Microprocessor & Assembly Language

Lecture No.6

Lecture Outlines

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.

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.

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
```

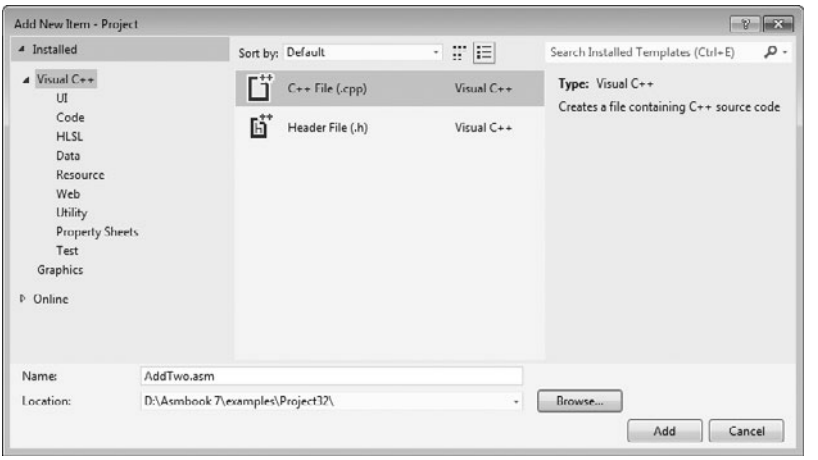
If you add any more lines to a program after the END directive, they will be ignored by the assembler. You can put anything there—program comments, copies of your code, etc.—it doesn't matter.

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.



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
```

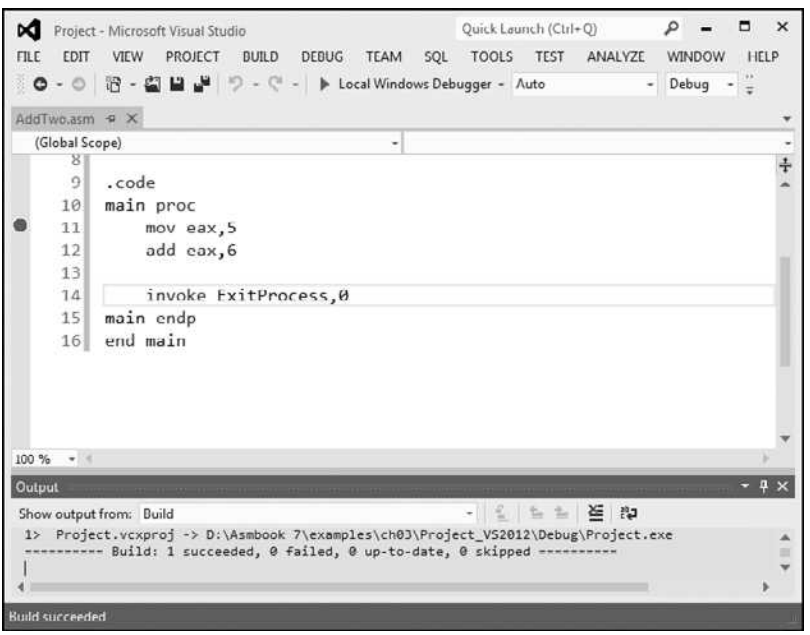
7. Select *Build Project* from the Project menu, and look for error messages at the bottom of the Visual Studio workspace. It's called the *Error List* window. Figure 3-2 shows our sample program after it has been opened and assembled. Notice that the status line on the bottom of the window says *Build 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.



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.

