

## Lesson I: Overview of the 80x86 Family

The 80x86 family was first started in 1981 with the 8086 and the newest member is the Pentium which was released thirteen years later in 1994. They are all backwards compatible with each other but each new generation has added features and more speed than the previous chip. Today there are very few computers in use that have the 8088 and 8086 chips in them as they are very outdated and slow. The number of 286 or 386 based computers around is declining as today's software becomes more and more demanding. Even 486's are being replaced by Pentiums. With the Pentium PRO and the MMX based CPUs Intel keeps increasing performance and features.

### Representation of numbers in binary

Before we begin to understand how to program in assembly it is best to try to understand how numbers are represented in computers. Numbers are stored in binary, base two. There are several terms which are used to describe different size numbers and I will describe what these mean.

1 BIT: 0

One bit is the simplest piece of data that exists. Its either a one or a zero.

1 NIBBLE: 0000

4 BITS

The nibble is four bits or half a byte. Note that it has a maximum value of 15 ( $1111 = 15$ ). This is the basis for the hexadecimal (base 16) number system which is used as it is far easier to understand. Hexadecimal numbers go from 1 to F and are followed by a h to state that they are in hex. i.e. Fh = 15 decimal. Hexadecimal numbers that begin with a letter are prefixed with a 0 (zero).

1 BYTE 00000000

2 NIBBLES

8 BITS

A byte is 8 bits or 2 nibbles. A byte has a maximum value of FFh (255 decimal). Because a byte is 2 nibbles the hexadecimal representation is two hex digits in a row i.e. 3Dh. The byte is also that size of the 8-bit registers which we will be covering later.

1 WORD 0000000000000000

2 BYTES

4 NIBBLES

16 BITS

A word is two bytes that are stuck together. A word has a maximum value of FFFFh (65,536). Since a word is four nibbles, it is represented by four hex digits. This is the size of the 16-bit registers.

## Registers

Registers are a place in the CPU where a number can be stored and manipulated. There are three sizes of registers: 8-bit, 16-bit and on 386 and above 32-bit. There are four different types of registers; general purpose registers, segment registers, index registers and stack registers. Firstly here are descriptions of the main registers. Stack registers and segment registers will be covered later.

15	7	0	
AH	AL		AX
BH	BL		BX
CH	CL		CX
DH	DL		DX

*General Purpose Registers*

## General Purpose Registers

These are 16-bit registers. There are four general purpose registers; AX, BX, CX and DX. They are split up into 8-bit registers. AX is split up into AH which contains the high byte and AL which contains the low byte. On 386's and above there are also 32-bit registers, these have the same names as the 16-bit registers but with an 'E' in front i.e. EAX. You can use AL, AH, AX and EAX separately and treat them as separate registers for some tasks.

If AX contained 24689 decimal:

AH	AL
01100000	01110001

AH would be 96 and AL would be 113. If you added one to AL it would be 114 and AH would be unchanged. SI, DI, SP and BP can also be used as general purpose registers but have more specific uses. They are not split into two halves.

## Index Registers

These are sometimes called pointer registers and they are 16-bit registers. They are mainly used for string instructions. There are three index registers SI (source index), DI (destination index) and IP (instruction pointer). On 386's and above there are also 32-bit index registers: EDI and ESI. You can also use BX to index strings.

IP is a index register but it can't be manipulated directly as it stores the address of the next instruction.

## Stack registers

BP and SP are stack registers and are used when dealing with the stack. They will be covered when we talk about the stack later on.

## Segments and offsets

The original designers of the 8088 decided that nobody will ever need to use more than one megabyte of memory so they built the chip so it couldn't access above that. The problem is to access a whole megabyte 20 bits are needed. Registers only have 16 bits and they didn't want to use two because that would be 32 bits and they thought that this would be too much for anyone. They came up with what they thought was a clever way to solve this problem: segments and offsets. This is a way to do the addressing with two registers but not 32 bits.

$$\text{OFFSET} = \text{SEGMENT} * 16$$

$$\text{SEGMENT} = \text{OFFSET} / 16 \text{ (the lower 4 bits are lost)}$$

One register contains the segment and another register contains the offset. If you put the two registers together you get a 20-bit address.

SEGMENT	0010010000010000----
OFFSET	----0100100000100010
20-bit Address	00101000100100100010

DS stores the Segment and SI stores the offset. As they are both 16 bits long the addresses overlap. This is how DS:SI is used to make a 20 bit address. The segment is in DS and the offset is in SI. The standard notation for a Segment/Offset pair is: SEGMENT:OFFSET

Segment registers are: CS, DS, ES, SS. On the 386+ there are also FS and GS.

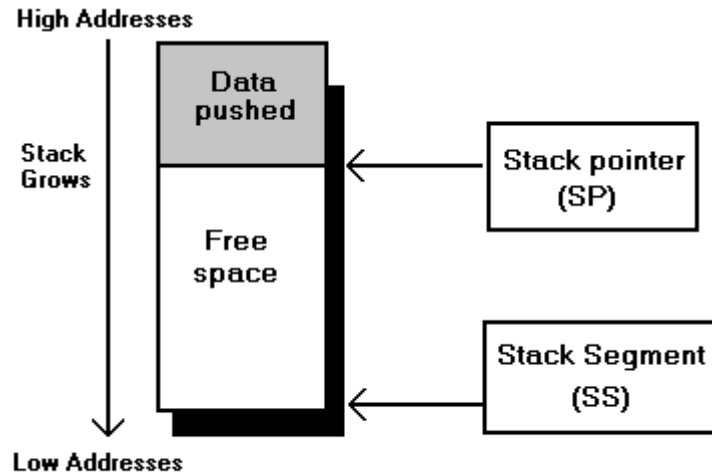
Offset registers are: BX, DI, SI, BP, SP, IP. In 386+ protected mode, ANY general register (not a segment register) can be used as an Offset register. (Except IP, which you can't manipulate directly).

If you are now thinking that assembly must be really hard and you don't understand segments and offsets at all then don't worry. I didn't understand them at first but I struggled on and found out that they were not so hard to use in practice.

## The Stack

As there are only six registers that are used for most operations, you're probably wondering how do you get around that. It's easy. There is something called a stack which is an area of memory which you can save and restore values to.

This is an area of memory that is like a stack of plates. The last one you put on is the first one that you take off. This is sometimes referred to as Last On First Off (LOFO) or Last In First Out (LIFO).



*How the stack is organised*

If another piece of data is put on the stack it grows downwards. As you can see the stack starts at a high address and grows downwards. You have to make sure that you don't put too much data in the stack or it will overflow.

## Lesson 2 : An Introduction to Assembly Instructions

There are a lot of instructions in assembly but there are only about twenty that you have to know and will use very often. Most instructions are made up of three characters and have an operand then a comma then another operand. For example to put a data into a register you use the MOV instruction.

```
mov ax,10      ; put 10 into ax
mov bx,20      ; put 20 into bx
mov cx,30      ; put 30 into cx
mov dx,40      ; put 40 into dx
```

Notice that in assembler anything after a ; (semicolon) is ignored. This is very useful for commenting your code.

### Push and Pop: Two Instructions to use the Stack

You know about the stack but not how to put data in an out of it. There are two simple instructions that you need to know: push and pop. Here is the syntax for their use:

**PUSH:** Puts a piece of data onto the top of the stack

Syntax:

```
push data
```

**POP:** Puts the piece of data from the top of the stack into a specified register or variable.

Syntax:

```
pop register (or variable)
```

This example of code demonstrates how to use the push and pop instructions

```
push cx ; put cx on the stack
```

```
push ax ; put ax on the stack
```

```
pop cx ; put value from stack into cx
```

```
pop ax ; put value from stack into ax
```

Notice that the values of CX and AX will be exchanged. There is an instruction to exchange two registers: XCHG, which would reduce the previous fragment to "xchg ax,cx".

## Types of Operand

There are three types of operands in assembler: immediate, register and memory. Immediate is a number which will be known at compilation and will always be the same for example '20' or 'A'. A register operand is any general purpose or index register for example AX or SI. A memory operand is a variable which is stored in memory which will be covered later.

## Some Instructions that you will need to know

This is a list of some important instructions that you need to know before you can understand or write assembly programs.

**MOV:** moves a value from one place to another.

Syntax:

```
MOV destination, source
```

for example:

```
mov ax,10      ; moves an immediate value into ax
mov bx,cx      ; moves value from cx into bx
mov dx,Number  ; moves the value of Number into dx
```

**INT:** calls a DOS or BIOS function which are subroutines to do things that we would rather not write a function for e.g. change video mode, open a file etc.

Syntax:

INT interrupt number

For example:

```
int 21h        ; Calls DOS service
int 10h        ; Calls the Video BIOS interrupt
```

Most interrupts have more than one function, this means that you have to pass a number to the function you want. This is usually put in AH. To print a message on the screen all you need to do is this:

```
mov ah,9 ; subroutine number 9
int 21h  ; call the interrupt
```

But first you have to specify what to print. This function needs

DS:DX to be a far pointer to where the string is. The string has to be terminated with a dollar sign (\$). This would be easy if DS could be manipulated directly, to get round this we have to use AX.

