2  1

       .  .  .  .  0  0  0      Only reading
       .  .  .  .  0  0  1      Only writing
       .  .  .  .  0  1  0      Reading/Writing
       .  .  .  x  .  .  .      RESERVED
```

3EH FUNCTION

Use:

Close file (handle).

Call registers:

AH = 3EH
BX = Assigned handle

Return registers:

CF = 0 if there were no mistakes, otherwise CF will be 1 and AX will contain the error code: 06H if the handle is invalid.

This function dates the file and frees the handle it was using.

3FH FUNCTION

Use:

To read a specific quantity of bytes from an open file and store them on a specific buffer.

5.4.2 10H INTERRUPTION

Purpose: To call on diverse BIOS video function

Syntax:

Int 10H

This interruption has several functions, all of them control the video input/output, to access each one of them it is necessary that the function number which is required at the moment of calling the interruption is in the Ah register.

In this tutorial we will see some functions of the 10h interruption.

Common functions of the 10h interruption

02H Function, select the cursor position
09H Function, write attribute and character of the cursor
0AH Function, write a character in the cursor position
0EH Function, Alphanumeric model of the writing characters

02H FUNCTION

Use:

Moves the cursor on the computer screen using text model.

Call registers:

AH = 02H

BH = Video page where the cursor is positioned.

DH = row

DL = Column

Return Registers:

None.

The cursor position is defined by its coordinates, starting from the position 0,0 to position 79,24. This means from the left per computer screen corner to right lower computer screen. Therefore the numeric values that the DH and DL registers get in text model are: from 0 to 24 for rows and from 0 to 79 for columns.

09H FUNCTION

Use:

Shows a defined character several times on the computer screen with a defined attribute, starting with the actual cursor position.

Call registers:

AH = 09H
AL = Character to display
BH = Video page, where the character will display it;
BL = Attribute to use
number of repetition.

Return registers:

None

This function displays a character on the computer screen several times, using a specified number in the CX register but without changing the cursor position on the computer screen.

0AH FUNCTION

Use:

Displays a character in the actual cursor position.

Call registers:

AH = 0AH
AL = Character to display
BH = Video page where the character will display it
BL = Color to use (graphic mode only).
CX = number of repetitions

Return registers:

None.

The main difference between this function and the last one is that this one doesn't allow modifications on the attributes neither does it change the cursor position.

0EH FUNCTION

Use:

Displays a character on the computer screen dates the cursor position.

Call registers:

AH = 0EH
AL = Character to display
BH = Video page where the character will display it
BL = Color to use (graphic mode only).

Return registers:

None

5.4.3 16H INTERRUPTION

We will see two functions of the 16 h interruption, these functions are called by using the AH register.

Functions of the 16h interruption

00H Function, reads a character from the keyboard.
01H Function, reads the keyboard state.

00H FUNCTION USE:

Reads a character from the keyboard.

Call registers:

AH = 00H

Return registers:

AH = Scan code of the keyboard
AL = ASCII value of the character

When we use this interruption, the program executing is halted until a character is typed, if this is an ASCII value; it is stored in the Ah register, Else the scan code is stored in the AL register and the AH register contents the value 00h.

The proposal of the scan code is to use it with the keys without ASCII representation as [ALT][CONTROL], the function keys and so on.

01H FUNCTION

Use:

Reads the keyboard state

Call registers:

AH = 01H

Return registers:

If the flag register is zero, this means, there is information on the buffer memory, else, there is no information in the buffer memory. Therefore the value of the Ah register will be the value key stored in the buffer memory.

5.4.4 17H INTERRUPTION

Purpose: Handles the printer input/output.

Syntax:

Int 17H

This interruption is used to write characters on the printer, sets printer and reads the printer state.

Functions of the 16h interruptions

00H Function, prints value ASCII out
01H Function, sets printer
02H Function, the printer state

00H FUNCTION

Use:

Writes a character on the printer.

Call registers:

AH = 00H
AL = Character to print.
DX = Port to use.

Return registers:

AH = Printer device state.

The port to use is in the DX register, the different values are: LPT1 = 0, LPT2 = 1, LPT3 = 2 ...

The printer device state is coded bit by bit as follows:

BIT	1/0	MEANING
0	1	The waited time is over
1	-	
2	-	
3	1	input/output error
4	1	Chosen printer
5	1	out-of-paper
6	1	communication recognized
7	1	The printer is ready to use

1 and 2 bits are not relevant

Most BIOS sport 3 parallel ports, although there are BIOS which sport 4 parallel ports.

01H FUNCTION

Use:

Sets parallel port.

Call registers:

AH = 01H
DX = Port to use

Return registers:

AH = Printer status

Port to use is defined in the DX register, for example: LPT=0, LPT2=1, and so on.

The state of the printer is coded bit by bit as follows:

BIT	1/0	MEANING
0	1	The waited time is over
1	-	
2	-	
3	1	input/output error
4	1	Chosen printer
5	1	out-of-paper
6	1	communication recognized
7	1	The printer is ready to use

1 and 2 bits are not relevant

Most BIOS sport 3 parallel ports, although there are BIOS which sport 4 parallel ports.

02H FUNCTION

Uses:

Gets the printer status.

Call registers:

AH = 01H
DX = Port to use

Return registers

AH = Printer status.

Port to use is defined in the DX register, for example: LPT=0, LPT2=1, and so on

The state of the printer is coded bit by bit as follows:

BIT	1/0	MEANING
0	1	The waited time is over
1	-	
2	-	
3	1	input/output error
4	1	Chosen printer
5	1	out-of-paper
6	1	communication recognized
7	1	The printer is ready to use

1 and 2 bits are not relevant

Most BIOS sport 3 parallel ports, although there are BIOS which sport 4 parallel ports.

5.5 Ways of working with files

There are two ways to work with files, the first one is by means of file control blocks or "FCB" and the second one is by means of communication channels, also known as "handles".

The first way of file handling has been used since the CPM operative system, predecessor of DOS, thus it assures certain compatibility with very old files from the CPM as well as from the 1.0 version of the DOS, besides this method allows us to have an unlimited number of open files at the same time. If you want to create a volume for the disk the only way to achieve this is by using this method.

Even after considering the advantages of the FCB, the use of the communication channels it is much simpler and it allows us a better handling of errors, besides, since it is much newer it is very probable that the files created this way maintain themselves compatible through later versions of the operative system.

