Lecture No.7

Lecture Outlines

3.4 Defining Data

3.4.1 Intrinsic Data Types

The assembler recognizes a basic set of *intrinsic data types,* which describe types in terms of their size (byte, word, doubleword, and so on), whether they are signed, and whether they are integers or reals. There's a fair amount of overlap in these types—for example, the DWORD type (32-bit, unsigned integer) is interchangeable with the SDWORD type (32-bit, signed integer). You might say that programmers use SDWORD to communicate to readers that a value will contain a sign, but there is no enforcement by the assembler. The assembler only evaluates the sizes of operands. So, for example, you can only assign variables of type DWORD, SDWORD, or REAL4 to a 32-bit integer. Table 3-2 contains a list of all the intrinsic data types. The notation IEEE in some of the table entries refers to standard real number formats published by the IEEE Computer Society.

3.4.2 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 (Table 3-2). A data definition has the following syntax:

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

Table 3-2 Intrinsic Data Types.

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer. D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

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 (Section 3.1.8).

Directive The directive in a data definition statement can be BYTE, WORD, DWORD, SBYTE, SWORD, or any of the types listed in Table 3-2. In addition, it can be any of the legacy data definition directives shown in Table 3-3.

Table 3-3 Legacy Data Directives.

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

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 literal or integer expression matching the size of the variable's type, such as BYTE or WORD. If you 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 have the same binary value.

3.4.3 Adding a Variable to the AddTwo Program

Let's create a new version of the *AddTwo* program we introduced at the beginning of this chapter, which we will now call *AddTwoSum*. This version introduces a variable named **sum**, which appears in the complete program listing:

```
 1: ; AddTwoSum.asm - Chapter 3 example
 2:
 3:    .386
 4:    .model flat,stdcall
 5:    .stack 4096
 6:    ExitProcess PROTO, dwExitCode:DWORD
 7:
 8:    .data
 9:    sum DWORD 0
10:
11:    .code
12:    main PROC
13:       mov eax,5
14:       add eax,6
15:       mov sum,eax
16:
17:       INVOKE ExitProcess,0
18:    main ENDP
19:    END main
```

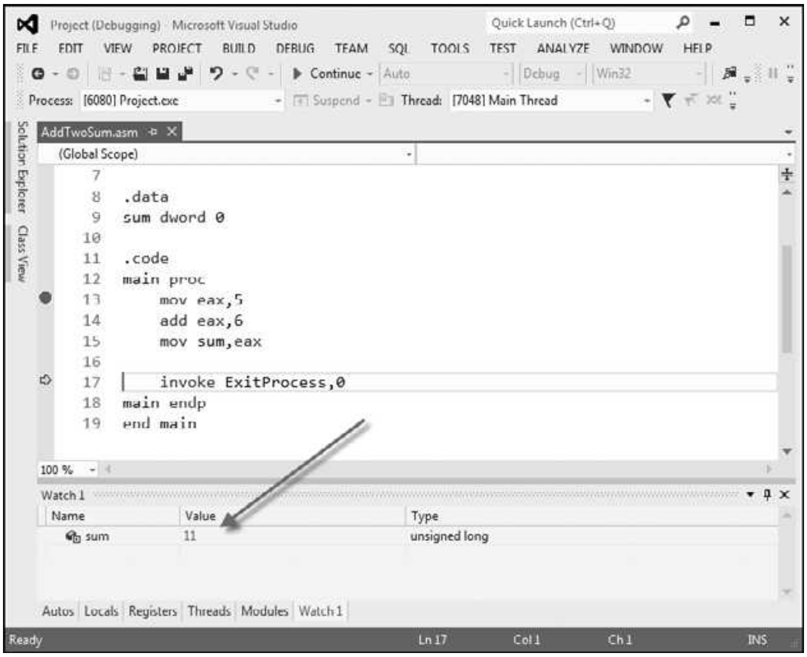
You can run this in the debugger by setting a breakpoint on line 13 and stepping through the program one line at a time. After executing line 15, hover the mouse over the **sum** variable to see its value. Or, you can open a Watch window. To do that, select *Windows* from the Debug menu (during a debugging session), select *Watch*, and select one of the four available choices (*Watch1, Watch2, Watch3*, or *Watch4*). Then, highlight the **sum** variable with the mouse and drag it into the Watch window. Figure 3-10 shows a sample, with a large arrow pointing at the current value of **sum** after executing line 15.

