Defining Data in Assembly Language

Data Types

Assembly language defines intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type. The essential characteristic of each type is its size in bits: 8, 16, 32, 48, 64, and 80. Other characteristics (such as signed, pointer, or floating-point) are optional and are mainly for the benefit of programmers who want to be reminded about the type of data held in the variable. A variable declared as DWORD, for example, logically holds an unsigned 32-bit integer. In fact, it could hold a signed 32-bit integer, a 32-bit single precision real, or a 32-bit pointer. The assembler is not case sensitive, so a directive such as DWORD can be written as **dword**, **Dword**, **dWord**, and so on.

Data Definition Statement

A data definition statement sets aside storage in memory for a variable, with an optional name. Data definition statements create variables based on intrinsic data types such as BYTE, WORD, DWORD SBYTE, SWORD etc. A data definition has the following syntax:

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

This is an example of a data definition statement:

count DWORD 12345

Name: The optional name assigned to a variable must conform to the rules for identifiers

Directive: The directive in a data definition statement can be BYTE, WORD, DWORD, SBYTE, SWORD, or any other types.

Initializer: At least one *initializer* is required in a data definition, even if it is zero. Additional initializers, if any, are separated by commas. For integer data types, *initializer* is an integer constant or expression matching the size of the variable's type, such as BYTE or WORD. If one prefer to leave the variable uninitialized (assigned a random value), the **?** symbol can be used as the initializer. All initializers, regardless of their format, are converted to binary data by the assembler. Initializers such as 00110010b, 32h, and 50d all end up being having the same binary value.

Defining BYTE and SBYTE Data

The BYTE (define byte) and SBYTE (define signed byte) directives allocate storage for one or more unsigned or signed values. Each initializer must fit into 8 bits of storage. For example,

```
value1 BYTE 'A' ; character constant
value2 BYTE 0 ; smallest unsigned byte
value3 BYTE 255 ; largest unsigned byte
value4 SBYTE -128 ; smallest signed byte
value5 SBYTE +127 ; largest signed byte
```

A question mark (?) initializer leaves the variable uninitialized, implying it will be assigned a value at runtime:

```
value6 BYTE ?
```

Multiple Initializers

If multiple initializers are used in the same data definition, its label refers only to the offset of the first initializer. In the following example, assume **list** is located at offset 0000. If so, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offset 0003:

list BYTE 10,20,30,40

Memory Layout of a Byte Sequence.

Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

Within a single data definition, its initializers can use **different radixes.** Character and string constants can be freely mixed. In the following example, **list1** and **list2** have the same contents:

```
list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h
```

Defining Strings

To define a string of characters, enclose them in single or double quotation marks. The most common type of string ends with a null byte (containing 0). Called a null-terminated string, strings of this type are used in many programming languages:

```
greeting1 BYTE "Good afternoon",0
greeting2 BYTE 'Good night',0
```

Strings are an exception to the rule that byte values must be separated by commas. Without that exception, **greeting1** would have to be defined as `greeting1 BYTE 'G','o','o','d'`....etc. which would be exceedingly tedious. A string can be divided between multiple lines without having to supply a label for each line:

```
greeting1 BYTE "Welcome to the Encryption Demo program "
BYTE "created by Kip Irvine.",0dh,0ah
BYTE "If you wish to modify this program, please "
BYTE "send me a copy.",0dh,0ah,0
```

The hexadecimal codes 0Dh and 0Ah are alternately called CR/LF (carriage-return line-feed) or end-of-line characters. When written to standard output, they move the cursor to the left column of the line following the current line. The line continuation character (\) concatenates two source code lines into a single statement. It must be the last character on the line. The following statements are equivalent:

```
greeting1 BYTE "Welcome to the Encryption Demo program "
and
greeting1 \
BYTE "Welcome to the Encryption Demo program "
```

