3

Assembly Language Fundamentals

3.1 Basic Language Elements
- 3.1.1 First Assembly Language Program
- 3.1.2 Integer Literals
- 3.1.3 Constant Integer Expressions
- 3.1.4 Real Number Literals
- 3.1.5 Character Literals
- 3.1.6 String Literals
- 3.1.7 Reserved Words
- 3.1.8 Identifiers
- 3.1.9 Directives
- 3.1.10 Instructions
- 3.1.11 Section Review

3.2 Example: Adding and Subtracting Integers
- 3.2.1 The *AddTwo* Program
- 3.2.2 Running and Debugging the AddTwo Program
- 3.2.3 Program Template
- 3.2.4 Section Review

3.3 Assembling, Linking, and Running Programs
- 3.3.1 The Assemble-Link-Execute Cycle
- 3.3.2 Listing File
- 3.3.3 Section Review

3.4 Defining Data
- 3.4.1 Intrinsic Data Types
- 3.4.2 Data Definition Statement
- 3.4.3 Adding a Variable to the AddTwo Program
- 3.4.4 Defining BYTE and SBYTE Data
- 3.4.5 Defining WORD and SWORD Data
- 3.4.6 Defining DWORD and SDWORD Data
- 3.4.7 Defining QWORD Data
- 3.4.8 Defining Packed BCD (TBYTE) Data
- 3.4.9 Defining Floating-Point Types
- 3.4.10 A Program that Adds Variables
- 3.4.11 Little-Endian Order
- 3.4.12 Declaring Uninitialized Data
- 3.4.13 Section Review

3.5 Symbolic Constants
- 3.5.1 Equal-Sign Directive
- 3.5.2 Calculating the Sizes of Arrays and Strings
- 3.5.3 EQU Directive
- 3.5.4 TEXTEQU Directive
- 3.5.5 Section Review

3.6 64-Bit Programming

3.7 Chapter Summary

3.8 Key Terms
- 3.8.1 Terms
- 3.8.2 Instructions, Operators, and Directives

3.9 Review Questions and Exercises
- 3.9.1 Short Answer
- 3.9.2 Algorithm Workbench

3.10 Programming Exercises

This chapter focuses on the basic building blocks of the Microsoft MASM assembler. You will see how constants and variables are defined, standard formats for numeric and string literals, and how to assemble and run your first programs. We particularly emphasize the Visual Studio debugger in this chapter, as an excellent tool for understanding how programs work. The important thing in this chapter is to move one step at a time, mastering each detail before you move to the next step. You are building a foundation that will greatly help you in upcoming chapters.

3.1 Basic Language Elements

3.1.1 First Assembly Language Program

Assembly language programming might have a reputation for being obscure and tricky, but we like to think of it another way—it is a language that gives you nearly total information. You get to see everything that is going on, even in the CPU's registers and flags! With this powerful ability, however, you have the responsibility to manage data representation details and instruction formats. You work at a very detailed level. To see how this works, let's look at a simple assembly language program that adds two numbers and saves the result in a register. We will call it the *AddTwo* program:

```
1: main PROC
2:    mov eax,5                ; move 5 to the eax register
3:    add eax,6                ; add 6 to the eax register
4:
5:    INVOKE ExitProcess,0     ; end the program
6: main ENDP
```

Although line numbers have been inserted in the beginning of each line to aid our discussion, you never actually type line numbers when you create assembly programs. Also, don't try to type in and run this program just yet—it's missing some important declarations that we will include later on in this chapter.

Let's go through the program one line at a time: Line 1 starts the **main** procedure, the entry point for the program. Line 2 places the integer 5 in the **eax** register. Line 3 adds 6 to the value in EAX, giving it a new value of 11. Line 5 calls a Windows service (also known as a function) named **ExitProcess** that halts the program and returns control to the operating system. Line 6 is the ending marker of the main procedure.

You probably noticed that we included comments, which always begin with a semicolon character. We've left out a few declarations at the top of the program that we can show later, but essentially this is a working program. It does not display anything on the screen, but we could run it with a utility program called a *debugger* that would let us step through the program one line at a time and look at the register values. Later in this chapter, we will show how to do that.

Adding a Variable

Let's make our program a little more interesting by saving the results of our addition in a variable named sum. To do this, we will add a couple of markers, or declarations, that identify the code and data areas of the program:

```
 1: .data                     ; this is the data area
 2: sum DWORD 0               ; create a variable named sum
 3:
 4: .code                     ; this is the code area
 5: main PROC
 6:    mov eax,5              ; move 5 to the eax register
 7:    add eax,6              ; add 6 to the eax register
 8:    mov sum,eax
 9:
10:    INVOKE ExitProcess,0   ; end the program
11: main ENDP
```

The **sum** variable is declared on Line 2, where we give it a size of 32 bits, using the DWORD keyword. There are a number of these size keywords, which work more or less like data types. But they are not as specific as types you might be familiar with, such as int, double, float, and so on. They only specify a size, but there's no checking into what actually gets put inside the variable. Remember, you are in total control.

By the way, those code and data areas we mentioned, which were marked by the .code and .data directives, are called *segments*. So you have the code segment and the data segment. Later on, we will see a third segment named **stack**.

Next, let's dive deeper into some of the language details, showing how to declare literals (also known as constants), identifiers, directives, and instructions. You will probably have to read this chapter a couple of times to retain it all, but it's definitely worth the time. By the way, throughout this chapter, when we refer to syntax rules imposed by the assembler, we really mean rules imposed by the Microsoft MASM assembler. Other assemblers are out there with different syntax rules, but we will ignore them. We will probably save at least one tree (somewhere in the world) by not reprinting the word MASM every time we refer to the assembler.

3.1.2 Integer Literals

An *integer literal* (also known as an *integer constant*) is made up of an optional leading sign, one or more digits, and an optional radix character that indicates the number's base:

```
[{+ | - }] digits [ radix ]
```

We will use Microsoft syntax notation throughout the book. Elements within square brackets [..] are optional and elements within braces {..} require a choice of one of the enclosed elements, separated by the | character. Elements in *italics* identify items that have known definitions or descriptions.

So, for example, 26 is a valid integer literal. It doesn't have a radix, so we assume it's in decimal format. If we wanted it to be 26 hexadecimal, we would have to write it as 26h. Similarly, the number 1101 would be considered a decimal value until we added a "b" at the end to make it 1101b (binary). Here are the possible radix values:

h	hexadecimal	r	encoded real
q/o	octal	t	decimal (alternate)
d	decimal	y	binary (alternate)
b	binary		

And here are some integer literals declared with various radixes. Each line contains a comment:

```
26                          ; decimal
26d                         ; decimal
11010011b                   ; binary
42q                         ; octal
42o                         ; octal
1Ah                         ; hexadecimal
0A3h                        ; hexadecimal
```

A hexadecimal literal beginning with a letter must have a leading zero to prevent the assembler from interpreting it as an identifier.

3.1.3 Constant Integer Expressions

A *constant integer expression* is a mathematical expression involving integer literals and arithmetic operators. Each expression must evaluate to an integer, which can be stored in 32 bits (0 through FFFFFFFFh). The arithmetic operators are listed in Table 3-1 according to their precedence order, from highest (1) to lowest (4). The important thing to realize about constant integer expressions is that they can only be evaluated at assembly time. From now on, we will just call them *integer expressions*.

Table 3-1 Arithmetic Operators.