3.4.4 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 literal
value2 BYTE   0                ; smallest unsigned byte
value3 BYTE  255               ; largest unsigned byte
value4 SBYTE −128              ; smallest signed byte
value5 SBYTE +127              ; largest signed byte
```

Figure 3–10 Using a *Watch* window in a debugging session.


Project (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Continue
Auto
Debug
Win32
Process: [6080] Project.exe
Suspend
Thread: [7048] Main Thread
Solution Explorer
Class View
AddTwoSum.asm
(Global Scope)
100 %
Watch 1
Autos Locals Registers Threads Modules Watch 1
Ready
Ln 17
Col 1
Ch 1
INS

```

.data
sum dword 0

.code
main proc
    mov eax,5
    add eax,6
    mov sum,eax

    invoke ExitProcess,0
main endp
end main
```

Name	Value	Type
sum	11	unsigned long

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

```
value6 BYTE ?
```

The optional name is a label marking the variable's offset from the beginning of its enclosing segment. For example, if **value1** is located at offset 0000 in the data segment and consumes one byte of storage, **value2** is automatically located at offset 0001:

```
value1 BYTE 10h
value2 BYTE 20h
```

The DB directive can also define an 8-bit variable, signed or unsigned:

```
val1 DB 255                    ; unsigned byte
val2 DB -128                   ; signed byte
```

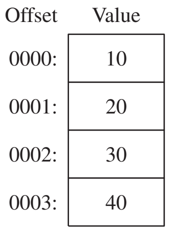
Multiple Initializers

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
```

Figure 3-11 shows **list** as a sequence of bytes, each with its own offset.

Figure 3–11 Memory layout of a byte sequence.



Not all data definitions require labels. To continue the array of bytes begun with **list**, for example, we can define additional bytes on the next lines:

```
list BYTE 10,20,30,40
     BYTE 50,60,70,80
     BYTE 81,82,83,84
```

Within a single data definition, its initializers can use different radixes. Character and string literals 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
```

Each character uses a byte of storage. 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 integer 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"
```

3.4.5 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
```

The legacy DW directive can also be used:

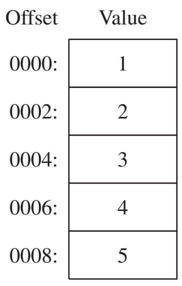
```
val1  DW 65535                ; unsigned
val2  DW -32768               ; signed
```

Array of 16-Bit Words Create an array of words by listing the elements or using the DUP operator. The following array contains a list of values:

```
myList  WORD 1,2,3,4,5
```

Figure 3-12 shows a diagram of the array in memory, assuming **myList** starts at offset 0000. The addresses increment by 2 because each value occupies 2 bytes.

Figure 3–12 Memory layout, 16-bit word array.



The DUP operator provides a convenient way to declare an array:

```
array WORD 5 DUP(?)           ; 5 values, uninitialized
```

3.4.6 Defining DWORD and SDWORD Data

The DWORD directive (define doubleword) and SDWORD directive (define signed doubleword) allocate storage for one or more 32-bit integers:

```
val1 DWORD   12345678h          ; unsigned
val2 SDWORD −2147483648         ; signed
val3 DWORD   20 DUP(?)          ; unsigned array
```

The legacy DD directive can also be used to define doubleword data.