For a greater facility on later explanations I will refer to the file control blocks as FCBs and to the communication channels as handles.

5.6 FCB method

Contents

5.6.1 Introduction
5.6.2 Open files
5.6.3 Create a new file
5.6.4 Sequential writing
5.6.5 Sequential reading
5.6.6Random reading and writing
5.6.7 Close a file

5.6.1 Introduction

There are two types of FCB, the normal, whose length is 37 bytes and the extended one of 44 bytes. On this tutorial we will only deal with the first type, so from now on when I refer to an FCB, I am really talking about a 37 bytes FCB.

The FCB is composed of information given by the programmer and by information which it takes directly from the operative system.

When thesetypes of files are used it is only possible to work on the current directory since the FCBs do not provide sport for the use of the organization by directories of DOS.

The FCB is formed by the following fields:

POSITION	LENGTH	MEANING
00H	1 Byte	Drive
01H	8 Bytes	File name
09H	3 Bytes	Extension
0CH	2 Bytes	Block number
0EH	2 Bytes	Register size
10H	4 Bytes	File size
14H	2 Bytes	Creation date
16H	2 Bytes	Creation hour
18H	8 Bytes	Reserved
20H	1 Bytes	Current register
21H	4 Bytes	Random register

To select the work drive the next format is followed: drive A = 1; drive B = 2; etc. If 0 is used the drive being used at that moment will be taken as option.

The name of the file must be justified to the left and in case it is necessary the remaining bytes will have to be filled with spaces, and the extension of the file is placed the same way.

The current block and the current register tell the computer which register will be accessed on reading or writing operations. A block is a group of 128 registers. The first block of the file is the block 0. The first register is the register 0, therefore the last register of the first block would be the 127, since the numbering started with 0 and the block can contain 128 registers in total.

5.6.2 Opening files

To open an FCB file the 21H interruption, 0FH function is used. The unit, the name and extension of the file must be initialized before opening it.

The DX register must point to the block. If the value of FFH is returned on the AH register when calling on the interruption then the file was not found, if everything came out well a value of 0 will be returned.

If the file is opened then DOS initializes the current block to 0, the size of the register to 128 bytes and the size of the same and its date are filled with the information found in the directory.

5.6.3 Creating a new file

For the creation of files the 21H interruption 16H function is used. DX must point to a control structure whose requirements are that at least the logic unit, the name and the extension of the file be defined. In case there is a problem the FFH value will be returned on AL, otherwise this register will contain a value of 0.

5.6.4 Sequential writing

Before we can perform writing to the disk it is necessary to define the data transfer area using for this end the 1AH function of the 21H interruption.

The 1AH function does not return any state of the disk nor or the operation, but the 15H function, which is the one we will use to write to the disk, does it on the AL register, if this one is equal to zero there was no error and the fields of the current register and block are dated.

5.6.5 Sequential reading

Before anything we must define the file transfer area or DTA. In order to sequentially read we use the 14H function of the 21H interruption.

The register to be read is the one which is defined by the current block and register. The AL register returns to the state of the operation, if AL contains a value of 1 or 3 it means we have reached the end of the file. A value of 2 means that the FCB is wrongly structured.

In case there is no error, AL will contain the value of 0 and the fields of the current block and register are dated.

5.6.6 Random reading and writing

The 21H function and the 22H function of the 21H interruption are the ones in charge of realizing the random readings and writings respectively.

The random register number and the current block are used to calculate the relative position of the register to read or write.

The AL register returns the same information for the sequential reading of writing. The information to be read will be returned on the transfer area of the disk, likewise the information to be written resides on the DTA.

5.6.7 Closing a file

To close a file we use the 10H function of the 21H interruption.

If after invoking this function, the AL register contains the FFH value, this means that the file has changed position, the disk was changed or there is error of disk access.

5.7 Channels of communication

Contents

5.7.1 Working with handles
5.7.2 Functions to use handles

5.7.1 Working with handles

The use of handles to manage files greatly facilitates the creation of files and programmer can concentrate on other aspects of the programming without worrying on details which can be handled by the operative system.

The easy use of the handles consists in that to operate o a file, it is only necessary to define the name of the same and the number of the handle to use, all the rest of the information is internally handled by the DOS.

When we use this method to work with files, there is no distinction between sequential or random accesses, the file is simply taken as a chain of bytes.

5.7.2 Functions to use handles

The functions used for the handling of files through handles are described in unit 6: Interruptions, in the section dedicated to the 21H interruption.

6 Macros and procedures

Contents

6.1 Procedures
6.2 Macros

6.1 Procedure

Definition of procedure

A procedure is a collection of instructions to which we can direct the flow of our program, and once the execution of these instructions is over control is given back to the next line to process of the code which called on the procedure.

Procedures help us to create legible and easy to modify programs.

At the time of invoking a procedure the address of the next instruction of the program is kept on the stack so that, once the flow of the program has been transferred and the procedure is done, one can return to the next line of the original program, the one which called the procedure.

Syntax of a Procedure

There are two types of procedures, the intrasegments, which are found on the same segment of instructions, and the inter-segments which can be stored on different memory segments.

When the intrasegment procedures are used, the value of IP is stored on the stack and when the intrasegments are used the value of CS:IP is stored.

To divert the flow of a procedure (calling it), the following directive is used:

CALL NameOfTheProcedure

The part which make a procedure are:

Declaration of the procedure
Code of the procedure
Return directive
Termination of the procedure

For example, if we want a routine which adds two bytes stored in AH and AL each one, and keep the addition in the BX register:

```
Adding Proc Near  ; Declaration of the procedure
Mov Bx, 0         ; Content of the procedure
Mov B1, Ah
Mov Ah, 00
Add Bx, Ax
Ret               ; Return directive
Add Endp          ; End of procedure declaration
```

On the declaration the first word, Adding, corresponds to the name of out procedure, Proc declares it as such and the word Near indicates to the MASM that the procedure is intrasegment.

The Ret directive loads the IP address stored on the stack to return to the original program, lastly, the Add Endp directive indicates the end of the procedure.

To declare an inter segment procedure we substitute the word Near for the word FAR.

The calling of this procedure is done the following way:

Call Adding

Macros offer a greater flexibility in programming compared to the procedures, nonetheless, these last ones will still be used.

6.2 Macros

Contents

6.2.1 Definition of a macro
6.2.2 Syntax of a macro
6.2.3 Macro libraries

6.2.1 Definition of the macro

A macro is a group of repetitive instructions in a program which are codified only once and can be used as many times as necessary.

The main difference between a macro and a procedure is that in the macro the passage of parameters is possible and in the procedure it is not, this is only applicable for the TASM - there are other programming languages which do allow it. At the moment the macro is executed each parameter is substituted by the name or value specified at the time of the call.

We can say then that a procedure is an extension of a determined program, while the macro is a module with specific functions which can be used by different programs.

Another difference between a macro and a procedure is the way of calling each one, to call a procedure the use of a directive is required, on the other hand the call of macros is done as if it were an assembler instruction.

6.2.2 Syntax of a Macro

The parts which make a macro are:

Declaration of the macro
Code of the macro
Macro termination directive

The declaration of the macro is done the following way:

NameMacro MACRO [parameter1, parameter2...]

Even though we have the functionality of the parameters it is possible to create a macro which does not need them.

The directive for the termination of the macro is: **ENDM**

An example of a macro, to place the cursor on a determined position on the screen is:

```
Position  MACRO  Row,  Column
  PUSH AX
  PUSH BX
  PUSH DX
  MOV AH,  02H
  MOV DH, Row
  MOV DL, Column
  MOV BH, 0
  INT 10H
  POP DX
  POP BX
  POP AX
ENDM
```

To use a macro it is only necessary to call it by its name, as if it were another assembler instruction, since directives are no longer necessary as in the case of the procedures.

Example:

Position 8, 6

6.2.3 Macro Libraries

One of the facilities that the use of macros offers is the creation of libraries, which are groups of macros which can be included in a program from a different file.

The creation of these libraries is very simple, we only have to write a file with all the macros which will be needed and save it as a text file.

To call these macros it is only necessary to use the following instruction Include NameOfTheFile, on the part of our program where we would normally write the macros, this is, at the beginning of our program, before the declaration of the memory model.

The macros file was saved with the name of MACROS.TXT, the instruction Include would be used the following way:

```
  ;Beginning of the program
Include MACROS.TXT
.MODEL SMALL
.DATA
  ;The data goes here
.CODE
Beginning:
  ;The code of the program is inserted here
.STACK
  ;The stack is defined
End beginning
  ;Our program ends
```

More debug program examples

In this section we provide you several assembler programs to run in the debug program. You can execute each assembler program using the "t" (trace) command, to see what each instruction does.

First example

```
-a0100
297D:0100     MOV     AX,0006        ; Puts value 0006 at register AX
297D:0103     MOV     BX,0004        ;Puts value 0004 at register BX
297D:0106     ADD     AX,BX         ;Adds BX to AX contents
297D:0108      INT     20          ;Causes end of the Program
```

The only thing that this program does is to save two values in two registers and add the value of one to the other.

Second example