Operator	Name	Precedence Level
( )	Parentheses	1
+, −	Unary plus, minus	2
*, /	Multiply, divide	3
MOD	Modulus	3
+, −	Add, subtract	4

Operator precedence refers to the implied order of operations when an expression contains two or more operators. The order of operations is shown for the following expressions:

```
4 + 5 * 2                   Multiply, add
12 -1 MOD 5                 Modulus, subtract
-5 + 2                      Unary minus, add
(4 + 2) * 6                 Add, multiply
```

The following are examples of valid expressions and their values:

Expression	Value
16 / 5	3
−(3 + 4) * (6 − 1)	−35

Expression	Value
−3 + 4 * 6 − 1	20
25 mod 3	1

Suggestion: Use parentheses in expressions to clarify the order of operations so you don't have to remember precedence rules.

3.1.4 Real Number Literals

Real number literals (also known as *floating-point literals*) are represented as either decimal reals or encoded (hexadecimal) reals. A *decimal real* contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent:

```
[sign]integer.[integer][exponent]
```

These are the formats for the sign and exponent:

```
sign       {+,-}
exponent   E[{+,-}]integer
```

Following are examples of valid decimal reals:

```
2.
+3.0
-44.2E+05
26.E5
```

At least one digit and a decimal point are required.

An *encoded real* represents a real number in hexadecimal, using the IEEE floating-point format for short reals (see Chapter 12). The binary representation of decimal +1.0, for example, is

```
0011 1111 1000 0000 0000 0000 0000 0000
```

The same value would be encoded as a short real in assembly language as

```
3F800000r
```

We will not be using real-number constants for a while, because most of the x86 instruction set is geared toward integer processing. However, Chapter 12 will show how to do arithmetic with real numbers, also known as floating-point numbers. It's very interesting, and very technical.

3.1.5 Character Literals

A *character literal* is a single character enclosed in single or double quotes. The assembler stores the value in memory as the character's binary ASCII code. Examples are

```
'A'
"d"
```

Recall that Chapter 1 showed that character literals are stored internally as integers, using the ASCII encoding sequence. So, when you write the character constant "A," it's stored in memory as the number 65 (or 41 hex). We have a complete table of ASCII codes on the inside back cover of this book, so be sure to look over them from time to time.

3.1.6 String Literals

A *string literal* is a sequence of characters (including spaces) enclosed in single or double quotes:

```
'ABC'
'X'
"Good night, Gracie"
'4096'
```

Embedded quotes are permitted when used in the manner shown by the following examples:

```
"This isn't a test"
'Say "Good night," Gracie'
```

Just as character constants are stored as integers, we can say that string literals are stored in memory as sequences of integer byte values. So, for example, the string literal “ABCD” contains the four bytes 41h, 42h, 43h, and 44h.

3.1.7 Reserved Words

Reserved words have special meaning and can only be used in their correct context. Reserved works, by default, are not case-sensitive. For example, MOV is the same as mov and Mov. There are different types of reserved words:

- Instruction mnemonics, such as MOV, ADD, and MUL
- Register names
- Directives, which tell the assembler how to assemble programs
- Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD
- Operators, used in constant expressions
- Predefined symbols, such as @data, which return constant integer values at assembly time

A common list of reserved words can be found in Appendix A.

3.1.8 Identifiers

An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label. There are a few rules on how they can be formed:

- They may contain between 1 and 247 characters.
- They are not case sensitive.
- The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. Subsequent characters may also be digits.
- An identifier cannot be the same as an assembler reserved word.

Tip: You can make all keywords and identifiers case sensitive by adding the –Cp command line switch when running the assembler.

In general, it's a good idea to use descriptive names for identifiers, as you do in high-level programming language code. Although assembly language instructions are short and cryptic, there's no reason to make your identifiers hard to understand also! Here are some examples of well-formed names:

```
lineCount     firstValue    index     line_count
myFile        xCoord        main      x_Coord
```

The following names are legal, but not as desirable:

```
_lineCount    $first        @myFile
```

Generally, you should avoid the @ symbol and underscore as leading characters, since they are used both by the assembler and by high-level language compilers.

3.1.9 Directives

A *directive* is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime, but they let you define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. Directives are not, by default, case sensitive. For example, **.data**, **.DATA**, and **.Data** are equivalent.

The following example helps to show the difference between directives and instructions. The DWORD directive tells the assembler to reserve space in the program for a doubleword variable. The MOV instruction, on the other hand, executes at runtime, copying the contents of **myVar** to the EAX register:

```
myVar  DWORD 26
mov    eax,myVar
```

Although all assemblers for Intel processors share the same instruction set, they usually have different sets of directives. The Microsoft assembler's REPT directive, for example, is not recognized by some other assemblers.

Defining Segments One important function of assembler directives is to define program sections, or *segments*. Segments are sections of a program that have different purposes. For example, one segment can be used to define variables, and is identified by the .DATA directive:

```
.data
```

The .CODE directive identifies the area of a program containing executable instructions:

```
.code
```

The .STACK directive identifies the area of a program holding the runtime stack, setting its size:

```
.stack 100h
```

Appendix A contains a useful reference for directives and operators.

3.1.10 Instructions

An *instruction* is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. An instruction contains four basic parts:

- Label (optional)
- Instruction mnemonic (required)
- Operand(s) (usually required)
- Comment (optional)

This is how the different parts are arranged:

```
[label:] mnemonic [operands] [;comment]
```

Let's explore each part separately, beginning with the *label* field.

Label

A *label* is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address. There are two types of labels: Data labels and Code labels.

A *data label* identifies the location of a variable, providing a convenient way to reference the variable in code. The following, for example, defines a variable named count:

```
count  DWORD 100
```

The assembler assigns a numeric address to each label. It is possible to define multiple data items following a label. In the following example, array defines the location of the first number (1024). The other numbers following in memory immediately afterward:

```
array  DWORD 1024, 2048
       DWORD 4096, 8192
```

Variables will be explained in Section 3.4.2, and the MOV instruction will be explained in Section 4.1.4.

A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named **target**, creating a loop:

```
target:
    mov   ax,bx
    ...
    jmp   target
```

A code label can share the same line with an instruction, or it can be on a line by itself:

```
L1: mov   ax,bx
L2:
```

Label names follow the same rules we described for identifiers in Section 3.1.8. You can use the same code label more than once in a program as long as each label is unique within its enclosing procedure. We will show how to create procedures in Chapter 5.

Instruction Mnemonic

An *instruction mnemonic* is a short word that identifies an instruction. In English, a *mnemonic* is a device that assists memory. Similarly, assembly language instruction mnemonics such as mov, add, and sub provide hints about the type of operation they perform. Following are examples of instruction mnemonics:

Mnemonic	Description
MOV	Move (assign) one value to another
ADD	Add two values
SUB	Subtract one value from another
MUL	Multiply two values
JMP	Jump to a new location
CALL	Call a procedure

Operands

An operand is a value that is used for input or output for an instruction. Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, integer expression, or input–output port. We discussed register names in Chapter 2, and we discussed integer expressions in Section 3.1.2. There are different ways to create memory operands—using variable names, registers surrounded by brackets, for example. We will go into more details about that later. A variable name implies the address of the variable and instructs the computer to reference the contents of memory at the given address. The following table contains several sample operands:

Example	Operand Type
96	*Integer literal*
2 + 4	Integer expression
eax	Register
count	Memory

Let's look at examples of assembly language instructions having varying numbers of operands. The STC instruction, for example, has no operands:

```
stc                        ; set Carry flag
```

The INC instruction has one operand:

```
inc  eax                   ; add 1 to EAX
```

The MOV instruction has two operands:

```
mov count,ebx              ; move EBX to count
```

There is a natural ordering of operands. When instructions have multiple operands, the first one is typically called the destination operand. The second operand is usually called the *source operand*. In general, the contents of the destination operand are modified by the instruction. In a MOV instruction, for example, data is copied from the source to the destination.

