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
1.1.4. Practice: Task to be performed 10
1.1.4.1. Code fragments to use 10

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

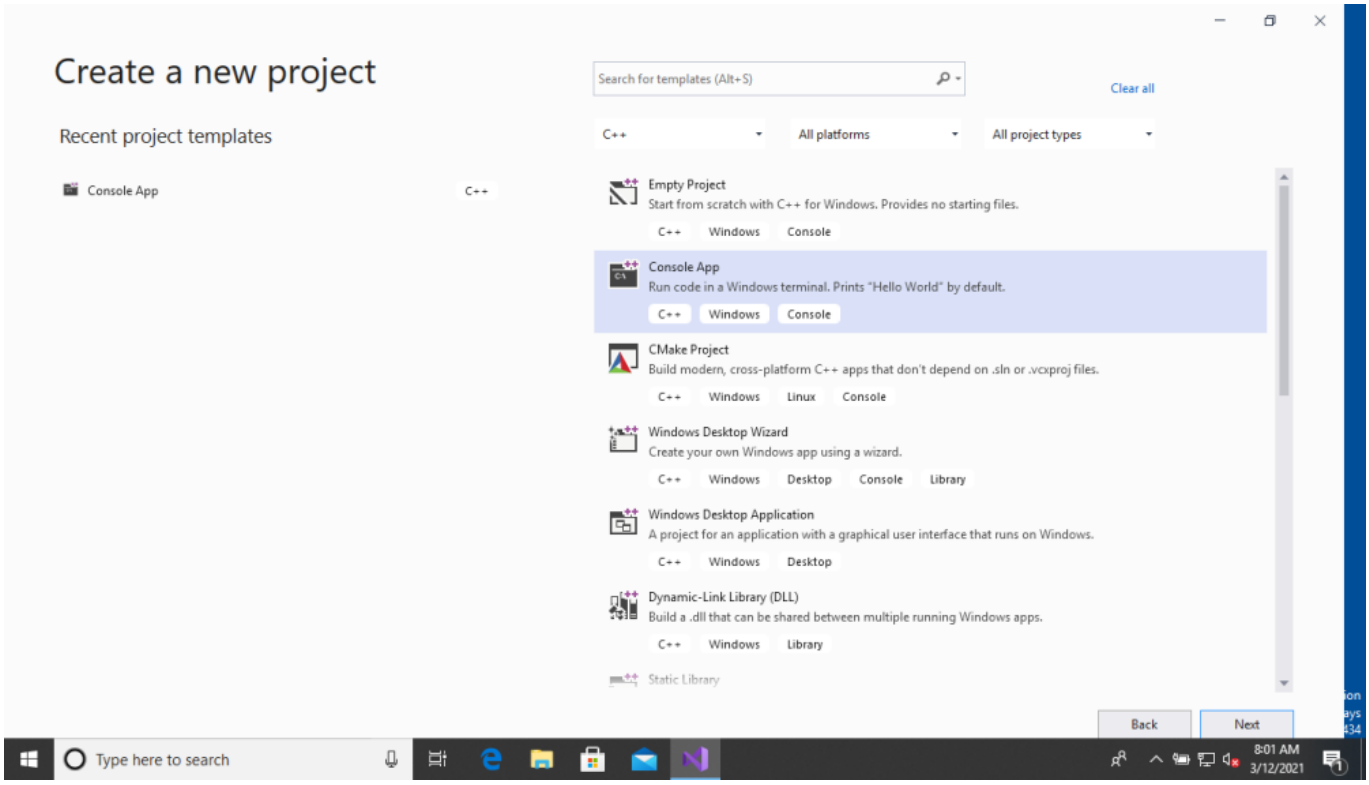


Fig. 1: Creating a C++ project, native code

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:

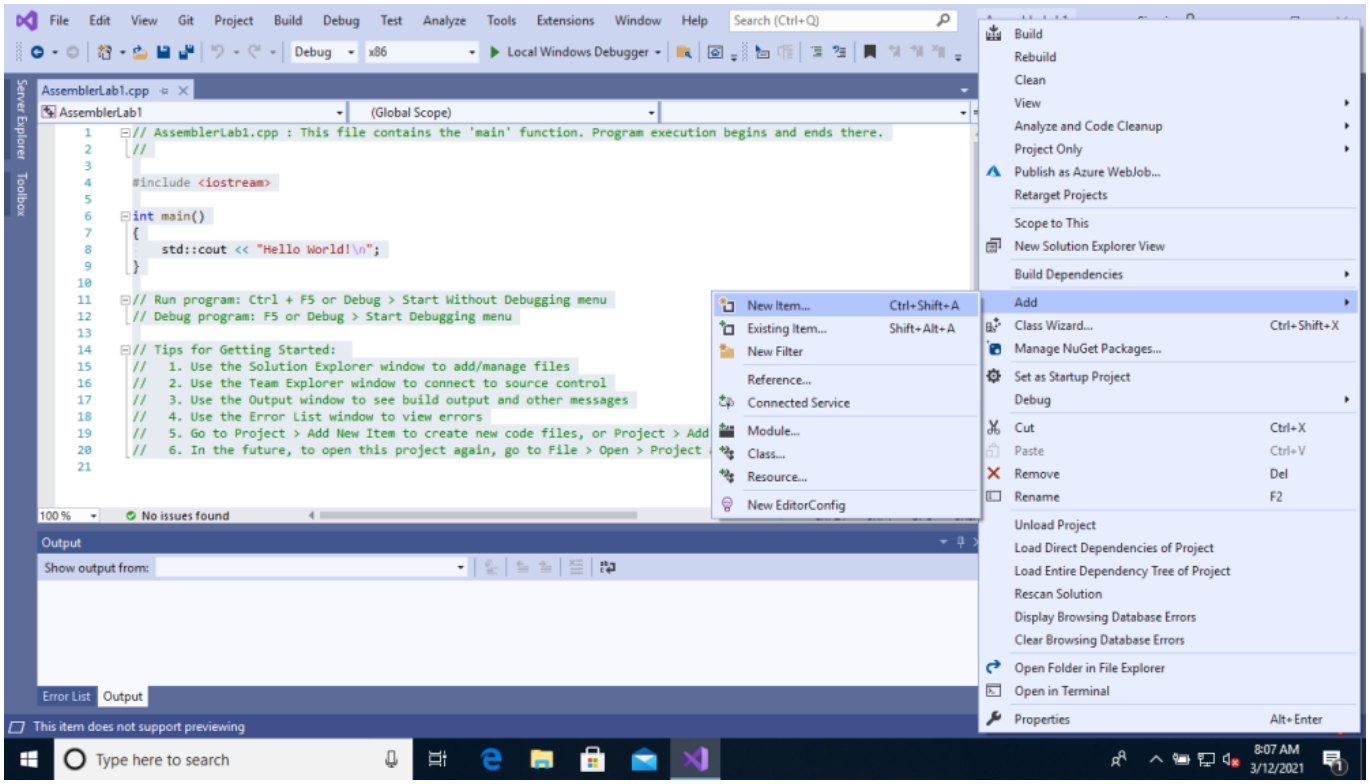


Fig. 2: Adding a new element

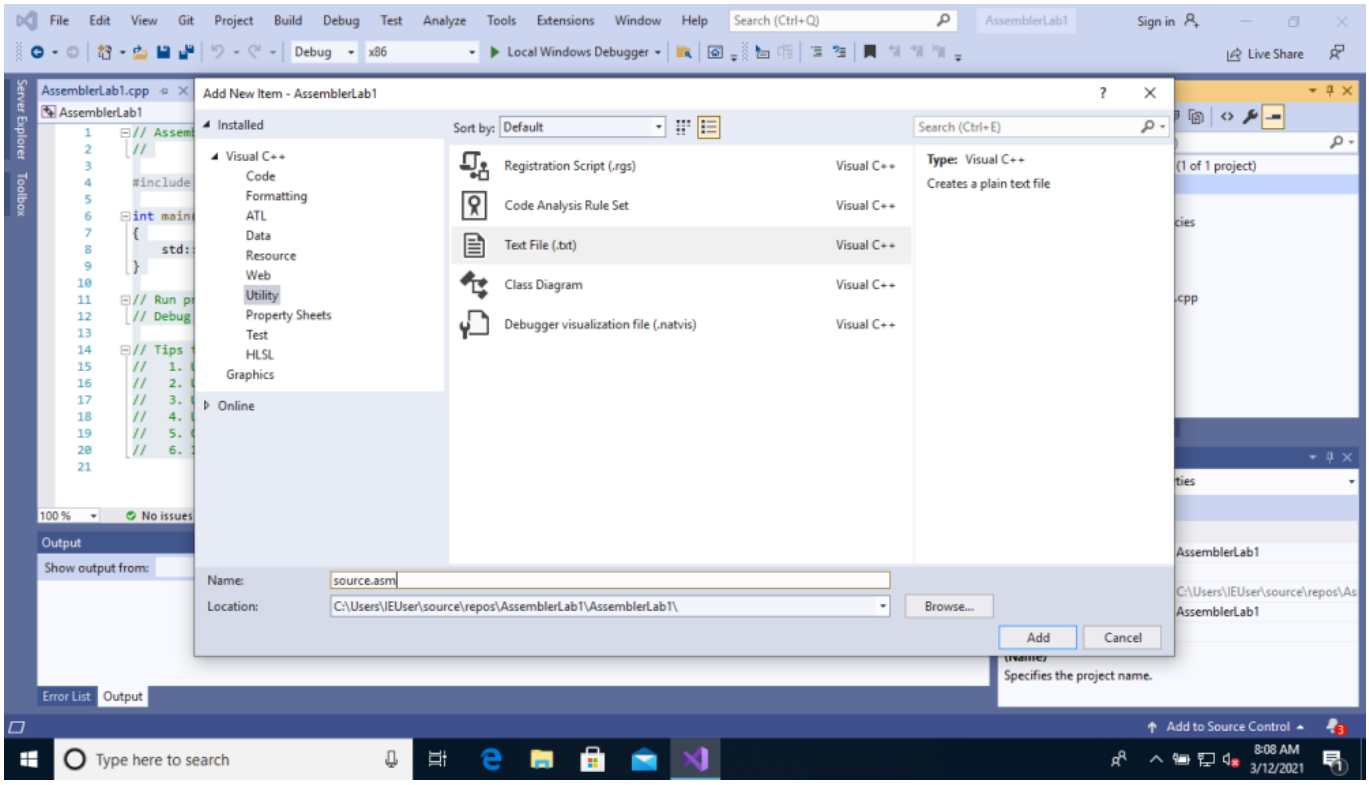


Fig. 3: Adding a new element - asm file

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.

