Table of Contents

1. Programming in Assembler Laboratory 2
1.1. Laboratory 1 2
1.1.1. Instroduction 2
1.1.2. Functions and functions calling from assembler 2
1.1.2.1. Function calling convention 3
1.1.3. Practice: Creating a C++ project and conversion to ASM 4
1.1.3.1. Creating a source code file 5
1.1.3.2. Converting a project from C++ to ASM 6
1.1.3.3. Creating a starting point and completing libraries 7
1.1.3.4. Marking an ASM file for compilation 9

1. Programming in Assembler Laboratory

1.1. Laboratory 1

Programming in Assembler Laboratory, exercise 1

1.1.1. Instroduction

In this exercise, you will create a console application in pure assembler for x64 (Intel/AMD processors, 64 bits) that will interact with the Windows operating system (Windows 7, 8, 8.1, 10, 11) by calling its functions via the Windows API.

This is just an example, creating complex applications with advanced user interfaces in this way is possible although rather pointless. On the other hand, it can be very useful when making simple computational services, tools, etc., which require compact code and execution of tasks at maximum speed. Note that the examples given are not limited to user interface functions, but cover the entire Windows API.

Since there is no ready-made project model for pure assembly applications in Visual Studio, we will use the C++ console application model and adjust it for pure assembler code compilation instead of C++ code.

This approach requires manual reconfiguration of several parameters in the project properties, in particular:

- changing the compiler to `masm`;
- changing the target platform to x64;
- providing the name of the main function as the entry point ([Entry Point]) so that the operating system knows which function to call when the application is launched - in many programming languages and environments, this is the `main()` function by default, but in the case of assembler, there is no such rule and you have to specify which function you are launching;
- supplementing missing libraries: since version 2019 of the Visual Studio environment some standard functions previously available by default in libraries linked to each project have been disabled and must be explicitly indicated.



Note! Because we are using `asm` files, which Visual Studio does not natively recognise as source code files, they are not automatically saved when you invoke compilation like `.cpp, .cs` files. Please remember to always click save the file (Ctrl+s) before compiling.

1.1.2. Functions and functions calling from assembler

When writing code in assembler, each byte must be allocated, declared and used in an appropriate manner, especially in the context of calling Windows API functions.

Since we are working in an operating system, it is necessary to follow the rules of function invocation known as calling convention.

1.1.2.1. Function calling convention

We pass function parameters via registers, as long as their number does not exceed four. Any additional parameter we pass through the stack (fifth and following parameters). However, it is mandatory to allocate space on the stack for arguments even if we pass them via registers. [1)]. It is called shadow space.

1.1.2.1.1. Passing integer arguments

When passing integer numeric arguments to a function, in particular addresses and pointers, the following registers should be used in the order: - argument 1 → RCX, - argument 2 → RDX, - argument 3 → R8, - argument 4 → R9, - each subsequent argument is placed on the stack, maintaining 16-bit data alignment (for performance reasons). Sample C++ code:

```
func1(int a, int b, int c, int d, int e, int f);
// a in RCX, b in RDX, c in R8, d in R9, f, and then e, are pushed onto the stack
```

1.1.2.1.2. Passing floating-point arguments

When passing floating-point arguments, we load subsequent values into XMM registers: - argument 1 → XMM0, - argument 2 → XMM1, - argument 3 → XMM2, - argument 4 → XMM3, - each subsequent parameter, similarly as for fixed-point, through the stack. Sample C++ code:

```
func2(float a, double b, float c, double d, float e, float f);
// a is XMM0, b in XMM1, c in XMM2, d in XMM3, f and then e are pushed onto the stack
```

1.1.2.1.3. Passing mixed arguments

When mixing floating point and decimal arguments, the positional system is used depending on the type, so if we pass, for example, int1, float1, float2, int2, the values should be placed in registers (in order): - int1 → RCX, - float1 → XMM1, - float2 → XMM2, - int2 → R9, - fifth and next argument via the stack. Sample C++ code:

```
func3(int a, float b, float c, int d, int e, float f); // a in RCX, b in XMM1, c in XMM2, d in R9, f, then e, via the stack
```



Registers values are changed after a function call, so you should not expect that after a function call, registers RCX, RDX, R8, R9, XMM0-XMM3 will have the same values as before the function call

1.1.2.1.4. Returning a value from a function

An integer value (including a pointer) is returned in register RAX.
Floating-point values are returned in register XMM0.

1.1.2.1.5. Volatile and non-volatile registers



Some registers must remain unchanged upon return from the function, in relation to their contents at the beginning, right after the function call (so-called non-volatile). Other registers can be changed freely as it will not affect the operation of the operating system (volatile).
If it is necessary to use this first group, the contents of the register can be pushed onto the stack or saved in memory and restored before executing the `ret` instruction.