```
- a100
0C1B:0100 jmp 125 ; Jumps to direction 125H
0C1B:0102 [Enter]
- e 102  'Hello, How are you ?' 0d 0a '$'
- a125
0C1B:0125 MOV DX,0102 ; Copies string to DX register
0C1B:0128 MOV CX,000F ; Times the string will be displayed
0C1B:012B MOV AH,09 ; Copies 09 value to AH register
0C1B:012D INT 21 ; Displays string
0C1B:012F DEC CX ; Reduces in 1 CX
0C1B:0130 JCXZ 0134 ; If CX is equal to 0 jumps to 0134
0C1B:0132 JMP 012D ; Jumps to direction 012D
0C1B:0134 INT 20 ; Ends the program
```

This program displays on the screen 15 times a character string.

Third example

```
-a100
297D:0100       MOV     AH,01 ;Function to change the cursor
297D:0102       MOV     CX,0007 ;Forms the cursor
297D:0105       INT     10      ;Calls for BIOS
297D:0107       INT     20      ;Ends the program
```

This program is good for changing the form of the cursor.

Fourth example

```
-a100
297D:0100         MOV     AH,01 ; Funtion  1 (reads keyboard)
297D:0102         INT     21 ; Calls for DOS
297D:0104        CMP     AL,0D ; Compares if what is read is a carriage return
297D:0106        JNZ     0100 ; If it is not, reads another character
297D:0108        MOV     AH,02 ; Funtion 2 (writes on the screen)
297D:010A       MOV     DL,AL ; Character to write on AL
297D:010C       INT     21 ; Calls for DOS
297D:010E       INT     20 ; Ends the program
```

This program uses DOS 21H interruption. It uses two functions of the same: the first one reads the keyboard (function 1) and the second one writes on the screen. It reads the keyboard characters until it finds a carriage
return.

Fifth example

```
-a100
297D:0100       MOV     AH,02  ; Function 2 (writes on the screen)
297D:0102       MOV     CX,0008; Puts value 0008 on register CX
297D:0105       MOV     DL,00  ; Puts value 00 on register DL
297D:0107       RCL     BL,1   ; Rotates the byte in BL to the left by one bit
                               ;   through the carry flag
297D:0109       ADC     DL,30  ; Converts flag register to1
297D:010C       INT     21     ; Calls for DOS
297D:010E       LOOP    0105   ; Jumps if CX > 0  to direction 0105
297D:0110       INT     20     ; Ends the program
```

This program displays on the screen a binary number through a conditional cycle (LOOP) using byte rotation.

Sixth example

```
-a100
297D:0100       MOV     AH,02  ; Function 2 (writes on the screen)
297D:0102       MOV     DL,BL  ; Puts BL's value on DL
297D:0104       ADD     DL,30  ; Adds value 30 to DL
297D:0107       CMP     DL,3A  ; Compares 3A value with DL's contents without
                                ; affecting its value only modifying the state
of
                              ; the car
297D:010A       JL      010F   ; jumps if < direction 010f
297D:010C       ADD     DL,07  ;  Adds 07 value on DL
297D:010F       INT     21     ; Calls for Dos
297D:0111       INT     20     ; Ends the Program
```

This program prints a zero value on hex digits

Seventh example

```
-a100
297D:0100      MOV     AH,02  ; Function 2 (writes on the screen)
297D:0102      MOV     DL,BL  ; Puts BL value on DL
297D:0104      AND     DL,0F  ; Carries ANDing numbers bit by bit
297D:0107      ADD     DL,30  ; Adds 30 to Dl
297D:010A      CMP     DL,3A  ; Compares Dl with 3A
297D:010D      JL      0112   ; Jumps if < 0112 direction
297D:010F      ADD     DL, 07 ; Adds  07 to DL
297D:0112      INT     21     ; Calls for Dos
297D:0114      INT     20     ; Ends the program
```

This program is used to print two digit hex numbers.

Eight example

