Lab 3: Defining Data and Symbolic Constants

Contents

3.1 MASM Data Types

MASM defines various intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type. The following table lists these data types. The first 9 data types are used to define integer data, while the last 3 are used to define real data according to the IEEE standard real number formats.

Type	Usage
BYTE	8-bit unsigned integer (range 0 to 255)
SBYTE	8-bit signed integer (range –128 to +127)
WORD	16-bit unsigned integer (range 0 to 65535 = 2^{16} – 1)
SWORD	16-bit signed integer (range –32768 to +32767 = 2^{15} – 1)
DWORD	32-bit unsigned integer (Double Word: range 0 to 4,294,967,295 = 2^{32} – 1)
SDWORD	32-bit signed integer (range –2,147,483,648 to +2,147,483,647 = 2^{31} – 1)
FWORD	48-bit integer (FAR pointer in protected mode)
QWORD	64-bit integer (Quad Word)
TBYTE	80-bit (Ten Byte) integer
REAL4	32-bit (4-byte) IEEE floating-point number
REAL8	64-bit (8-byte) IEEE double-precision floating-point number
REAL10	80-bit (10-byte) IEEE extended-precision floating-point number

Data definition has the following syntax:

```
[name] directive initializer [,initializer] . . .
```

For more information, refer to your textbook or class notes.

3.2 Lab Work: Defining Integer Data

The following program demonstrates integer data definition under the .DATA section. You may open and view this program in ConTEXT or any other text editor. Assemble and link this program to produce the *IntegerDef.exe* executable file.

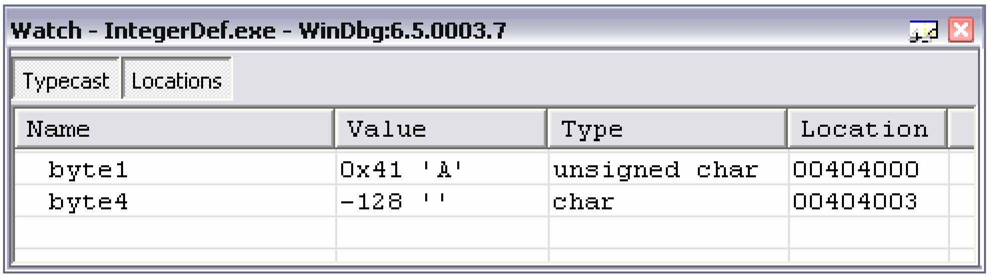
```
TITLE Integer Data Definitions            (File:IntegerDef.asm)

; Examples Demonstrating Integer Data Definition

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.data
; ---------------- Byte Values --------------------
byte1  BYTE   'A'            ; 'A' = 65 = 41h
byte2  BYTE    0             ; smallest unsigned byte value
byte3  BYTE    255           ; largest unsigned byte value
byte4  SBYTE   -128          ; smallest signed byte value
byte5  SBYTE   +127          ; largest signed byte value   ;
byte6  BYTE    ?             ; uninitialized

; ---------------- Word Values --------------------
word1  WORD    65535         ; largest unsigned word value
word2  SWORD   -32768        ; smallest signed word value
word3  WORD    ?             ; uninitialized

; -------------- DoubleWord Values ----------------
dword1 DWORD   0FFFFFFFFh    ; largest unsigned value in hex
dword2 SDWORD  -2147483648   ; smallest signed value in decimal

; -------------- QuadWord Value -------------------
quad1  QWORD   0123456789ABCDEFh

.code
main PROC

; No instructions to execute
      exit
main ENDP
END main
```

3.3 Lab Work: Watching Variables using the Windows Debugger

Now run the Windows debugger to watch the variables and the memory content. You may run the debugger from ConTEXT **Tools** menu (if it is properly configured) or from the command prompt by typing: **windbg –QY –G IntegerDef.exe**