The IMUL instruction has three operands, in which the first operand is the destination, and the following two operands are source operands, which are multiplied together:

```
imul eax,ebx,5
```

In this case, EBX is multiplied by 5, and the product is stored in the EAX register.

Comments

Comments are an important way for the writer of a program to communicate information about the program's design to a person reading the source code. The following information is typically included at the top of a program listing:

- Description of the program's purpose
- Names of persons who created and/or revised the program
- Program creation and revision dates
- Technical notes about the program's implementation

Comments can be specified in two ways:

- Single-line comments, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.
- Block comments, beginning with the COMMENT directive and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears. Here is an example:

```
COMMENT  !
     This line is a comment.
     This line is also a comment.
!
```

We can also use any other symbol, as long as it does not appear within the comment lines:

```
COMMENT &
    This line is a comment.
    This line is also a comment.
&
```

Of course, you should provide comments throughout a program, particularly where the intent of your code is not obvious.

The NOP (No Operation) Instruction

The safest (and the most useless) instruction is NOP (no operation). It takes up 1 byte of program storage and doesn't do any work. It is sometimes used by compilers and assemblers to align code to efficient address boundaries. In the following example, the first MOV instruction generates three machine code bytes. The NOP instruction aligns the address of the third instruction to a doubleword boundary (even multiple of 4):

```
00000000  66 8B C3    mov ax,bx
00000003  90          nop                ; align next instruction
00000004  8B D1       mov edx,ecx
```

x86 processors are designed to load code and data more quickly from even doubleword addresses.

3.1.11 Section Review

1. Using the value –35, write it as an integer literal in decimal, hexadecimal, octal, and binary formats that are consistent with MASM syntax.
2. *(Yes/No):* Is A5h a valid hexadecimal literal?
3. *(Yes/No):* Does the multiplication operator (*) have a higher precedence than the division operator (/) in integer expressions?
4. Create a single integer expression that uses all the operators from Section 3.1.2. Calculate the value of the expression.
5. Write the real number -6.2×10^4 as a real number literal using MASM syntax.
6. *(Yes/No):* Must string literals be enclosed in single quotes?
7. Reserved words can be instruction mnemonics, attributes, operators, predefined symbols, and __________.
8. What is the maximum length of an identifier?

3.2 Example: Adding and Subtracting Integers

3.2.1 The *AddTwo* Program

Let's revisit the *AddTwo* program we showed at the beginning of this chapter and add the necessary declarations to make it a fully operational program. Remember, the line numbers are not really part of the program:

```
 1: ; AddTwo.asm - adds two 32-bit integers
 2: ; Chapter 3 example
 3:
 4: .386
 5: .model flat,stdcall
 6: .stack 4096
 7: ExitProcess PROTO, dwExitCode:DWORD
 8:
 9: .code
10: main PROC
11:    mov    eax,5      ; move 5 to the eax register
12:    add    eax,6      ; add 6 to the eax register
13:
14:    INVOKE ExitProcess,0
15: main ENDP
16: END main
```

Line 4 contains the .386 directive, which identifies this as a 32-bit program that can access 32-bit registers and addresses. Line 5 selects the program's memory model (*flat*), and identifies the calling convention (named *stdcall*) for procedures. We use this because 32-bit Windows services require the stdcall convention to be used. (Chapter 8 explains how *stdcall* works.) Line 6 sets aside 4096 bytes of storage for the runtime stack, which every program must have.

Line 7 declares a prototype for the **ExitProcess** function, which is a standard Windows service. A *prototype* consists of the function name, the **PROTO** keyword, a comma, and a list of input parameters. The input parameter for ExitProcess is named **dwExitCode**. You might think of it as a return value passed back to the Window operating system. A return value of zero usually means our program was successful. Any other integer value generally indicates an error code number. So, you can think of your assembly programs as subroutines, or processes, which are called by the operating system. When your program is ready to finish, it calls ExitProcess and returns an integer that tells the operating system that your program worked just fine.

More Info: You might be wondering why the operating system wants to know if your program completed successfully. Here's why: system administrators often create script files than execute a number of programs in sequence. At each point in the script, they need to know if the most recently executed program has failed, so they can exit the script if necessary. It often goes something like the script shown below, where *ErrorLevel 1* indicates that the process return code from the previous step was greater than or equal to 1:

```
call program_1
if ErrorLevel 1 goto FailedLabel
call program_2
if ErrorLevel 1 goto FailedLabel
:SuccessLabel
Echo Great, everything worked!
```

Let's return to our listing of the AddTwo program. Line 16 uses the **end** directive to mark the last line to be assembled, and it identifies the program entry point (main). The label main was declared on Line 10, and it marks the address at which the program will begin to execute.

Tip: Visual Studio's syntax highlighting and wavy lines under keywords are not consistent when displaying assembly language code. If you want to disable it, here's how: Choose *Options* from the Tools menu, select *Text Editor,* select *C/C++*, select *Advanced,* and under the *Intellisense* heading, set *Disable Squiggles* to True. Click *OK* to close the Options window. Also, remember that MASM is not case-sensitive, so you can capitalize or not capitalize keywords in any combination.

A Review of the Assembler Directives

Let's review some of the most important assembler directives we used in the sample program. First, the .MODEL directive tells the assembler which memory model to use:

```
.model flat,stdcall
```

In 32-bit programs, we always use the flat memory model, which is associated with the processor's protected mode. We talked about protected mode in Chapter 2. The stdcall keyword tells the assembler how to manage the runtime stack when procedures are called. That's a complicated subject that we will address in Chapter 8. Next, the .STACK directive tells the assembler how many bytes of memory to reserve for the program's runtime stack:

```
.stack 4096
```

The value 4096 is probably more than we will ever use, but it happens to correspond to the size of a memory page in the processor's system for managing memory. All modern programs use a stack when calling subroutines—first, to hold passed parameters, and second, to hold the address of the code that called the function. The CPU uses this address to return when the function call finishes, back to the spot where the function was called. In addition, the runtime stack can hold local variables, that is, variables declared inside a function.

The .CODE directive marks the beginning of the code area of a program, the area that contains executable instructions. Usually the next line after .CODE is the declaration of the program's entry point, and by convention, it is usually a procedure named **main**. The entry point of a program is the location of the very first instruction the program will execute. We used the following lines to convey this information:

```
.code
main PROC
```

The ENDP directive marks the end of a procedure. Our program had a procedure named main, so the endp must use the same name:

```
main ENDP
```

Finally, the END directive marks the end of the program, and references the program entry point:

```
END main
```

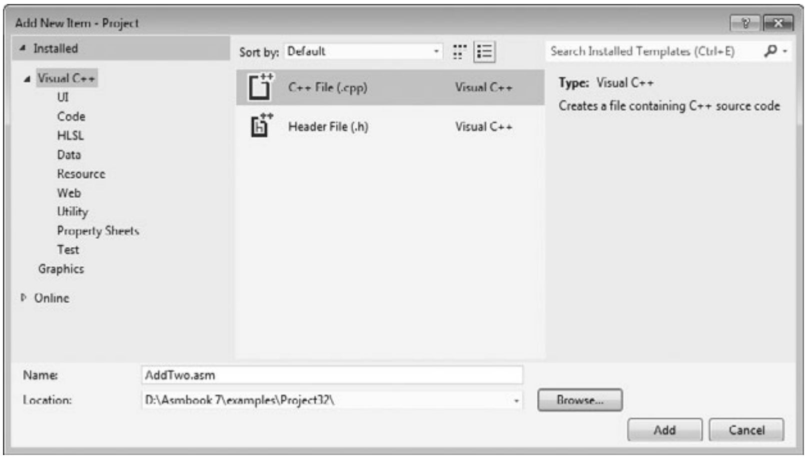
If you add any more lines to a program after the END directive, they will be ignored by the assembler. You can put anything there—program comments, copies of your code, etc.—it doesn't matter.