```
-a100
297D:0100       MOV     AH,02 ; Function 2 (writes on the screen)
297D:0102       MOV     DL,BL ; Puts BL value on DL
297D:0104       MOV     CL,04 ;  Puts 04 value on CL
297D:0106       SHR     DL,CL ;  Moves per four bits of your number to the
                              ; rightmost nibble
297D:0108       ADD     DL,30 ; Adds  30 to DL
297D:010B       CMP     L,3A  ; Compares Dl with 3A
297D:010E       JL      0113  ; Jumps if < 0113 direction
297D:0110       ADD     DL,07 ; Adds 07 to DL
297D:0113       INT     21    ; Calls for Dos
297D:0115       INT     20    ; Ends the program
```

This program works for printing the first of two digit hex numbers

Ninth example

```
-a100
297D:0100      MOV     AH,02 ; Function 2 (writes on the screen)
297D:0102      MOV     DL,BL ; Puts BL value on DL
297D:0104      MOV     CL,04 ; Puts 04 value on CL
297D:0106      SHR     DL,CL ; Moves per four bits of your number to the
                             ;rightmost nibble
297D:0108      ADD     DL,30 ; Adds 30 to DL
297D:010B      CMP     DL,3A ; Compares Dl with 3A
297D:010E      JL      0113  ; Jumps if < 0113 direction
297D:0110      ADD     DL,07 ; Adds 07 to DL
297D:0113      INT     21    ; Calls for Dos
```

```
297D:0115      MOV     DL,BL ; Puts Bl value on DL
297D:0117      AND     DL,0F ; Carries ANDing numbers bit by bit
297D:011A      ADD     DL,30 ; Adds 30 to DL
297D:011D      CMP     DL,3A ; Compares Dl with 3A
297D:0120       JL      0125 ; Jumps if < 125 direction
297D:0122      ADD     DL,07 ; Adds 07 to DL
297D:0125      INT     21    ; Calls for Dos
297D:0127      INT     20    ; Ends the Program
```

This program works for printing the second of two digit hex numbers

Tenth example

```
-a100
297D:0100       MOV     AH,01 ; Function 1 (reads keyboard)
297D:0102       INT     21    ; Calls for  Dos
297D:0104       MOV     DL,AL ; Puts Al value on DL
297D:0106       SUB     DL,30 ;  Subtracts 30 from DL
297D:0109       CMP     DL,09 ; Compares DL with 09
297D:010C       JLE     0111  ; Jumps if <=  0111 direction
297D:010E       SUB     DL,07 ; Subtracts 07 from DL
297D:0111       MOV     CL,04 ; Puts 04 value on CL register
297D:0113       SHL     DL,CL ; It inserts zeros to the right
297D:0115       INT     21    ; Calls for Dos
297D:0117       SUB     AL,30 ; Subtracts 30 from AL
297D:0119       CMP     AL,09 ; Compares AL with 09
297D:011B       JLE     011F  ; Jumps if <= 011f direction
297D:011D       SUB     AL,07 ; Subtracts 07 from AL
297D:011F       ADD     DL,AL ; Adds Al to DL
297D:0121       INT     20    ; Ends the Program
```

This program can read two digit hex numbers

Eleventh example

```
-a100
297D:0100      CALL    0200 ;  Calls for a procedure
297D:0103      INT     20 ;Ends the program

-a200
297D:0200      PUSH    DX    ;  Puts DX value on the stack
297D:0201      MOV     AH,08 ; Function 8
297D:0203      INT     21    ; Calls for Dos
297D:0205      CMP     AL,30 ; Compares AL with 30
297D:0207      JB      0203  ; Jumps if CF is activated towards 0203 direction
297D:0209      CMP     AL,46 ; Compares AL with 46
297D:020B      JA      0203  ; jumps if <> 0203 direction
297D:020D      CMP     AL,39 ; Compares AL with 39
297D:020F      JA      021B  ; Jumps if <> 021B direction
```

