**Chapter Four**

**Directives and Operators**

1. **Directives and Operators**

Just as High Level Programming languages must have non-executable statements to pre-assign values, reserve storage, assign names to constants, form data structures, terminate compilation etc…, assembler languages must have directives to perform similar tasks. These features provide the basic rules and frame works for the assembly language program.

1. **Program Comments**
	* To improve clarity of a program
	* Begins with semi colon
	* Generate no machine code
	* One can include any number of comments without affecting the size of the assembled program.
2. **Reserved words**
	* These are certain words that are reserved for other operations. They may include Instructions (such as MOV, ADD), Directives (such as SEGMENT, PROC), Operators (such as SIZE, OFFSET), Predefined symbols if there are any (such as $). These reserved words cannot be used as identifiers (Labels, Variables names, procedure names, segment names etc).
3. **Identifiers or Symbols**
	* A name that you can apply to an item in a program that you expect to reference. The following are some of the types of identifiers:
		1. **Name**: refers to the address of the first byte of a data item. See the Data Definition and Storage directive later in this note.
		2. **Label**: Refers to the first byte of an address of an instruction, procedure or segment.

**Note:** There might be assembler specific rules to allow or disallow certain characters in identifiers (in addition to reserved words).

1. **Statements:**
	* An assembler language program consists of a set of statements generally of two types:
		1. **Instructions**: which are translated into machine code by the assembler.
		2. **Directives**: which direct the assembler to perform a specific action. These are not translated into machine language instructions. The following section describes some of these directives.
		3. **Pseudo-opcodes:** Pseudo-opcodes were discussed in the previous chapter separately from directives. However, in this chapter, pseudo -opcodes are explained together with directives.
	1. **Directives**

Any assembler program should support a number of directives to control the way in which a source program is assembled. The directives act only during the assembling process and generate no machine code. The following is a discussion on some of the known directives common to all assembly language programs.

* + 1. **Data Definition and Storage (variable declaration)**

**General form:**

*Variable Mnemonic Operand,…………., Operand ; Comment*

* + ***Variable****:* is optional, but if it is present it will be associated with the Offset (EA) of the first byte that is reserved by the directives.
	+ ***Mnemonic****:* it is the actual directives that determines the size of each operand and can be one of the following:
		1. *DB (Define Byte): each operand datum occupies one byte.*
		2. *DW (Define Word): Each operand datum occupies one word with its lower part being in the first byte.*
		3. *DD (Define Double Word): Each operand datum is two words (32 bit) long with the low order word followed by the high order word.*
		4. *DQ: eight byte variable*
		5. *DT: Ten byte variable*
	+ ***Operand(s):***indicate the data to be pre-assigned or the amount of space to be reserved.
	+ ***Comments****:* is optional, and used to describe the data assignment.

***Examples:***

*Data\_byte DB 10,4, 10h*

*Message DB ‘Hello’*

*Abc DW 0, ?, ?, 7*

Pre-assignment by individually listing the contents of the locations is satisfactory if there are only a few locations to be filled. But if there are several operands, such an approach will be burdensome. Therefore, just as high level languages allow for duplication, assemblers permit the use of duplication operator DUP. Several operands or operand patterns can be replaced with a form:

***Expression DUP (Operand,……….., Operand)***

*Where Expression is an expression that evaluates to a positive integer (or just a positive integer) that defines the number of repetitions.*

*Examples:*

 *Array2 DB 2 DUP (0, 1, 2, ?)*

 *Array2 DB 100 DUP (0, 2 DUP (1,2), 0, 3)*

* + 1. **Segment Definition**
	+ Physical memory address is obtained by adding an offset to 16 times a segment address that is contained in a segment register. One of the tasks an assembler must perform is to assign the offsets of the labels & variables as it translates the instructions into machine language.
	+ The assembler must also pass to the linker (via the object modules) all of the information that the link will need in putting the various segments & modules together to form a program.

Therefore, to be able to assign the variable & label offsets the assembler must knew the exact structure of segment.

*Syntax of Segment Directives:*

***Segment\_name segment***

***(Storage definition, allocation, & alignment directives) or***

***(Instructions and instruction related directives)***

 ***Segment\_name Ends***

*Segment Name should be a valid identifier*

The **Assume** directive is used to associate segments to their corresponding registers, with the following syntax.