This example shows how it works:

```
mov dx,OFFSET Message ; DX contains offset of message
mov ax,SEG Message    ; AX contains segment of message
mov ds,ax              ; DS:DX points to message
mov ah,9               ; function 9 - display string
int 21h                ; call dos service
```

The words OFFSET and SEG tell the compiler that you want the segment or the offset of the message put in the register not the contents of the message. Now we know how to set up the code to display the message we need to declare the message. In the data segment we declare the message like this:

Message DB "Hello World!\$"

Notice that the string is terminated with an dollar sign. What does 'DB' mean? DB is short for declare byte and the message is an array of bytes (an ASCII character takes up one byte). Data can be declared in a number of sizes: bytes (DB), words (DW) and double words (DD). You don't have to worry about double words at the moment as you need a 32-bit register, such as EAX, to fit them in.

Here are some examples of declaring data:

```
Number1 db ?
```

```
Number2 dw ?
```

The question mark (?) on the end means that the data isn't initialised i.e. it has no value in to start with. That could as easily be written as:

```
Number1 db 0
```

```
Number2 dw 1
```

This time Number1 is equal to 0 and Number2 is equal to 1 when you program loads. Your program will also be three bytes longer.

If you declare a variable as a word you cannot move the value of this variable into a 8-bit register and you can't declare a variable as a byte and move the value into a 16-bit register. For examples:

```
mov al,Number1      ; ok
```

```
mov ax,Number1     ; error
```

```
mov bx,Number2     ; ok
```

```
mov bl,Number2    ; error
```

All you have to remember is that you can only put bytes into 8-bit registers and words into 16-bit registers.

### **Your first assembly program**

Now that you know some basic instructions and a little about data it is time that we looked at a full assembly program which can be compiled.

Listing 1: 1STPROG.ASM

```
; This is a simple program which displays "Hello World!"
```

; on the screen.

.model small

.stack

.data

Message db "Hello World!\$" ; message to be display

.code

start:

mov dx,OFFSET Message ; offset of Message is in DX

mov ax,SEG Message ; segment of Message is in AX

mov ds,ax ; DS:DX points to string

mov ah,9 ; function 9 - display string

int 21h ; call dos service

mov ax,4c00h ; return to dos DOS

int 21h

END start ;end here

## Compilation Instructions

These are some instructions to compile and link programs. If you have a compiler other than TASM or A86 then see your instruction manual.

## Turbo Assembler

tasm file.asm

tlink file [/t]

The /t switch makes a .COM file. This will only work if the memory model is declared as tiny in the source file.

## A86

a86 file.asm

This will compile your program to a .COM file. It doesn't matter what the memory model is.

### **Lesson 3: Making things easier**

The way we entered the address of the message we wanted to print was a bit cumbersome. It took three lines and it isn't the easiest thing to remember.

```
mov dx,OFFSET MyMessage
mov ax,SEG MyMessage
mov ds,ax
```

We can replace all this with just one line. This makes the code easier to read and it easier to remember.

```
mov dx,OFFSET MyMessage
```

To make this work at the beginning of your code add these lines:

```
mov ax,@data
mov ds,ax
```

Note: for A86 you need to change the first line to:

```
mov ax,data
```

This is because all the data in the segment has the same SEG value. Putting this in DS saves us reloading this every time we want to use another thing in the same segment.

### **Keyboard Input**

We are going to use interrupt 16h, function 00h to read the keyboard. this gets a key from the keyboard buffer. If there isn't one, it waits until there is. It returns the SCAN code in AH and the ASCII translation in AL.

```
xor ah,ah ; function 00h - get character
int 16h ; interrupt 16h
```

All we need to worry about for now is the ascii value which is in AL.

Note: XOR performs a Boolean Exclusive OR. It is commonly used to erase a register or variable.

## Printing a Character

The problem is that we have the key that has been pressed in ah. How do we display it? We can't use function 9h because for that we need to have already defined the string which has to end with a dollar sign. this is what we do instead:

; after calling function 00h of interrupt 16h

```
mov dl,al ; move al (ascii code) into dl
```

```
mov ah,02h ; function 02h of interrupt 21h
```

```
int 21h ; call interrupt 21h
```

If you want to save the value of AH then push AX before and pop it afterwards.

## Control Flow

In assembly there is a set of commands for control flow like in any other language. Firstly the most basic command:

```
jmp label
```

All this does it to move to the label specified and start executing the code there. For example:

```
jmp ALabel
```

```
.  
.br/>.
```

```
ALabel:
```

What do we do if we want to compare something? We have just got a key from the user but we want to do something with it. Lets print something out if it is equal to something else. How do we do that? It is easy. We use the jump on condition commands.

Firstly you compare the operand with the data and then use the correct command.

```
cmp ax,3 ; is AX = 3?
```

```
je correct ; yes
```

Here is a list of them:

### Jump on Condition Instructions

JAJumps if the first number was above the second number	
JA	same as above, but will also jump if they are equal
JB	jumps if the first number was below the second
JBE	Same as above, but will also jump if they are equal
JNA	jumps if the first number was NOT above (JBE)
JNAE	jumps if TDe first number was NOT above or TDe same as (JNB)
JNB	jumps if the first number was NOT below (JAE)
JNBE	jumps if the first number was NOT below or the same as (JA)
JZ	jumps if the two numbers were equal
JE	same as JZ, just a different name
JNZ	jumps if the two numbers are NOT equal
JNE	same as above
JC	jump if carry flag is set

Note: the jump can only be a maximum of 127 bytes in either direction.

**CMP:** compare a value

Syntax:

CMP register or variable, value

jxx destination

An example of this is:

```
cmp al,'Y' ; compare the value in al with Y
```

```
je ItsYES ; if it is equal then jump to ItsYES
```

Every instruction takes up a certain amount of code space. You will get a warning if you try and jump over 127 bytes in either direction from the compiler. You can solve this by changing a sequence like this:

```
cmp ax,10      ; is AX 10?
```

```
je done      ; yes, lets finish
```

to something like this:

```
cmp ax,10      ; is AX 10?
```

```
jne notdone    ; no it is not
```

```
jmp done      ; we are now done
```

notdone:

This solves the problem but you may want to think about reordering your code or using procedures if this happens often.

Now we are going to look at a program which demonstrates input, output and control flow.

Listing 2: PROGFLOW.ASM

; a program to demonstrate program flow and input/output

```
.model tiny
```

```
.code
```

```
org 100h
```

```
start:
```

```
mov dx,OFFSET Message      ; display a message on the screen
```

```
mov ah,9                    ; using function 09h
```

```
int 21h                      ; of interrupt 21h
```

```
mov dx,OFFSET Prompt       ; display a message on the screen
```

```
mov ah,9                    ; using function 09h
```

```
int 21h                      ; of interrupt 21h
```

```
jmp First_Time
```

```
Prompt_Again:
```

```
mov dx,OFFSET Another      ; display a message on the screen
```

```
mov ah,9                    ; using function 09h
```

```
int 21h                      ; of interrupt 21h
```

First\_Time:

mov dx,OFFSET Again ; display a message on the screen

mov ah,9 ; using function 09h

int 21h ; of interrupt 21h

xor ah,ah ; function 00h of

int 16h ; interrupt 16h gets a character

mov bl,al ; save to bl

mov dl,al ; move al to dl

mov ah,02h ; function 02h - display character

int 21h ; call DOS service

cmp bl,'Y' ; is al=Y?

je Prompt\_Again ; if yes then display it again

cmp bl,'y' ; is al=y?

je Prompt\_Again ; if yes then display it again

theEnd:

mov dx,OFFSET GoodBye ; print goodbye message

mov ah,9 ; using function 9

int 21h ; of interrupt 21h

mov ah,4Ch ; terminate program

int 21h

.DATA

CR equ 13 ; enter

LF equ 10 ; line-feed

Message DB "A Simple Input/Output Program\$"

Prompt DB CR,LF,"Here is your first prompt.\$"

Again DB CR,LF,"Do you want to be prompted again? \$"

Another DB CR,LF,"Here is another prompt!\$"

GoodBye DB CR,LF,"Goodbye then."

end start

#### **Lesson 4: Some instructions that you need to know**

This is just a list of some basic assembly instructions that are very important and are used often.

**ADD:** Add the contents of one number to another

Syntax:

ADD operand1,operand2

This adds operand2 to operand1. The answer is stored in operand1. Immediate data cannot be used as operand1 but can be used as operand2.

**SUB:** Subtract one number from another

Syntax:

SUB operand1,operand2

This subtracts operand2 from operand1. Immediate data cannot be used as operand1 but can be used as operand2.

**MUL:** Multiplies two unsigned integers (always positive)

**IMUL:** Multiplies two signed integers (either positive or negative)

Syntax:

MUL register or variable

IMUL register or variable

This multiplies AL or AX by the register or variable given. AL is multiplied if a byte sized operand is given and the result is stored in AX. If the operand is word sized AX is multiplied and the result is placed in DX:AX.

On a 386, 486 or Pentium the EAX register can be used and the answer is stored in EDX:EAX.