```
297D:0211      MOV     AH,02 ; Function 2 (writes on the screen)
297D:0213      MOV     DL,AL ; Puts Al value on DL
297D:0215      INT     21    ; Calls for Dos
297D:0217      SUB     AL,30 ; Subtracts 30 from AL
297D:0219      POP     DX    ;  Takes DX value out of the stack
297D:021A      RET           ; Returns control to the main program
297D:021B      CMP     AL,41 ; Compares AL with 41
297D:021D      JB      0203  ; Jumps if CF is activated towards 0203 direction
297D:021F      MOV     AH,02 ; Function 2 (writes on the screen)
297D:022       MOV     DL,AL ; Puts AL value on DL
297D:0223      INT     21    ; Calls for Dos
297D:0225      SUB     AL,37 ; Subtracts 37 from AL
297D:0227      POP     DX    ; Takes DX value out of the stack
297D:0228      RET           ; Returns control to the main program
```

This program keeps reading characters until it receives one that can be converted to a hex number

More Assembler programs examples(using TASM program)

```
;name of the program:one.asm
;
.model small
.stack
.code
        mov AH,1h       ;Selects the 1  D.O.S. function
        Int 21h         ;reads character and return ASCII code to register AL
        mov DL,AL       ;moves the ASCII code to register DL
         sub DL,30h       ;makes the operation minus 30h to convert 0-9 digit
number
        cmp DL,9h       ;compares if digit number it was between 0-9
        jle digit1      ;If it true gets the first number digit (4 bits long)
          sub DL,7h        ;If it false, makes operation minus 7h to convert
letter A-F
digit1:
        mov CL,4h       ;prepares to multiply by 16
        shl DL,CL       ; multiplies to convert into four bits upper
        int 21h         ;gets the next character
        sub AL,30h      ;repeats the conversion operation
        cmp AL,9h       ;compares the value 9h with the content of register AL
        jle digit2      ;If true, gets the second digit number
        sub AL,7h       ;If no, makes the minus operation 7h
digit2:
        add DL,AL       ;adds the second number digit
        mov AH,4CH
        Int 21h         ;21h interruption
        End             ; finishs the program code
```

This program reads two characters from the keyboard and prints them on the screen.

```
;name the program:two.asm
.model small
.stack
.code
PRINT_A_J       PROC
        MOV DL,'A'      ;moves the A character to register DL
        MOV CX,10       ;moves the decimal value 10 to register cx
                        ;This number value its the time to print out after the
A                                       ;character
PRINT_LOOP:
        CALL WRITE_CHAR ;Prints A character out
        INC DL          ;Increases the value of register DL
        LOOP PRINT_LOOP ;Loop to print out ten characters
        MOV AH,4Ch      ;4Ch function of the 21h interruption
        INT 21h         ;21h interruption
PRINT_A_J       ENDP    ;Finishes the procedure
```

```
WRITE_CHAR      PROC
        MOV AH,2h       ;2h function of the 21 interruption
        INT 21h         ;Prints character out from the register DL
        RET             ;Returns the control to procedure called
WRITE_CHAR      ENDP    ;Finishes the procedure
        END  PRINT_A_J  ;Finishes the program code
```

This program prints the a character through j character on the screen

```
;name of the program :three.asm
.model small
.STACK
.code

TEST_WRITE_HEX    PROC
        MOV DL,3Fh     ;moves the value 3Fh to the register DL
        CALL WRITE_HEX ;Calls the procedure
        MOV AH,4CH     ;4Ch function
        INT 21h        ;Returns the control to operating system
TEST_WRITE_HEX ENDP    ;Finishes the procedure

        PUBLIC WRITE_HEX
;.....................................................;
; This procedure converts into hexadecimal number the byte is in the register
DL and show the digit number;
;Use:WRITE_HEX_DIGIT                                  ;
;.....................................................;

WRITE_HEX    PROC
        PUSH CX     ;pushes the value of the register CX to the stack memory
        PUSH DX     ;pushes the value of the register DX to the stack memory
      MOV DH,DL     ;moves the value of the register DL to register DH
        MOV CX,4    ;moves the value numeric 4 to register CX
        SHR DL,CL
        CALL  WRITE_HEX_DIGIT  ;shows  on  the  computer  screen,  the  first
hexadecimal number
        MOV DL,DH   ;moves the value of the register DH to the register DL
        AND DL,0Fh  ;ANDing the upper bit
        CALL  WRITE_HEX_DIGIT  ;  shows  on  the  computer  screen,  the  second
hexadecimal number
        POP DX      ;pops the value of the register DX to register DX
        POP CX      ; pops the value of the register DX to register DX
        RET         ;Returns the control of the procedure called
WRITE_HEX  ENDP