*Assume Assignment , …………….., Assignment*

 where each assignment is written as

 *Segment register name: Segment name*.

E.g. *Assume CS: code\_seg, DS:data\_ seg1, ES:data\_seg2*

The same segment can be assigned to two-segment register.

 *Assume CS: Code\_seg, DS: Code\_seg*.

* + 1. **Procedure Directive**

The code segment may contain the executable code for a program, which consists of one or more Procedures defined initially with the *PROC* directive & ended with the *ENDP* directive.

The Syntax:

 *Procedure name Proc*

 *(Instructions and other directives)*

 *Procedure name Endp*.

Procedures provide the following benefits:

* + Reduce the amount of code, because a common procedure can be called anytime.
	+ Encourage better program organization.
	+ Defects can easily be isolated.
		1. **Structures**

All elements allocated by a single storage definition statement must be of the same type. It is desirable for a variable to have several fields, with each filed having its own type. The STRUC directive handles this.

A Structure definition gives the ***pattern*** of the structure & may have the simplified form:

*Structure name STRUC*

 *Sequence or DB & DW & DD directive*

*(The variables associated with this are called field identifiers)*

 *Structure name ENDS*

The structure definition doesn’t reserve storage or directly pre-assign values; it just defines a pattern. Therefore, to reserve the necessary space it must be accompanied by a statement for invoking the structure-with the following format.

***Variable structure name 〈Preassignment Specification 〉***

Example: Personnel\_data STRUC

Initials DB ‘xx’

Last name DB 5 Dup(?)

ID DB O,O

Age DB ?

Weight DW ?

Personnel\_data ENDS

 Employee\_1 personnel\_data 〈‘JR, , ,’ 35〉

A variable that is associated with a structure is referred to by an instruction by modifying the memory referencing operand with pathnames of the form.

*Offset rotation. Filed Name*

Example:. Mov AL, employee\_1. Last name[SI]

*(The SIth element of the last name filedin the structure employee1 defined by the personnel data)*

* + 1. **Assigning names to expression (EQU directive)**

If an expression appears several times in a program, it is sometimes more convenient to give it a name & refer to it by name.

The statement that assigns a name to an expression has the form.

 *Expression name EQU Expression*

*Where the expression name should be a valid identifier.*

The expression may have the format of any valid operand, be any expression that evaluates to a constant or be a valid mnemonic.

Examples Constant EQU 256

 Alpha EQU 7

 BETA EQU Alpha-2

Another important reason for using names in place of instructions is that they permit a program to be easily modified.

* + 1. **Program Termination (END Directive)**

***END Label***

It is used to indicate the end of a set of assembler language code.

The label must also appear in the instruction to be executed first, the point at which the program begins.

Example: A complete program to perform addition of the contents of *OPER1* & *OPER2* and stores the absolute value of the sum in *result*.

 Data segment

 OPER1 DW 12

 OPER2 DW 230

 Result DW ?

Data ends

Code\_seg segment

 Assume CS:code\_seg, DS:Data

Start: Mov AX,Data

 Mov DS, AX

 Mov AX, Oper1

 ADD AX, Oper2

 Mor Result, AX

 HLT

 Code\_seg ends

 END START

* + 1. **Value returning attributes operators.**

They are used to make the programmer’s task a little easier by providing flexibility, increasing readability & causing the assembler to fill in some of the numbers.

These operators are:

 *LENGTH, SIZE, OFFSET, SEG & TYPE*

*(All these operators return values related to the expressions, which follows them within their operands).*

* ***LENGTH*** returns the number of units assigned to a variable.

*Example: FEES DW 100 DUP (0)*

 *Therefore,* Mov CX, length fees is equivalent to *Mov CX,100*

* ***SIZE***operator is the same as the LENGTH operator except that it returns the number of bytes instead of the number of units.

*Example: FEES DW 100 DUP (0)*

 *Therefore,* Mov CX, size fees is equivalent to *Mov CX,200*

* The **OFFSET** operator returns the value of the offset of a label or variable.

