1. Programming in Assembler

Laboratory manual

Exercise 9

Integrating .NET and x64 native assembler code in one solution

using Visual Studio

Exercise goal:

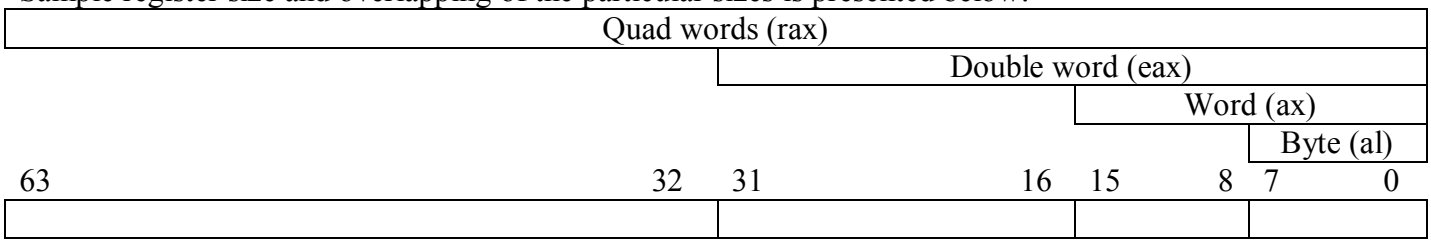
Students get familiarized with x64 programming and integration with .NET framework - by building sample Windows Presentation Foundation application and handling calculations in x64 assembler (native code) to speed up processing and calculations. Students also get familiarized with SSE and AVX extensions providing SIMD computation model.

1. General purpose x64 registers

The x64 capable processors (both AMD and Intel) provided extension to the 8 existing registers from 32 to 64 bits as well as adds new, 8 64bit registers unknown for x86. The general purpose registers (as on March 2015) for 64bit operations are juxtaposed in the table below:

64 bit	32 bit	16 bit	8 bit
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Sample register size and overlapping of the particular sizes is presented below:



One should note that whenever writing a 32 bit value into the 64 bit registers, the more significant part is automatically zero-extended but 16 and 8 bit values are NOT zero-extended automatically (this behavior is compatible with x86 default behavior).

2. SIMD registers

By the side of the general purpose registers, SIMD registers were extended along with x64 introduction - existing 64bit MMX registers were supplied and overlapped with 128 bit SSE and the 256 bit AVX registers (AVX registers are subject of future 512 bit and 1024 bit length extensions in forthcoming processor families as on Q1 2015). The exact set of the registers varies depending on the processor family, moreover XMM registers are accessible in x86 mode, however only **xmm0** through **xmm7** (first 8 of the xmm registers). The remaining 8 xmm registers (**xmm8** through **xmm15** were introduced in first generation of the

64bit processors). The AVX 256bit registers noted as ymm0 through ymm15 overlap xmm registers where i.e. **xmm0** register is simply a lower significant 128bits of the **ymm0** register. Similar way, when processor is working in 32 bit mode, only first 8 **ymm0** through **ymm7** registers are available for operations:

	AVX	SSE		
Bits	255 128	127 0		
	ymm0	xmm0	64 bit (x64)	32 bit (x86)
	ymm1	xmm1		
	ymm2	xmm2		
	ymm3	xmm3		
	ymm4	xmm4		
	ymm5	xmm5		
	ymm6	xmm6		
	ymm7	xmm7		
	ymm8	xmm8		
	ymm9	xmm9		
	ymm10	xmm10		
	ymm11	xmm11		
	ymm12	xmm12		
	ymm13	xmm13		
	ymm14	xmm14		
	ymm15	xmm15		

Future extension to the AVX registers on its length is also expected to increase the total number of registers to 32.

3. x64 calling convention

Calling convention is unified a method of passing arguments to/from the procedure. It applies both to the .NET-to-native code as well as to the native (pure assembler code) calls.
Unlike the x86, fortunately there is only one x64 calling convention, sometime referenced as *fastcall*, as uses increased amount of 64-bit registers. The stack is used when the amount of the arguments is above of the scope of the via-registry passing. The details are presented in the following chapters. The caller (the party that invokes the procedure/function) passes up to 4 arguments via registry, but also reserves space on the stack for arguments passed in registers. Any additional arguments are passed on the stack only.
There are three kind of arguments to be considered: integers, floats and pointers.

3.1. Passing of integer values and pointers (references)

Parameters up to 4 are passed via 64bit registers (in any case one should operate using 64bit values, when using 32, 16 or 8 bit arguments the need to be extended to 64bits or passed within the structure using a pointer).
The order is left to right, as follows: first function parameter->**rcx**, second->**rdx**, third->**r8**, fourth->**r9**.
The pointer is passed as 64bit value (an address) and it is callee's duty to handle it appropriately.