        PUBLIC WRITE_HEX_DIGIT
;.....................................................................;
;                                                                     ;
; This procedure converts the lower 4 bits of the register DL into hexadecimal
;number and show them in the computer screen                          ;
;Use: WRITE_CHAR                                                      ;
;.....................................................................;

WRITE_HEX_DIGIT    PROC
         PUSH DX          ;Pushes the value of the register DX in the stack
memory
        CMP DL,10       ;compares if the bit number is minus than number ten
        JAE HEX_LETTER  ;No , jumps HEX_LETER
        ADD DL,"0"      ;yes, it converts into digit number
        JMP Short WRITE_DIGIT ;writes the character
```

```
HEX_LETTER:
        ADD DL,"A"-10   ;converts a character into hexadecimal number
WRITE_DIGIT:
        CALL WRITE_CHAR ;shows the character in the computer screen
         POP DX     ;Returns the initial value of the register DX to register
DL
        RET             ;Returns the control of the procedure called
WRITE_HEX_DIGIT   ENDP

        PUBLIC WRITE_CHAR
;.................................................................;
;This procedure shows the character in the computer screen using the D.O.S. ;
;.................................................................;

WRITE_CHAR   PROC
        PUSH AX   ;pushes the value of the register AX in the stack memory
        MOV AH,2  ;2h Function
        INT 21h   ;21h Interruption
         POP AX    ;Pops the initial value of the register AX to the register
AX
        RET       ;Returns the control of the procedure called
WRITE_CHAR   ENDP

        END TEST_WRITE_HEX ;finishes the program code
```

This program prints a predefined value on the screen

```
;name of the program:five.asm
.model small
.stack
.code

PRINT_ASCII       PROC
        MOV DL,00h      ;moves the value 00h to register DL
        MOV  CX,255       ;moves the value decimal number 255. this decimal
number
                        ;will be 255 times to print out after the character A
PRINT_LOOP:
        CALL WRITE_CHAR    ;Prints the characters out
        INC DL             ;Increases the value of the register DL content
        LOOP PRINT_LOOP    ;Loop to print out ten characters
        MOV AH,4Ch         ;4Ch function
        INT 21h            ;21h Interruption
PRINT_ASCII       ENDP     ;Finishes the procedure

WRITE_CHAR      PROC
        MOV AH,2h          ;2h function to print character out
        INT 21h            ;Prints out the character in the register DL
        RET                ;Returns the control to the procedure called
```

```
WRITE_CHAR      ENDP       ;Finishes the procedure

        END  PRINT_ASCII   ;Finishes the program code
```

This program prints the 256 ASCII code on the screen

```
dosseg
.model small
.stack
.code
write   proc
        mov ah,2h;
        mov dl,2ah;
        int 21h
        mov ah,4ch
        int 21h
write  endp

        end  write
```

This program prints a defined character using an ASCII code on the screen.

```
.model small; the name of the program is seven.asm
.stack;
.code;

EEL:    MOV AH,01 ; 1 function (reads one character from the keyboard)
        INT 21h ; 21h interruption
        CMP AL,0Dh ; compares the value with  0dh
        JNZ EEL ;jumps if no equal of the label eel
        MOV AH,2h ; 2 function (prints the character out on the screen)
        MOV DL,AL ;moves the value of the register  AL to the register  DL
        INT 21h  ;21 interruption
         MOV AH,4CH ;4C function (returns the control to the D.O.S. operating
system)
        INT 21h ;21 interruption

        END ;finishes the program
```

This program reads characters form the keyboard and prints them on the screen until find the return character.