*Example: Mov BX, offset Oper1* is equivalent to *LEA BX, OPER1*

* The **SEG** operator similarly causes the segment address of the variable or label to be inserted as an immediate operand. (Although the actual insertion is mad by the linker).

*Example:* If *data\_seg* is assigned to the block of memory *beginning* at *05000* & *oper1* is in *data\_seg*, then the instruction:

 ***Mov BX, seg oper1*** *w*ould put 0500 in BX

* The **TYPE** operator is used primarily with variables and structure names and returns the number of bytes associated with the variable or structure.

*Example:* ***ADD SI, Type array*** would add 1 to the SI register if ARRAY is defined in a DB statement, 2 if defined in A DW, 4 if defined in a DD statement

* 1. **Directives, Instructions and Pseudo Opcodes in brief**
		1. **Directive in Brief**

The **HEX** Directive

The HEX Directive directs the assembler to treat values in the source file that begins with the character, which is defined with the HEX directive as numeric constant in hexadecimal notation.

The **SEGMENT** Directive

A segment directive defines the logical segment to which subsequent instructions and data allocation statement belong.

 *Format:* Segment\_Name SEGMENT

 :

 :

 :

 Segment\_Name ENDS

The **END** Directive

The END directive is simply an advisory to the assembler alerting it that it has reached the end of the program.

The **PROC** Directive

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

 *Format:* proc\_name PROC

 :

 :

 proc\_name ENDP

* + 1. **Instruction statements in brief**

The **MOV** Instruction

The MOV instruction copies the contents of the source operand into the destination operand.

 *Format:* MOV destination, source

The **INC** Instruction

The INC (INCrement) instruction increases the contents of the operand by the value of 1.

 *Format:* INC operand

The **DEC** Instruction

The DEC (DECrement) subtracts a value of 1 from the contents of the operand.

 *Format:* DEC operand

The **INT** Instruction

The mnemonic INT stands for INTerrupt. It is a kind of subroutine call

 *Format:* INT $XX

INT $XX means call special subroutine number $XX

The **CALL** Instruction

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

 Format: CALL *procname*

Where *procname* is the name of the procedure

The **RET** Instruction

The RET (RETurn) instruction transfers program flow back from a subroutine to its parent.

The **JMP** instruction

The JMP instruction make unconditional jump or transfer the program flow to the instruction at specific place in the program which is addressed by a label.

 *Format:* JMP ***label***

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

A conditional jump instruction will test the state of some specified status flag or flags and direct program flow accordingly. The following are some conditional jumps:

Jump if above JA Jump if not above JNA

Jump if above or equal JAE Jump if neither above nor equal JNAE

Jump if below JB Jump if not below JNB

Jump if below or equal JBE Jump if neither below or equal JNBE

Jump if carry set JC Jump if no carry JNC

Jump if equal JE Jump if not eqal JNE

Jump if zero JZ Jump if not zero JNZ

Jump if greater JG Jump if not greater JNG

Jump if greater or equal JGE Jump if neither greater or equal JNGE

Jump if less JL Jump if not less JNL

Jump if less or equal JLE Jump if neither less or equal JNLE

Jump if overflow JO Jump if not overflow JO

Jump if parity even JPE Jump if parity odd JPO

Jump if parity JP Jump if no parity JNP

Jump if sign negative JS Jump if sign positive JNS

Jump if CX register is zero

The **LOOP** Instruction

The LOOP instruction decrements the contents of the CX register and, if the result is not zero, directs program flow to the instruction at the address referenced by label.

 *Format:* LOOP **label**

* + 1. **Pseudo Opcodes (Data Definition Statements) in brief**

The ***data\_definition\_type*** is one of the following:

 DB (Define Byte) allocates 1 - byte variable

 DW (Define Word) “ 2 - byte variable

 DD (Define Double Word) “ 4 - byte variable

 DQ (Define and Word) “ 8 - byte variable

 DT (Define Ten Byte) “ 10 – byte

* 1. **Program Data**
		1. **Memory Reference**

The 8086/8088 supports instructions that can transfer data from its registers to memory or from memory to its registers. It also supports instructions that can perform some basic arithmetic operations directly on data stored in memory.