DUP Operator

The *DUP* operator allocates storage for multiple data items, using a constant expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data:

```
BYTE 20 DUP(0) ; 20 bytes, all equal to zero
BYTE 20 DUP(?) ; 20 bytes, uninitialized
BYTE 4 DUP("STACK") ; 20 bytes: "STACKSTACKSTACKSTACK"
```

Example 2

```
TITLE Add and Subtract, Version 2 (AddSub2.asm)
; This program adds and subtracts 32-bit unsigned
; integers and stores the sum in a variable.
INCLUDE Irvine32.inc
.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?
.code
main PROC
mov eax,val1 ; start with 10000h
add eax,val2 ; add 40000h
sub eax,val3 ; subtract 20000h
mov finalVal,eax ; store the result (30000h)
call DumpRegs ; display the registers
exit
main ENDP
END main
```

How does it work? First, the integer in **val1** is moved to EAX:

mov eax,val1 ; start with 10000h

Next, **val2** is added to EAX:

add eax,val2 ; add 40000h

Next, **val3** is subtracted from EAX:

sub eax,val3 ; subtract 20000h

EAX is copied to **finalVal**:

mov finalVal,eax ; store the result (30000h)

Defining WORD and SWORD Data

The WORD (define word) and SWORD (define signed word) directives create storage for one or more 16-bit integers:

```
word1 WORD 65535 ; largest unsigned value
word2 SWORD -32768 ; smallest signed value
word3 WORD ? ; uninitialized, unsigned
```

Alternatively the DW directive can also be used:

```
val1 DW 65535 ; unsigned
val2 DW -32768 ; signed
```

EQU and TEXTEQU Directives

EQU Directive

The *EQU directive* associates a symbolic name with an integer expression or some arbitrary text. There are three formats:

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

In the first format, *expression* must be a valid integer expression. In the second format, *symbol* is an existing symbol name, already defined with = or EQU. In the third format, any text may appear within the brackets <. . .>. When the assembler encounters *name* later in the program, it substitutes the integer value or text for the symbol. For example, can be defined using EQU:

```
PI EQU <3.1416>
```

```
pressKey EQU <"Press any key to continue...",0>
```

which is invoke below;

```
.data
prompt BYTE pressKey
```

TEXTEQU Directive

The *TEXTEQU directive*, similar to EQU, creates what is known as a *text macro*. There are three different formats: the first assigns text, the second assigns the contents of an existing text macro, and the third assigns a constant integer expression:

```
name TEXTEQU <text>
name TEXTEQU textmacro
name TEXTEQU %constExpr
```

For example, the **prompt1** variable uses the **continueMsg** text macro:

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
.data
prompt1 BYTE continueMsg
```

Text macros can build on each other. In the next example, **count** is set to the value of an integer expression involving **rowSize**. Then the symbol **move** is defined as **mov**. Finally, **setupAL** is built from **move** and **count**:

```
rowSize = 5
count TEXTEQU %(rowSize * 2)
move TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
```

Therefore, the statement setupAL would be assembled as mov al,10. A symbol defined by TEXTEQU can be redefined at any time.

Example 3 (AddSub3)

The following program implements various arithmetic expressions using the ADD, SUB, INC, DEC, and NEG instructions, and shows how certain status flags are affected:

```
TITLE Addition and Subtraction (AddSub3.asm)
INCLUDE Irvine32.inc
.data
Rval SDWORD ?
Xval SDWORD 26
Yval SDWORD 30
Zval SDWORD 40
.code
main PROC
; INC and DEC
mov ax,1000h
inc ax ; 1001h
dec ax ; 1000h
; Expression: Rval = -Xval + (Yval - Zval)
mov eax,Xval
neg eax ; -26
mov ebx,Yval
sub ebx,Zval ; -10
add eax,ebx
mov Rval,eax ; -36
; Zero flag example:
mov cx,1
sub cx,1 ; ZF = 1
mov ax,0FFFFh
inc ax ; ZF = 1
; Sign flag example:
mov cx,0
sub cx,1 ; SF = 1
mov ax,7FFFh
add ax,2 ; SF = 1
; Carry flag example:
mov al,0FFh
add al,1 ; CF = 1, AL = 00
; Overflow flag example:
mov al,+127
add al,1 ; OF = 1
mov al,-128
sub al,1 ; OF = 1
exit
```