**DIV:** Divides two unsigned integers (always positive)

**IDIV:** Divides two signed integers (either positive or negative)

Syntax:

DIV register or variable

IDIV register or variable

This works in the same way as MUL and IMUL by dividing the number in AX by the register or variable given. The answer is stored in two places. AL stores the answer and the remainder is in AH. If the operand is a 16 bit register than the number in DX:AX is divided by the operand and the answer is stored in AX and remainder in DX.

## Introduction to Procedures

In assembly a procedure is the equivalent to a function in C or Pascal. A procedure provides a easy way to encapsulate some calculation which can then be used without worrying how it works. With procedures that are properly designed you can ignore how a job is done.

This is how a procedure is defined:

```
PROC AProcedure
```

```
.
```

```
.           ; some code to do something
```

```
.
```

```
ret           ; if this is not here then your computer will crash
```

```
ENDP AProcedure
```

It is equally easy to run a procedure all you need to do is this:

```
call AProcedure
```

This next program is an example of how to use a procedure. It is like the first example we looked at, all it does is print "Hello World!" on the screen.

Listing 3: SIMPPROC.ASM

```
; This is a simple program to demonstrate procedures. It should
```

```
; print Hello World! on the screen when ran.
```

```
.model tiny
```

```
.code
```

```
org 100h
```

Start:

```
call Display_Hi      ; Call the procedure
mov ax,4C00h        ; return to DOS
int 21h
```

Display\_Hi PROC

```
mov dx,OFFSET HI
mov ah,9
int 21h
```

```
ret
```

Display\_Hi ENDP

```
HI DB "Hello World!" ; define a message
```

```
end Start
```

### **Procedures that pass parameters**

Procedures wouldn't be so useful unless you could pass parameters to modify or use inside the procedure. There are three ways of doing this and I will cover all three methods: in registers, in memory and in the stack.

There are three example programs which all accomplish the same task. They print a square block (ASCII value 254) in a specified place.

The sizes of the files when compiled are: 38 for register, 69 for memory and 52 for stack (in bytes).

#### **In registers**

The advantages of this is that it is easy to do and is fast. All you have to do is to move the parameters into registers before calling the procedure.

Listing 4: PROC1.ASM

```
; this a procedure to print a block on the screen using
; registers to pass parameters (cursor position of where to
```

; print it and colour).

.model tiny

.code

org 100h

Start:

mov dh,4 ; row to print character on

mov dl,5 ; column to print character on

mov al,254 ; ascii value of block to display

mov bl,4 ; colour to display character

call PrintChar ; print our character

mov ax,4C00h ; terminate program

int 21h

PrintChar PROC NEAR

push bx ; save registers to be destroyed

push cx

xor bh,bh ; clear bh - video page 0

mov ah,2 ; function 2 - move cursor

int 10h ; row and col are already in dx

pop bx ; restore bx

xor bh,bh ; display page - 0

mov ah,9 ; function 09h write char & attrib

mov cx,1 ; display it once

int 10h ; call bios service

pop cx ; restore registers

ret ; return to where it was called

```
PrintChar ENDP
```

```
end Start
```

### Passing through memory

The advantages of this method is that it is easy to do but it makes your program larger and can be slower.

To pass parameters through memory all you need to do is copy them to a variable which is stored in memory. You can use a variable in the same way that you can use a register but commands with registers are a lot faster.

Listing 5: PROC2.ASM

```
; this a procedure to print a block on the screen using memory  
; to pass parameters (cursor position of where to print it and  
; colour).
```

```
.model tiny
```

```
.code
```

```
org 100h
```

```
Start:
```

```
mov Row,4      ; row to print character  
mov Col,5      ; column to print character on  
mov Char,254   ; ascii value of block to display  
mov Colour,4   ; colour to display character
```

```
call PrintChar ; print our character  
mov ax,4C00h   ; terminate program  
int 21h
```

```
PrintChar PROC NEAR
```

```
push ax cx bx  ; save registers to be destroyed
```

```
xor bh,bh ; clear bh - video page 0
```

```
mov ah,2 ; function 2 - move cursor
```

```
mov dh,Row
```

```
mov dl,Col
```

```
int 10h ; call Bios service
```

```
mov al,Char
```

```
mov bl,Colour
```

```
xor bh,bh ; display page - 0
```

```
mov ah,9 ; function 09h write char & attrib
```

```
mov cx,1 ; display it once
```

```
int 10h ; call bios service
```

```
pop bx cx ax ; restore registers
```

```
ret ; return to where it was called
```

```
PrintChar ENDP
```

```
; variables to store data
```

```
Row db ?
```

```
Col db ?
```

```
Colour db ?
```

```
Char db ?
```

```
end Start
```

### **Passing through Stack**

This is the most powerful and flexible method of passing parameters the problem is that it is more complicated.

Listing 6: PROC3.ASM

```
; this a procedure to print a block on the screen using the  
; stack to pass parameters (cursor position of where to print it  
; and colour).
```

```
.model tiny
```

```
.code
org 100h
```

Start:

```
mov dh,4 ; row to print string on
mov dl,5 ; column to print string on
mov al,254 ; ascii value of block to display
mov bl,4 ; colour to display character
push dx ax bx ; put parameters onto the stack
```

```
call PrintString ; print our string
```

```
pop bx ax dx ;restore registers
mov ax,4C00h ;terminate program
int 21h
```

```
PrintString PROC NEAR
```

```
push bp ; save bp
mov bp,sp ; put sp into bp
push cx ; save registers to be destroyed
```

```
xor bh,bh ; clear bh - video page 0
mov ah,2 ; function 2 - move cursor
mov dx,[bp+8] ; restore dx
int 10h ; call bios service
```

```
mov ax,[bp+6] ; character
mov bx,[bp+4] ; attribute
xor bh,bh ; display page - 0
mov ah,9 ; function 09h write char & attrib
mov cx,1 ; display it once
int 10h ; call bios service
```

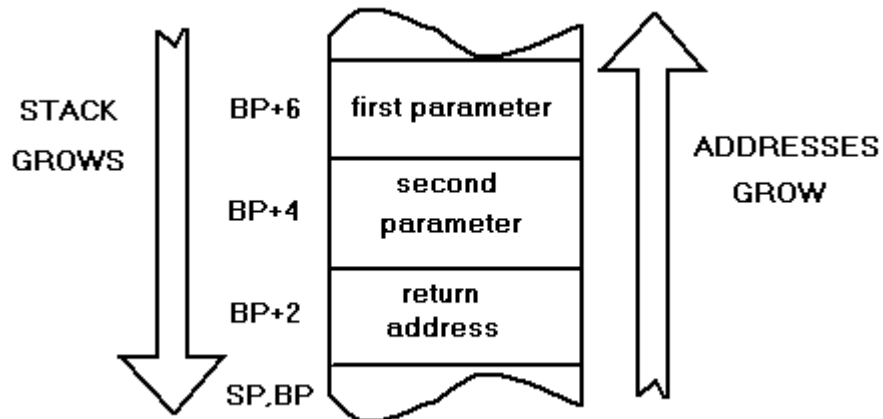
```
pop cx ; restore registers
```

pop bp

ret ; return to where it was called

PrintString ENDP

end Start



*This shows the stack for a procedure with two parameters*

To get a parameter from the stack all you need to do is work out where it is. The last parameter is at BP+2 and then the next and BP+4.

### What are "Memory Models"?

We have been using the .MODEL directive to specify what type of memory model we use, but what does this mean?

Syntax:

.MODEL MemoryModel

Where MemoryModel can be SMALL, COMPACT, MEDIUM, LARGE, HUGE, TINY or FLAT.

#### Tiny

This means that there is only one segment for both code and data. This type of program can be a .COM file.

#### Small

This means that by default all code is placed in one segment and all data declared in the data segment is also placed in one segment. This means that all procedures and variables are addressed as NEAR by pointing at offsets only.

#### Compact

This means that by default all elements of code are placed in one segment but each element of data can be placed in its own physical segment. This means that data elements are addressed by pointing at both at the segment and offset addresses. Code elements (procedures) are NEAR and variables are FAR.

### **Medium**

This is the opposite to compact. Data elements are NEAR and procedures are FAR.

### **Large**

This means that both procedures and variables are FAR. You have to point at both the segment and offset addresses.

### **Flat**

This isn't used much as it is for 32 bit unsegmented memory space. For this you need a DOS extender. This is what you would have to use if you were writing a program to interface with a C/C++ program that used a DOS extender such as DOS4GW or PharLap.

### **Macros (in Turbo Assembler)**

*Note:* All code examples given are for macros in Turbo Assembler.

Macros are very useful for doing something that is done often but for which a procedure can't be used. Macros are substituted when the program is compiled to the code which they contain.

This is the syntax for defining a macro:

```
Name_of_macro macro  
;  
;      a sequence of instructions  
;  
endm
```

These two examples are for macros that take away the boring job of pushing and popping certain registers:

SaveRegs macro

```
push ax  
push bx  
push cx  
push dx
```

```
endm
```

RestoreRegs macro

pop dx

pop cx

pop bx

pop ax

endm

Notice that the registers are popped in the reverse order to they were pushed. To use a macro in you program you just use the name of the macro as an ordinary instruction:

SaveRegs

; some other instructions

RestoreRegs

This example shows how you can use a macro to save typing in. This macro simply prints out a message to the screen.