Registers that can be changed with impunity inside the function, without having to restore the original value, are (x64 calling conventions):

- RAX, RCX, RDX, R8, R9, R10, R11, and XMM0-XMM5,
- additionally in processors that are equipped with YMM registers, the upper halves of the YMM0-YMM15 registers can also be changed without having to restore their values,
- and in processors equipped with functions AVX512VL, all registers ZMM, YMM and XMM16-XMM31.

Registers that must not be changed (or rather must be restored to their original values before exiting the function) are:

- RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15 and XMM6-XMM15.

1.1.3. Practice: Creating a C++ project and conversion to ASM

Create a new project: a command-line (console) application for C++, native (not C++ .NET) figure 1

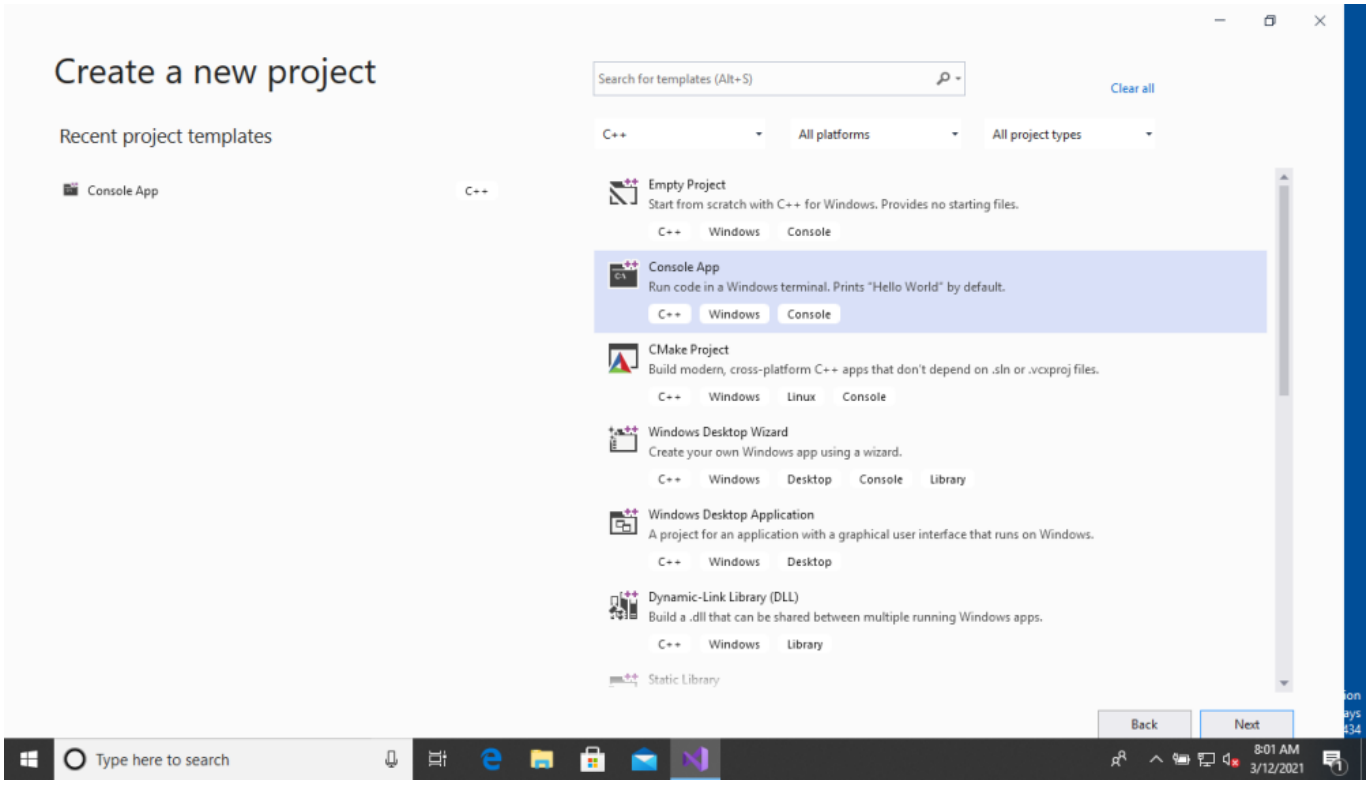


Fig. 1: Creating a C++ project, native code

1.1.3.1. Creating a source code file

Delete the generated .cpp file (from VS) and then add a new .asm file (this can be done by clicking on the project (not the solution!) and selecting to add a new element figure 2, then select the text file and change its name and extension figure 3:

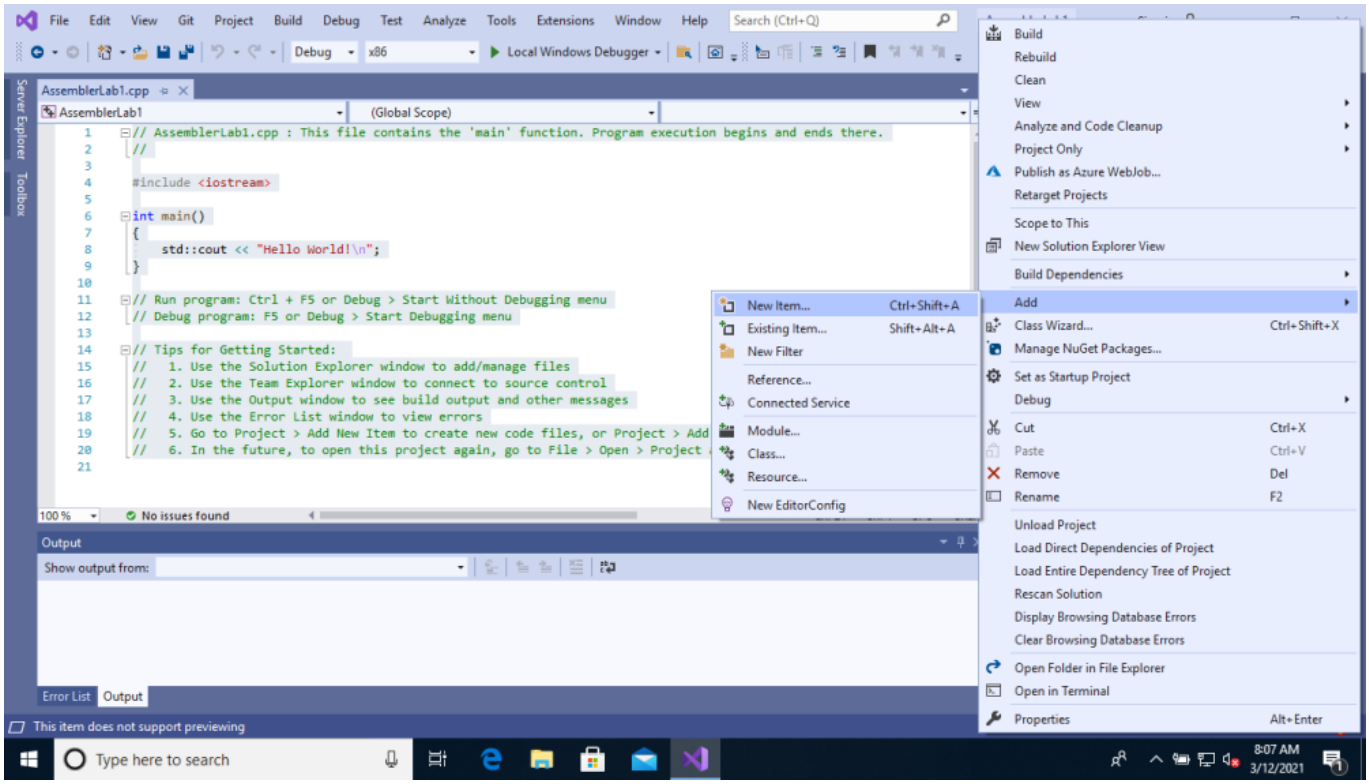


Fig. 2: Adding a new element

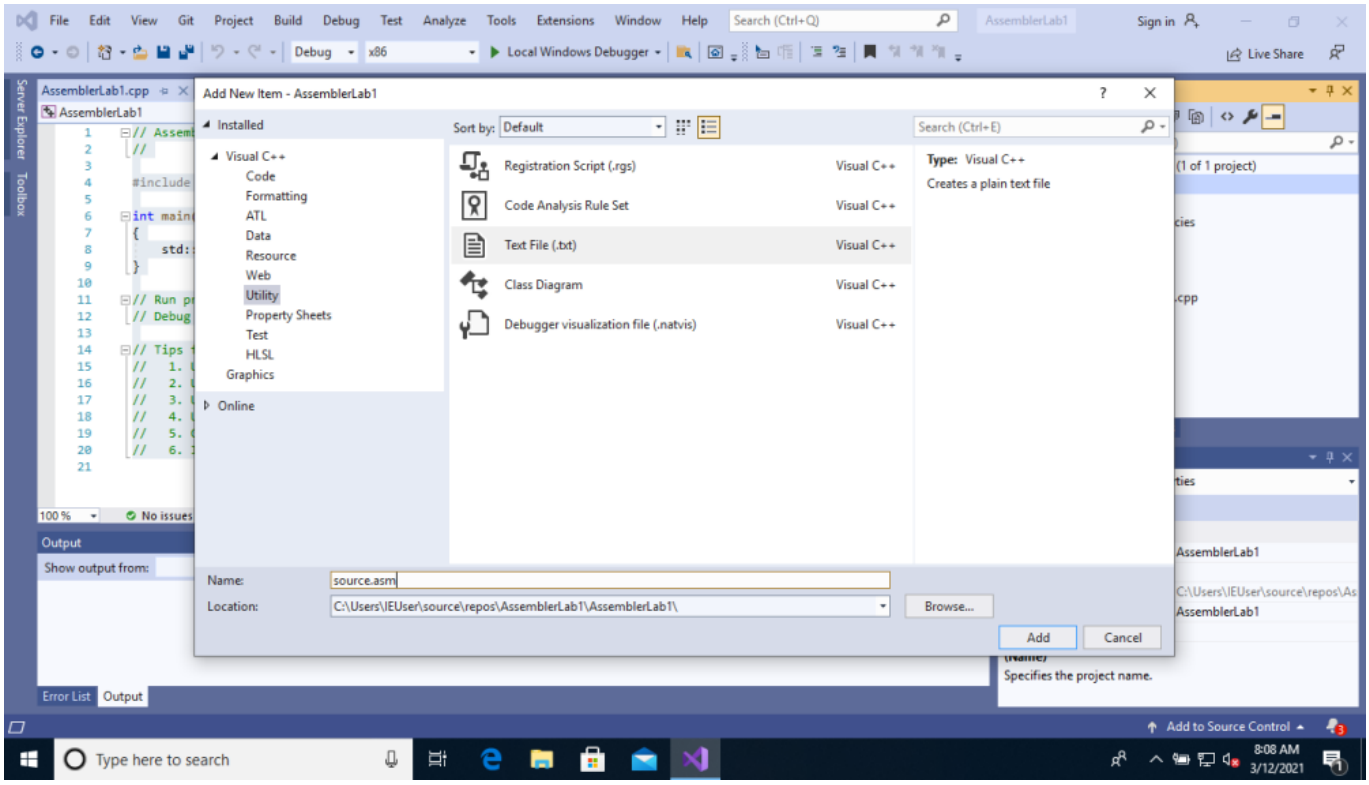


Fig. 3: Adding a new element - asm file

1.1.3.2. Converting a project from C++ to ASM

To convert, you need to tell VS that the source files will be compiled using MASM and not the C/C++ compiler. To do this, right-click on the project and select the option to customize the project build figure 4. In the settings, select `masm(.targets, .props)` figure 5.