Sample:

C#/.NET prototype:

```
sampleAdd(int a, int b, int c, int d)
```

(*a* is passed in **rcx**, *b* in **rdx**, *c* in **r8**, *d* in **r9**)

Assembler implementation:

```
sampleAdd proc
     mov rax, rcx ; rax<-a
     add rax, rdx ; rax+=b
     add rax, r8  ; rax+=c
     add rax, r9  ; rax+=d
     ret
sampleAdd endp
```

3.2. Passing of the floating point values

Parameters up to 4 are passed via first 4 SSE registers: **xmm0<->xmm3** (considering from left to right, first parameter->**xmm0**, second->**xmm1**, third->**xmm2**, fourth->**xmm3**. The register size is 128bits so floating point values up to 128bit length can be passed (i.e. .NET *float* that is 32bits long or .NET *double* that is 64 bit long).
Sample:

C#/.NET prototype:

```
sampleSub(double a, float b)
```

(*a* is passed in **xmm0**/**ymm0**, *b* is passed in **xmm1**/**ymm1**)

Assembler implementation:

```
sampleSub proc
     vsubpd ymm0, ymm0, ymm1 ; mind that ymm registers overlay xmms
     ret
sampleSub endp
```

3.3. Mixed types

When mixing integer and floating point arguments, the absolute argument position denotes the register used for passing the arguments i.e.:
C#/.NET prototype:

```
sampleSub(double a, int b, float c, int d)
```

(*a* is passed in **xmm0**/**ymm0**, *b* is passed in **rdx**, *c* in **xmm2**/**ymm2**, *d* in **r9**)

3.4. Stack allocation – passing more than 4 values to the function

More than 4 values are passed via stack. The 5th argument can be accessed as [**rsp**+28h], the following arguments are addressed linearly, every 8 bytes (64 bits). The first fife arguments represent (in the order:

- caller return address [**rsp**+0],
- argument 1 [**rsp**+8h],
- argument 2 [**rsp**+10h],
- argument 3 [**rsp**+18h],
- argument 4 [**rsp**+20h].

The argument 1 through 4 are those passed via registers and calling convention requires to reserve this space even if those arguments are not physically loaded into the stack. This way 5th argument is always present at [**rsp**+28h] even if arguments 1 through 4 are passed via registers only. This approach is

compatible with mixed types passing as presented in the previous chapters – the stack may contain mixed types because both integers and floating point values are passed as 64bit, so do pointers.

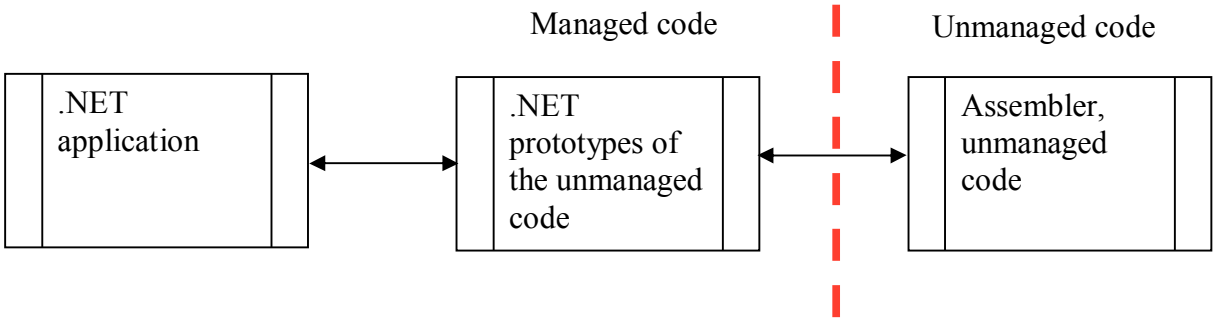
3.5. Return values

The callee returns the values in the appropriate register with respect to the value type:

- integer or pointer value is returned in **rax**,
- floating point value is returned in **xmm0** (a lower 128bits of the **ymm0** AVX extensions thus values can be also referenced as **ymm0**),
- multiple values are returned as pointer to the structure. The pointer is returned in **rax** register.