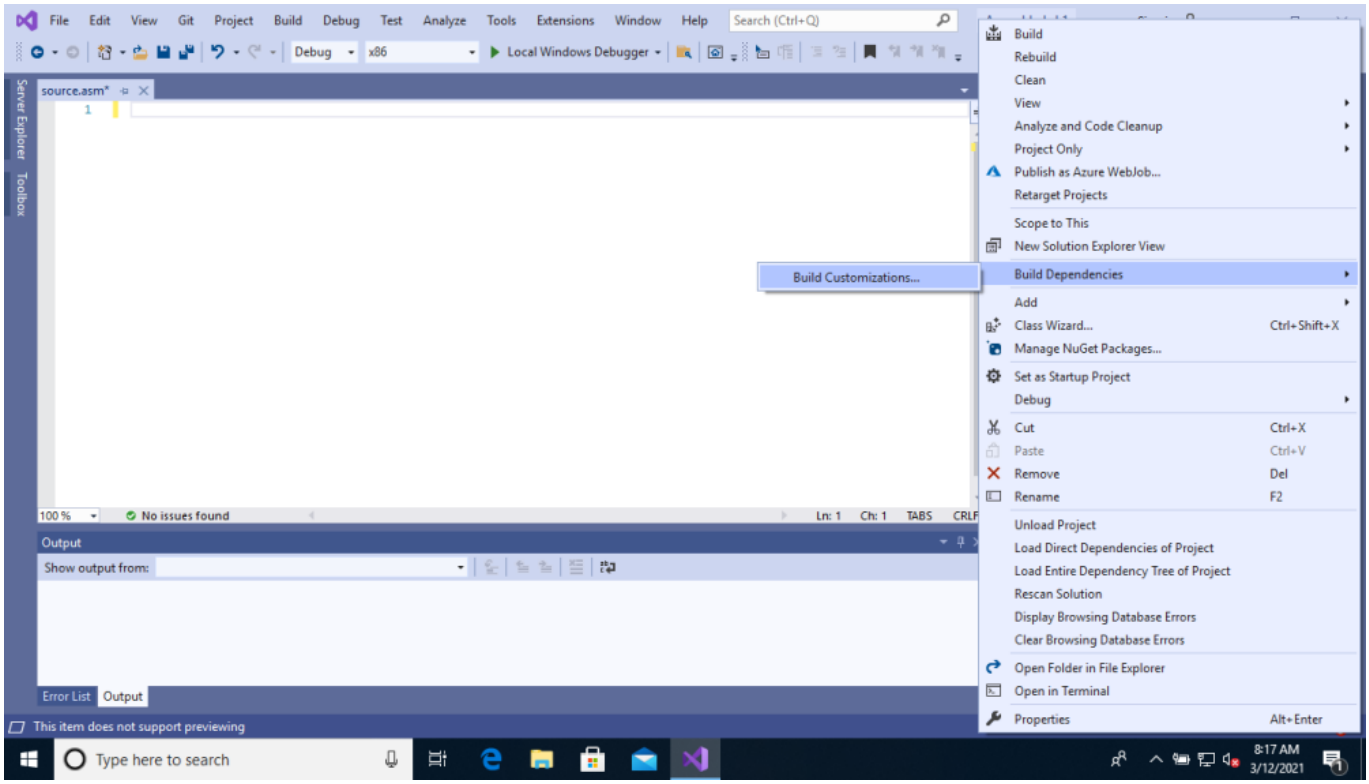


Fig. 4: Choosing the project build customization menu

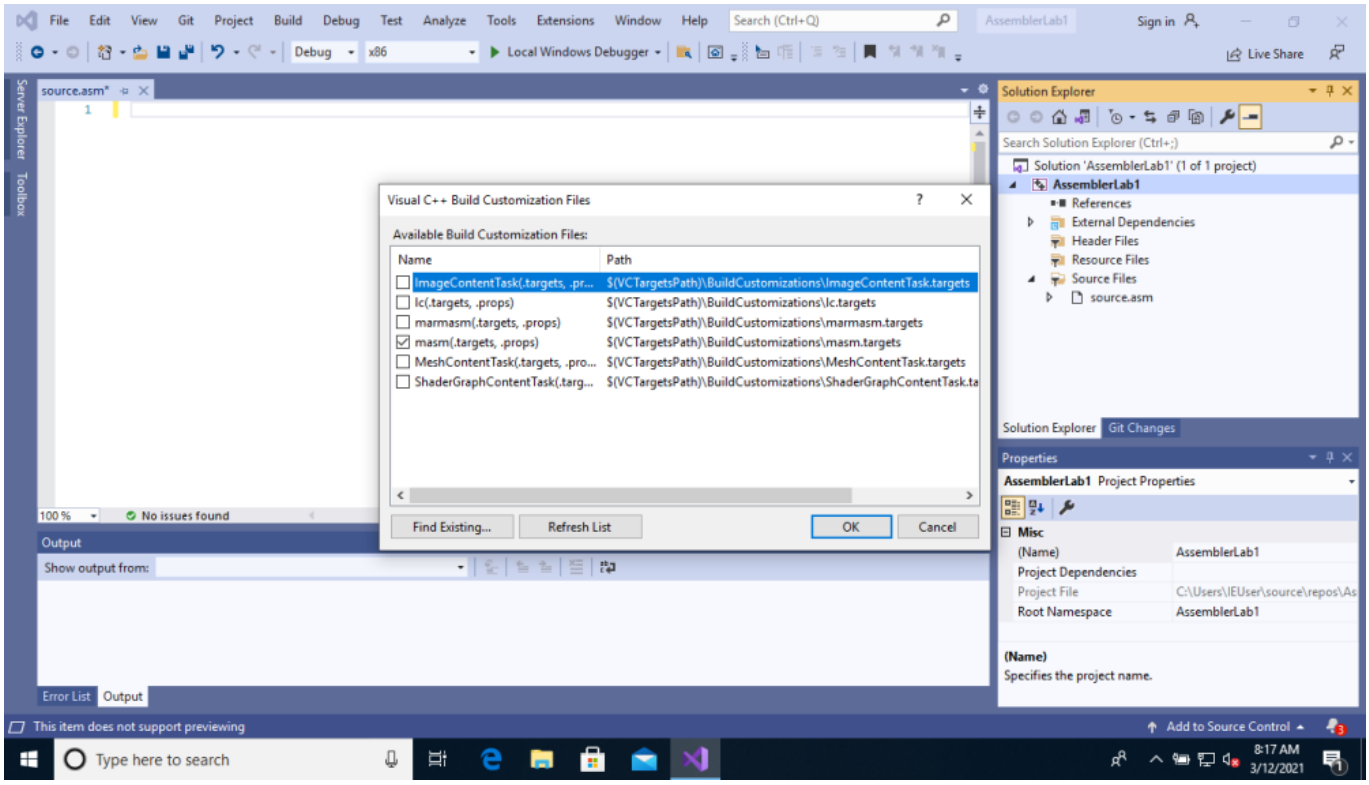


Fig. 5: Choosing the MASM compiler

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