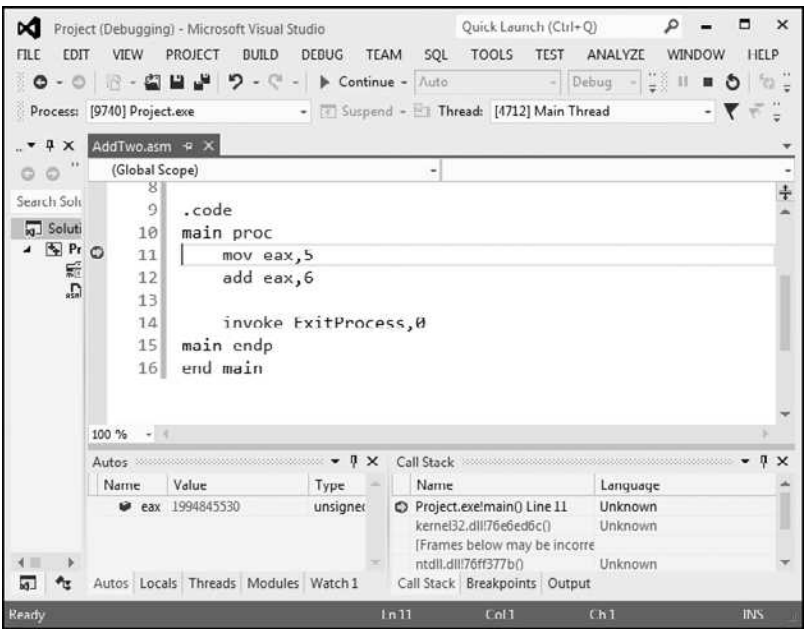


Figure 3–4 After executing lines 11 and 12 in the debugger.

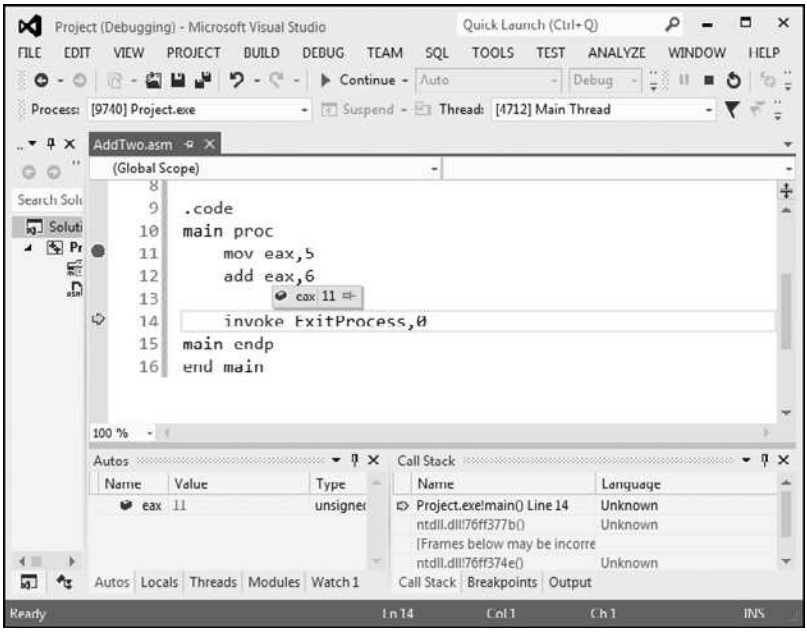
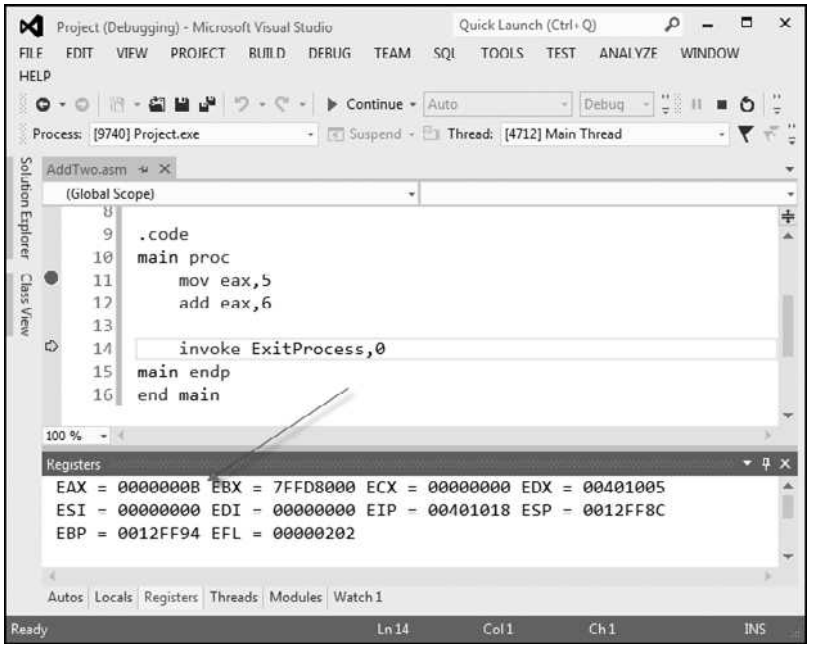


Figure 3–5 Adding the *Registers* window to a debugging session.