4. Integrating .NET and assembler native code:

Cooperation between managed and native code is possible with means of wrappers. Fortunately most of the job is done through .NET compiler but unfortunately, creating one solution with managed and unmanaged code is pretty tricky. This laboratory scenario assumes the user interface is created using .NET framework (C# language) while computation is done by the native (assembler) code. The .NET party is not limited to the WPF model as presented here – actually any .NET but also C++ native and managed code is capable to call pure, assembler, native code. For the simplicity, this lab uses WPF (Windows Presentation Foundation) direct model (in opposite to the MVVM model) but the .NET application could be anything from command line app to the web application. In any case, the assembler – native code is organized as a DLL library (originally as C++ dll model, then ported to the assembler code) with code organized as a set of computation functions, embedded into the main, .NET application. It is essential to note that native code is not managed by the .NET framework memory management (particularly Garbage Collector that may relocate variables and classes through stack and heap) so to persist the pointers between managed and unmanaged code it is essential to mark managed code prototypes as **unsafe** and .NET structures as **fixed**, to disable data relocation due to the .NET Garbage Collector tasks.



The following sections present hands-on lab on creating .NET framework WPF simple dialog then creating a managed code DLL implemented in assembler. If you already own the solution, you may omit the .NET implementation part and focus on the assembler functions.

Warning. The following hands-on lab assumes you're using Debug mode only. To generate Release it is necessary to configure separately some of the settings similar way it is presented for the Debug mode in the following chapter. It is not done by the compiler automatically, however!

4.1. Tasks to perform during labs

The WPF part assumes there is a dialog box with set of controls providing one (or many) functionalities:

- Adding at least 2 integer values given by values within text boxes.
- Adding at least 2 floating point values (mixed - float, double) given by values within text boxes.
- Calculating sum of an array of integers given by the table (int array).

- Computing the weighted average of the four products given by the double and integer each, using SIMD and mixed mode.
- Performing some operation on the image (byte array).

The underlined scenario's implementation is presented in the following chapters.