The general scheme for performing an operation on a datum in memory involves specifying a memory reference as an operand in an assembly language instruction.

Only five kinds of elements can be used as operands for instructions in 8086/8088 assembly language: registers, immediate values, labels, procnames and memory references.

In assembly language, the address of any datum in memory can be expressed in any one of several different formats. Those formats constituted the addressing mode of the processor.

Each datum in memory is identified by two parameters, its type and its address. The type of a datum specifies the number of bytes memory it occupies. The 8086/8088 supports five different types of data.

**DATA TYPE LENGTH OF DATUM**

 BYTE 1 BYTE

 WORD 2 BYTES

 DWORD 4 BYTES

 QWORD 8 BYTES

 TBYTE 10 BYTES

Every memory reference must consist of three components

* A data type (BYTE, WORD, DWORD, QWORD, OR TBYTE)
* A segment register (CS, DS, ES, QS)
* An Offset (a numeric value in the range $0000 through $FFFF

The general format for expressing a memory reference as an operand in an instruction is

 **{type} {Sreg:} offset**

The offset component of a memory reference can be expressed in terms of a displacement, in terms of an index register, in terms of a base register, or in terms of some combination of three.

The square brackets surrounding the offset component of a memory reference are permissible but unnecessary, unless omitting them would result in an ambiguity. Brackets may be necessary to distinguish a displacement form an immediate value

 MOV AX, [$0006] move the datum at offset $0006 to AX

 MOV AX, $0006 move the immediate value, $0006 to AX.

Or to distinguish a memory reference via a register form the contents of that register.

 MOV AX, [DI] move the datum at offset DI to AX

 MOV AX, DI move the contents of the DI register to AX

There are two index registers: the DI register and the SI register, and there are two base registers the BX and the BP registers.

The assembler must be able to identify the type of a memory reference in order to determine exactly which machine language instruction corresponds to a given assembly language instruction.

**Example:**

In the case of INC [SI] the operand [SI] is ambiguous. It may mean a byte sized or a word sized unit of memory. The assembler must know the data type assigned to the datum at offset SI.

In the case of MOV BX, [SI] the assembler assume that the programmer refers the word of memory at offset SI and not the byte.

When an operand’s type is not given implicitly by its context, it must be specified explicitly with a type pointer. The assembler supports five data type pointers:

 BYTE PTR

 WORD PTR

 DWORD PTR

 QWORD PTR

 TBYTE PTR

If a memory reference does not designate a segment register by name, the CPU will select one by default according to the following rule: If the BP register appears as a component in the memory reference then the SS(stack segment) register will be used; if the BP register does not appear as a component in the memory reference, then the DS (Data Segment) register will be used.

* + 1. **Variables**

A variable is a datum in memory that has been given a name and that may be referred to by name in an instruction.

Variables are created and named and can be initialized by data allocation statements. The general format for a data allocation statement is:

 **{varname} data\_definition\_type {init ,{init}}**

***varname*** is an identifier.

The ***data\_definition\_type*** is one of the following:

 DB (Define Byte) allocates 1 - byte variable

 DW (Define Word) “ 2 - byte variable

 DD (Define Double Word) “ 4 - byte variable

 DQ (Define and Word) “ 8 - byte variable

 DT (Define Ten Byte) “ 10 - byte

 ***init***  is an initial value

* + 1. **Array**

You can allocate an array by specifying more than one init value in a data allocation statement. The statement:

 An\_array DB $11, $00, $44, $33

creates an array named An\_array, allocates 4 bytes to it, and initializes them with values of $11, $00, $44, and $33, respectively.

If an array contains more elements that can be entered on a single statement, it can be broken up and entered in two or more statements.

An\_array DB $11, $00

 DB $44, $33

**DUP** Construct

The DUP(DUPlicate) construct facilitates initialization of large arrays. The general format for the Dup construct is

 **expr DUP (init)**

Where expr is a numeric expression for the number of elements to be allocated and initialized, and where init is the initial value to be given to each of them.