3.2.2 Running and Debugging the AddTwo Program

You can easily use Visual Studio to edit, build, and run assembly language programs. The book's example files directory has a folder named *Project32* that contains a Visual Studio 2012 Windows Console project that has been configured for 32-bit assembly language programming. (Another folder named *Project64* is configured for 64-bit assembly.) The following instructions, modeled after Visual Studio 2012, tell you how to open the sample project and create the AddTwo program:

1. Open the *Project32* folder and double-click the file named *Project.sln.* This should launch the latest version of Visual Studio installed on your computer.
2. Open the Solution Explorer window inside Visual Studio. It should already be visible, but you can always make it visible by selecting *Solution Explorer* from the *View* menu.
3. Right-click the project name in Solution Explorer, select *Add* from the context menu, and then select *New Item* from the popup menu.
4. In the *Add New File* dialog window (see Figure 3-1), name the file *AddTwo.asm*, and choose an appropriate disk folder for the file by filling in the *Location* entry.
5. Click the *Add* button to save the file.

Figure 3–1 Adding a new source code file to a Visual Studio project.



Type: Visual C++
Creates a file containing C++ source code

Name: AddTwo.asm
Location: D:\Asmbook 7\examples\Project32\

6. Type in the program's source code, shown here. The capitalization of keywords here is not required:

```
; AddTwo.asm - adds two 32-bit integers.

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD

.code
main PROC
  mov  eax,5
  add  eax,6

  INVOKE ExitProcess,0
main ENDP
END main
```

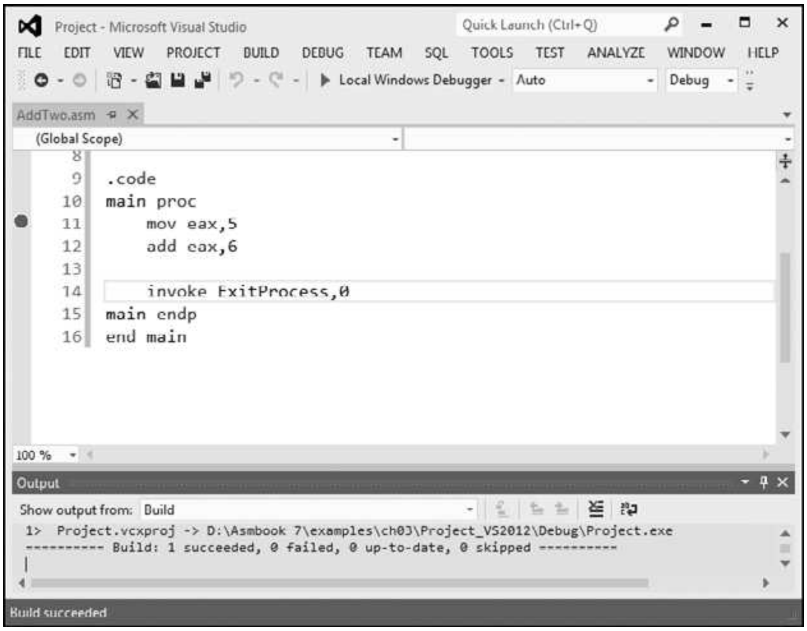
7. Select *Build Project* from the Project menu, and look for error messages at the bottom of the Visual Studio workspace. It's called the *Error List* window. Figure 3-2 shows our sample program after it has been opened and assembled. Notice that the status line on the bottom of the window says *Build succeeded* when there are no errors.

Debugging Demonstration

We will demonstrate a sample debugging session for the AddTwo program. We have not shown you a way to display variable values directly in the console window yet, so we will run the program in a debugging session. We will use Visual Studio 2012 for this demonstration, but it would work just as well in any version of Visual Studio from 2008 onward.

One way to run and debug a program is to, select *Step Over* from the Debug menu. Depending on how Visual Studio was configured, either the F10 function key or the Shift+F8 keys will execute the *Step Over* command.

Figure 3–2 Building the Visual Studio project.



```
 8
 9  .code
10  main proc
11      mov eax,5
12      add eax,6
13
14      invoke ExitProcess,0
15  main endp
16  end main
```

```
1>  Project.vcxproj -> D:\Asmbook 7\examples\ch03\Project_VS2012\Debug\Project.exe
---------- Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ----------
```

Another way to start a debugging session is to set a breakpoint on a program statement by clicking the mouse in the vertical gray bar just to the left of the code window. A large red dot will mark the breakpoint location. Then you can run the program by selecting *Start Debugging* from the Debug menu.

> ***Tip:*** If you try to set a breakpoint on a non-executable line, Visual Studio will just move the breakpoint forward to the next executable line when you run the program.

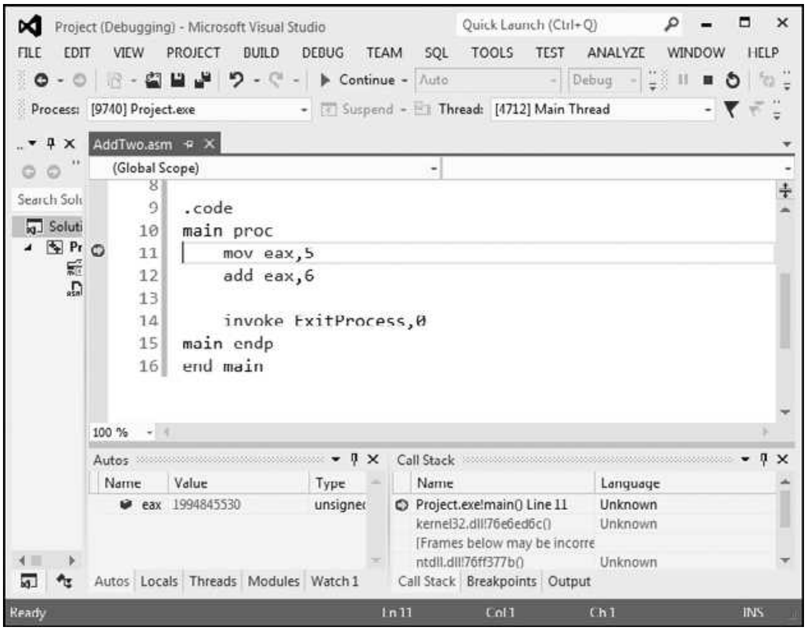
Figure 3-3 shows the program at the start of a debugging session. A breakpoint was set on Line 11, the first MOV instruction, and the debugger has paused on that line. The line has not executed yet. When the debugger is active, the bottom status line of the Visual Studio window turns orange. When you stop the debugger and return to edit mode, the status line turns blue. The visual cue is helpful because you cannot edit or save a program while the debugger is running.

Figure 3-4 shows the debugger after the user has stepped through lines 11 and 12, and is paused on line 14. By hovering the mouse over the EAX register name, we can see its current contents (11). We can then finish the program execution by clicking the *Continue* button on the toolbar, or by clicking the red *Stop Debugging* button (on the right side of the toolbar).

Customizing the Debugging Interface

You can customize the debugging interface while it is running. For example, you might want to display the CPU registers; to do this, select *Windows* from the Debug menu, and then select *Registers*. Figure 3-5 shows the same debugging session we used just now, with the *Registers* window visible. We also closed some other nonessential windows. The value shown in EAX, 0000000B, is the hexadecimal representation of 11 decimal. We've drawn an arrow in the

Figure 3–3 Debugger paused at a breakpoint.