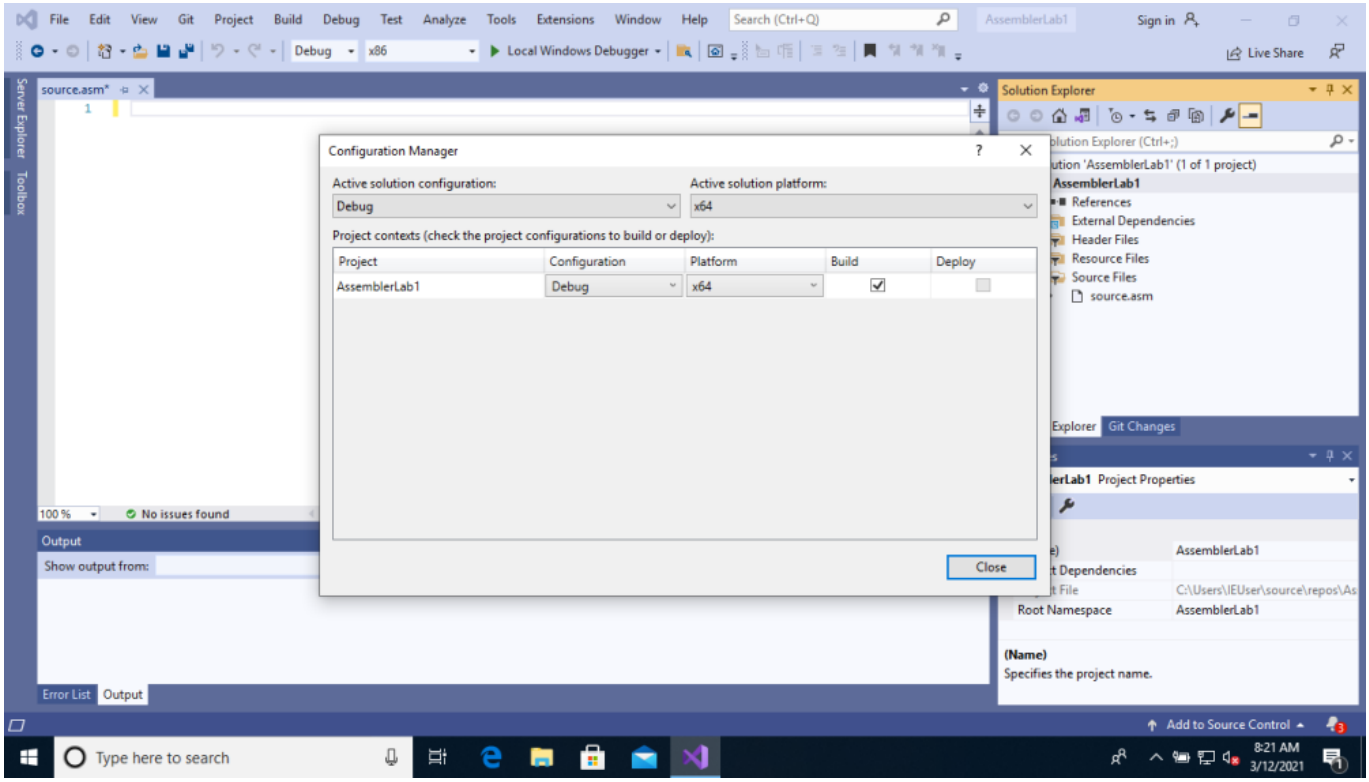


Fig. 6: Environment configuration - building the 64-bit version

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 an 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. In