Ex: To create an array of $ 10 bytes, each initialized to $00, the appropriate statement would be

 Array\_name DB $10 DUP ($00)

String constant can also be duplicated. The statement:

 Two\_hellos DB $01 DUP (‘Hello’)

is equivalent to

 Two\_hellos DB ‘Hello Hello’

Constant, text and Dup construct can be freely interleaved in the same statement.

 STUFF DB $03, $02 DUP ($00), ‘Hello,

The **SIZE** and **LENGTH** of an Array.

***SIZE*** and ***LENGTH*** are assembler operators that return the number of bytes associated with an array and the number of data elements associated with an array respectively

 Arrayone DB $04, $06 DUP ($00)

 Arraytwo DW $04, $06 DUP ($00)

Each of the statements allocates an array of seven elements that is, both arrays are of ***LENGTH*** $07. The seven elements in Arrayone occupy seven bytes, so Arrayone is of ***SIZE*** $07. But the seven elements in Arraytwo occupy 14 bytes, so Arraytwo is of SIZE $0E.

* + 1. **Program Constants**

A program constant is oriented with an ***EQU*** (Equate) directive. The general format for an ***EQU*** directive is:

 **identifier EQU expression.**

* 1. **Detail of this Chapter (Optional Part)**

The previous chapters discussed the basic format for data in memory. It also covered how a computer system physically organizes that data. This section finishes this discussion by connecting the concept of *data representation* to its actual physical representation. As the title implies, this chapter concerns itself with two main topics: variables and data structures. This chapter does not assume that you’ve had a formal course in data structures, though such experience would be useful.

* + 1. **Some Additional Instructions: LEA, LES, ADD, and MUL**

The purpose of this sectin is not to present the 80x86 instruction set. However, there are four additional instructions (above and beyond mov ) that will prove handy in the discussion throughout the rest of this chapter. These are the *load effective address* (lea), *load* es *and general purpose register* (les), *addition*(add), and*multiply*(mul). These instructions,along with the movinstruction, provide all the necessary power to access the differentdata types this section discusses.

The lea instruction takes the form: lea reg16, memory *reg16* is a 16 bit general purpose register. *Memory* is a memory location represented by a mod/reg/rm byte (except it must be a memory location, it cannot be a register).

This instruction loads the 16 bit register with the offset of the location speciÞed by the memory operand. lea ax,1000h[bx][si], for example, would load ax with the address of the memory location pointed at by 1000h[bx][si]. This, of course, is the value 1000h+bx+si. Lea is also quite useful for obtaining the address of a variable. If you have a variable I somewhere in memory, lea bx,I will load the bx register with the address (offset) of I.

The les instruction takes the form les reg16, memory32 .This instruction loads the es register and one of the 16 bit general purpose registers from the specified memory address. Note that any memory address you can specify with a mod/reg/rm byte is legal but like the lea instruction it must be a memory location, not a register.

The les instruction loads the specified general purpose register from the word at the given address, it loads the es register from the following word in memory. This instruction, and it’s companion lds (which loads ds) are the only instructions on pre-80386 machines that manipulate 32 bits at a time.

The add instruction, like it’s x86 counterpart, adds two values on the 80x86. This instruction takes several forms. There are five forms that concern us here. They are

add reg, reg

add reg, memory

add memory, reg

add reg, constant

add memory, constant

All these instructions add the second operand to the first leaving the sum in the first operand.

For example, add bx,5 computes bx = bx + 5. The last instruction to look at is the mul (multiply) instruction. This instruction has only a single operand and takes the form:

mul reg/memory

There are many important details concerning mul that this chapter ignores. For the sake of the discussion that follows, assume that the register or memory location is a 16 bit register or memory location. In such a case this instruction computes dx:ax =ax\*reg/mem. Note that there is no immediate mode for this instruction.

* + 1. **Declaring and Accessing Scalar Variables**

Scalar variables hold single values. The variables I and j in the preceding section are examples of scalar variables. Examples of data structures that are not scalars include arrays, records, sets, and lists. These latter data types are made up from scalar values. They are the *composite types*. You’ll see the composite types a little later; first you need to learn to deal with the scalar types.

To declare a variable in dseg, you would use a statement something like the following:

ByteVar byte ?