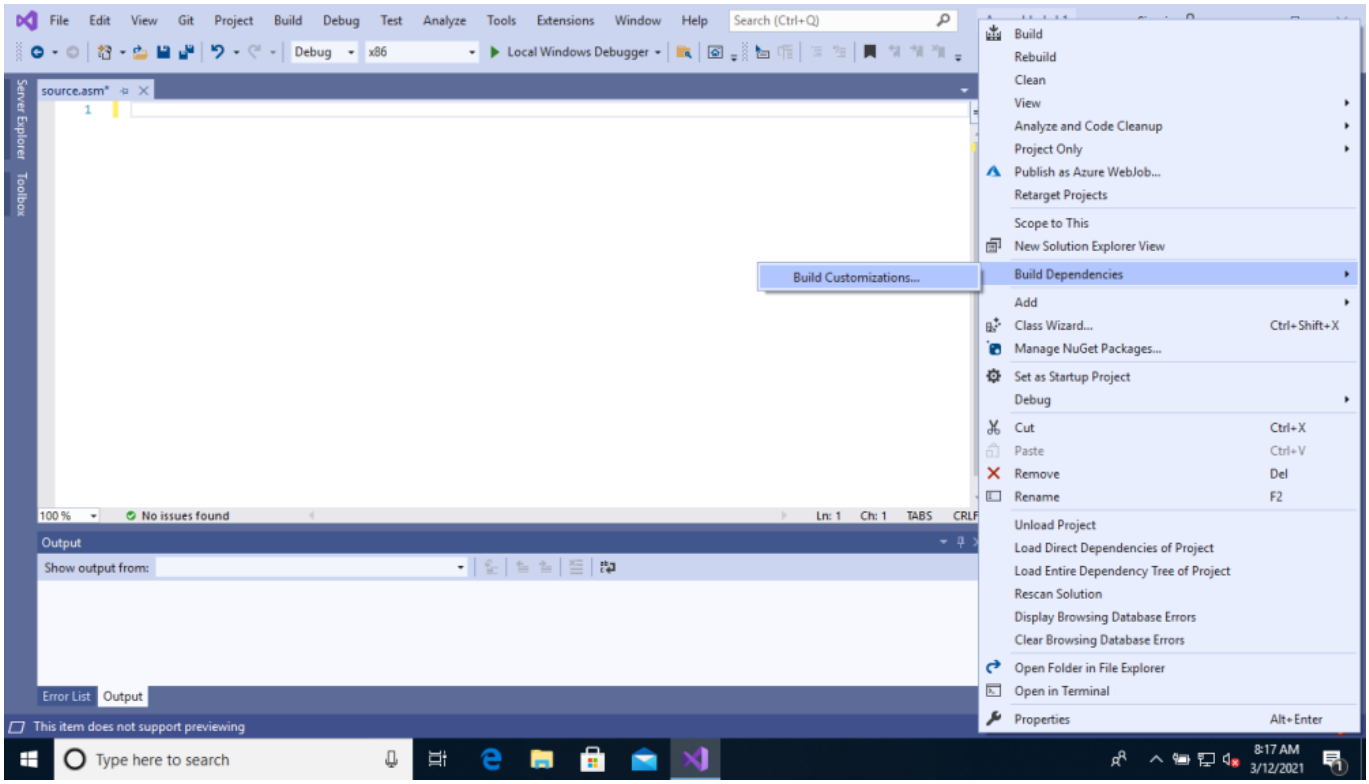


Fig. 4: Choosing the project build customization menu

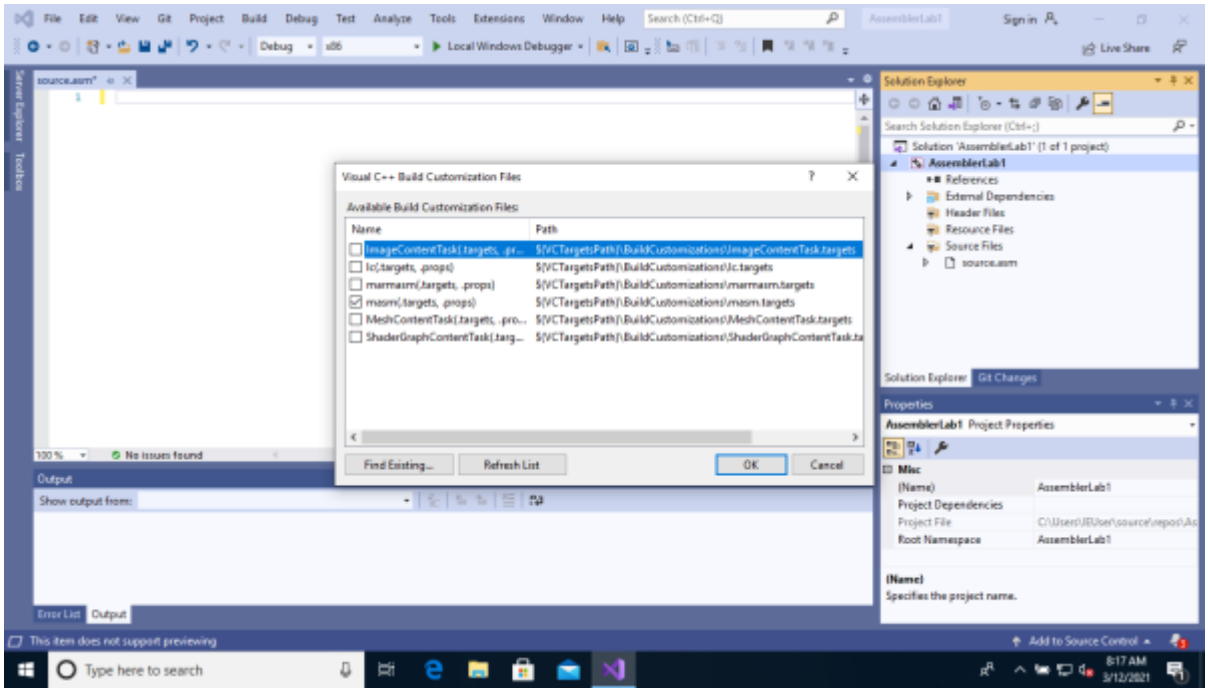


Fig. 5: Choosing the MASM compiler

Next, change the type of application being built from x86 (32-bit application) to x64 (64-bit application): Build→Configuration Manager menu figure 6.