3.3.1 Lab Work: Watch Window

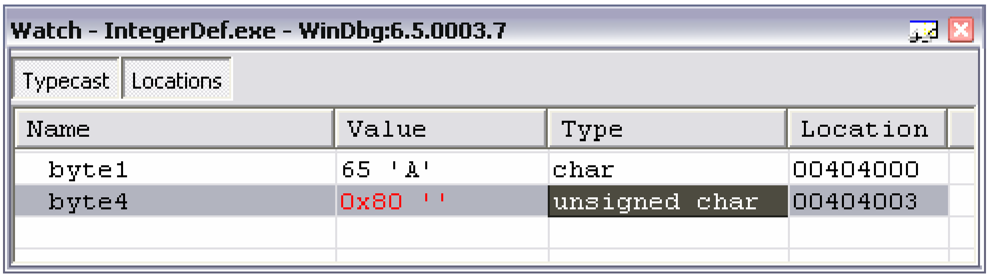
Open the **Watch window** (from the **View** menu). Insert the variable *byte1* and *byte4* under the *Name* column as shown below. To add a variable to the Watch list, click in the first empty cell in the Name column, enter the name of this variable, and press ENTER. You can watch variables in any order. It does not have to be the same order declared in the source program.



Name	Value	Type	Location
byte1	0x41 'A'	unsigned char	00404000
byte4	-128 ''	char	00404003

Observe the *Value* of these variables under the *Value* column. You can also view the **type** and **memory addresses** of these variables by pressing on the **Typecast** and **Locations** buttons. Observe that all the memory location addresses are in hexadecimal. The value of variables can be in hexadecimal or decimal depending on whether the type is unsigned or not. The windows debugger uses different type names other than the ones used in MASM. The type *BYTE* becomes *unsigned char* and *SBYTE* becomes *char*. The type *WORD* becomes *unsigned short* and *SWORD* becomes *short*. The type *DWORD* becomes *unsigned long*, *SDWORD* becomes *long*, and the type *QWORD* becomes *int64*.

Now, let us change the Type of *byte1* from *unsigned char* to *char*. This can be done easily by clicking on the type and editing its value. Observe that the value of *byte1* now appears in decimal (**65**) after it appeared in hexadecimal (**0x41**). Similarly, let us change the type of *byte4* from *char* to *unsigned char* as shown below. Observe that **changing the type of a variable to unsigned (or vice versa) does NOT change the value of the variable**. The value is still the same and is stored in binary in the memory of the computer. It is only **how we view the same value** as being either in signed decimal or hexadecimal.



Name	Value	Type	Location
byte1	65 'A'	char	00404000
byte4	0x80 ''	unsigned char	00404003

To delete a Name from the watch list, select that name and press the delete button to delete the name. The line will disappear from the watch list as soon as click outside.

Now insert all the variable names in the watch list and observe their values and locations. Fill in the following table **showing the Location (address) of these variables in the data segment in memory, as well as their values in hex and in decimal**. Observe that the variables occupy locations at successive addresses in memory, according to their order of appearance in the program.

Name	Location (hex)	Value (hex)	Value (decimal)
byte1	00404000	'A' = 41h	65
byte2			
byte3			
byte4			
byte5			
byte6			
word1			

word2			
word3			
dword1			
dword2			
quad1			

What is the total number of bytes allocated for data? ....