our case, it will be `hello_world_asm`.

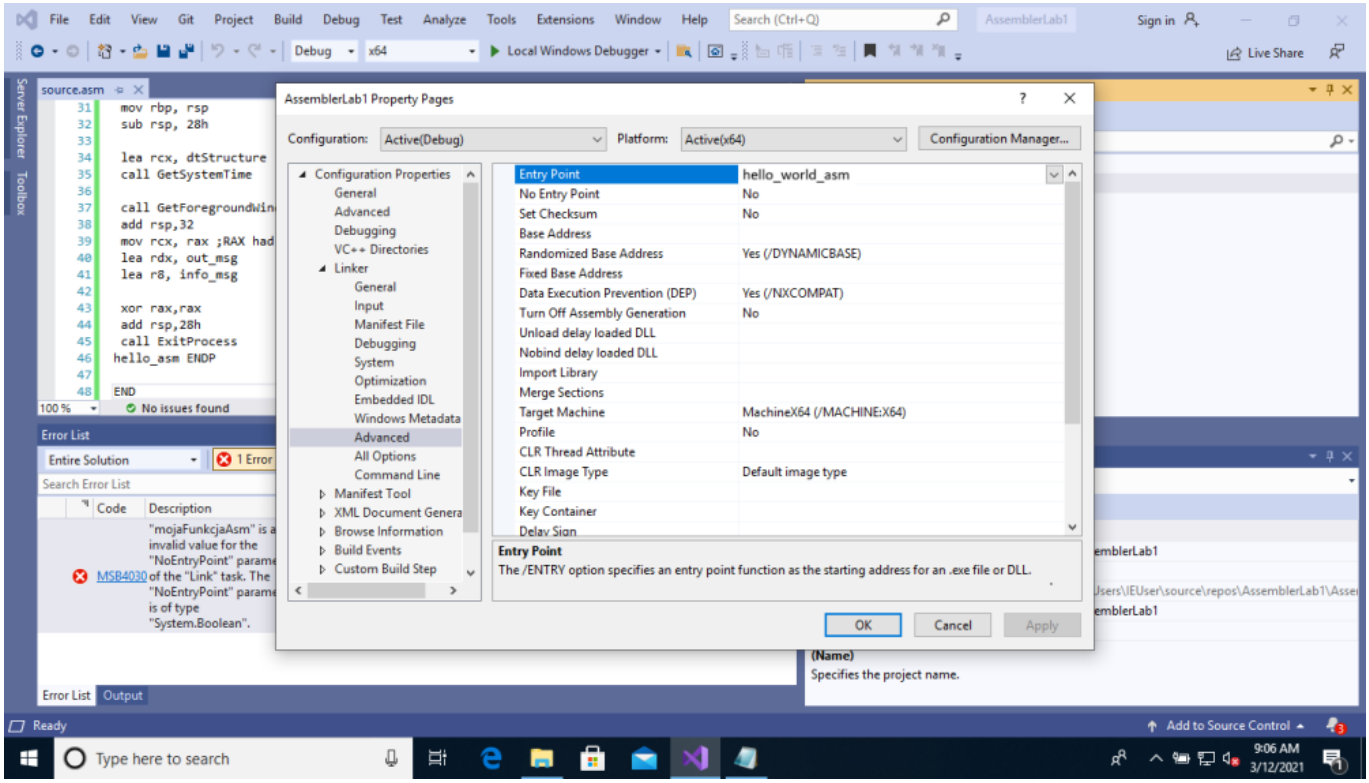


Fig. 7: Configuration of the program entry point

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.