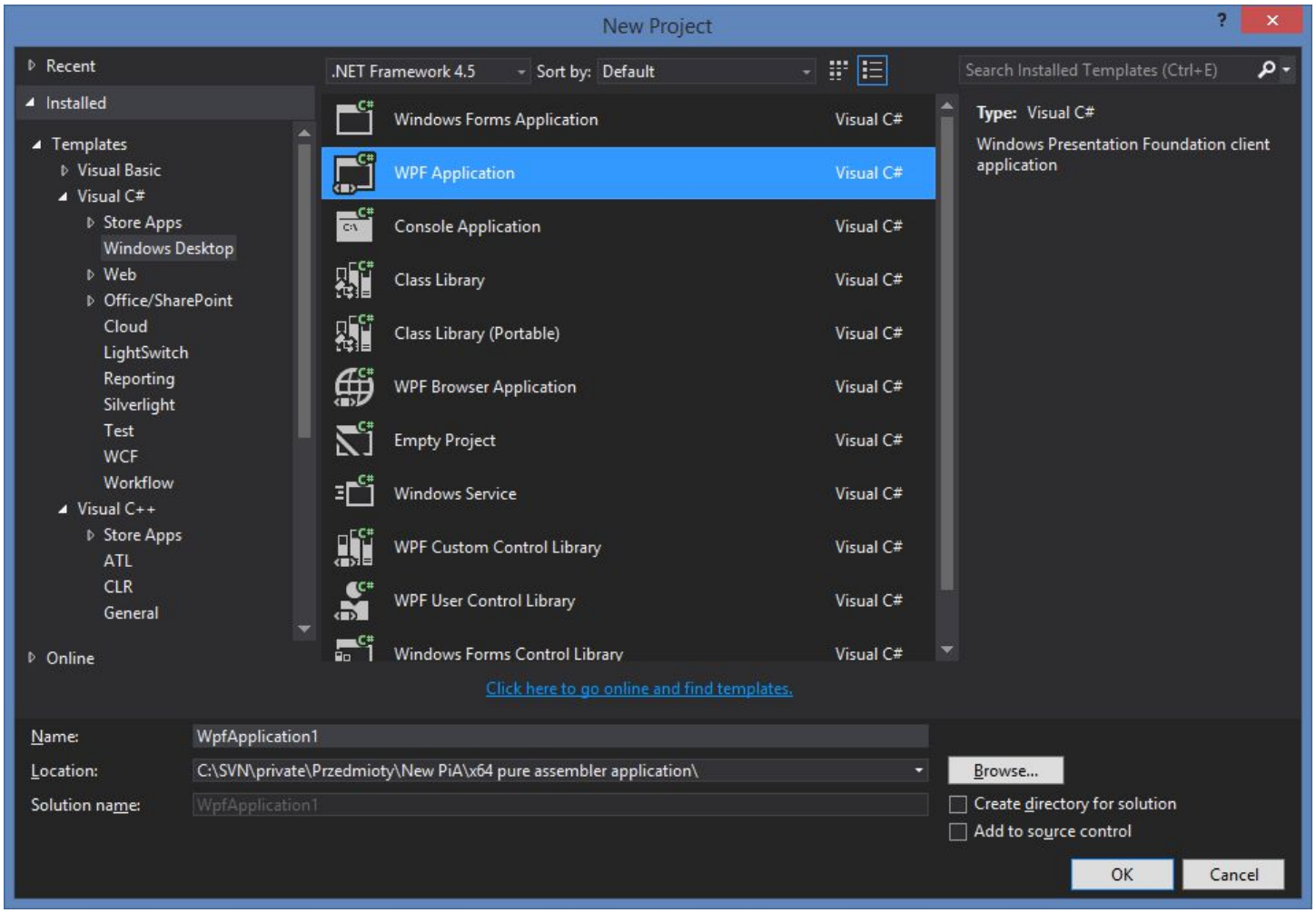
4.2. Solution

The .NET solution is composed of the two projects:

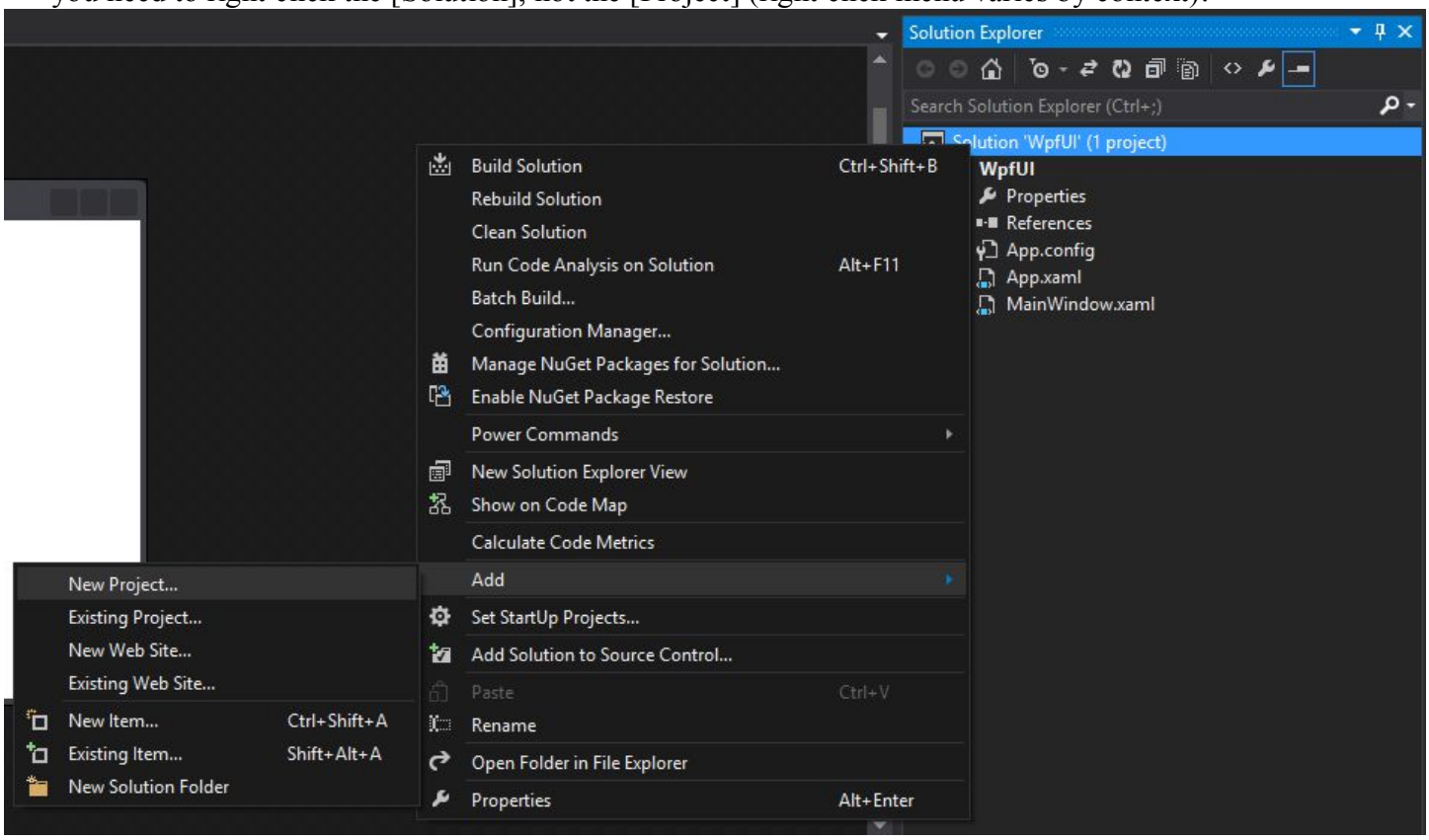
- a WPF dialog box (UI),
- a C++ DLL library project, converted to handle assembler code.