ByteVar is a *label*. It should begin at column one on the line somewhere in the dseg segment (that is, between the dseg segment and dseg ends statements). You’ll find out all about labels in a few chapters, for now you can assume that most legal Pascal/C/Ada identiÞers are also valid assembly language labels.

If you need more than one variable in your program, just place additional lines in the dseg segment declaring those variables. MASM will automatically allocate a unique storage location for the variable (it wouldn’t be too good to have I and j located at the same address now, would it?).

After declaring said variable, MASM will allow you to refer to that variable *by name* rather than by location in your program. For example, after inserting the above statement into the data segment (dseg), you could use instructions like mov ByteVar,al in your program.

The first variable you place in the data segment gets allocated storage at location DS:0. The next variable in memory gets allocated storage just beyond the previous variable. For example, if the variable at location zero was a byte variable, the next variable gets allocated storage at DS:1.

However, if the first variable was a word, the second variable gets allocated storage at location DS:2. MASM is always careful to allocate variables in such a manner that they do not overlap. Consider the following dseg definition:

dseg segment para public ‘data’

bytevar byte ? ;byte allocates bytes

wordvar word ? ;word allocates words

dwordvar dword ? ;dword allocs dbl words

byte2 byte ?

word2 word ?

dseg ends

MASM allocates storage for *bytevar* at location DS:0. Because *bytevar* is one byte long, the next available memory location is going to be DS:1. MASM, therefore, allocates storage

For *wordvar* at location DS:1. Since words require two bytes, the next available memory location after *wordvar* is DS:3 which is where MASM allocates storage for *dwordvar*. *Dwordvar* is four bytes long, so MASM allocates storage for *byte2* starting at location DS:7. Likewise,

MASM allocates storage for *word2* at location DS:8. Were you to stick another variable after *word2*, MASM would allocate storage for it at location DS:0A. Whenever you refer to one of the names above, MASM automatically substitutes the appropriate offset. For example, MASM would translate the mov ax,wordvar instruction into mov ax,ds:[1].

So now you can use symbolic names for your variables and completely ignore the fact that these variables are really memory locations with corresponding offsets into the data segment.

* + - 1. **Declaring and using BYTE Variables**

There are three main statements for declaring byte variables in a program. They are:

identifier db ?

identifier byte ?

and

identifier sbyte ?

*identiÞer* represents the name of your byte variable. ‘db’ is an older term that predates MASM 6.x. You will see this directive used quite a bit by other programmers (especially those who are not using MASM 6.x or later) but Microsoft considers it to be an obsolete term; you should always use the byte and sbyte declarations instead. The byte declaration declares unsigned byte variables. You should use this declaration for all byte variables *except* small signed integers. For signed integer values, use the sbyte (signed byte) directive.

Once you declare some byte variables with these statements, you may reference those variables within your program by their names:

i db ?

j byte ?

k sbyte ?

.

.

.

mov i, 0

mov j, 245

mov k, -5

mov al, i

mov j, al

etc.

Although MASM 6.x performs a small amount of type checking, you should not get the idea that assembly language is a strongly typed language. In fact, MASM 6.x will only check the values you’re moving around to verify that they will *fit* in the target location. All of the following are legal in MASM 6.x:

mov k, 255

mov j, -5

mov i, -127

Since all of these variables are byte-sized variables, and all the associated constants will fit into eight bits, MASM happily allows each of these statements. Yet if you look at them, they are logically incorrect. What does it mean to move -5 into an unsigned byte variable?

Since signed byte values must be in the range -128É127, what happens when you store the value 255 into a signed byte variable? Well, MASM simply converts these values to their eight bit equivalents (-5 becomes 0FBh, 255 becomes 0FFh [-1], etc.).

* + - 1. **Declaring and using WORD Variables**

You use the dw, word, and sword statements to declare word variables. The following examples demonstrate their use:

NoSignedWord dw ?

UnsignedWord word ?

SignedWord sword ?