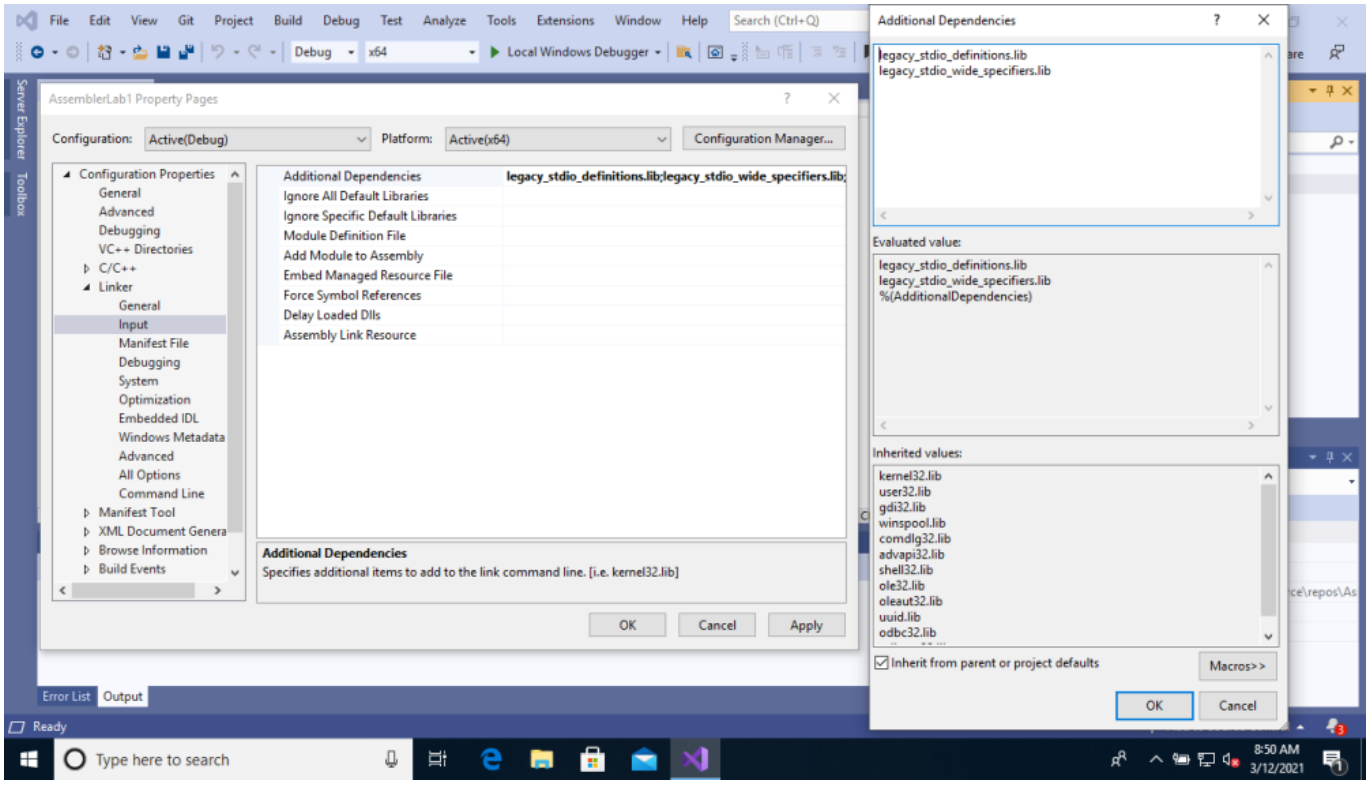


Fig. 8: Configuring additional libraries

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.