4.2.1. Create new projects and constitute solution

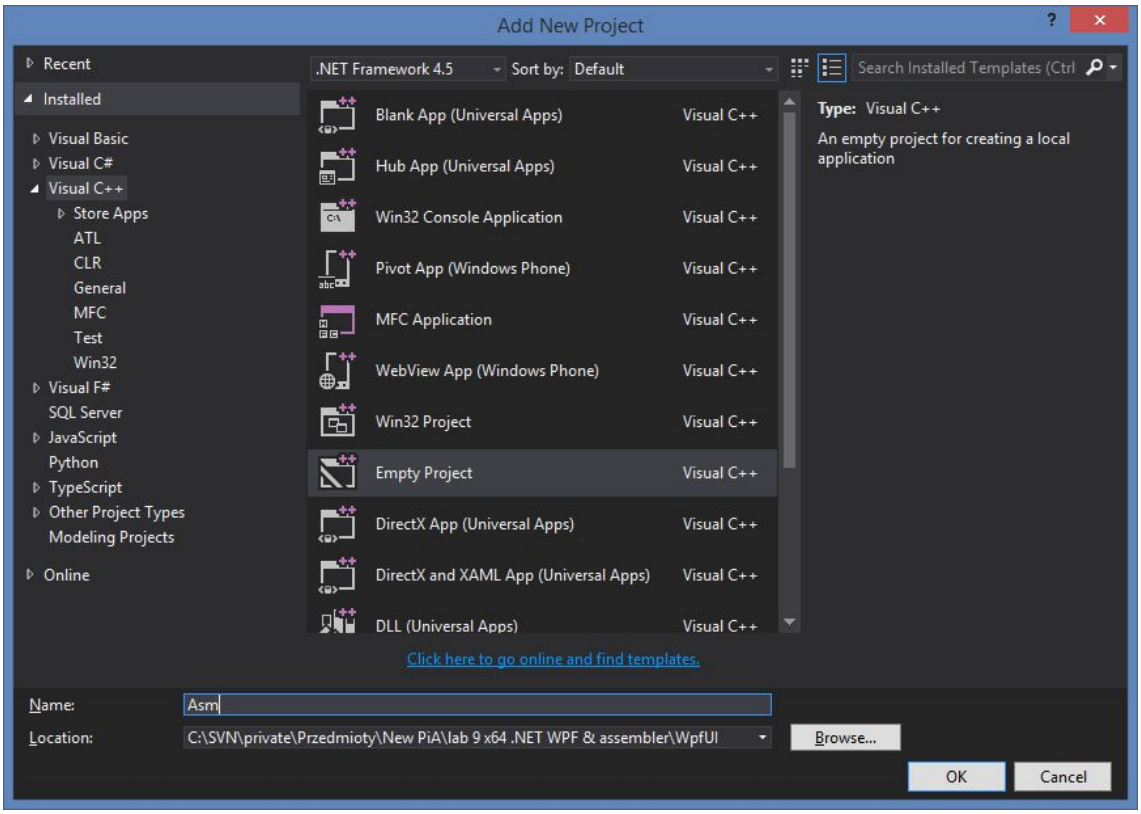
Start Visual Studio then create new project [Menu: File->New->Project]. Choose: [Installed->Visual C#->WPF Application]. This creates WPF application (a user interface). Give it some reasonable name.