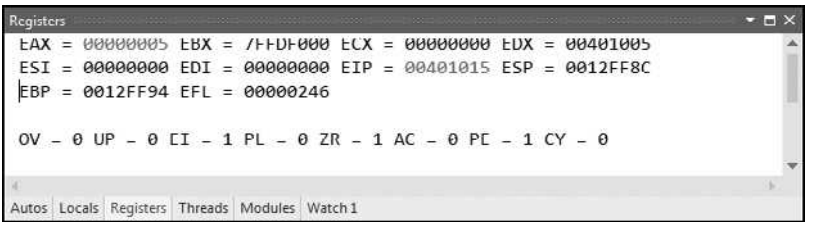


figure, pointing to the value. In the *Registers* window, the EFL register contains all the status flag settings (Zero, Carry, Overflow, etc.). If you right-click the *Registers* window and select *Flags* from the popup menu, the window will display the individual flag values. Figure 3-6 shows an example, 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.



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
```

```
.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.

3.3 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.

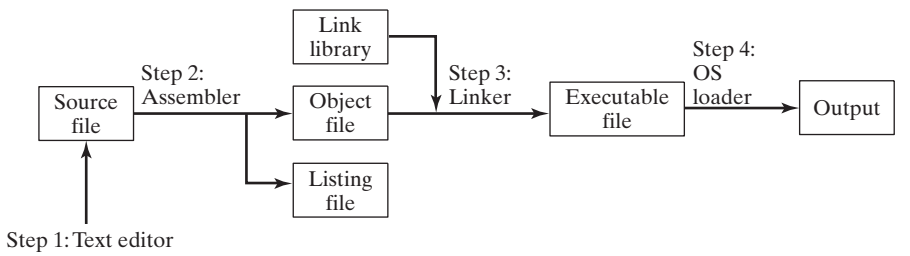
Step 1: A programmer uses a **text editor** to create an ASCII text file named the *source file*.

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.

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.

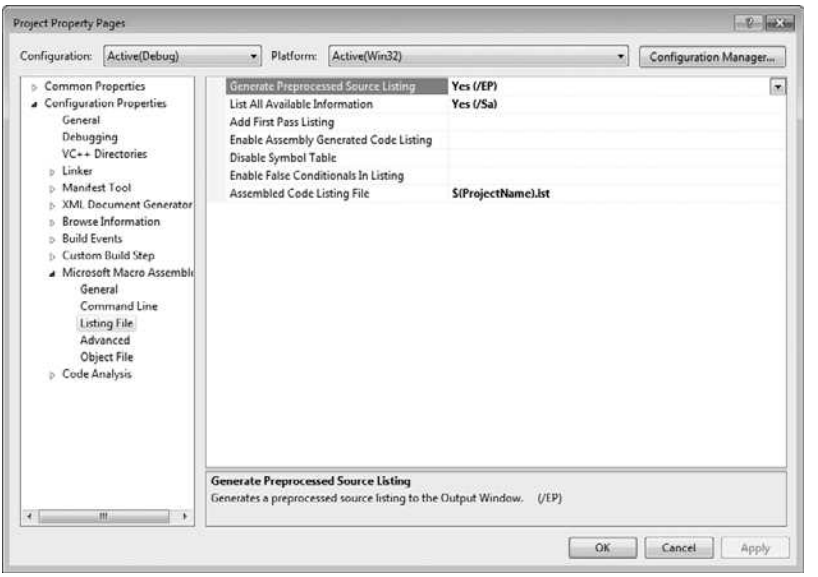
Line 14 contains the **invoke** directive. Notice how lines 15 and 16 seem to have been inserted into our code. This is because the INVOKE directive causes the assembler to generate the PUSH and CALL statements shown on lines 15 and 16. In Chapter 5 we will show how to use PUSH and CALL.

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.



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.