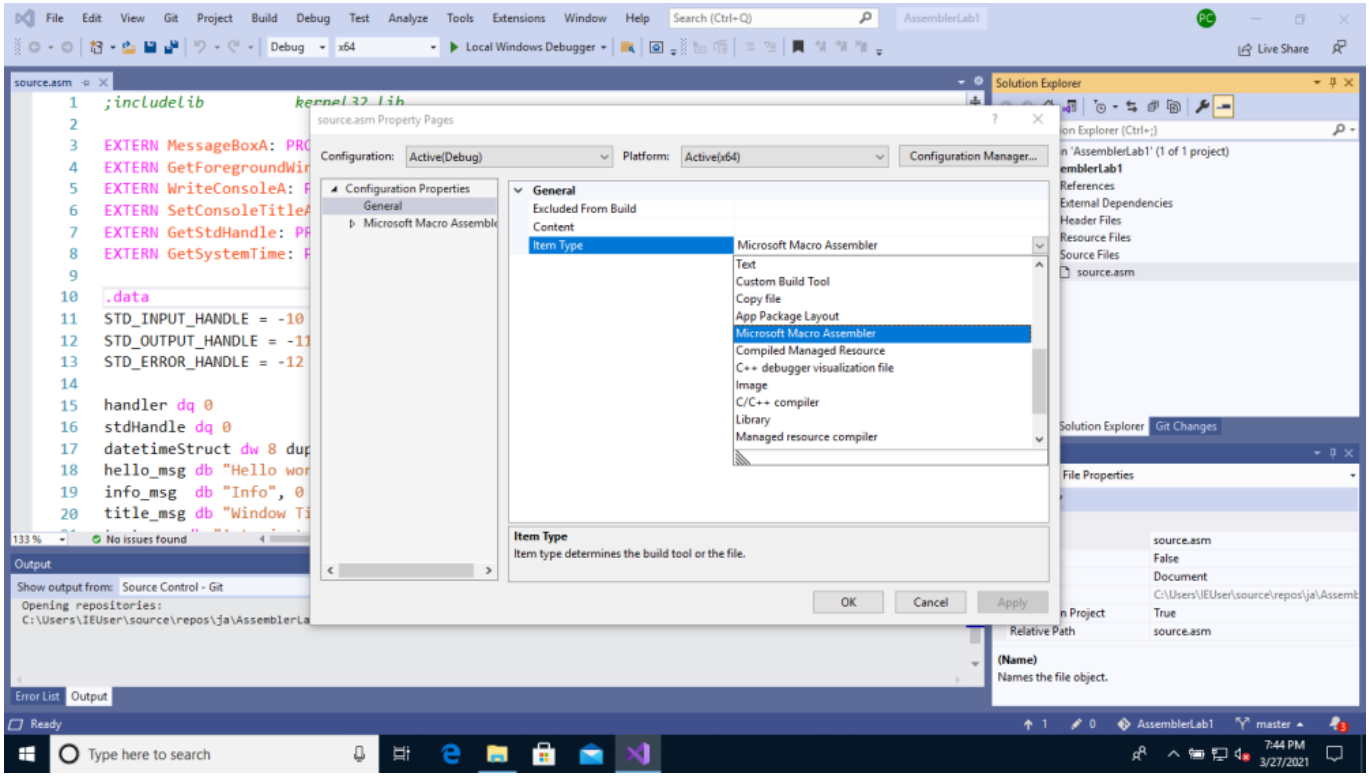


Fig. 9: Marking an ASM file as a source file with assembler code

1.1.4. Practice: Task to be performed

Implement reading of the current date (use the function `GetSystemTime`(sysinfoapi.h)[2] reading the structure `_SYSTEMTIME`(minwinbase.h)[3]. The structure consists of 8 16-bit values.



Note - do not declare the structure as such, just for simplicity declare a target, static array of 8 Word (16bit) values in the data section.

Convert the year, month and day to ASCII text form and display on the screen using:

- in the command line (cmd) using the `WriteConsoleA` function;
- in the dialog box, using the `MessageBoxA` function;



Pay attention to the position of individual words in the structure and their corresponding values.

You can also display the current hour, minutes and seconds.

1.1.4.1. Code fragments to use

Remember to declare the code and data sections using the `.data` and `.code` directives. At the end of the code, place the `END` clause.



Code and Data sections historically were named segments. You can still find such names in some sources.

1.1.4.1.1. Data section

Each text string must end with a byte with binary value 0. It is named the NULL character. We do not use international characters because we use the ASCII character set (8-bit), not Unicode.
A sample data segment that was used in the presentation looks like this:

```
...

.data
STD_INPUT_HANDLE = -10
STD_OUTPUT_HANDLE = -11
STD_ERROR_HANDLE = -12

handler dq 0
stdHandle dq 0
datetimeStruct dw 16 dup (0)
```

```
hello_msg db "Hello world", 0
info_msg db "Info", 0
title_msg db "Window Title JA",0
text_msg db "And this is a very long text to display in the console!",13,10
dummy dq 0

...
```

To use Windows API functions, you have to tell the assembler how they are named so the linker can match the names from your code to those in the library.
To do this, you need to use the EXTERN and PROC directives:
EXTERN means that the functions are not defined in your source code but present in external modules - here libraries. PROC specifies that the symbols represent the names of the functions.