```
val1 DD 12345678h               ; unsigned
val2 DD −2147483648             ; signed
```

The DWORD can be used to declare a variable that contains the 32-bit offset of another variable. Below, **pVal** contains the offset of **val3**:

```
pVal DWORD val3
```

Array of 32-Bt Doublewords Let's create an array of doublewords by explicitly initializing each value:

```
myList DWORD 1,2,3,4,5
```

Figure 3-13 shows a diagram of this array in memory, assuming **myList** starts at offset 0000. The offsets increment by 4.

3.4.7 Defining QWORD Data

The QWORD (define quadword) directive allocates storage for 64-bit (8-byte) values:

```
quad1 QWORD 1234567812345678h
```

The legacy DQ directive can also be used to define quadword data:

```
quad1 DQ 1234567812345678h
```

Figure 3–13 Memory layout, 32-bit doubleword array.

Offset	Value
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

3.4.8 Defining Packed BCD (TBYTE) Data

Intel stores a packed *binary coded decimal* (BCD) integers in a 10-byte package. Each byte (except the highest) contains two decimal digits. In the lower 9 storage bytes, each half-byte holds a single decimal digit. In the highest byte, the highest bit indicates the number's sign. If the highest byte equals 80h, the number is negative; if the highest byte equals 00h, the number is positive. The integer range is −999,999,999,999,999,999 to +999,999,999,999,999,999.

Example The hexadecimal storage bytes for positive and negative decimal 1234 are shown in the following table, from the least significant byte to the most significant byte:

Decimal Value	Storage Bytes
+1234	34 12 00 00 00 00 00 00 00 00
−1234	34 12 00 00 00 00 00 00 00 80

MASM uses the TBYTE directive to declare packed BCD variables. Constant initializers must be in hexadecimal because the assembler does not automatically translate decimal initializers to BCD. The following two examples demonstrate both valid and invalid ways of representing decimal −1234:

```
intVal TBYTE 800000000000001234h       ; valid
intVal TBYTE -1234                      ; invalid
```

The reason the second example is invalid is that MASM encodes the constant as a binary integer rather than a packed BCD integer.

If you want to encode a real number as packed BCD, you can first load it onto the floating-point register stack with the FLD instruction and then use the FBSTP instruction to convert it to packed BCD. This instruction rounds the value to the nearest integer:

```
.data
posVal REAL8 1.5
bcdVal TBYTE ?

.code
fld posVal          ; load onto floating-point stack
fbstp bcdVal        ; rounds up to 2 as packed BCD
```

If **posVal** were equal to 1.5, the resulting BCD value would be 2. In Chapter 7, you will learn how to do arithmetic with packed BCD values.

3.4.9 Defining Floating-Point Types

REAL4 defines a 4-byte single-precision floating-point variable. REAL8 defines an 8-byte double-precision value, and REAL10 defines a 10-byte extended-precision value. Each requires one or more real constant initializers:

```
rVal1      REAL4 -1.2
rVal2      REAL8  3.2E-260
rVal3      REAL10 4.6E+4096
ShortArray REAL4  20 DUP(0.0)
```

Table 3-4 describes each of the standard real types in terms of their minimum number of significant digits and approximate range:

The DD, DQ, and DT directives can define also real numbers:

```
rVal1 DD -1.2                  ; short real
rVal2 DQ  3.2E-260             ; long real
rVal3 DT  4.6E+4096            ; extended-precision real
```

Table 3-4 Standard Real Number Types.

Data Type	Significant Digits	Approximate Range
Short real	6	1.18×10^{-38} to 3.40×10^{38}
Long real	15	2.23×10^{-308} to 1.79×10^{308}
Extended-precision real	19	3.37×10^{-4932} to 1.18×10^{4932}

3.4.10 A Program That Adds Variables

The sample programs shown so far in this chapter added integers stored in registers. Now that you have some understanding of how to declare data, we will revise the same program by making it add the contents of three integer variables and store the sum in a fourth variable.

Clarification: The MASM assembler includes data types such as **real4** and **real8**, suggesting that the values they represent are real numbers. More correctly, the values are floating-point numbers, which have a limited amount of precision and range. Mathematically, a real number has unlimited precision and size.