```
main ENDP
END main
```

JMP and LOOP Instructions

By default, the CPU loads and executes programs sequentially. But the current instruction might be conditional, meaning that it transfers control to a new location in the program based on the values of CPU status flags (Zero, Sign, Carry, etc.). Assembly language programs use conditional instructions to implement high-level statements such as IF statements and loops. Each of the conditional statements involves a possible transfer of control (jump) to a different memory address. A transfer of control, or branch, is a way of altering the order in which statements are executed. There are two basic types of transfers:

• **Unconditional Transfer:** Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address. The JMP instruction does this.

• **Conditional Transfer:** The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

JMP Instruction

The JMP instruction causes an unconditional transfer to a destination, identified by a code label that is translated by the assembler into an offset. The syntax is

```
JMP destination
```

When the CPU executes an unconditional transfer, the offset of *destination* is moved into the instruction pointer, causing execution to continue at the new location.

Creating a Loop: The JMP instruction provides an easy way to create a loop by jumping to a label at the top of the loop:

```
top:
; instructions
; instructions
.
.
jmp top ; repeat the endless loop
```

JMP is unconditional, so a loop like this will continue endlessly unless another way is found to exit the loop.

LOOP Instruction

The LOOP instruction, formally known as Loop According to ECX Counter, repeats a block of statements a specific number of times. ECX is automatically used as a counter and is decremented each time the loop repeats. Its syntax is

```
LOOP destination
```

The execution of the LOOP instruction involves two steps: First, it subtracts 1 from ECX. Next, it compares ECX to zero. If ECX is not equal to zero, a jump is taken to the label identified by destination. Otherwise, if ECX equals zero, no jump takes place, and control passes to the instruction following the loop. In the following example, we add 1 to AX each time the loop repeats. When the loop ends,

```
AX =5 and ECX = 0:
mov ax,0
mov ecx,5
L1:
inc ax; increase content of ax by 1
loop L1
```
Rarely should one explicitly modify ECX inside a loop. If one do, the LOOP instruction may not work as expected.

Nested Loops: When creating a loop inside another loop, special consideration must be given to the outer loop counter in ECX. Instead save it in a variable:

```
.data
count DWORD ?
.code
mov ecx,100 ; set outer loop count
L1:
mov count,ecx ; save outer loop count
mov ecx,20 ; set inner loop count
L2:
.
.
loop L2 ; repeat the inner loop for 20times
mov ecx,count ; restore outer loop count i.e 100 times
loop L1 ; repeat the outer loop
```

Arrays

In assembly language, you would follow these steps in working with an array variable:
1. Assign the array's address to a register that will serve as an indexed operand.
2. Initialize the loop counter to the length of the array.
3. Assign zero to the register that accumulates the sum.
4. Create a label to mark the beginning of the loop.
5. In the loop body, add a single array element to the sum.
6. Point to the next array element.
7. Use a LOOP instruction to repeat the loop.

Steps 1 through 3 may be performed in any order. Here's a short program that sums an array of 16-bit integers.

```
TITLE Summing an Array (SumArray.asm)
INCLUDE Irvine32.inc
.data
intarray DWORD 10000h,20000h,30000h,40000h
.code
main PROC
mov edi,OFFSET intarray ; 1: EDI = address of intarray
```

```
mov ecx,LENGTHOF intarray ; 2: initialize loop counter
mov eax,0 ; 3: sum = 0
L1: ; 4: mark beginning of loop
add eax,[edi] ; 5: add an integer
add edi,TYPE intarray ; 6: point to next element
loop L1 ; 7: repeat until ECX = 0
exit
main ENDP
END main
```