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
Process: [9740] Project.exe
Suspend
Thread: [4712] Main Thread
AddTwo.asm
(Global Scope)
Search Solu
Soluti
100 %
Autos Locals Threads Modules Watch 1
Call Stack Breakpoints Output
Ready Ln 11 Col 1 Ch 1 INS

```
 8
 9  .code
10  main proc
11      mov eax,5
12      add eax,6
13
14      invoke ExitProcess,0
15  main endp
16  end main
```

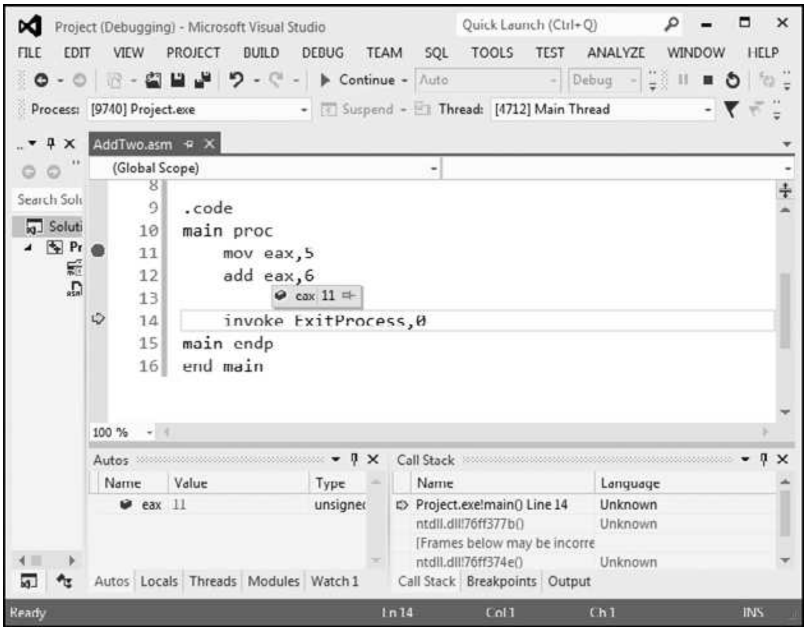
Autos

Name	Value	Type
eax	1994845530	unsignec

Call Stack

Name	Language
Project.exe!main() Line 11	Unknown
kernel32.dll!76e6ed6c()	Unknown
[Frames below may be incorre	
ntdll.dll!76ff377b0()	Unknown

Figure 3–4 After executing lines 11 and 12 in the debugger.


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
Process: [9740] Project.exe
Suspend
Thread: [4712] Main Thread
AddTwo.asm
(Global Scope)
Search Solu
Soluti
eax 11
100 %
Autos Locals Threads Modules Watch 1
Call Stack Breakpoints Output
Ready Ln 14 Col 1 Ch 1 INS

```
 8
 9  .code
10  main proc
11      mov eax,5
12      add eax,6
13
14      invoke ExitProcess,0
15  main endp
16  end main
```

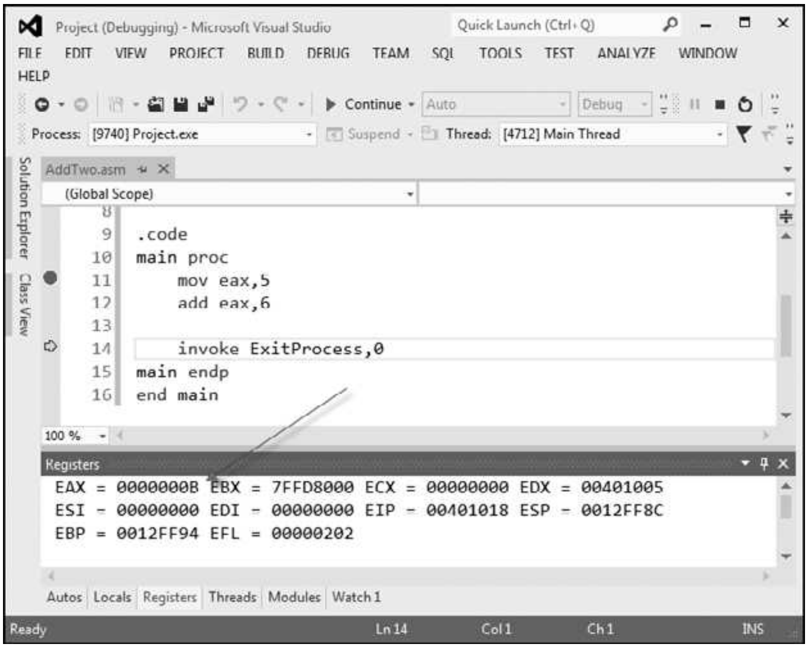
Autos

Name	Value	Type
eax	11	unsignec

Call Stack

Name	Language
Project.exe!main() Line 14	Unknown
ntdll.dll!76ff377b0()	Unknown
[Frames below may be incorre	
ntdll.dll!76ff374e0()	Unknown

Figure 3–5 Adding the *Registers* window to 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
Process: [9740] Project.exe
Suspend
Thread: [4712] Main Thread
AddTwo.asm
(Global Scope)
Solution Explorer
Class View
100 %
Registers
Autos Locals Registers Threads Modules Watch 1
Ready Ln 14 Col 1 Ch 1 INS

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

    invoke ExitProcess,0
main endp
end main
```

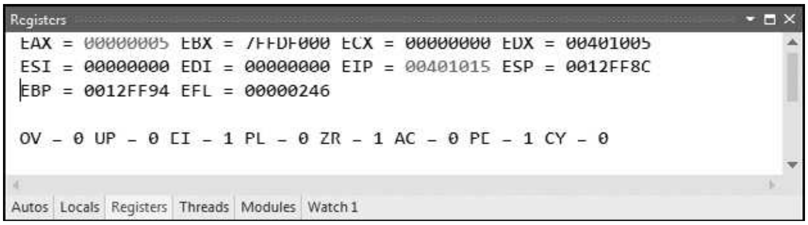
```
EAX = 0000000B EBX = 7FFD8000 ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000 EIP = 00401018 ESP = 0012FF8C
EBP = 0012FF94 EFL = 00000202
```

figure, pointing to the value. In the *Registers* window, the EFL register contains all the status flag settings (Zero, Carry, Overflow, etc.). If you right-click the *Registers* window and select *Flags* from the popup menu, the window will display the individual flag values. Figure 3-6 shows an example, where the flag values from left to right are: OV (overflow flag), UP (direction flag), EI (interrupt flag), PL (sign flag), ZR (zero flag), AC (auxiliary carry), PE (parity flag), and CY (carry flag). The precise meaning of these flags will be explained in Chapter 4.

One of the great things about the *Registers* window is that as you step through a program, any register whose value is changed by the current instruction will turn red. Although we cannot show it on the printed page (which is black and white), the red highlighting really jumps out at you, to let you know how your program is affecting the registers.

> ***Tip:*** The book's web site (*asmirvine.com*) has tutorials that show you how to assemble and debug assembly language programs.

Figure 3–6 Showing the CPU status flags in the *Registers* window.