```
 1: ; AddVariables.asm - Chapter 3 example
 2:
 3:    .386
 4:    .model flat,stdcall
 5:    .stack 4096
 6:    ExitProcess PROTO, dwExitCode:DWORD
 7:
 8:    .data
 9:    firstval  DWORD 20002000h
10:    secondval DWORD 11111111h
11:    thirdval  DWORD 22222222h
12:    sum       DWORD 0
13:
14:    .code
15:    main PROC
16:       mov eax,firstval
17:       add eax,secondval
18:       add eax,thirdval
19:       mov sum,eax
20:
21:       INVOKE ExitProcess,0
22:    main ENDP
23:    END main
```

Notice that we have initialized three variables with nonzero values (lines 9–11). Lines 16–18 add the variables. The x86 instruction set does not let us add one variable directly to another, but it does allow a variable to be added to a register. That is why lines 16–17 use EAX as an accumulator:

```
16:    mov eax,firstval
17:    add eax,secondval
```

After line 17, EAX contains the sum of **firstval** and **secondval**. Next, line 18 adds **thirdval** to the sum in EAX:

```
18:    add eax,thirdval
```

Finally, on line 19, the sum is copied into the variable named **sum**:

```
19:    mov sum,eax
```

As an exercise, we encourage you to run this program in a debugging session and examine each of the registers after each instruction executes. The final sum should be hexadecimal 53335333.

> ***Tip:*** During a debugging session, if you want to display the variable in hexadecimal, do the following: Hover the mouse over a variable or register for a second until a gray rectangle appears under the mouse. Right-click the rectangle and select *Hexadecimal Display* from the popup menu.

3.4.11 Little-Endian Order

x86 processors store and retrieve data from memory using *little-endian* order (low to high). The least significant byte is stored at the first memory address allocated for the data. The remaining bytes are stored in the next consecutive memory positions. Consider the doubleword 12345678h. If placed in memory at offset 0000, 78h would be stored in the first byte, 56h would be stored in the second byte, and the remaining bytes would be at offsets 0002 and 0003, as shown in Figure 3-14.

Figure 3–14 Little-endian representation of 12345678h.

0000:	78
0001:	56
0002:	34
0003:	12

Some other computer systems use *big-endian* order (high to low). Figure 3-15 shows an example of 12345678h stored in big-endian order at offset 0:

Figure 3–15 Big-endian representation of 12345678h.

0000:	12
0001:	34
0002:	56
0003:	78

3.4.12 Declaring Uninitialized Data

The .DATA? directive declares uninitialized data. When defining a large block of uninitialized data, the .DATA? directive reduces the size of a compiled program. For example, the following code is declared efficiently:

```
.data
smallArray DWORD 10 DUP(0)      ; 40 bytes
.data?
bigArray DWORD 5000 DUP(?)      ; 20,000 bytes, not initialized
```

The following code, on the other hand, produces a compiled program 20,000 bytes larger:

```
.data
smallArray DWORD 10 DUP(0)      ; 40 bytes
bigArray DWORD 5000 DUP(?)      ; 20,000 bytes
```

Mixing Code and Data The assembler lets you switch back and forth between code and data in your programs. You might, for example, want to declare a variable used only within a localized area of a program. The following example inserts a variable named **temp** between two code statements:

```
.code
mov eax,ebx
.data
temp DWORD ?
.code
mov temp,eax
. . .
```

Although the declaration of **temp** appears to interrupt the flow of executable instructions, MASM places **temp** in the data segment, separate from the segment holding compiled code. At the same time, intermixing .code and .data directives can cause a program to become hard to read.

3.5 Symbolic Constants

A *symbolic constant* (or *symbol definition*) is created by associating an identifier (a symbol) with an integer expression or some text. Symbols do not reserve storage. They are used only by the assembler when scanning a program, and they cannot change at runtime. The following table summarizes their differences:

	Symbol	Variable
Uses storage?	No	Yes
Value changes at runtime?	No	Yes

We will show how to use the equal-sign directive (=) to create symbols representing integer expressions. We will use the EQU and TEXTEQU directives to create symbols representing arbitrary text.

3.5.1 Equal-Sign Directive

The *equal-sign directive* associates a symbol name with an integer expression (see Section 3.1.3). The syntax is

