Using x86 Assembly Language with Microsoft Visual C++ 6.0

This tutorial continues the introduction to the Microsoft Visual C++ Integrated Development Environment (IDE) and addresses using assembly code in a project. It covers adding assembly code to a project containing a main() function in C++ and debugging a project with assembly.

Creating a C++/Assembly Project

The project we'll develop in this tutorial will consist of a main() function written in C++. It will call an assembly function named clear(). Since Visual C++ does not recognize assembly code, VC++ will have to be told what program to call to compile the assembly code. In this section, we'll cover the basic steps of creating a VC++ project, adding assembly code to it, specifying the Custom Build instructions, and building the project.

■ Step 1 Create Project

Create a standard Visual C++ project. Add a c/c++ source file with a main() function. For example, we will create a main() function that will call an assembler function named clear(). clear() is of type void and requires no parameters and will be located in a separate file called clear.asm. Since it is in a separate file, clear() will need to be declared at the beginning of the file. Our main() function will look like this:

```
// Declaration of assembly function
// extern "C" instructs compiler to use C calling conventions
extern "C" void clear();

int main() {
        clear();
        return 1;
}
```

The phrase "extern "C"" in the declaration tells the compiler to use the C calling convention. **This effects things like how parameters are passed to the routine and whether the calling function or the called function has responsibility for saving and restoring the registers.** Most importantly for us, it prevents name-mangling. When C++ code is compiled, the names of functions are changed drastically to include extra information. This is commonly called name-mangling. The language C also changes the name, but it merely pretends an underscore to the front of the name. The "extern "C"" tells the compiler to do this simpler, C-style name change instead of the more complex, standard C++ name-mangling. Leaving out the "extern "C"" will result in an error like:

```
asm_tut.obj : error LNK2001: unresolved external symbol
"void _cdecl clear(void)" (?clear@@YAXXZ)
```

■ Step 2　Add Assembly Code

Add the file that contains your assembly source code to the project. If this hasn't been created yet, you can do this by selecting FileView in the Project Window, right-clicking on the project's name and selecting "Add files to project..." When the dialog box appears, type in the name you want the assembly code file to be saved as (in our case, clear.asm). VC++ will warn you that the file does not exist and ask if you want to create a reference to it in the project anyway. Select Yes. Expand the tree listing in the project window until you see the name of the assembly file (clear.asm). Double-click the file name. VC++ will ask if you want to create a new file with that name. Select Yes. A new file will be created and opened in the editor.

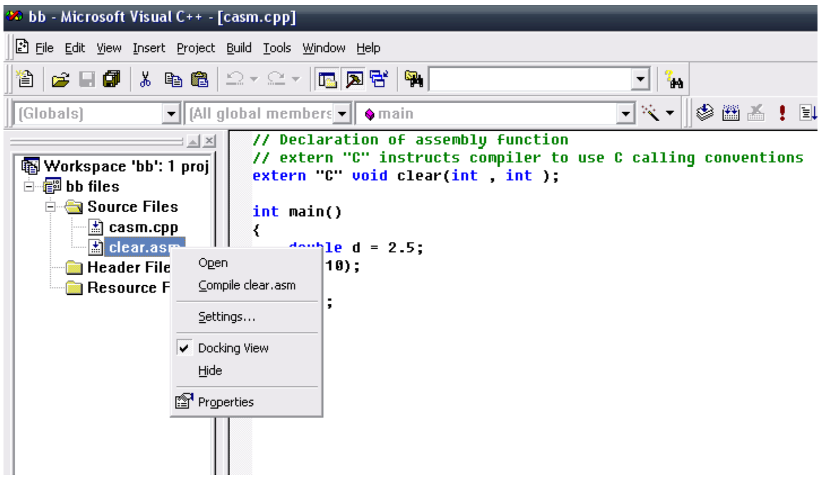
Enter you assembly code. For this tutorial, we will clear the EAX and EBX registers. To do this, we'll use this code:

```
.386                    ;Target processor.  Use instructions for Pentium class
machines
.MODEL FLAT, C          ;Use the flat memory model. Use C calling conventions
.STACK                  ;Define a stack segment of 1KB (Not required for this
example)
.DATA                   ;Create a near data segment.  Local variables are
declared after
                        ;this directive (Not required for this example)
.CODE                   ;Indicates the start of a code segment.

clear PROC
        xor eax, eax
        xor ebx, ebx

        ret
clear ENDP
END
```

■ Step 3 Set Custom Build Commands

We now provide the commands that VC++ will use to compile the assembly code. Visual C++ does not compile assembly source code. It must call an external program named ml.exe to perform the compilation. The commandline must be added to Custom Build options of the Project Settings.

1.To do this, right-click on the assembly filename (clear.asm) in the Project window. Select "Settings…" from the pop-up menu.



```
// Declaration of assembly function
// extern "C" instructs compiler to use C calling conventions
extern "C" void clear(int , int );

int main()
{
        ble d = 2.5;
     10);

     ;
```