3.4 Lab Work: Multiple Initializers, Defining Strings, and the DUP Operator

You can create arrays of bytes, words, double words, etc., either by explicitly using multiple initializers or by using the **DUP** (Duplicate) operator. Multiple initializers are separated by commas and are used to initialize each element of the array with an explicit number. The **DUP** operator generates a repeated storage allocation, using a constant expression as a counter. The initializers and the **DUP** operator can be combined together and can be nested. You can also create string data definition by enclosing a sequence of characters in quotation marks (either single quotes or double quotes can be used). Strings are commonly terminated with a null character, a byte containing the value 0. We will follow the convention of terminating all strings with a null char.

```
TITLE Multiple Initializers        (MultipleInitializers.asm)

; Examples showing multiple initializers and the DUP operator

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

; ----------------- Byte Values ----------------
.data

list1   BYTE  10, 32, 41h, 00100010b
list2   BYTE  0Ah, 20h, 'A', 22h
array1  BYTE  8 DUP(0)        ; 8 bytes initialized to 0

greeting BYTE "Good  afternoon",0

; ----------------- Word Values ---------------------

myList  WORD   1,2,3,4        ; array of words

; --------------- DoubleWord Values --------------

array2  DWORD   4 DUP(01234567h)

.code
main PROC

; No instructions to execute

      exit
main ENDP
END main
```

Now open *MultipleInitializers.asm* and assemble and link the file. You can use the *make32* batch file from the command prompt or from the ConTEXT editor's **Tools** menu.

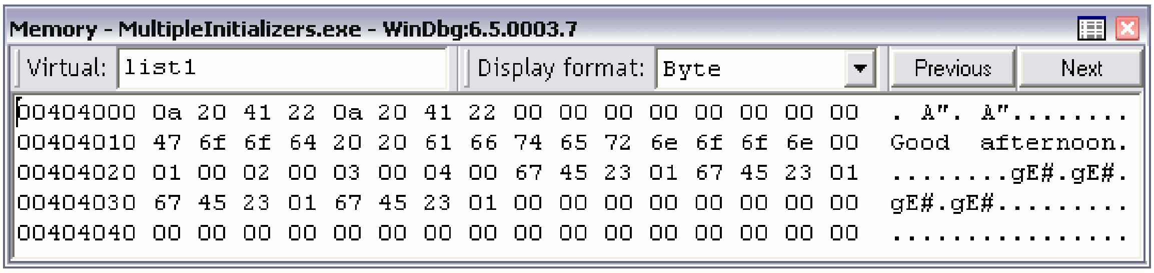
3.5 Lab Work: Watching Memory using the Windows Debugger

Now run the Windows debugger to view the memory content. You may open the debugger from the ConTEXT editor's **Tools** menu (if it is properly configured) or from the command prompt by typing: **windbg –QY –G MultipleInitializers.exe**

3.5.1 Lab Work: Memory Window

From the **View** menu, select **Memory** to open the **Memory Window**. This window will allow you to watch the content of memory. Under **Virtual** enter **list1**, the name of the first variable. The **virtual address** of *list1* is **00404000** (in hexadecimal) as shown in the first column of the Memory Window. We call this address a virtual address because it is not a real address. The Windows Operating System maps virtual addresses onto real addresses, but we are not concerned here about the details.

In the **Memory window** shown below, the **Display format** is shown as **Byte**, and the Byte values in memory appear in hexadecimal. You may resize the Memory window so that exactly 16 bytes are displayed on each line. The virtual address goes by increments of 10h = 16 bytes.