```
EAX = 00000005 EBX = 7FFDF000 ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000 EIP = 00401015 ESP = 0012FF8C
EBP = 0012FF94 EFL = 00000246

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0 PE = 1 CY = 0
```

When you run an assembly language program inside Visual Studio, it launches inside a console window. This is the same window you see when you run the program named *cmd.exe* from the Windows *Start* menu. Alternatively, you could open up a command prompt in the project's *Debug\Bin* folder and run the application directly from the command line. If you did this, you would only see the program's output, which consists of text written to the console window. Look for an executable filename having the same name as your Visual Studio project.

3.2.3 Program Template

Assembly language programs have a simple structure, with small variations. When you begin a new program, it helps to start with an empty shell program with all basic elements in place. You can avoid redundant typing by filling in the missing parts and saving the file under a new name. The following program (*Template.asm*) can easily be customized. Note that comments have been inserted, marking the points where your own code should be added. Capitalization of keywords is optional:

```
; Program template (Template.asm)

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data
    ; declare variables here
.code
main PROC
    ; write your code here

    INVOKE ExitProcess,0
main ENDP
END main
```

Use Comments It's a very good idea to include a program description, the name of the program's author, creation date, and information about subsequent modifications. Documentation of this kind is useful to anyone who reads the program listing (including you, months or years from now). Many programmers have discovered, years after writing a program, that they must become reacquainted with their own code before they can modify it. If you're taking a programming course, your instructor may insist on additional information.

3.2.4 Section Review

1. In the AddTwo program, what is the meaning of the INCLUDE directive?
2. In the AddTwo program, what does the .CODE directive identify?
3. What are the names of the two segments in the AddTwo program?
4. In the AddTwo program, which register holds the sum?
5. In the AddTwo program, which statement halts the program?

3.3 Assembling, Linking, and Running Programs

A source program written in assembly language cannot be executed directly on its target computer. It must be translated, or *assembled* into executable code. In fact, an assembler is very similar to a *compiler*, the type of program you would use to translate a C++ or Java program into executable code.

The assembler produces a file containing machine language called an *object file*. This file isn't quite ready to execute. It must be passed to another program called a *linker*, which in turn produces an *executable file*. This file is ready to execute from the operating system's command prompt.

3.3.1 The Assemble-Link-Execute Cycle

The process of editing, assembling, linking, and executing assembly language programs is summarized in Figure 3-7. Following is a detailed description of each step.

Step 1: A programmer uses a **text editor** to create an ASCII text file named the *source file*.

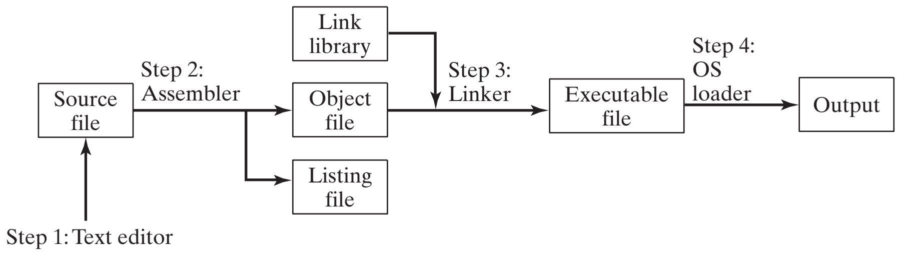
Step 2: The **assembler** reads the source file and produces an *object file,* a machine-language translation of the program. Optionally, it produces a *listing file*. If any errors occur, the programmer must return to Step 1 and fix the program.

Step 3: The **linker** reads the object file and checks to see if the program contains any calls to procedures in a link library. The **linker** copies any required procedures from the link library, combines them with the object file, and produces the *executable file*.

Step 4: The operating system **loader** utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.

See the topic "Getting Started" on the author's Web site (www.asmirvine.com) for detailed instructions on assembling, linking, and running assembly language programs using Microsoft Visual Studio.

Figure 3–7 Assemble-Link-Execute cycle.



3.3.2 Listing File

A *listing file* contains a copy of the program's source code, with line numbers, the numeric address of each instruction, the machine code bytes of each instruction (in hexadecimal), and a symbol table. The symbol table contains the names of all program identifiers, segments, and related information. Advanced programmers sometimes use the listing file to get detailed information about the program. Figure 3-8 shows a partial listing file for the *AddTwo* program. Let's examine it in more detail. Lines 1–7 contain no executable code, so they are copied directly from the source file without changes. Line 9 shows that the beginning of the code segment is located at address 00000000 (in a 32-bit program, addresses display as 8 hexadecimal digits). This address is relative to the beginning of the program's memory footprint, but it will be converted into an absolute memory address when the program is loaded into memory. When that happens, the program might start at an address such as 00040000h.

Figure 3–8 Excerpt from the AddTwo source listing file.

```
 1:    ; AddTwo.asm - adds two 32-bit integers.
 2:    ; Chapter 3 example
 3:
 4:    .386
 5:    .model flat,stdcall
 6:    .stack 4096
 7:    ExitProcess PROTO,dwExitCode:DWORD
 8:
 9:     00000000                     .code
10:     00000000                     main PROC
11:     00000000  B8 00000005           mov  eax,5
12:     00000005  83 C0 06              add  eax,6
13:
14:                                     invoke ExitProcess,0
15:     00000008  6A 00                 push   +000000000h
16:     0000000A  E8 00000000 E         call   ExitProcess
17:     0000000F                     main ENDP
18:                                  END main
```

Lines 10 and 11 also show the same starting address of 00000000, because the first executable statement is the MOV instruction on line 11. Notice on line 11 that several hexadecimal bytes appear between the address and the source code. These bytes (B8 00000005) represent the machine code instruction (B8), and the constant 32-bit value (00000005) that is assigned to EAX by the instruction:

```
11:  00000000  B8 00000005  mov eax,5
```

The value B8 is also known as an *operation code* (or just *opcode*), because it represents the specific machine instruction to move a 32-bit integer into the **eax** register. In Chapter 12 we explain the structure of x86 machine instructions in great detail.

Line 12 also contains an executable instruction, starting at offset 00000005. That offset is a distance of 5 bytes from the beginning of the program. Perhaps you can guess how that offset was calculated.

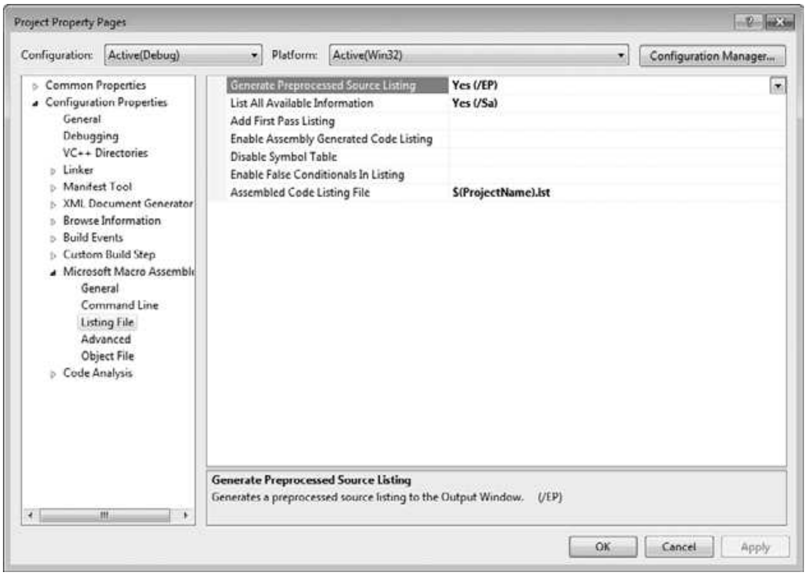
Line 14 contains the **invoke** directive. Notice how lines 15 and 16 seem to have been inserted into our code. This is because the INVOKE directive causes the assembler to generate the PUSH and CALL statements shown on lines 15 and 16. In Chapter 5 we will show how to use PUSH and CALL.

The sample listing file in Figure 3-8 shows that the machine instructions are loaded into memory as a sequence of integer values, expressed here in hexadecimal: B8, 00000005, 83, C0, 06, 6A, 00, EB, 00000000. The number of digits in each number indicates the number of bits: a 2-digit number is 8 bits, a 4-digit number is 16 bits, an 8-digit number is 32 bits, and so on. So our machine instructions are exactly 15 bytes long (two 4-byte values and seven 1-byte values).

Whenever you want to make sure the assembler is generating the correct machine code bytes based on your program, the listing file is your best resource. It is also a great teaching tool if you're just learning how machine code instructions are generated.

> ***Tip:*** To tell Visual Studio to generate a listing file, do the following when a project is open: Select *Properties* from the Project menu. Under *Configuration Properties*, select *Microsoft Macro Assembler*. Then select *Listing File*. In the dialog window, set *Generate Preprocessed Source Listing* to *Yes*, and set *List All Available Information* to *Yes*. The dialog window is shown in Figure 3-9.

Figure 3–9 Configuring Visual Studio to generate a listing file.



Configuration: Active(Debug)   Platform: Active(Win32)

Generate Preprocessed Source Listing	Yes (/EP)
List All Available Information	Yes (/Sa)
Add First Pass Listing	
Enable Assembly Generated Code Listing	
Disable Symbol Table	
Enable False Conditionals In Listing	
Assembled Code Listing File	$(ProjectName).lst

Generate Preprocessed Source Listing
Generates a preprocessed source listing to the Output Window. (/EP)

The rest of the listing file contains a list of structures and unions, as well as procedures, parameters, and local variables. We will not show those elements here, but we will discuss them in later chapters.

3.3.3 Section Review

1. What types of files are produced by the assembler?
2. *(True/False):* The linker extracts assembled procedures from the link library and inserts them in the executable program.
3. *(True/False):* When a program's source code is modified, it must be assembled and linked again before it can be executed with the changes.
4. Which operating system component reads and executes programs?
5. What types of files are produced by the linker?

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

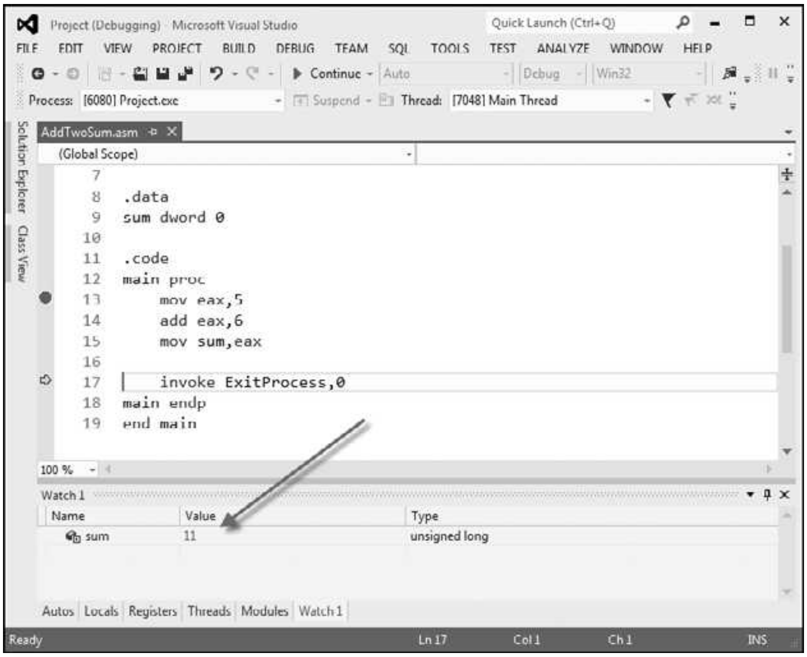
Let's create a new version of the *AddTwo* program we introduced at the beginning of this chapter, which we will now call *AddTwoSum*. This version introduces a variable named **sum**, which appears in the complete program listing:

```
; AddTwoSum.asm - Chapter 3 example

   .386
   .model flat,stdcall
   .stack 4096
   ExitProcess PROTO, dwExitCode:DWORD

   .data
   sum DWORD 0

   .code
   main PROC
      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.

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

```
 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
```

Name	Value	Type
sum	11	unsigned long

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
```

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
```

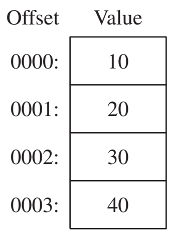
Multiple Initializers

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:

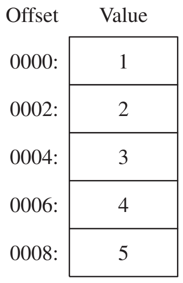
```
val1  DW 65535              ; unsigned
val2  DW -32768             ; signed
```

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

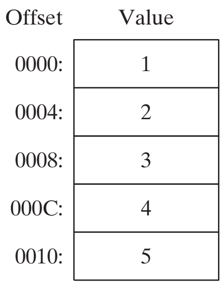
The QWORD (define quadword) directive allocates storage for 64-bit (8-byte) values:

```
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.



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:

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}

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
```

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.

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.

```
; AddVariables.asm - Chapter 3 example

   .386
   .model flat,stdcall
   .stack 4096
   ExitProcess PROTO, dwExitCode:DWORD

   .data
   firstval  DWORD 20002000h
   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

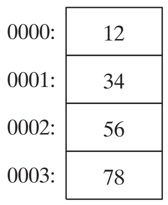
x86 processors store and retrieve data from memory using *little-endian* order (low to high). The least significant byte is stored at the first memory address allocated for the data. The remaining bytes are stored in the next consecutive memory positions. Consider the doubleword 12345678h. If placed 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.



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.

3.4.13 Section Review

1. Create an uninitialized data declaration for a 16-bit signed integer.
2. Create an uninitialized data declaration for an 8-bit unsigned integer.
3. Create an uninitialized data declaration for an 8-bit signed integer.

4. Create an uninitialized data declaration for a 64-bit integer.
5. Which data type can hold a 32-bit signed integer?

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.

3.5.5 Section Review

1. Declare a symbolic constant using the equal-sign directive that contains the ASCII code (08h) for the Backspace key.
2. Declare a symbolic constant named **SecondsInDay** using the equal-sign directive and assign it an arithmetic expression that calculates the number of seconds in a 24-hour period.
3. Write a statement that causes the assembler to calculate the number of bytes in the following array, and assign the value to a symbolic constant named **ArraySize**:

   ```
   myArray WORD 20 DUP(?)
   ```

4. Show how to calculate the number of elements in the following array, and assign the value to a symbolic constant named **ArraySize**:

   ```
   myArray DWORD 30 DUP(?)
   ```

5. Use a TEXTEQU expression to redefine "proc" as "procedure."
6. Use TEXTEQU to create a symbol named **Sample** for a string constant, and then use the symbol when defining a string variable named **MyString**.
7. Use TEXTEQU to assign the symbol **SetupESI** to the following line of code:

   ```
   mov esi,OFFSET myArray
   ```

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:

```
; AddTwoSum_64.asm - Chapter 3 example.

ExitProcess PROTO

.data
sum DWORD 0

.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.

3.7 Chapter Summary

A constant integer expression is a mathematical expression involving integer literals, symbolic constants, and arithmetic operators. *Precedence* refers to the implied order of operations when an expression contains two or more operators.

A *character literal* is a single character enclosed in quotes. The assembler converts a character to a byte containing the character's binary ASCII code. A *string literal* is a sequence of characters enclosed in quotes, optionally ending with a null byte.