Once created, add another project that will hold the assembler code inside. This is C++ DLL project. To do so, right click the [Solution] in the Solution Explorer window, then choose: [Add->New Project]. Mind that you need to right click the [Solution], not the [Project] (right click menu varies by context):

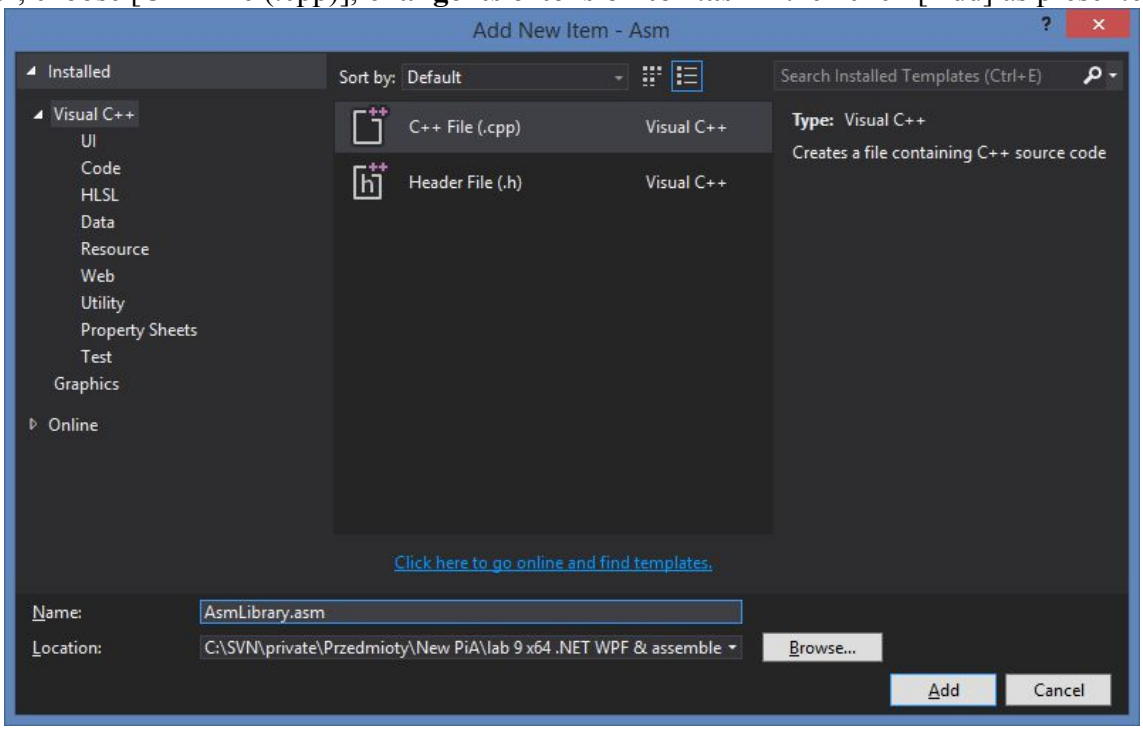


then choose [Empty Project Visual C++], give it some name and accept:



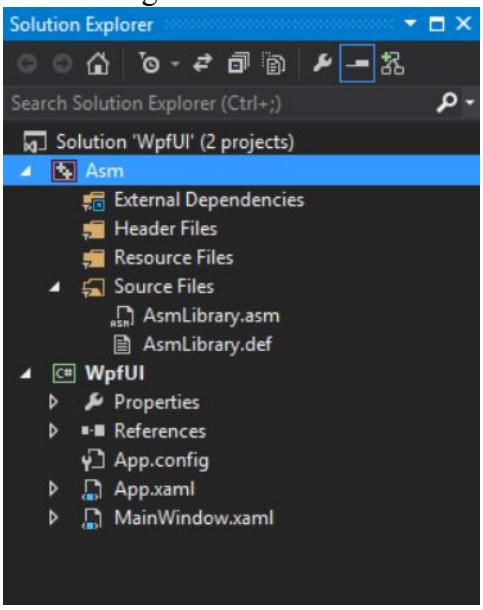
4.2.2. Add source files

The WPF User Interface has created some empty dialog box (will return to this later) but assembler project is empty. To create an empty source file in assembler project, right click the [Project] in the Solution Explorer, choose [C++ File (.cpp)], **change its extension to ".asm"** then click [Add] as presented below:



Repeat the step above to add the exporting definition file (.def) for the linker. Mind that the file needs to have the same name as your .asm file and the extension should be .def.

Your solution should look similar to the image below:



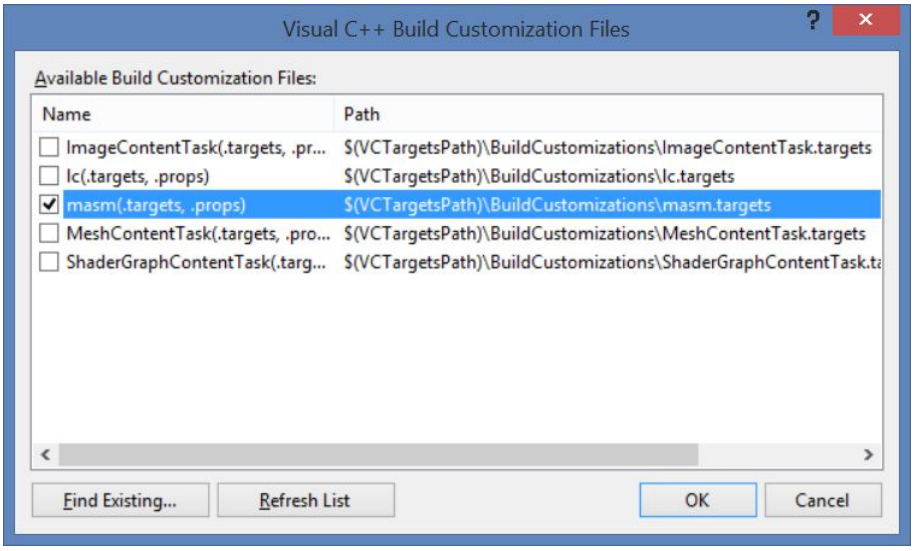
In the example above, the compiler will create Asm.dll (the dll name is the project name, not a source file name).

The assembler source file (.asm) contains assembler functions that will constitute the library, while .def file informs the compiler, which function are provided for the rest of the code (here the caller is the WPF UI

app). Whenever there is new function added to the assembler code, the .def file should be updated along with appropriate function name.

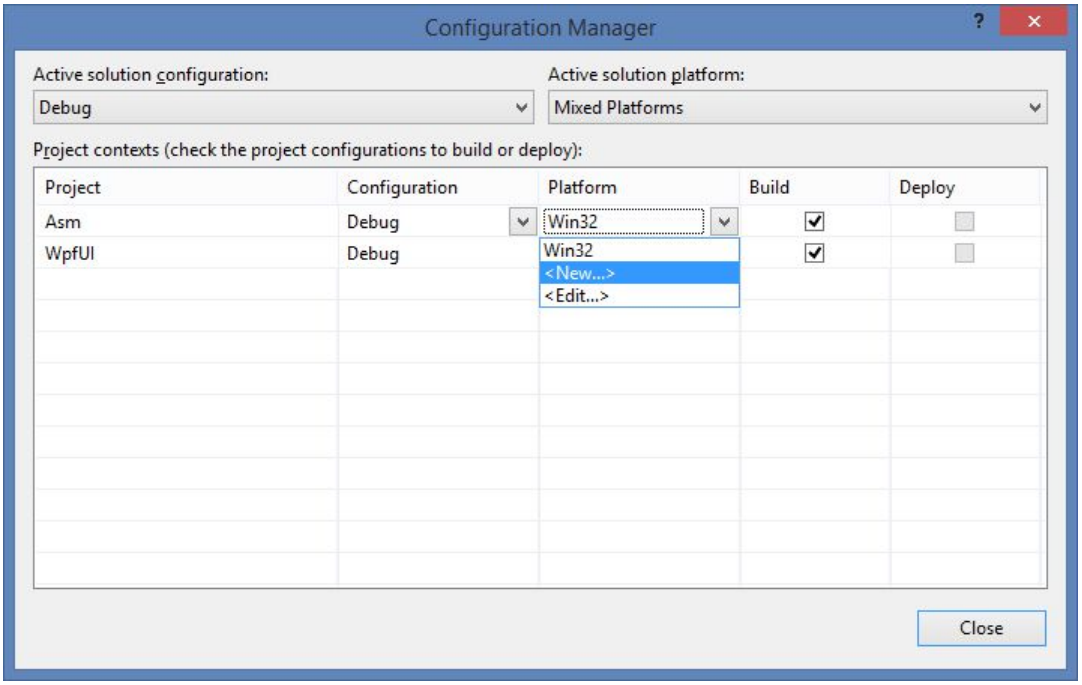
4.2.3. Constructing the building order dependencies and compiler configuration

The compiler uses somehow "unusual"[1] combination of the managed and native code, so you need to inform the compiler how to compile the assembler library when building the solution (and how to maintain the rebuild order, when necessary) It is done via building customization. To do so, right click the assembler library project and choose [Build Dependencies->Building Customization] then check: [masm(.targets, .props)] and accept clicking [OK] as presented below:

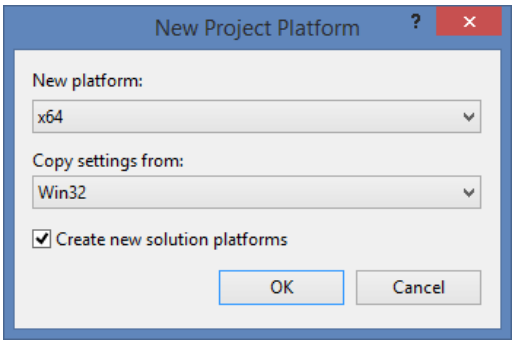


[1] The Visual Studio is capable to compile dozens of different targets and platforms. "Unusual" does not mean "unknown" – the native assembler coding is considered somehow uncommon now and for geeks/professionals than for regular programmers as it provides great performance to the results but is quite hard and not so user friendly as high level languages. One need to "enable" this feature of the compiler before use.