Virtual: list1

Display format: Byte

```
00404000 0a 20 41 22 0a 20 41 22 00 00 00 00 00 00 00 00  . A". A"........
00404010 47 6f 6f 64 20 20 61 66 74 65 72 6e 6f 6f 6e 00  Good  afternoon.
00404020 01 00 02 00 03 00 04 00 67 45 23 01 67 45 23 01  ........gE#.gE#.
00404030 67 45 23 01 67 45 23 01 00 00 00 00 00 00 00 00  gE#.gE#.........
00404040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

It is also possible to write the virtual address of the variable in the Virtual address box. For example, we could have written **00404000** in the virtual address box, but it is easier to refer to memory by name, rather than by address. The addresses are still listed in the first column of the Memory Window in hexadecimal. Observe also that printable characters are displayed on the right. For example, the byte at address **00404010** is **47h = G**, and the byte at address **00404013** is **64h = d**.

You can view memory starting at any address or at any variable. For example, if you want to view *myList* then type the variable name in the **Virtual** address box.

Now answer the following:

What is the virtual address (in hexadecimal) of *myList*? ....

What is the virtual address (in hexadecimal) of *array2*? ....

How many bytes are allocated for *myList*? ....

How many bytes are allocated for *array2*? ....

What is the byte value (in hex) at virtual address **00404018**? ....

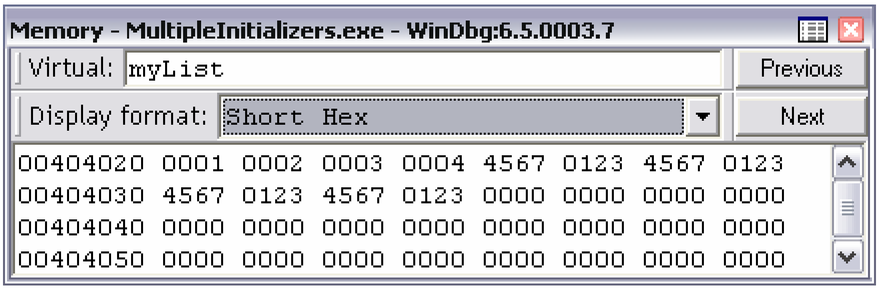
What is the byte value (in hex) at virtual address **00404032**? ....

3.5.2 Little Endian Order

The variable *myList* is an array of words and each word occupies 2 bytes of memory. The first element of *myList* has the value **1** and occupies 2 bytes of memory. The least significant byte has value **01h** and the most significant byte has value **00h**. The least significant byte is stored at address **00404020h** (low byte address) and the most significant byte is stored at address **00404021h** (high byte address) as shown in the above Memory window. Similarly, observe the byte order of the elements of *array2*. Each element of *array2* is a double word and occupies 4 bytes of memory. Each element has a value **01234567h**, where the least significant byte **67h** is stored at the first byte address, while the most significant byte **01h** is stored at the last byte address. This byte ordering, from least significant byte to most significant byte, is called **Little Endian order**. This byte ordering is used by the Intel processors to store values that occupy more than one byte.

3.5.3 Lab Work: Changing the Display Format

The Byte display format is not convenient to view arrays of words or double words. Let us now change the **Display format** to **Short Hex** to have a WORD view of *myList* in hexadecimal, as shown below. The **Short Unsigned** display format can be used to view an unsigned WORD array in decimal, rather than in hex. The **Short** display format can be used to view the elements of an SWORD (Signed WORD) array in decimal.



Virtual: myList

Display format: Short Hex

```
00404020 0001 0002 0003 0004 4567 0123 4567 0123
00404030 4567 0123 4567 0123 0000 0000 0000 0000
00404040 0000 0000 0000 0000 0000 0000 0000 0000
00404050 0000 0000 0000 0000 0000 0000 0000 0000
```

For *array2*, which is an array of double words, use the **Long Hex** display format to display the values. Observe that each element of *array2* occupies 4 bytes of memory. We can also use the **Long** and **Long Unsigned** display formats to display the numbers as signed or unsigned decimal, rather than hex.

Using the **Long Hex** format for *array2*, write the address and the four double word values of *array2*, as they appear in the Memory window:

Other display formats are also available and can be explored. The **Quad** display formats can be used to display QWORD (8-byte) numbers either signed or unsigned, in decimal or in hex. The **Real** display formats can be used to display floating-point numbers of different sizes.

3.6 Lab Work: Data Related Operators

In the following program, you will learn about **TYPE**, **LENGTHOF**, **SIZEOF**, **OFFSET**, and **PTR** operators. These operators are not instructions and are not executed by the processor. Instead, they are only processed by the assembler during assembly time.

TYPE	Size (in bytes) of each element in an array.
LENGTHOF	Number of elements in an array.
SIZEOF	Number of bytes used by an array initializer.
OFFSET	Virtual address of a variable.
PTR	Used to override a variable's default size.

For more details about these operators, refer to the lecture notes or your textbook. Open *Operators.asm* using ConTEXT or any other text editor and assemble and link the file.