OutMsg macro SomeText

local PrintMe,SkipData

jmp SkipData

PrintMe db SomeText,'\$'

SkipData:

push ax dx cs

mov dx,OFFSET cs:PrintMe

mov ah,9

int 21h

pop cs dx ax

endm

The only problems with macros is that if you overuse them it leads to you program getting bigger and bigger and that you have problems with multiple definition of labels and variables. The correct way to solve this problem is to use the LOCAL directive for declaring names inside macros.

Syntax:

LOCAL name

Where name is the name of a local variable or label.

### **Macros with parameters**

Another useful property of macros is that they can have parameters. The number of parameters is only restricted by the length of the line. Syntax:

```
Name_of_Macro macro par1,par2,par3
```

```
;
```

```
; commands go here
```

```
;
```

```
endm
```

This is an example that adds the first and second parameters and puts the result in the third:

```
AddMacro macro num1,num2,result
```

```
push ax ; save ax from being destroyed
```

```
mov ax,num1 ; put num1 into ax
```

```
add ax,num2 ; add num2 to it
```

```
mov result,ax ; move answer into result
```

```
pop ax ; restore ax
```

```
endm
```

### **Lesson 5: Files and how to use them**

Files can be opened, read and written to. DOS has some ways of doing this which save us the trouble of writing our own routines. Yes, more interrupts. Here is a list of helpful functions of interrupt 21h that deal with files.

Note: Bits are numbered from right to left.

### **Function 3Dh: open file**

Opens an existing file for reading, writing or appending on the specified drive and filename.

#### **INPUT:**

AH = 3Dh

AL = bits 0-2 Access mode

000 = read only

001 = write only

010 = read/write

bits 4-6 Sharing mode (DOS 3+)

000 = compatibility mode

001 = deny all

010 = deny write

011 = deny read

100 = deny none

DS:DX = segment:offset of ASCIIZ pathname

#### **OUTPUT:**

CF = 0 function is successful

AX = handle

CF = 1 error has occurred

AX = error code

01h missing file sharing software

02h file not found

03h path not found or file does not exist

04h no handle available

05h access denied

0Ch access mode not permitted

What does ASCIIZ mean? An ASCIIZ string like a ASCII string with a zero on the end instead of a dollar sign.

Important: Remember to save the file handle it is needed for later.

### **How to save the file handle**

It is important to save the file handle because this is needed to do anything with the file. Well how is this done? There are two methods we could use: copy the file handle into another register and don't use that register or copy it to a variable in memory.

The disadvantages with the first method is that you will have to remember not to use the register you saved it in and it wastes a register that can be used for something more useful. We are going to use the second. This is how it is done:

```
FileHandle DW 0 ; use this for saving the file handle
```

```
.  
.br/>.
```

```
mov FileHandle,ax ; save the file handle
```

### **Function 3Eh: close file**

Closes a file that has been opened.

#### **INPUT:**

AX = 3Eh

BX = file handle

#### **OUTPUT:**

CF = 0 function is successful

AX = destroyed

CF = 1 function not successful

AX = error code - 06h file not opened or unauthorised handle.

Important: Don't call this function with a zero handle because that will close the standard input (the keyboard) and you won't be able to enter anything.

### **Function 3Fh: read file/device**

Reads bytes from a file or device to a buffer.

#### **INPUT:**

AH = 3Fh

BX = handle

CX = number of bytes to be read

DS:DX = segment:offset of a buffer

### OUTPUT:

CF = 0 function is successful

AX = number of bytes read

CF = 1 an error has occurred

05h access denied

06h illegal handle or file not opened

If CF = 0 and AX = 0 then the file pointer was already at the end of the file and no more can be read. If CF = 0 and AX is smaller than CX then only part was read because the end of the file was reached or an error occurred.

This function can also be used to get input from the keyboard. Use a handle of 0, and it stops reading after the first carriage return, or once a specified number of characters have been read. This is a good and easy method to use to only let the user enter a certain amount of characters.

### Listing 7: READFILE.ASM

; a program to demonstrate creating a file and then reading from

; it

.model small

.stack

.code

mov ax,@data ; base address of data segment

mov ds,ax ; put this in ds

mov dx,OFFSET FileName ; put address of filename in dx

mov al,2 ; access mode - read and write

mov ah,3Dh ; function 3Dh -open a file

int 21h ; call DOS service

mov Handle,ax ; save file handle for later

jc ErrorOpening ; jump if carry flag set - error!

```
mov dx,offset Buffer      ; address of buffer in dx
mov bx,Handle             ; handle in bx
mov cx,100               ; amount of bytes to be read
mov ah,3Fh               ; function 3Fh - read from file
int 21h                  ; call dos service
```

```
jc ErrorReading          ; jump if carry flag set - error!
```

```
mov bx,Handle            ; put file handle in bx
mov ah,3Eh               ; function 3Eh - close a file
int 21h                  ; call DOS service
```

```
mov cx,100               ; length of string
mov si,OFFSET Buffer      ; DS:SI - address of string
xor bh,bh                ; video page - 0
mov ah,0Eh               ; function 0Eh - write character
```

NextChar:

```
lodsb                    ; AL = next character in string
int 10h                  ; call BIOS service
loop NextChar
```

```
mov ax,4C00h             ; terminate program
int 21h
```

ErrorOpening:

```
mov dx,offset OpenError ; display an error
mov ah,09h               ; using function 09h
int 21h                  ; call DOS service
mov ax,4C01h             ; end program with an errorlevel =1
int 21h
```

ErrorReading:

```
mov dx,offset ReadError ; display an error
mov ah,09h                ; using function 09h
int 21h                   ; call DOS service
mov ax,4C02h              ; end program with an errorlevel =2
int 21h
```

```
.data
Handle DW ?                ; to store file handle
FileName DB "C:\test.txt",0 ; file to be opened
OpenError DB "An error has occurred(opening)!$"
ReadError DB "An error has occurred(reading)!$"
Buffer DB 100 dup (?)      ; buffer to store data

END
```

### Function 3Ch: Create file

Creates a new empty file on a specified drive with a specified pathname.

#### INPUT:

AH = 3Ch

CX = file attribute

bit 0 = 1 read-only file

bit 1 = 1 hidden file

bit 2 = 1 system file

bit 3 = 1 volume (ignored)

bit 4 = 1 reserved (0) - directory

bit 5 = 1 archive bit

bits 6-15 reserved (0)

DS:DX = segment:offset of ASCII pathname

#### OUTPUT:

CF = 0 function is successful

AX = handle

CF = 1 an error has occurred

03h path not found

04h no available handle

05h access denied

Important: If a file of the same name exists then it will be lost. Make sure that there is no file of the same name. This can be done with the function below.

### **Function 4Eh: find first matching file**

Searches for the first file that matches the filename given.

#### **INPUT:**

AH = 4Eh CX = file attribute mask (bits can be combined)

bit 0 = 1 read only

bit 1 = 1 hidden

bit 2 = 1 system

bit 3 = 1 volume label

bit 4 = 1 directory

bit 5 = 1 archive

bit 6-15 reserved

DS:DX = segment:offset of ASCIIZ pathname

#### **OUTPUT:**

CF = 0 function is successful

[DTA] Disk Transfer Area = FindFirst data block

#### **The DTA block:**

##### **Offset Size in bytes Meaning**

0	21	Reserved
21	1	File attributes
22	2	Time last modified
24	2	Date last modified
26	4	Size of file (in bytes)
30	13	File name (ASCIIZ)

An example of checking if file exists:

```
File DB "C:\file.txt",0          ; name of file that we want
mov dx,OFFSET File              ; address of filename
mov cx,3Fh                      ; file mask 3Fh - any file
mov ah,4Eh                      ; function 4Eh - find first file
int 21h                         ; call DOS service
```

```
jc NoFile
```

```
; print message saying file exists
```

```
NoFile:
```

```
; continue with creating file
```

This is an example of creating a file and then writing to it.