```
name = expression
```

Ordinarily, expression is a 32-bit integer value. When a program is assembled, all occurrences of *name* are replaced by *expression* during the assembler's preprocessor step. Suppose the following statement occurs near the beginning of a source code file:

```
COUNT = 500
```

Further, suppose the following statement should be found in the file 10 lines later:

```
mov eax, COUNT
```

When the file is assembled, MASM will scan the source file and produce the corresponding code lines:

```
mov eax, 500
```

Why Use Symbols? We might have skipped the COUNT symbol entirely and simply coded the MOV instruction with the literal 500, but experience has shown that programs are easier to read and maintain if symbols are used. Suppose COUNT were used many times throughout a program. At a later time, we could easily redefine its value:

```
COUNT = 600
```

Assuming that the source file was assembled again, all instances of COUNT would be automatically replaced by the value 600.

Current Location Counter One of the most important symbols of all, shown as $, is called the *current location counter*. For example, the following declaration declares a variable named **selfPtr** and initializes it with the variable's offset value:

```
selfPtr DWORD $
```

Keyboard Definitions Programs often define symbols that identify commonly used numeric keyboard codes. For example, 27 is the ASCII code for the Esc key:

```
Esc_key = 27
```

Later in the same program, a statement is more self-describing if it uses the symbol rather than an integer literal. Use

```
mov  al,Esc_key                 ; good style
```

rather than

```
mov  al,27                      ; poor style
```

Using the DUP Operator Section 3.4.4 showed how to use the DUP operator to create storage for arrays and strings. The counter used by DUP should be a symbolic constant, to simplify program maintenance. In the next example, if COUNT has been defined, it can be used in the following data definition:

```
array dword COUNT DUP(0)
```

Redefinitions A symbol defined with = can be redefined within the same program. The following example shows how the assembler evaluates COUNT as it changes value:

```
COUNT = 5
mov al,COUNT                   ; AL = 5
COUNT = 10
mov al,COUNT                   ; AL = 10
COUNT = 100
mov al,COUNT                   ; AL = 100
```

The changing value of a symbol such as COUNT has nothing to do with the runtime execution order of statements. Instead, the symbol changes value according to the assembler's sequential processing of the source code during the assembler's preprocessing stage.

3.5.2 Calculating the Sizes of Arrays and Strings

When using an array, we usually like to know its size. The following example uses a constant named **ListSize** to declare the size of **list**:

```
list BYTE 10,20,30,40
ListSize = 4
```

Explicitly stating an array's size can lead to a programming error, particularly if you should later insert or remove array elements. A better way to declare an array size is to let the assembler calculate its value for you. The $ operator (*current location counter*) returns the offset associated with the current program statement. In the following example, **ListSize** is calculated by subtracting the offset of **list** from the current location counter ($):

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

ListSize must follow immediately after **list**. The following, for example, produces too large a value (24) for **ListSize** because the storage used by **var2** affects the distance between the current location counter and the offset of **list**:

```
list BYTE 10,20,30,40
var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

Rather than calculating the length of a string manually, let the assembler do it:

```
myString  BYTE "This is a long string, containing"
          BYTE "any number of characters"
myString_len = ($ − myString)
```

Arrays of Words and DoubleWords When calculating the number of elements in an array containing values other than bytes, you should always divide the total array size (in bytes) by the size of the individual array elements. The following code, for example, divides the address range by 2 because each word in the array occupies 2 bytes (16 bits):

```
list  WORD  1000h,2000h,3000h,4000h
ListSize = ($ – list) / 2
```

Similarly, each element of an array of doublewords is 4 bytes long, so its overall length must be divided by four to produce the number of array elements:

```
list  DWORD  10000000h,20000000h,30000000h,40000000h
ListSize = ($ –list) / 4
```

3.5.3 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 (see Section 3.1.3). 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.