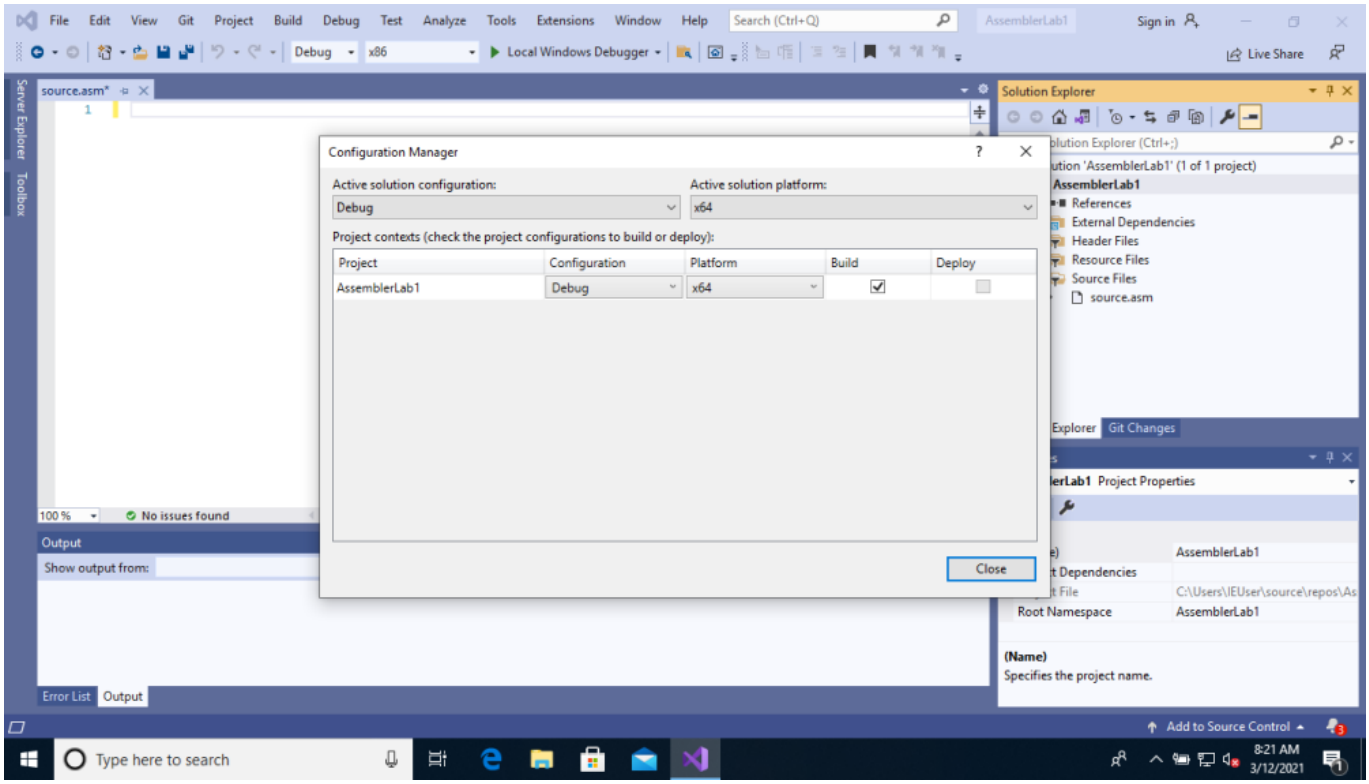


Fig. 6: Environment configuration - building the 64-bit version

1.1.3.3. Creating a starting point and completing libraries

In this step, we will inform the linker what starting point to enter into the executable file. In high-level languages, this is the `main()` function by default. In the case of assembler, there is no such assumption, so the source code must be consistent with the value set here.



The name of the code entry point function given to the linker must match the source code. In case of discrepancies, the program will work incorrectly, will not start or will throw an exception at startup!

In the given example, the name of the function is `myFunctionAsm`. To get to the configuration window, you should select the project (not the solution!) in the project tree, then right-click on the project, select `Properties`, or use the shortcut [Alt+Enter]. A window will appear as in the figure 7. Then in the `Linker` section, select the `Advanced` option and in the `Entry Point` field enter the name of the function.