Listing 8: CREATE.ASM

```
; This example program creates a file and then writes to it.
```

```
.model small
```

```
.stack
```

```
.code
```

```
mov ax,@data                    ; base address of data segment
```

```
mov ds,ax                      ; put it in ds
```

```
mov dx,offset StartMessage
```

```
mov ah,09h
```

```
int 21h
```

```
mov dx,offset FileName         ; put offset of filename in dx
```

```
xor cx,cx                     ; clear cx - make ordinary file
```

```
mov ah,3Ch                    ; function 3Ch - create a file
```

```
int 21h                       ; call DOS service
```

```
jc CreateError                ; jump if there is an error
```

```
mov dx,offset FileName         ; put offset of filename in dx
```

```

mov al,2          ; access mode -read and write
mov ah,3Dh        ; function 3Dh - open the file
int 21h          ; call dos service

jc OpenError     ; jump if there is an error
mov Handle,ax    ; save value of handle

mov dx,offset WriteMe ; address of information to write
mov bx,Handle    ; file handle for file
mov cx,38        ; 38 bytes to be written
mov ah,40h       ; function 40h - write to file
int 21h          ; call dos service

jc WriteError    ; jump if there is an error
cmp ax,cx        ; was all the data written?
jne WriteError   ; no it wasn't - error!

mov bx,Handle    ; put file handle in bx
mov ah,3Eh       ; function 3Eh - close a file
int 21h          ; call dos service

mov dx,offset EndMessage
mov ah,09h
int 21h

ReturnToDOS:

mov ax,4C00h     ; terminate program
int 21h

WriteError:
mov dx,offset WriteMessage
jmp EndError

OpenError:
mov dx,offset OpenMessage

```

```
jmp EndError
```

```
CreateError:
```

```
mov dx,offset CreateMessage
```

```
EndError:
```

```
mov ah,09h
```

```
int 21h
```

```
mov ax,4C01h
```

```
int 21h
```

```
.data
```

```
CR equ 13
```

```
LF equ 10
```

```
StartMessage DB "This program creates a file called NEW.TXT"
```

```
DB ,"on the C drive.$"
```

```
EndMessage DB CR,LF,"File create OK, look at file to"
```

```
DB ,"be sure.$"
```

```
WriteMessage DB "An error has occurred (WRITING)$"
```

```
OpenMessage DB "An error has occurred (OPENING)$"
```

```
CreateMessage DB "An error has occurred (CREATING)$"
```

```
WriteMe DB "HELLO, THIS IS A TEST, HAS IT WORKED?",0
```

```
FileName DB "C:\new.txt",0 ; name of file to open
```

```
Handle DW ? ; to store file handle
```

```
END
```

This is an example of how to delete a file after checking it exists:

Listing 9: DELFILE.ASM

```
; a demonstration of how to delete a file. The file new.txt on  
; c: is deleted (this file is created by create.exe). We also  
; check if the file exists before trying to delete it
```

.model small

.stack

.data

CR equ 13

LF equ 10

File db "C:\new.txt",0

Deleted db "Deleted file c:\new.txt\$"

NoFile db "c:\new.txt doesn't exists - exiting\$"

ErrDel db "Can't delete file - probably write protected\$"

.code

mov ax,@data

mov ds,ax

mov dx,OFFSET File ; address of filename to look for

mov cx,3Fh ; file mask 3Fh - any file

mov ah,4Eh ; function 4Eh - find first file

int 21h ; call dos service

jc FileDontExist

mov dx,OFFSET File ; DS:DX points to file to be killed

mov ah,41h ; function 41h - delete file

int 21h ; call DOS service

jc ErrorDeleting ; jump if there was an error

mov dx,OFFSET Deleted ; display message

jmp Endit

ErrorDeleting:

mov dx,OFFSET ErrDel

jmp Endit

FileDontExist:

```
mov dx,OFFSET NoFile
```

EndIt:

```
mov ah,9
```

```
int 21h
```

```
mov ax,4C00h ; terminate program and exit to DOS
```

```
int 21h ; call DOS service
```

```
end
```

## Using the FindFirst and FindNext Functions

Listing 10: DIRC.ASM

```
; this program demonstrates how to look for files. It prints
```

```
; out the names of all the files in the c:\drive and names of
```

```
; the sub-directories
```

```
.model small
```

```
.stack
```

```
.data
```

```
FileName db "c:\*.*",0 ;file name
```

```
DTA db 128 dup(?) ;buffer to store the DTA
```

```
ErrorMsg db "An Error has occurred - exiting.$"
```

```
.code
```

```
mov ax,@data ; set up ds to be equal to the
```

```
mov ds,a ; data segment
```

```
mov es,ax ; also es
```

```
mov dx,OFFSET DTA ; DS:DX points to DTA
```

```
mov ah,1AH ; function 1Ah - set DTA
```

```
int 21h ; call DOS service
```

```
mov cx,3Fh          ; attribute mask - all files
mov dx,OFFSET FileName ; DS:DX points ASCIZ filename
mov ah,4Eh         ; function 4Eh - find first
int 21h           ; call DOS service
```

```
jc error ; jump if carry flag is set
```

LoopCycle:

```
mov dx,OFFSET FileName ; DS:DX points to file name
mov ah,4Fh            ; function 4Fh - find next
int 21h              ; call DOS service
```

```
jc exit ; exit if carry flag is set
```

```
mov cx,13          ; length of filename
mov si,OFFSET DTA+30 ; DS:SI points to filename in DTA
xor bh,bh         ; video page - 0
mov ah,0Eh        ; function 0Eh - write character
```

NextChar:

```
lodsb ; AL = next character in string
int 10h ; call BIOS service
```

loop NextChar

```
mov di,OFFSET DTA+30 ; ES:DI points to DTA
mov cx,13            ; length of filename
xor al,al           ; fill with zeros
rep stosb           ; erase DTA
```

```
jmp LoopCycle ; continue searching
```

error:

```
mov dx,OFFSET ErrorMessage    ; display error message
```

```
mov ah,9
```

```
int 21h
```

```
exit:
```

```
mov ax,4C00h    ; exit to DOS
```

```
int 21h
```

```
end
```

## Lesson 6: String Instructions

In assembly there are some very useful instructions for dealing with strings. Here is a list of the instructions and the syntax for using them:

**MOV\*** Move String: moves byte, word or double word at DS:SI to ES:DI

Syntax:

```
movsb    ; move byte
```

```
movsw    ; move word
```

```
movsd    ; move double word
```

**CMPS\*** Compare string: compares byte, word or double word at DS:SI to ES:DI

Syntax:

```
cmpsb    ; compare byte
```

```
cmpsw    ; compare word
```

```
cmpsd    ; compare double word
```

Note: This instruction is normally used with the REP prefix.

**SCAS\*** Search string: search for AL, AX, or EAX in string at ES:DI

Syntax:

```
scasb    ; search for AL
```

```
scasw    ; search for AX
```

```
scasd    ; search for EAX
```

Note: This instruction is usually used with the REPZ or REPNZ prefix.

**REP** Prefix for string instruction repeats instruction CX times

Syntax:

```
rep StringInstruction
```

**STOS\*** Move byte, word or double word from AL, AX or EAX to ES:DI

Syntax:

```
stosb    ; move AL into ES:DI
```

```
stosw    ; move AX into ES:DI
```

```
stosd    ; move EAX into ES:DI
```

**LODS\*** Move byte, word or double word from DS:SI to AL, AX or EAX

Syntax:

```
lodsb    ; move ES:DI into AL
```

```
lodsw    ; move ES:DI into AX
```

```
lodsd    ; move ES:DI into EAX
```

The next example demonstrates how to use these instructions.

Listing 11: STRINGS.ASM

```
.model small
.stack
.code

mov ax,@data          ; ax points to of data segment
mov ds,ax             ; put it into ds
mov es,ax             ; put it in es too
mov ah,9              ; function 9 - display string
mov dx,OFFSET Message1 ; ds:dx points to message
int 21h               ; call dos function

cld                   ; clear direction flag
mov si,OFFSET String1 ; make ds:si point to String1
mov di,OFFSET String2 ; make es:di point to String2
mov cx,18              ; length of strings
rep movsb             ; copy string1 into string2
```

```
mov ah,9          ; function 9 - display string
mov dx,OFFSET Message2    ; ds:dx points to message
int 21h          ; call dos function
```

```
mov dx,OFFSET String1     ; display String1
int 21h          ; call DOS service
```

```
mov dx,OFFSET Message3    ; ds:dx points to message
int 21h          ; call dos function
```

```
mov dx,OFFSET String2     ; display String2
int 21h          ; call DOS service
```

```
mov si,OFFSET Diff1 ; make ds:si point to Diff1
mov di,OFFSET Diff2     ; make es:di point to Diff2
mov cx,39                ; length of strings
repz cmpsb              ; compare strings
jnz Not_Equal           ; jump if they are not the same
```

```
mov ah,9          ; function 9 - display string
mov dx,OFFSET Message4    ; ds:dx points to message
int 21h          ; call dos function
```

```
jmp Next_Operation
```

```
Not_Equal:
```

```
mov ah,9          ; function 9 - display string
mov dx,OFFSET Message5 ; ds:dx points to message
int 21h          ; call dos function
```

```
Next_Operation:
```

```
mov di,OFFSET SearchString ; make es:di point to string
mov cx,36                  ; length of string
mov al,'H'                 ; character to search for
repne scasb                ; find first match
jnz Not_Found
```

```
mov ah,9          ; function 9 - display string
mov dx,OFFSET Message6    ; ds:dx points to message
int 21h          ; call dos function
jmp Lodsb_Example
```

Not\_Found:

```
mov ah,9          ; function 9 - display string
mov dx,OFFSET Message7    ; ds:dx points to message
int 21h          ; call dos function
```

Lodsb\_Example:

```
mov ah,9          ; function 9 - display string
mov dx,OFFSET NewLine    ; ds:dx points to message
int 21h          ; call dos function
```

```
mov cx,17          ; length of string
mov si,OFFSET Message    ; DS:SI - address of string
xor bh,bh          ; video page - 0
mov ah,0Eh         ; function 0Eh - write character
```

NextChar:

```
lodsb              ; AL = next character in string
int 10h           ; call BIOS service
loop NextChar
```

```
mov ax,4C00h      ; return to DOS
int 21h
```

.data

CR equ 13

LF equ 10

NewLine db CR,LF,"\$"

String1 db "This is a string!\$"

String2 db 18 dup(0)

```

Diff1 db "This string is nearly the same as Diff2$"
Diff2 db "This string is nearly the same as Diff1$"
Equal1 db "The strings are equal$"
Equal2 db "The strings are not equal$"
Message db "This is a message"
SearchString db "1293ijdkfjiu938uHello983fjksj98934$"

Message1 db "Using String instructions example program.$"
Message2 db CR,LF,"String1 is now: $"
Message3 db CR,LF,"String2 is now: $"
Message4 db CR,LF,"Strings are equal!$"
Message5 db CR,LF,"Strings are not equal!$"
Message6 db CR,LF,"Character was found.$"
Message7 db CR,LF,"Character was not found.$"

end

```

### **How to find out the DOS Version**

In many programs it is necessary to find out what the DOS version is. This could be because you are using a DOS function that needs the revision to be over a certain level.