```
TITLE Operators                (File: Operators.asm)
; Demonstration of TYPE, LENGTHOF, SIZEOF, OFFSET, and PTR operators

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.data
byte1    BYTE  10,20,30,40
array1   WORD  30 DUP(?),0,0
array2   WORD  5 DUP(3 DUP(?))
array3   DWORD 01234567h,2,3,4
digitStr BYTE  '12345678',0
myArray  BYTE  10h,20h,30h,40h,50h,60h,70h,80h,90h

.code
main PROC
      ; Demonstrating TYPE operator
      mov al, TYPE byte1
      mov bl, TYPE array1
      mov cl, TYPE array3
      mov dl, TYPE digitStr

      ; Demonstrating LENGTHOF operator
      mov eax, LENGTHOF array1
      mov ebx, LENGTHOF array2
      mov ecx, LENGTHOF array3
      mov edx, LENGTHOF digitStr

      ; Demonstrating SIZEOF operator
      mov eax, SIZEOF array1
      mov ebx, SIZEOF array2
      mov ecx, SIZEOF array3
      mov edx, SIZEOF digitStr

      ; Demonstrating OFFSET operator
      mov eax, OFFSET byte1
      mov ebx, OFFSET array1
      mov ecx, OFFSET array2
      mov edx, OFFSET array3
      mov esi, OFFSET digitStr
      mov edi, OFFSET myArray
      ; Demonstrating PTR operator
      mov al,  BYTE  PTR array3
      mov bx,  WORD  PTR array3
      mov cx,  WORD  PTR myArray
      mov edx, DWORD PTR myArray

      exit
main ENDP
END main
```

3.6.1 Lab Work: Tracing Program Execution and Watching Registers

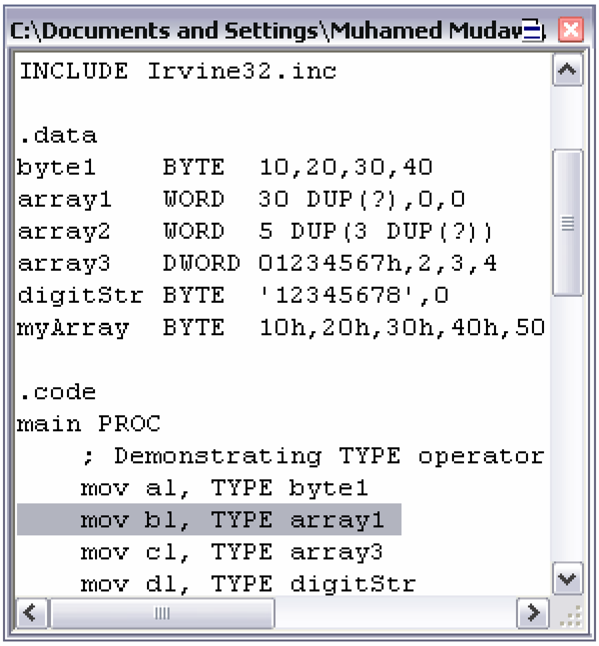
Run the Windows debugger from the ConTEXT editor, or from the command prompt by typing: **windbg –QY –G Operators.exe**. From the **View** menu, select **Registers** to open the **Registers Window**. Customize the registers to appear in the order that you want. Press the **Customize** button and enter:

al bl cl dl ax bx cx dx eax ebx ecx edx esi edi

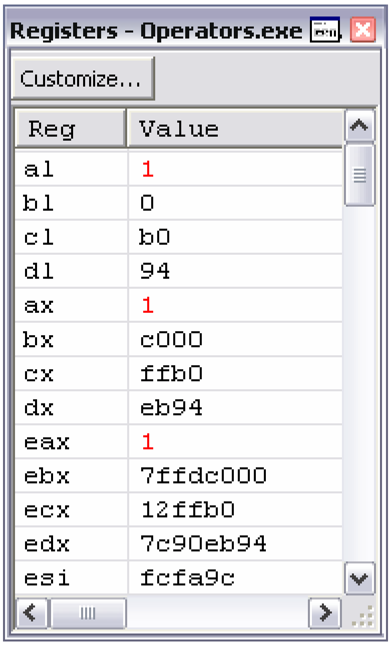
We only care to view these registers. The rest is not important. The list can be customized differently for different programs. You can also make this register window always floating or you may dock it, by selecting the option in the upper drop-down menu.

Now place the cursor at the beginning of the main procedure and press **F7** to start debugging the main procedure. You may also place the cursor at any instruction and press **F7** to run the program until that instruction. This is useful if you don't want to view the execution of each instruction separately.

Now press **F10** to watch the execution of each instruction separately in the main procedure, and to view the changes in the registers at each step. These value changes appear in red. The next instruction in the source window is highlighted. This instruction is not executed yet, but will be executed the next time that you press **F10**.