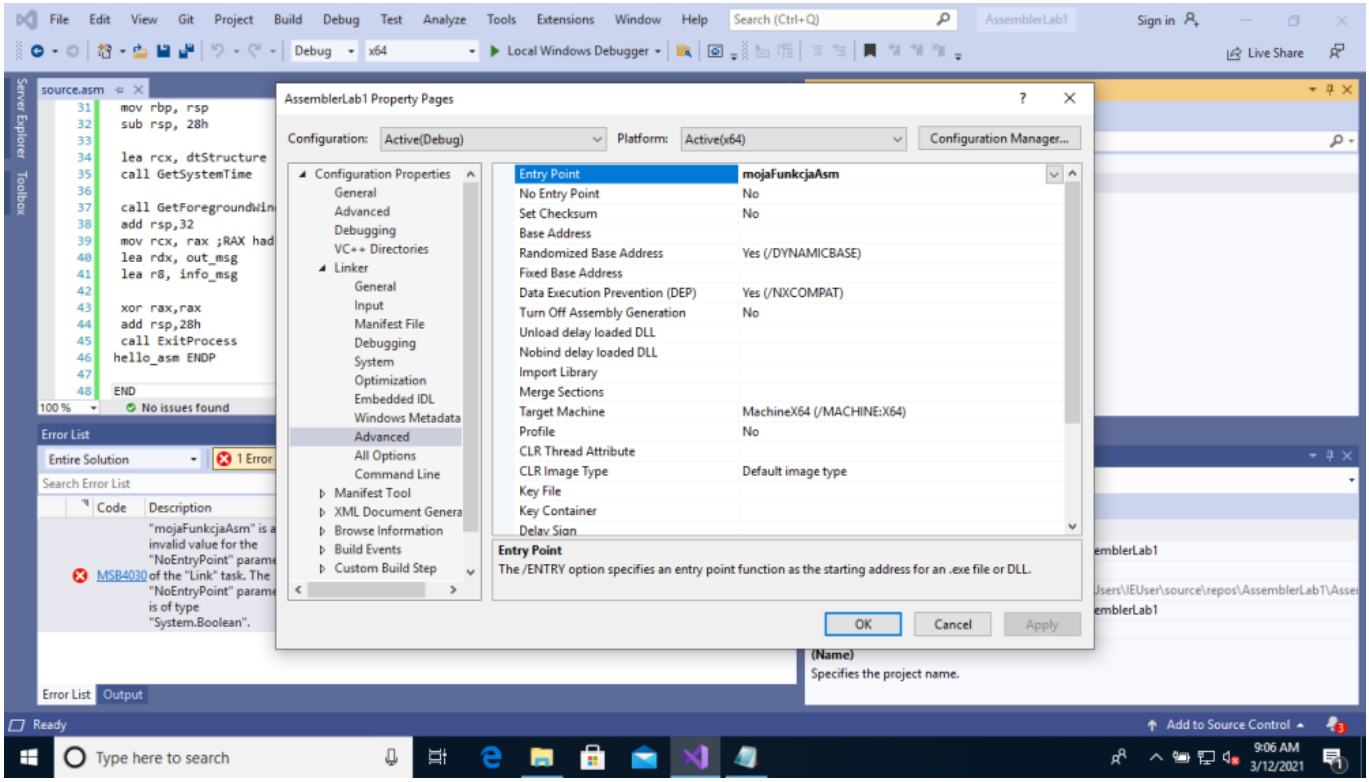


Fig. 7: Configuration of the program entry point

In this section, you should also add the missing libraries that we will use. Note - depending on the Windows version and current update level, this step may not be necessary, but it will not hurt. In particular, these libraries contain functions such as `sprintf`.
Note - the given libraries should be added to those that are already there by default. Removing the libraries located there (including `kernel32.lib`, `user32.lib`) will result in the inability to use standard dialog boxes and interact with the Windows API. To add libraries, select the `Input` item in the `Linker` menu, then select `Additional Dependencies` and the `<Edit..>` option after expanding the selection list. Then, in the upper field, enter the names of library files that are to be additionally available during compilation (one item per line).
Libraries to be added are:

- `legacy_stdio_definitions.lib`
- `legacy_stdio_wide_specifiers.lib`

Details in figure 8.