Assembly language has a set of *reserved words* with special meanings that may only be used in the correct context. An *identifier* is a programmer-chosen name identifying a variable, a symbolic constant, a procedure, or a code label. Identifiers cannot be reserved words.

A *directive* is a command embedded in the source code and interpreted by the assembler. An *instruction* is a source code statement that is executed by the processor at runtime. An *instruction mnemonic* is a short keyword that identifies the operation carried out by an instruction. A *label* is an identifier that acts as a place marker for instructions or data.

Operands are values passed to instructions. An assembly language instruction can have between zero and three operands, each of which can be a register, memory operand, integer expression, or input-output port number.

Programs contain *logical segments* named code, data, and stack. The code segment contains executable instructions. The stack segment holds procedure parameters, local variables, and return addresses. The data segment holds variables.

A *source file* contains assembly language statements. A *listing file* contains a copy of the program's source code, suitable for printing, with line numbers, offset addresses, translated machine code, and a symbol table. A source file is created with a text editor. An *assembler* is a program that reads the source file, producing both object and listing files. The *linker* is a program that reads one or more object files and produces an executable file. The latter is executed by the operating system loader.

MASM recognizes intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type:

- BYTE and SBYTE define 8-bit variables.
- WORD and SWORD define 16-bit variables.
- DWORD and SDWORD define 32-bit variables.
- QWORD and TBYTE define 8-byte and 10-byte variables, respectively.
- REAL4, REAL8, and REAL10 define 4-byte, 8-byte, and 10-byte real number variables, respectively.

A *data definition statement* sets aside storage in memory for a variable, and may optionally assign it a name. If multiple initializers are used in the same data definition, its label refers only to the offset of the first initializer. To create a string data definition, enclose a sequence of characters in quotes. The DUP operator generates a repeated storage allocation, using a constant expression as a counter. The current location counter operator ($) is used in address-calculation expressions.

x86 processors store and retrieve data from memory using *little-endian* order: The least significant byte of a variable is stored at its starting (lowest) address value.

A *symbolic constant* (or symbol definition) associates an identifier with an integer or text expression. Three directives create symbolic constants:

- The equal-sign directive (=) associates a symbol name with a constant integer expression.
- The EQU and TEXTEQU directives associate a symbolic name with a constant integer expression or some arbitrary text.

3.8 Key Terms

3.8.1 Terms

assembler
big endian
binary coded decimal (BCD)
calling convention
character literal
code label
code segment
compiler
constant integer expression
data definition statement
data label
data segment
decimal real
directive
encoded real
executable file
floating-point literal
identifier
initializer
instruction
instruction mnemonic
integer constant
integer literal
intrinsic data type
label
linker
link library
listing file
little-endian order
macro
memory model
memory operand
object file
operand
operator precedence
packed binary coded decimal
process return code
program entry point
real number literal
reserved word
source file
stack segment
string literal
symbolic constant
system function

3.8.2 Instructions, Operators, and Directives

+	(add, unary plus)	END
=	(assign, compare for equality)	ENDP
/	(divide)	DUP
*	(multiply)	EQU
( )	(parentheses)	MOD
−	(subtract, unary minus)	MOV
ADD		NOP
BYTE		PROC
CALL		SBYTE
.CODE		SDWORD
COMMENT		.STACK
.DATA		TEXTEQU
DWORD		

3.9 Review Questions and Exercises

3.9.1 Short Answer

1. Provide examples of three different instruction mnemonics.
2. What is a calling convention, and how is it used in assembly language declarations?
3. How do you reserve space for the stack in a program?
4. Explain why the term *assembler language* is not quite correct.
5. Explain the difference between big endian and little endian. Also, look up the origins of this term on the Web.
6. Why might you use a symbolic constant rather than an integer literal in your code?
7. How is a source file different from a listing file?
8. How are data labels and code labels different?
9. *(True/False):* An identifier cannot begin with a numeric digit.
10. *(True/False):* A hexadecimal literal may be written as 0x3A.
11. *(True/False):* Assembly language directives execute at runtime.
12. *(True/False):* Assembly language directives can be written in any combination of uppercase and lowercase letters.
13. Name the four basic parts of an assembly language instruction.
14. *(True/False):* MOV is an example of an instruction mnemonic.
15. *(True/False):* A code label is followed by a colon (:), but a data label does not end with a colon.
16. Show an example of a block comment.
17. Why is it not a good idea to use numeric addresses when writing instructions that access variables?

18. What type of argument must be passed to the ExitProcess procedure?
19. Which directive ends a procedure?
20. In 32-bit mode, what is the purpose of the identifier in the END directive?
21. What is the purpose of the PROTO directive?
22. *(True/False):* An Object file is produced by the Linker.
23. *(True/False):* A Listing file is produced by the Assembler.
24. *(True/False):* A link library is added to a program just before producing an Executable file.
25. Which data directive creates a 32-bit signed integer variable?
26. Which data directive creates a 16-bit signed integer variable?
27. Which data directive creates a 64-bit unsigned integer variable?
28. Which data directive creates an 8-bit signed integer variable?
29. Which data directive creates a 10-byte packed BCD variable?

3.9.2 Algorithm Workbench

1. Define four symbolic constants that represent integer 25 in decimal, binary, octal, and hexadecimal formats.
2. Find out, by trial and error, if a program can have multiple code and data segments.
3. Create a data definition for a doubleword that stored it in memory in big endian format.
4. Find out if you can declare a variable of type DWORD and assign it a negative value. What does this tell you about the assembler's type checking?
5. Write a program that contains two instructions: (1) add the number 5 to the EAX register, and (2) add 5 to the EDX register. Generate a listing file and examine the machine code generated by the assembler. What differences, if any, did you find between the two instructions?
6. Given the number 456789ABh, list out its byte values in little-endian order.
7. Declare an array of 120 uninitialized unsigned doubleword values.
8. Declare an array of byte and initialize it to the first 5 letters of the alphabet.
9. Declare a 32-bit signed integer variable and initialize it with the smallest possible negative decimal value. (Hint: Refer to integer ranges in Chapter 1.)
10. Declare an unsigned 16-bit integer variable named **wArray** that uses three initializers.
11. Declare a string variable containing the name of your favorite color. Initialize it as a nullterminated string.
12. Declare an uninitialized array of 50 signed doublewords named **dArray**.
13. Declare a string variable containing the word "TEST" repeated 500 times.
14. Declare an array of 20 unsigned bytes named **bArray** and initialize all elements to zero.

15. Show the order of individual bytes in memory (lowest to highest) for the following doubleword variable:

```
val1 DWORD 87654321h
```

3.10 Programming Exercises

★ **1. Integer Expression Calculation**

Using the *AddTwo* program from Section 3.2 as a reference, write a program that calculates the following expression, using registers: A = (A + B) – (C + D). Assign integer values to the EAX, EBX, ECX, and EDX registers.

★ **2. Symbolic Integer Constants**

Write a program that defines symbolic constants for all seven days of the week. Create an array variable that uses the symbols as initializers.

★★ **3. Data Definitions**

Write a program that contains a definition of each data type listed in Table 3-2 in Section 3.4. Initialize each variable to a value that is consistent with its data type.

★ **4. Symbolic Text Constants**

Write a program that defines symbolic names for several string literals (characters between quotes). Use each symbolic name in a variable definition.

★★★★ **5. Listing File for AddTwoSum**

Generate a listing file for the AddTwoSum program and write a description of the machine code bytes generated for each instruction. You might have to guess at some of the meanings of the byte values.

★★★ **6. AddVariables Program**

Modify the AddVariables program so it uses 64-bit variables. Describe the syntax errors generated by the assembler and what steps you took to resolve the errors.