In assembler there is no difference between function and procedure - every subroutine is named procedure.

```
...

includelib kernel32.lib

EXTERN MessageBoxA: PROC
EXTERN GetForegroundWindow: PROC
EXTERN WriteConsoleA: PROC
EXTERN SetConsoleTitleA: PROC
EXTERN GetStdHandle: PROC
EXTERN GetSystemTime: PROC

...
```

We also usually include information about the library.

1.1.4.1.2. Code section

After running the program, we need to save the values of certain registers, in order to restore them later. At the very end, you should also call a function that ends the program and returns a so-called return code to the operating system.

The minimal set of instructions looks like this:

```
...
.code
hello_world_asm PROC
push rbp ; save frame pointer
mov rbp, rsp ; fix stack pointer
sub rsp, 8 * (4 + 2)
```

```
.... ; your program code here

mov rsp, rbp
pop rbp
ret
hello_world_asm ENDP
END
```



Note, in the above example, the main procedure is `hello_world_asm` and it was indicated as the [Entry Point] to the program, in the configuration described above. Adjust to your naming.

As you can see, the whole thing comes down to reserving and putting space on the stack for the so-called shadow space, i.e. for saving the values of registers and restoring them after the program has finished executing.

In the program, we use interaction with graphic elements such as the main window (Foreground Window) and the data stream (STD_OUT). When calling the function, it is necessary to clearly indicate which graphic object we want to interact with or create a child object (e.g. MessageBox).
To do this, we need to get two handles:

- a handle to the current main application window at the operating system level (so-called Foreground Window)
- a handle to a terminal-type data stream, into which information can be entered.



Note! To display something on the console, you should always press Enter, even virtually. Therefore, any string of characters redirected to STD_OUT will appear on the console as long as there is a special code at the end indicating the ENTER character. These are two numeric values, in the order 13 and 10. Hence, in the example in the data segment, after the content there is `,13,10`. If you don't do this, you will need to go to the window and manually press ENTER to display the string of characters redirected there from your program. Otherwise, the window will remain empty and the text will not appear!

To read the handles, two functions from the Windows API will be useful, which will return pointers (handles) to the appropriate objects. These values should be saved and can be used multiple times.

Reading the handle to the main window, this handle should be used when invoking a dialogue window of the Message Box type (and any other):

```
...
call GetForegroundWindow
mov handler, rax
...
```

In the above example, the value returned by the function (via the RAX register) is saved in the variable `handler`, of the DQ type (64 bits, i.e. simply an address in memory).

To read the handle to the inside of the terminal, you need to use another function:

```
...
mov rcx, STD_OUTPUT_HANDLE
call GetStdHandle
mov stdHandle, rax
...
```

However, this function requires one argument, which data stream we want. There are three possibilities: * STD_INPUT_HANDLE = -10 - standard input - it can be used to read data from the console * STD_OUTPUT_HANDLE = -11 - standard output - it is used to write something in the terminal; remember the ending ENTER, i.e. `13,10`!; * STD_ERROR_HANDLE = -12 - stream for errors;

An example code that displays the text in the variable (at the address) `text_msg` on the console is below. The first parameter is the console handle read above. In the third parameter, you need to specify how many characters should be displayed. The fourth parameter is a pointer to where the function will return information about how many characters it actually displayed (it's not entirely clear what this is for).

```
...
; WINAPI WriteConsoleA(
; RCX => _In_ HANDLE hConsoleOutput,
; RDX => _In_ const VOID *lpBuffer,
; R8 => _In_ DWORD nNumberOfCharsToWrite,
; R9 => _Out_opt_ LPDWORD lpNumberOfCharsWritten,
mov rcx, stdHandle
lea rdx, text_msg
mov r8, sizeof text_msg
mov r9, dummy
call WriteConsoleA
...
```

To display a dialog box, use a pointer to the main application window. The parameters passed in this order are: a handle, two pointers, to texts (they must end with a zero, RDX is a pointer to the content that the dialog box will display, and R8 is a pointer to the window header that will be given to the window) that are to appear in header and dialog body and finally, a binary mask specifying which buttons and icons to use in the dialog box [4)].

```
...
; WINUSERAPI int WINAPI MessageBoxA(
; RCX => _In_opt_ HWND hWnd,
; RDX => _In_opt_ LPCSTR lpText,
; R8 => _In_opt_ LPCSTR lpCaption,
; R9 => _In_ UINT uType);
mov rcx, handler
mov rdx, offset hello_msg
mov r8, offset info_msg
mov r9, 0 ; MB_OK
and rsp, not 8
call MessageBoxA
```

```
...
```



Note - in the above case, the stack was explicitly aligned to a value divisible by 16 (`and rsp, not 8`). In most cases, this is not necessary, because the data organization is usually 64-bit. However, if it turns out that a single byte or word has been pushed somewhere on the stack, this will change and the program may throw an exception. Therefore, in such a case, you should take care to align the stack pointer so that it is divisible by 16.

1) https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160
2) https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemtime
3) https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-systemtime
4) https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa

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=1731963358

Last update: **2024/11/18 21:55**