EQU can be useful when defining a value that does not evaluate to an integer. A real number constant, for example, can be defined using EQU:

```
PI EQU <3.1416>
```

Example The following example associates a symbol with a character string. Then a variable can be created using the symbol:

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

Example Suppose we would like to define a symbol that counts the number of cells in a 10-by-10 integer matrix. We will define symbols two different ways, first as an integer expression and second as a text expression. The two symbols are then used in data definitions:

```
matrix1  EQU   10 * 10
matrix2  EQU  <10 * 10>
.data
M1 WORD matrix1
M2 WORD matrix2
```

The assembler produces different data definitions for **M1** and **M2**. The integer expression in **matrix1** is evaluated and assigned to **M1**. On the other hand, the text in **matrix2** is copied directly into the data definition for **M2**:

```
M1 WORD  100
M2 WORD  10 * 10
```

No Redefinition Unlike the = directive, a symbol defined with EQU cannot be redefined in the same source code file. This restriction prevents an existing symbol from being inadvertently assigned a new value.

3.5.4 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.

3.6 64-Bit Programming

With the advent of 64-bit processors by AMD and Intel, there has been increased interest in 64-bit programming. MASM supports 64-bit code, and the 64-bit version of the assembler is installed with all full versions of Visual Studio 2012 (Ultimate, Premium, or Professional) and with the Visual Studio 2012 Express for Desktop. In each chapter, beginning with this one, we will include 64-bit versions of some of the sample programs. We will also discuss the *Irvine64* subroutine library supplied with this book.

Let's borrow the *AddTwoSum* program shown earlier in this chapter, and modify it for 64-bit programming. We will use the 64-bit register RAX to accumulate two integers, and store their sum in a 64-bit variable:

```
 1: ; AddTwoSum_64.asm - Chapter 3 example.
 2:
 3: ExitProcess PROTO
 4:
 5: .data
 6: sum DWORD 0
 7:
 8: .code
 9: main PROC
10:    mov  eax,5
11:    add  eax,6
12:    mov  sum,eax
13:
14:    mov  ecx,0
15:    call ExitProcess
16: main ENDP
17: END
```

Here's how this program is different from the 32-bit version we showed earlier in the chapter:

- The following three lines, which were in the 32-bit version of the AddTwoSum program are not used in the 64-bit version:

  ```
  .386
  .model flat,stdcall
  .stack 4096
  ```

- Statements using the PROTO keyword do not have parameters in 64-bit programs. This is from Line 3:

  ```
  ExitProcess PROTO
  ```

  This was our earlier 32-bit version:

  ```
  ExitProcess PROTO,dwExitCode:DWORD
  ```

- Lines 14–15 use two instructions to end the program (mov and call). The 32-bit version used an INVOKE statement to do the same thing. The 64-bit version of MASM does not support the INVOKE directive.
- In line 17, the end directive does not specify a program entry point. The 32-bit version of the program did.

Using 64-Bit Registers

In some applications, you may need to perform arithmetic with integers that are larger than 32 bits. In that case, you can use 64-bit registers and variables. For example, this is how we could make our sample program use 64-bit values:

- In line 6, we would change DWORD to QWORD when declaring the **sum** variable.
- In lines 10–12, we would change EAX to its 64-bit version, named RAX.

This is how lines 6–12 would appear after we made the changes:

```
 6: sum QWORD 0
 7:
 8: .code
 9: main PROC
10:    mov  rax,5
11:    add  rax,6
12:    mov  sum,rax
```

Whether you write 32-bit or 64-bit assembly programs is largely a matter of preference. Here's something to remember: the 64-bit version of MASM 11.0 (shipped with Visual Studio 12) does not support the INVOKE directive. Also, you must be running the 64-bit version of Windows in order to run 64-bit programs.

You can find instructions at the author's web site (asmirvine.com) to help you configure Visual Studio for 64-bit programming.