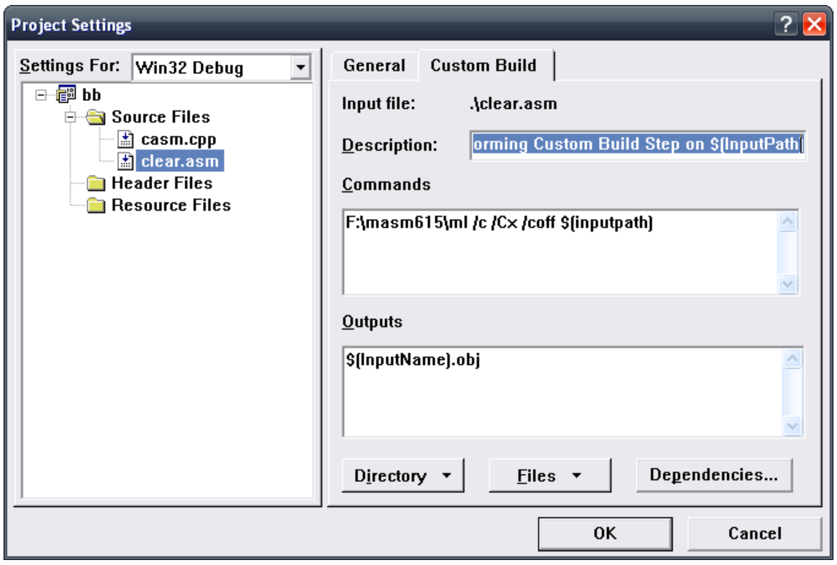
2.Select the "Custom Build" tab. In the "Commands" text box, type

c:\masm615\ml /c /Cx /coff $(inputpath)

THE SWITCHES IN THIS COMMAND ARE CASE SENSITIVE. /c and /C are very different and will create errors.

3.The string $(inputpath) is a macro which expands to the path and name of the assembly code file. Under "Outputs", type:

$(InputName).obj



Input file: .\clear.asm

Description: orming Custom Build Step on $(InputPath)

Commands

```
F:\masm615\ml /c /Cx /coff $(inputpath)
```

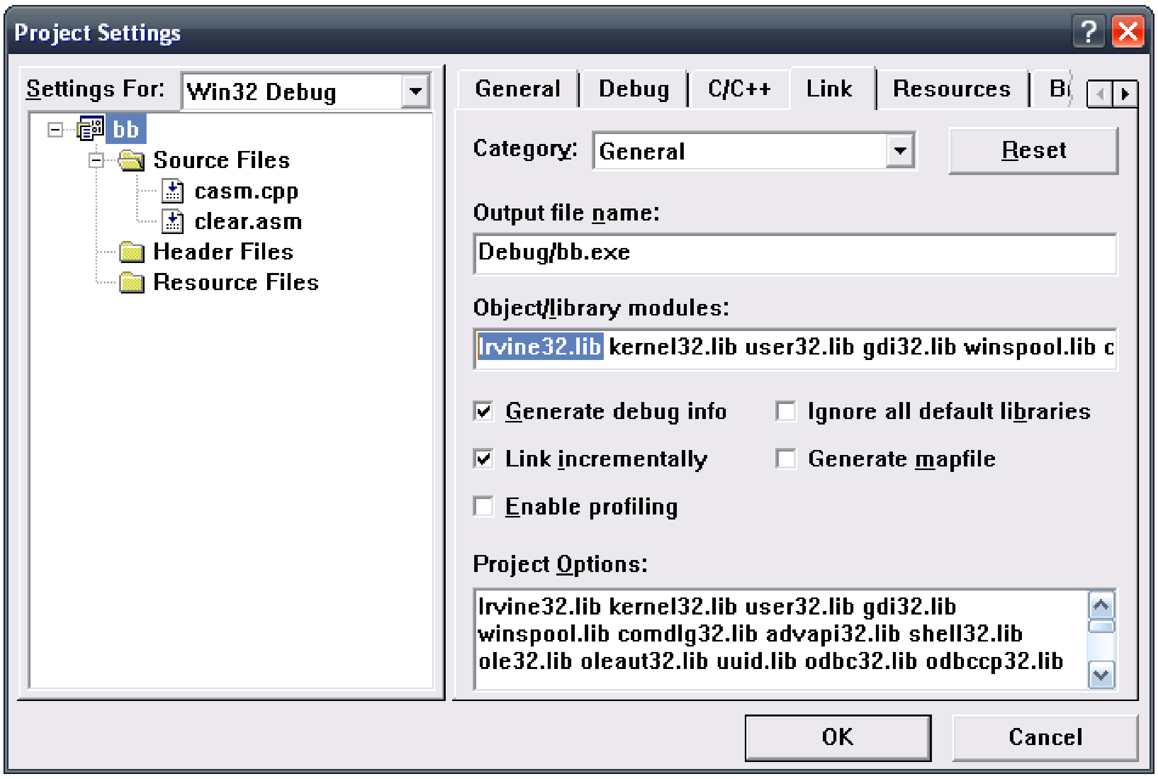
Outputs

```
$(InputName).obj
```

This tells VC++ that the results of the Custom Build will be an obj file with the same name as the assembly file (e.g., our sample project will generate a file named clear.obj which will be linked into the final program). Select "OK" to close the dialog box.

4.Copy ***Irvine32.lib*** from C:/MASM615/LIB to C:/Program Files/Microsoft Visual Studio/VC98/Lib

5.Select Project -> Settings from Visual C++ 6.0. Then select tab “Link”.
Insert “Irvine32.lib” in the line “Object/library modules:”. Then Select “OK”.



Category: General

Output file name: Debug/bb.exe

Object/library modules: Irvine32.lib kernel32.lib user32.lib gdi32.lib winspool.lib c

[✓] Generate debug info
[ ] Ignore all default libraries
[✓] Link incrementally
[ ] Generate mapfile
[ ] Enable profiling

Project Options:

```
Irvine32.lib kernel32.lib user32.lib gdi32.lib
winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

■ Step 4 Compile and Link

The project may now be compiled, linked, and run like any other VC++ project. Press F7 to Build the project and F5 to execute the program.