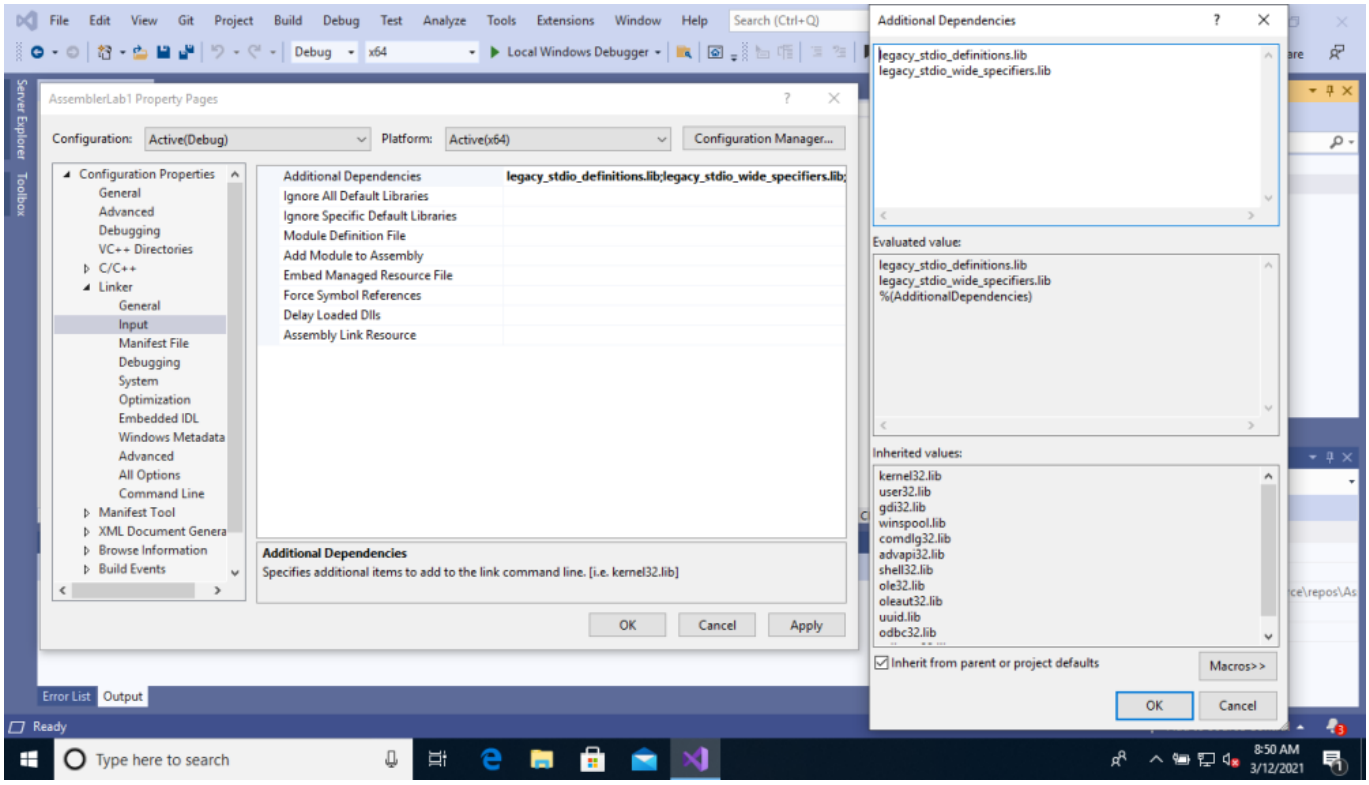


Fig. 8: Configuring additional libraries

1.1.3.4. Marking an ASM file for compilation

In some cases, the `.asm` file is not automatically recognized as a source file with assembler code. Therefore, you must explicitly inform the environment about this. To do this, right-click the file name in the project tree and select `Properites`, then in the `General` section, select `Item Type` and set its value to `Microsoft Macro Assembler`. Details in figure 9.

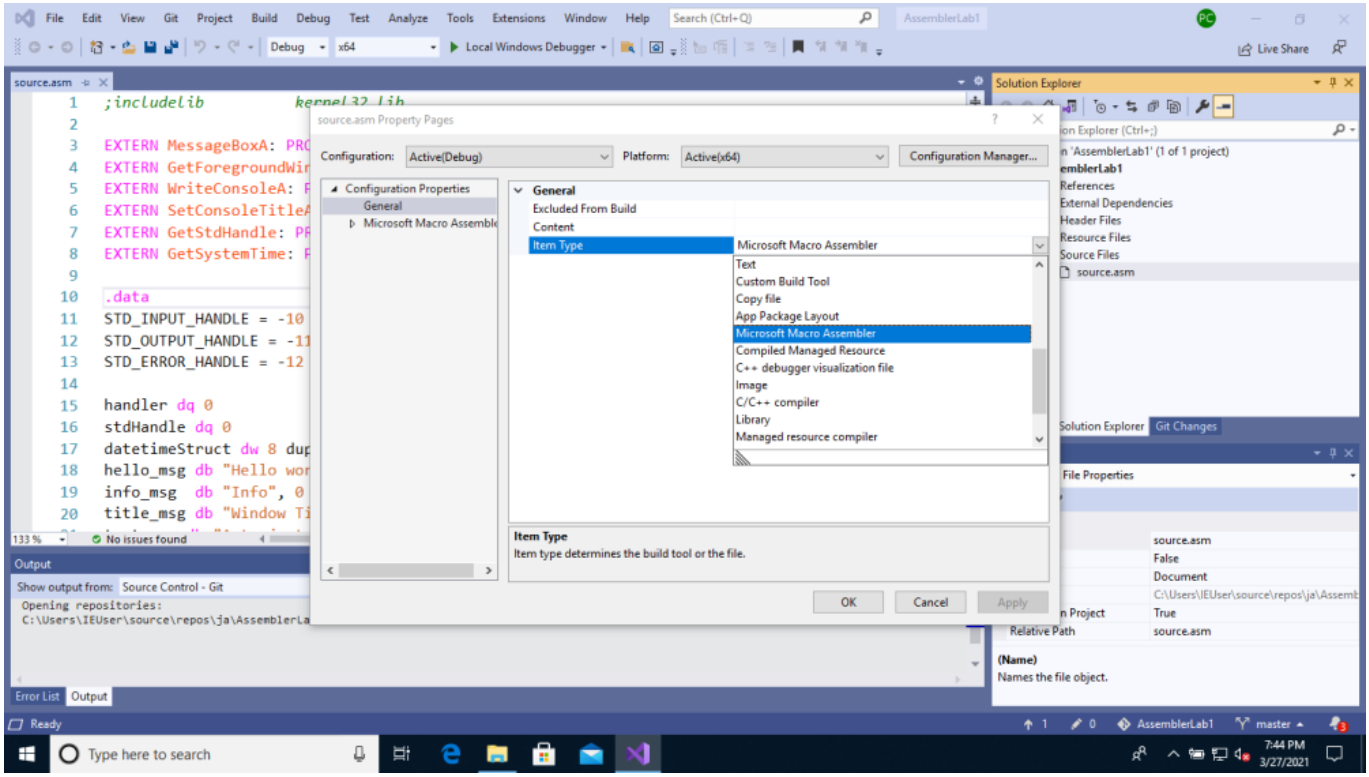


Fig. 9: Marking an ASM file as a source file with assembler code
[1)]

https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s1:apl:lab:lab1?rev=1731961152

Last update: **2024/11/18 21:19**