Initialized0 word 0

InitializedM1 sword -1

InitializedBig word 65535

InitializedOfs dw NoSignedWord

Most of these declarations are slight modiÞcations of the byte declarations you saw in the last section. Of course you may initialize any word variable to a value in the range -32768É65535 (the union of the range for signed and unsigned 16 bit constants). The last declaration above, however, is new. In this case a label appears in the operand field (specifiÞcally, the name of the NoSignedWord variable). When a label appears in the operand field the assembler will substitute the offset of that label (within the variableÕs segment). If these were the only declarations in *dseg* and they appeared in this order, the last declaration above would initialize *InitializedOfs* with the value zero since *NoSignedWord*Õs offset is zero within the data segment. This form of initialization is quite useful for initializing *pointers*. But more on that subject later.

* + - 1. **Declaring and using DWORD Variables**

You may use the dd, dword, and sdword instructions to declare four-byte integers, pointers, and other variables types. Such variables will allow values in the range 2,147,483,648É4,294,967,295 (the union of the range of signed and unsigned four-byte integers). You use these declarations like the word declarations:

NoSignedDWord dd ?

UnsignedDWord dword ?

SignedDWord sdword ?

InitBig dword 4000000000

InitNegative sdword -1

InitPtr dd InitBig

The last example initializes a double word pointer with the segment:offset address of the InitBig variable.

Once again, it’s worth pointing out that the assembler doesn’t check the types of these variables when looking at the initialization values. If the value fits into 32 bits, the assembler will accept it. Size checking, however, is strictly enforced. Since the only 32 bit mov instructions on processors earlier than the 80386 are les and lds, you will get an error if you attempt to access dword variables on these earlier processors using a mov instruction. Of course, even on the 80386 you cannot move a 32 bit variable into a 16 bit register, you must use the 32 bit registers. Later, you’ll learn how to manipulate 32 bit variables, even on a 16 bit processor. Until then, just pretend that you can’t.

* + - 1. **Declaring and using FWORD, QWORD, and TBYTE Variables**

MASM 6.x also lets you declare six-byte, eight-byte, and ten-byte variables using the df/fword*,* dq/qword*,* and dt/tbyte statements. Declarations using these statements were originally intended for floating point and BCD values. There are better directives for the floating point variables and you don’t need to concern yourself with the other data types you’d use these directives for. The following discussion is for completeness’ sake.

The df/fword statement’s main utility is declaring 48 bit pointers for use in 32 bit protected mode on the 80386 and later. Although you could use this directive to create an arbitrary six byte variable, there are better directives for doing that. You should only use this directive for 48 bit far pointers on the 80386.

dq/qword lets you declare *quadword* (eight byte) variables. The original purpose of this directive was to let you create 64 bit double precision floating point variables and 64 bit integer variables. There are better directives for creating floating point variables. As for 64 bit integers, you won’t need them very often on the 80x86 CPU (at least, not until Intel releases a member of the 80x86 family with 64 bit general purpose registers).

The dt/tbyte directives allocate ten bytes of storage. There are two data types indigenous to the 80x87 (math coprocessor) family that use a ten byte data type: ten byte BCD values and extended precision (80 bit) floating point values. This text will pretty much ignore the BCD data type. As for the floating point type, once again there is a better way to do it.

* + 1. **Composite Data Types**

Composite data types are those that are built up from other (generally scalar) data types. An array is a good example of a composite data type Ð it is an aggregate of elements all the same type. Note that a composite data type need not be composed of scalar data types, there are arrays of arrays for example, but ultimately you can decompose a composite data type into some primitive, scalar, types.

This section will cover two of the more common composite data types: arrays and records. It’s a little premature to discuss some of the more advanced composite data types.

* + - 1. **Declaring Arrays in Your Data Segment**

****

Figure 4.1 Single Dimension Array Implementation

Before you access elements of an array, you need to set aside storage for that array. Fortunately, array declarations build on the declarations youÕve seen so far. To allocate *n* elements in an array, you would use a declaration like the following:

arrayname basetype n dup (?)

*Arrayname* is the name of the array variable and *basetype* is the type of an element of that array. This sets aside storage for the array. To obtain the base address of the array, just use *arrayname*.

The n dup (?) operand tells the assembler to duplicate the object inside the parentheses *n* times. Since a question mark appears inside the parentheses, the definition above would create *n* occurrences of an uninitialized value. Now let’s look at some specific examples:

CharArray char 128 dup (?) ;array[0..127] of char

IntArray integer 8 dup (?) ;array[0..7] of integer

BytArray byte 10 dup (?) ;array[0..9] of byte

PtrArray dword 4 dup (?) ;array[0..3] of dword

The first two examples, of course, assume that you’ve used the typedef statement to define the char and integer data types.