Firstly this method simply finds out what the version is.

```

mov ah,30h      ; function 30h - get MS-DOS version
int 21h        ; call DOS function

```

This function returns the major version number in AL and the minor version number in AH. For example if it was version 4.01, AL would be 4 and AH would be 01. The problem is that if on DOS 5 and higher

SETVER can change the version that is returned. The way to get round this is to use this method.

```

mov ah,33h      ; function 33h - actual DOS version
mov al,06h      ; subfunction 06h
int 21h        ; call interrupt 21h

```

This will only work on DOS version 5 and above so you need to check using the former method. This will return the actual version of DOS even if SETVER has changed the version. This returns the major version in BL and the minor version in BH.

### **Multiple Pushes and Pops**

You can push and pop more than one register on a line in TASM and A86. This makes your code easier to understand.

```
push ax bx cx dx    ; save registers
```

```
pop dx cx bx ax     ; restore registers
```

When TASM (or A86) compiles these lines it translates it into separate pushes and pops. This way just saves you time typing and makes it easier to understand.

Note: To make these lines compile in A86 you need to put commas (,) in between the registers.

### **The PUSH/PUSHAD and POPA/POPAD Instructions**

PUSHA is a useful instruction that pushes all general purpose registers onto the stack. It accomplishes the same as the following:

```
temp = SP
```

```
push ax
```

```
push cx
```

```
push dx
```

```
push bx
```

```
push temp
```

```
push bp
```

```
push si
```

```
push di
```

The main advantage is that it is less typing, a smaller instruction and it is a lot faster. POPA does the reverse and pops these registers off the stack. PUSHAD and POPAD do the same but with the 32-bit registers ESP, EAX, ECX, EDX, EBX, EBP, ESI and EDI.

### **Using Shifts for faster Multiplication and Division**

Using MUL's and DIV's is very slow and should be only used when speed is not needed. For faster multiplication and division you can shift numbers left or right one or more binary positions. Each shift is to a power of 2. This is the same as the << and >> operators in C.

There are four different ways of shifting numbers either left or right one binary position.

**SHL** Unsigned multiple by two

**SHR** Unsigned divide by two

**SAR** Signed divide by two

**SAL** same as SHL

The syntax for all four is the same:

SHL operand1,operand2

Note: The 8086 cannot have the value of operand2 other than 1. The 286/386 cannot have operand2 higher than 31.

## Loops

Using Loop is a better way of making a loop than using JMP's. You place the amount of times you want it to loop in the CX register and every time it reaches the loop statement it decrements CX (CX-1) and then does a short jump to the label indicated. A short jump means that it can only 128 bytes before or 127 bytes after the LOOP instruction.

Syntax:

```
mov cx,100 ; 100 times to loop
```

Label:

```
.  
. .  
. .
```

```
Loop Label: ; decrement CX and loop to Label
```

This is exactly the same as the following piece of code without using loop:

```
mov cx,100 ; 100 times to loop
```

Label:

```
dec cx ; CX = CX-1
```

```
jnz Label ; continue until done
```

Which do you think is easier to understand? Using DEC/JNZ is faster on 486's and above and it is useful as you don't have to use CX.

This works because JNZ will jump if the zero flag has not been set. Setting CX to 0 will set this flag.

## How to use a debugger

This is a good time to use a debugger to find out what your program is actually doing. I am going to demonstrate how to use Turbo Debugger to check what the program is actually doing. First we need a program which we can look at.

#### Listing 12: DEBUG.ASM

; example program to demonstrate how to use a debugger

```
.model tiny
```

```
.code
```

```
org 100h
```

```
start:
```

```
push ax ; save value of ax
```

```
push bx ; save value of bx
```

```
push cx ; save value of cx
```

```
mov ax,10 ; first parameter is 10
```

```
mov bx,20 ; second parameter is 20
```

```
mov cx,3 ; third parameter is 3
```

Call ChangeNumbers

```
pop cx ; restore cx
```

```
pop bx ; restore bx
```

```
pop ax ; restore dx
```

```
mov ax,4C00h ; exit to dos
```

```
int 21h
```

ChangeNumbers PROC

```
add ax,bx ; adds number in bx to ax
```

```
mul cx ; multiply ax by cx
```

```
mov dx,ax ; return answer in dx
```

```
ret
```

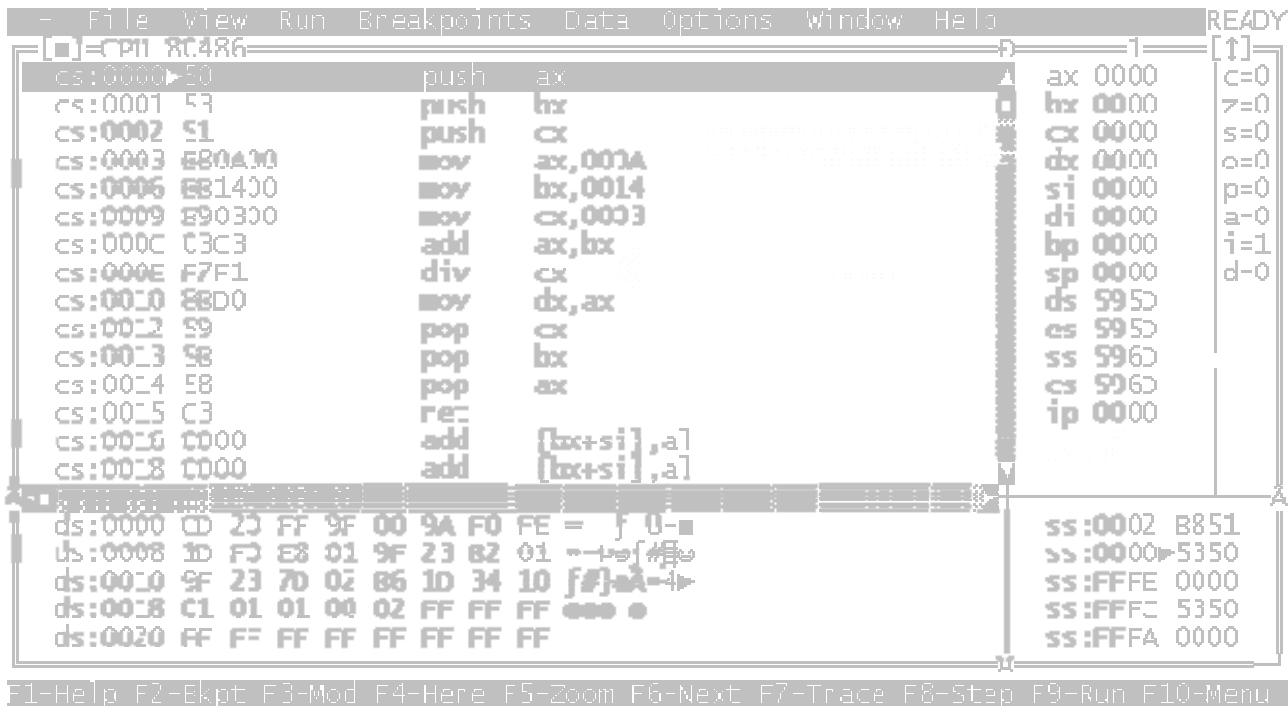
ChangeNumbers ENDP

end start

Now compile it to a .COM file and then type:

td name of file

Turbo Debugger then loads. You can see the instructions that make up your program.



For example the first few lines of this program is shown as:

```
cs:0000 50      push ax
cs:0001 53      push bx
cs:0002 51      push cx
```

Some useful keys for Turbo Debugger:

- F5 Size Window
- F7 Next Instruction
- F9 Run
- ALT F4 Step Backwards

The numbers that are moved into the registers are different that the ones that in the source code because they are represented in their hex form (base 16) as this is the easiest base to convert to and from binary and that it is easier to understand than binary also.