```
INCLUDE Irvine32.inc

.data
byte1     BYTE  10,20,30,40
array1    WORD  30 DUP(?),0,0
array2    WORD  5 DUP(3 DUP(?))
array3    DWORD 01234567h,2,3,4
digitStr  BYTE  '12345678',0
myArray   BYTE  10h,20h,30h,40h,50

.code
main PROC
    ; Demonstrating TYPE operator
    mov al, TYPE byte1
    mov bl, TYPE array1
    mov cl, TYPE array3
    mov dl, TYPE digitStr
```



Reg	Value
al	1
bl	0
cl	b0
dl	94
ax	1
bx	c000
cx	ffb0
dx	eb94
eax	1
ebx	7ffdc000
ecx	12ffb0
edx	7c90eb94
esi	fcfa9c

Try first to guess and understand the values of the registers in the above program. Write these values in decimal for the **TYPE**, **LENGTHOF**, and **SIZEOF** operators, and in hexadecimal for the **OFFSET** and **PTR** operators. By default, all register values appear in hexadecimal. You will have to convert them to decimal for the **LENGTHOF** and **SIZEOF** operators.

TYPE Operator

al =	**bl =**	**cl =**	**dl =**

LENGTHOF Operator

eax (decimal) =	**ecx (decimal) =**
ebx (decimal) =	**edx (decimal) =**

SIZEOF Operator

eax (decimal) =	**ecx (decimal) =**
ebx (decimal) =	**edx (decimal) =**

OFFSET Operator

eax (hex) =	**edx (hex) =**
ebx (hex) =	**esi (hex) =**
ecx (hex) =	**edi (hex) =**

PTR Operator

al (hex) =	**cx (hex) =**
bx (hex) =	**edx (hex) =**

3.7 Lab Work: Symbolic Constants and the EQU and = directives

A *symbolic constant* is created by associating an *identifier* (a *symbol*) with either an integer expression or some text. Unlike a variable definition, which reserves storage, a **symbolic constant does not use any storage**. The value of a symbolic constant is defined by the assembler and does not change at run time.

The **EQU** (Equal) directive associates a symbolic name with either an integer expression or some arbitrary text, according to the following formats:

```
name EQU expression
name EQU <text>
```

A symbol defined with **EQU** cannot be redefined in the same source code file. This prevents an existing symbol from being inadvertently assigned a new value.

Unlike the **EQU** directive, the = directive can redefine a symbol any number of times. The = directive can associate a symbolic name with an integer expression only according to the following format. However, the = directive cannot associate a symbol with text.