These examples all allocate storage for uninitialized arrays. You may also specify that the elements of the arrays be initialized to a single value using declarations like the following:

RealArray real4 8 dup (1.0)

IntegerAry integer 8 dup (1)

These definitions both create arrays with eight elements. The Þrst deÞnition initializes each four-byte real value to 1.0, the second declaration initializes each integer element to one. This initialization mechanism is fine if you want each element of the array to have the same value. What if you want to initialize each element of the array with a (possibly) different value? Well, that is easily handled as well. The variable declaration statements you’ve seen thus far offer yet another initialization form:

name type value1, value2, value3, …, valuen

This form allocates *n* variables of type *type*. It initializes the Þrst item to *value1*, the second item to *value2*, etc. So by simply enumerating each value in the operand Þeld, you can create an array with the desired initial values. In the following integer array, for example, each element contains the square of its index:

Squares integer 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100

If your array has more elements than will Þt on one line, there are several ways to continue the array onto the next line. The most straight-forward method is to use another integer statement *but without a label*:

Squares integer 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100

integer 121, 144, 169, 196, 225, 256, 289, 324

integer 361, 400

Another option, that is better in some circumstances, is to use a backslash at the end of each line to tell MASM 6.x to continue reading data on the next line:

Squares integer 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, \

121, 144, 169, 196, 225, 256, 289, 324, \

361, 400

Of course, if your array has several thousand elements in it, typing them all in will not be very much fun. Most arrays initialized this way have no more than a couple hundred entries, and generally far less than 100.

You need to learn about one final technique for initializing single dimension arrays before moving on. Consider the following declaration:

BigArray word 256 dup (0,1,2,3)

This array has 1024 elements, not 256. The n dup (xxxx) operand tells MASM to duplicate xxxx *n* times, not create an array with *n* elements. If xxxx consists of a single item, then the dup operator will create an *n* element array. However, if xxxx contains two items separated by a comma, the dup operator will create an array with 2\**n* elements. If xxxx contains three items separated by commas, the dup operator creates an array with 3\**n* items, and so on. Since there are four items in the parentheses above, the dup operator creates 256\*4 or 1024 items in the array. The values in the array will initially be 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 ...

* + - 1. **Accessing Elements of Array**

To access an element of a zero-based array, you can use the simplified formula:

Element\_Address = Base\_Address + index \* Element\_Size

For the Base\_Address entry you can use the name of the array (since MASM associates the address of the Þrst operand with the label). The Element\_Size entry is the number of bytes for each array element. If the object is an array of bytes, the Element\_Size Þeld is one (resulting in a very simple computation). If each element of the array is a word (or integer, or other two-byte type) then Element\_Size is two. And so on. To access an element of the Squares array in the previous section, youÕd use the formula:

Element\_Address = Squares + index\*2

The 80x86 code equivalent to the statement AX:=Squares[index] is

mov bx, index

add bx, bx ;Sneaky way to compute 2\*bx

mov ax, Squares [bx]

There are two important things to notice here. First of all, this code uses the add instruction rather than the mul instruction to compute 2\*index. The main reason for choosing add is that it was more convenient (remember, mul doesnÕt work with constants and it only operates on the ax register). It turns out that add is a *lot* faster than mul on many processors, but since you probably didnÕt know that, it wasnÕt an overriding consideration in the choice of this instruction.

The second thing to note about this instruction sequence is that it does not explicitly compute the sum of the base address plus the index times two. Instead, it relies on the indexed addressing mode to implicitly compute this sum. The instruction mov ax, Squares[bx] loads ax from location Squares+bx which is the base address plus index\*2 (since bx contains index\*2). Sure, you could have used

lea ax, Squares

add bx, ax

mov ax, [bx]

in place of the last instruction, but why use three instructions where one will do the same job? This is a good example of why you should know your addressing modes inside and out. Choosing the proper addressing mode can reduce the size of your program, thereby speeding it up.