At the left of this display there is a box showing the contents of the registers. At this time all the main registers are empty. Now press F7 this means that the first line of the program is run. As the first line pushed the AX register into the

stack, you can see that the stack pointer (SP) has changed. Press F7 until the line which contains `mov ax,000A` is highlighted. Now press it again.

Now if you look at the box which contains the contents of the registers you can see that AX contains A. Press it again and BX now contains 14, press it again and CX contains 3. Now if you press F7 again you can see that AX now contains 1E which is A+14. Press it again and now AX contains 5A, 1E multiplied by 3. Press F7 again and you will see that DX now also contains 5A. Press it three more times and you can see that CX, BX and AX are all set back to their original values of zero.

## Lesson 6: More output in text modes

I am going to cover some more ways of outputting text in text modes. This first program is an example of how to move the cursor to display a string of text where you want it to go.

Listing 12: TEXT1.ASM

```
.model tiny
.code
org 100h
start:

mov dh,12                ; cursor col
mov dl,32                ; cursor row
mov ah,02h              ; move cursor to the right place
xor bh,bh               ; video page 0
int 10h                 ; call bios service

mov dx,OFFSET Text     ; DS:DX points to message
mov ah,9               ; function 9 - display string
int 21h                ; all dos service

mov ax,4C00h           ; exit to dos
int 21h

Text DB "This is some text$"

end start
```

This next example demonstrates how to write to the screen using the file function 40h of interrupt 21h.

Listing 13: TEXT2.ASM

```
.model small
.stack
.code

mov ax,@data          ; set up ds as the segment for data
mov ds,ax

mov ah,40h           ; function 40h - write file
mov bx,1             ; handle = 1 (screen)
mov cx,17            ; length of string
mov dx,OFFSET Text   ; DS:DX points to string
int 21h              ; call DOS service

mov ax,4C00h         ; terminate program
int 21h

.data

Text DB "This is some text"

end
```

The next program shows how to set up and call function 13h of interrupt 10h - write string. This has the advantages of being able to write a string anywhere on the screen in a specified colour but it is hard to set up.

Listing 14: TEXT3.ASM

```
.model small
.stack
.code

mov ax,@data          ; set up ds as the segment for data
mov es,ax              ; put this in es

mov bp,OFFSET Text     ; ES:BP points to message
```

```

mov ah,13h          ; function 13 - write string
mov al,01h         ; attrib in bl,move cursor
xor bh,bh          ; video page 0
mov bl,5           ; attribute - magenta
mov cx,17          ; length of string
mov dh,5           ; row to put string
mov dl,5           ; column to put string
int 10h           ; call BIOS service

```

```

mov ax,4C00h       ; return to DOS
int 21h

```

```

.data

```

```

Text DB "This is some text"

```

```

end

```

The next program demonstrates how to write to the screen using rep stosw to put the writing in video memory.

Listing 15: TEXT4.ASM

```

.model small

```

```

.stack

```

```

.code

```

```

mov ax,0B800h      ; segment of video buffer
mov es,ax          ; put this into es
xor di,di          ; clear di, ES:DI points to video memory
mov ah,4           ; attribute - red
mov al,"G"         ; character to put there
mov cx,4000        ; amount of times to put it there
cld                ; direction - forwards
rep stosw          ; output character at ES:[DI]

```

```

mov ax,4C00h       ; return to DOS
int 21h

```

end

The next program demonstrates how to write a string to video memory.

Listing 15: DIRECTWR.ASM

; write a string direct to video memory

.model small

.stack

.code

mov ax,@data

mov ds,ax

mov ax,0B800h ; segment of video buffer

mov es,ax ; put this into es

mov ah,3 ; attribute - cyan

mov cx,17 ; length of string to print

mov si,OFFSET Text ; DX:SI points to string

xor di,di

Wr\_Char:

lodsb ; put next character into al

mov es:[di],al ; output character to video memory

inc di ; move along to next column

mov es:[di],ah ; output attribute to video memory

inc di

loop Wr\_Char ; loop until done

mov ax,4C00h ; return to DOS

int 21h

.data

Text DB "This is some text"

end

It is left as an exercise to the reader to modify it so that only one write is made to video memory.

### Mode 13h

Mode 13h is only available on VGA, MCGA cards and above. The reason that I am talking about this card is that it is very easy to use for graphics because of how the memory is arranged.

### First check that mode 13h is possible

It would be polite to tell the user if their computer cannot support mode 13h instead of just crashing it computer without warning. This is how it is done.

Listing 16: CHECK13.ASM

```
.model small
.stack
.data

NoSupport db "Mode 13h is not supported on this computer."
           db ,"You need either a MCGA or VGA video"
           db ,"card/monitor.$"

Supported db "Mode 13h is supported on this computer.$"

.code

mov ax,@data           ; set up DS to point to data segment
mov ds,ax              ; use ax

call Check_Mode_13h    ; check if mode 13h is possible

jc Error               ; if cf=1 there is an error

mov ah,9               ; function 9 - display string
mov dx,OFFSET Supported ; DS:DX points to message
int 21h                ; call DOS service
```

```
jmp To_DOS ; exit to DOS
```

Error:

```
mov ah,9 ; function 9 - display string
mov dx,OFFSET NoSupport ; DS:DX points to message
int 21h ; call DOS service
```

To\_DOS:

```
mov ax,4C00h ; exit to DOS
int 21h
```

```
Check_Mode_13h PROC ; Returns: CF = 1 Mode 13h not possible
```

```
mov ax,1A00h ; Request video info for VGA
int 10h ; Get Display Combination Code
cmp al,1Ah ; Is VGA or MCGA present?
je Mode_13h_OK ; mode 13h is supported
stc ; mode 13h isn't supported CF=1
```

Mode\_13h\_OK:

```
ret
```

```
Check_Mode_13h ENDP
```

```
end
```

Just use this to check if mode 13h is supported at the beginning of your program to make sure that you can go into that mode.

### Setting the Video Mode

It is very simple to set the mode. This is how it is done.

```
mov ax,13h ; set mode 13h
int 10h ; call BIOS service
```

Once we are in mode 13h and have finished what we are doing we need to set it to the video mode that it was in previously.

This is done in two stages. Firstly we need to save the video mode and then reset it to that mode.

VideoMode db ?

```
.  
.  
mov ah,0Fh ; function 0Fh - get current mode  
int 10h ; Bios video service call  
mov VideoMode,al ; save current mode
```

; program code here

```
mov al,VideoMode ; set previous video mode  
xor ah,ah ; clear ah - set mode  
int 10h ; call bios service
```

```
mov ax,4C00h ; exit to dos  
int 21h ; call dos function
```

Now that we can get into mode 13h lets do something. Firstly lets put some pixels on the screen.

### **Function 0Ch: Write Graphics Pixel**

This makes a colour dot on the screen at the specified graphics coordinates.

INPUT:

AH = 0Ch

AL = Color of the dot

CX = Screen column (x coordinate)

DX = Screen row (y coordinate)

OUTPUT:

Nothing except pixel on screen.

Note: This function performs exclusive OR (XOR) with the new colour value and the current context of the pixel of bit 7 of AL is set.

This program demonstrates how to plot pixels. It should plot four red pixels into the middle of the screen.

Listing 17: PIXINT.ASM

; example of plotting pixels in mode 13 using bios services -

; INT 10h

.model tiny

.code

org 100h

start:

mov ax,13 ; mode = 13h

int 10h ; call bios service

mov ah,0Ch ; function 0Ch

mov al,4 ; color 4 - red

mov cx,160 ; x position = 160

mov dx,100 ; y position = 100

int 10h ; call BIOS service

inc dx ; plot pixel downwards

int 10h ; call BIOS service

inc cx ; plot pixel to right

int 10h ; call BIOS service

dec dx ; plot pixel up

int 10h ; call BIOS service

xor ax,ax ; function 00h - get a key

int 16h ; call BIOS service

mov ax,3 ; mode = 3

int 10h ; call BIOS service

mov ax,4C00h ; exit to DOS

int 21h

end start

## Some Optimizations

This method isn't too fast and we could make it a lot faster. How? By writing direct to video memory. This is done quite easily.

The VGA segment is 0A000h. To work out where each pixel goes you use this simple formula to get the offset.

$$\text{Offset} = X + (Y * 320)$$

All we do is to put a number at this location and there is now a pixel on the screen. The number is what colour it is.

There are two instructions that we can use to put a pixel on the screen, firstly we could use stosb to put the value in AL to ES:DI or we can use a new form of the MOV instruction like this:

```
mov es:[di], color
```

Which should we use? When we are going to write pixels to the screen we need to do so as fast as it is possible.

Instruction	Pentium	486	386	286	86
-------------	---------	-----	-----	-----	----

STOSB	3	5	4	3	11
-------	---	---	---	---	----

MOV AL to SEG:OFF	1	1	4	3	10
-------------------	---	---	---	---	----

If you use the MOV method you may need to increment DI (which STOSB does).

If we had a program which used sprites which need to be continuously draw, erased and then redraw you will have problems with flicker. To avoid this you can use a 'double buffer'. This is another part of memory which you write to and then copy all the information onto the screen.