```
name = expression
```

The following program illustrates the definition of symbolic constants:

```
TITLE Symbolic Constants           (File: Constants.asm)
; Demonstration of EQU and = directives

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.data
Rows       EQU   3
Cols       EQU   3
Elements   EQU   Rows * Cols
CR         EQU   10
LF         EQU   13
PromptText EQU   <"Press any key to continue ...",CR,LF,0>

matrix     WORD  Elements DUP(0)
prompt     BYTE  PromptText

COUNT = 10h
COUNT = 100h
COUNT = 1000h
COUNT = SIZEOF matrix

.code
main PROC
      exit
main ENDP
END main
```

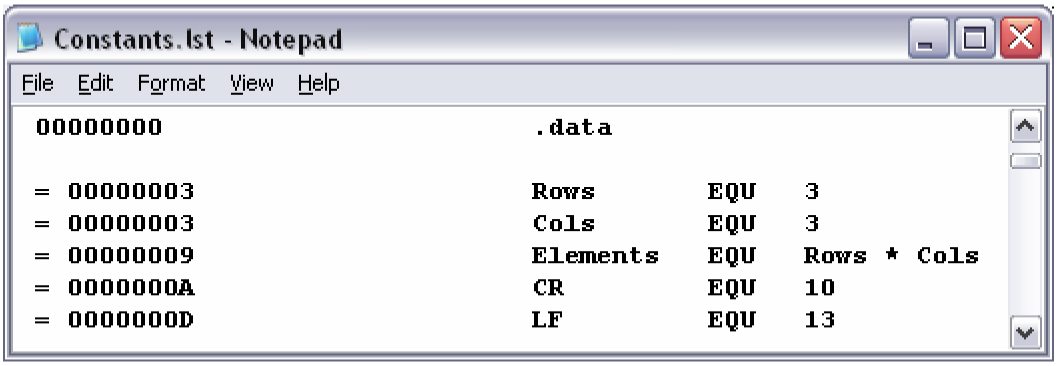
Fill the table below, listing only the symbolic constants in the above Program and their values in hexadecimal. If a symbolic constant is redefined then it should be listed multiple times.

Symbolic Constant	Value (hexadecimal)

What is the total number of bytes allocated for data? ....

3.8 Lab Work: Viewing Symbolic Constants in the Listing (.lst) File

Open a command prompt and type: **ml –c –Zi –Fl –coff Constants.asm**. The **ML** program is the assembler. It will generate the *Constants.obj* and *Constants.lst* files. The *Constants.obj* is the object file and contains the machine code. The *Constants.lst* file is a listing file produced optionally when you use the **–Fl** option with the **ML** command. Open the *Constants.lst* file and examine its contents. This file shows the work of the assembler and contains a copy of the assembly language source code, offset addresses, translated machine code, procedures, and symbols. The symbolic constants and their values are listed in this file. Examine the content of the *Constants.lst* file, and check the values of the symbolic constants. Notice that the symbolic constant values are listed in hexadecimal.



```
 00000000                       .data

 = 00000003                     Rows        EQU    3
 = 00000003                     Cols        EQU    3
 = 00000009                     Elements    EQU    Rows * Cols
 = 0000000A                     CR          EQU    10
 = 0000000D                     LF          EQU    13
```

Review Questions

1. Write a data declaration for an 8-bit unsigned integer variable.

2. Write a data declaration for a 32-bit signed integer variable.

3. Declare a 16-bit signed integer and initialize it with the smallest negative 16-bit number.

4. Declare an unsigned 16-bit integer variable *wArray* that uses three initializers.

5. Declare an uninitialized array of 50 unsigned 32-bit integers named *dArray*.

6. Declare a string variable containing the word “TEST” repeated 100 times.

7. Show the order of individual bytes in memory (lowest to highest) for the following:
   `dvar DWORD 5012AB6Fh`

8. Consider the following array declaration: **`myArray DWORD 30 DUP(5 DUP(0))`**

   Define a symbolic constant *Elements* that calculates the number of elements in *myArray*.

   Define a symbolic constant *Size* that calculates the number of bytes in *myArray*.