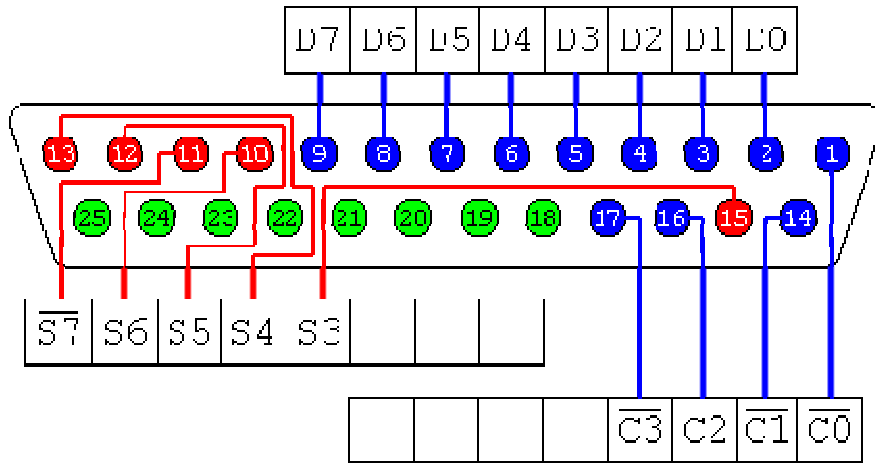
## Lesson 8 : Computer Hardware Interfacing using Assembly Language

### Parallel Ports (0278H – 0378H)

The original IBM-PC's Parallel Printer Port had a total of **12 digital outputs** and **5 digital inputs** accessed via 3 consecutive 8-bit ports in the processor's I/O space.

- **8 output pins accessed via the DATA Port**
- **5 input pins (one inverted) accessed via the STATUS Port**
- **4 output pins (three inverted) accessed via the CONTROL Port**

- The remaining 8 pins are grounded



**25-way Female D-Type Connector**

Below is a table of the "Pin Outs" of the D-Type 25 Pin connector and the Centronics 34 Pin connector. The D-Type 25 pin connector is the most common connector found on the Parallel Port of the computer, while the Centronics Connector is commonly found on printers. The IEEE 1284 standard however specifies 3 different connectors for use with the Parallel Port. The first one, 1284 Type A is the D-Type 25 connector found on the back of most computers. The 2nd is the 1284 Type B which is the 36 pin Centronics Connector found on most printers.

IEEE 1284 Type C however, is a 36 conductor connector like the Centronics, but smaller. This connector is claimed to have a better clip latch, better electrical properties and is easier to assemble. It also contains two more pins for signals which can be used to see whether the other device connected, has power. 1284 Type C connectors are recommended for new designs, so we can look forward on seeing these new connectors in the near future.

Pin No (D-Type 25)	Pin No (Centronics)	SPP Signal	Direction In/out	Register	Hardware Inverted
1	1	nStrobe	In/Out	Control	Yes

2	2	Data 0	Out	Data	
3	3	Data 1	Out	Data	
4	4	Data 2	Out	Data	
5	5	Data 3	Out	Data	
6	6	Data 4	Out	Data	
7	7	Data 5	Out	Data	
8	8	Data 6	Out	Data	
9	9	Data 7	Out	Data	
10	10	nAck	In	Status	
11	11	Busy	In	Status	Yes
12	12	Paper-Out / Paper-End	In	Status	
13	13	Select	In	Status	
14	14	nAuto-Linefeed	In/Out	Control	Yes
15	32	nError / nFault	In	Status	
16	31	nInitialize	In/Out	Control	
17	36	nSelect-Printer / nSelect-In	In/Out	Control	Yes
18 - 25	19-30	Ground	Gnd		

Table 1. Pin Assignments of the D-Type 25 pin Parallel Port Connector.

The above table uses "n" in front of the signal name to denote that the signal is active low. e.g. nError. If the printer has occurred an error then this line is low. This line normally is high, should the printer be functioning correctly. The "Hardware Inverted" means the signal is inverted by the Parallel card's hardware. Such an example is the Busy line. If +5v (Logic 1) was applied to this pin and the status register read, it would return back a 0 in Bit 7 of the Status Register.

The output of the Parallel Port is normally TTL logic levels. The voltage levels are the easy part. The current you can sink and source varies from port to port. Most Parallel Ports implemented in ASIC, can sink and source around 12mA. However these are just some of the figures taken from Data sheets, Sink/Source 6mA, Source 12mA/Sink 20mA, Sink 16mA/Source 4mA, Sink/Source 12mA. As you can see they vary quite a bit. The best bet is to use a buffer, so the least current is drawn from the Parallel Port.

## Experiment # 1

### BINARY COUNTER

PC parallel port can be very useful I/O channel for connecting your own circuits to PC. The port is very easy to use when you first understand some basic tricks. This document tries to show those tricks in easy to understand way.

**WARNING:** PC parallel port can be damaged quite easily if you make mistakes in the circuits you connect to it. If the parallel port is integrated to the motherboard (like in many new computers) repairing damaged parallel port may be expensive (in many cases it is cheaper to replace the whole motherboard than repair that port). Safest bet is to buy an inexpensive I/O card which has an extra parallel port and use it for your experiment. If you manage to damage the parallel port on that card, replacing it is easy and inexpensive.

#### How to connect circuits to parallel port

PC parallel port is 25 pin D-shaped female connector in the back of the computer. It is normally used for connecting computer to printer, but many other types of hardware for that port is available today.

Not all 25 are needed always. Usually you can easily do with only 8 output pins (data lines) and signal ground. I have presented those pins in the table below. Those output pins are adequate for many purposes.

pin function

2 D0

3 D1

4 D2

5 D3

6 D4

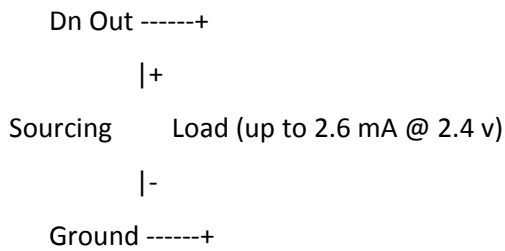
7 D5

8 D6

9 D7

Pins 18,19,20,21,22,23,24 and 25 are all ground pins.

Those datapins are TTL level output pins. This means that they put out ideally 0V when they are in low logic level (0) and +5V when they are in high logic level (1). In real world the voltages can be something different from ideal when the circuit is loaded. The output current capacity of the parallel port is limited to only few milliamperes.

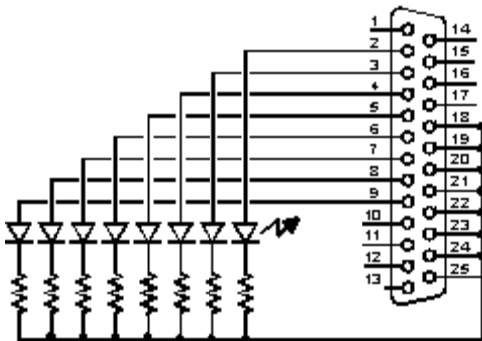


### Simple LED driving circuits

You can make simple circuit for driving a small led through PC parallel port. The only components needed are one LED and one 470 ohm resistors. You simply connect the diode and resistor in series. The resistors is needed to limit the current taken from parallel port to a value which light up acceptably normal LEDs and is still safe value (not overloading the parallel port chip). In practical case the output current will be few milliampres for the LED, which will cause a typical LED to somewhat light up visibly, but not get the full brightness.



Then you connect the circuit to the parallel port so that one end of the circuit goes to one data pin (that one you with to use for controlling that LED) and another one goes to any of the ground pins. Be sure to fit the circuit so that the LED positive lead (the longer one) goes to the datapin. If you put the led in the wrong way, it will not light in any condition. You can connect one circuit to each of the parallel port data pins. In this way you get eight software controllable LEDs.



The software controlling is easy. When you send out 1 to the datapin where the LED is connected, that LED will light. When you send 0 to that same pin, the LED will no longer light.

### Control program (C)

```
outp(0x378,n);  
or  
outportb(0x378,n);
```

Where N is the data you want to output. The actual I/O port controlling command varies from compiler to compiler because it is not part of standardized C libraries.

```
# include<stdio.h>  
main()  
{  
    int n;  
    clrscr();  
    printf(" Enter a value:");  
    scanf("%d",&n);  
    outportb(0x378,n);  
    printf("The equivalent binary no. is display in the led.");  
    getch();  
}
```

### How to calculate your own values to send to program

You have to think the value you give to the program as a binary number. Every bit of the binary number control one output bit. The following table describes the relation of the bits, parallel port output pins and the value of those bits.

Pin	2	3	4	5	6	7	8	9
Bit	D0	D1	D2	D3	D4	D5	D6	D7
Value	1	2	4	8	16	32	64	128

For example if you want to set pins 2 and 3 to logic 1 (led on) then you have to output value  $1+2=3$ . If you want to set on pins 3,5 and 6 then you need to output value  $2+8+16=26$ . In this way you can calculate the value for any bit combination you want to output.