Current default configuration for the C++ project (assembler project) states it is Win32 (x86, 32-bit application). To switch from x86 to x64 code it is necessary to construct new solution platform – an Intel x64 code. To do it, right click [Solution] then choose [Configuration Manager]. In the dialog box select assembler project then expand the combo in the "Platform" column and select <New…>:

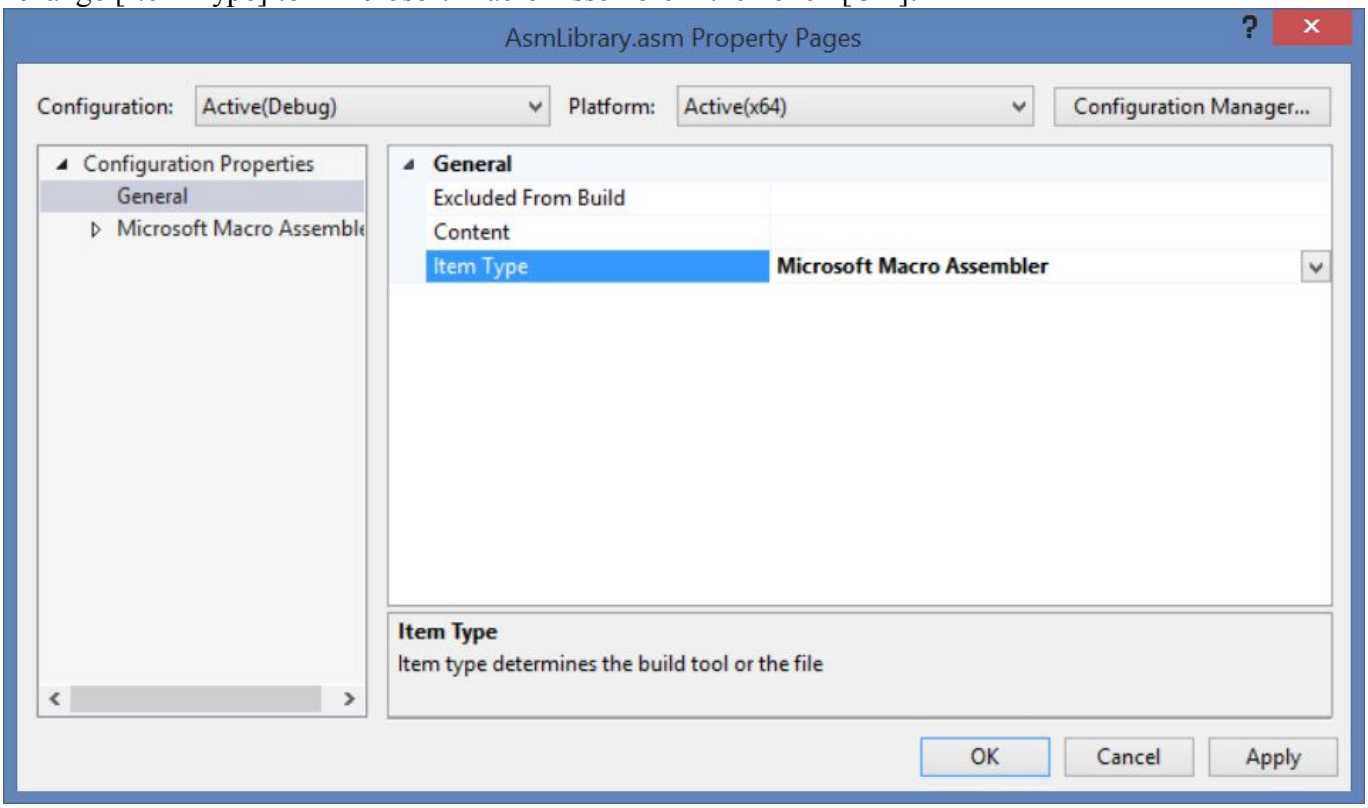


Once the "New Project Platform" appears, set "x64" in "New platform:" and set "Copy settings from:" to "Win32". This will preserve all setting done to the project till now:



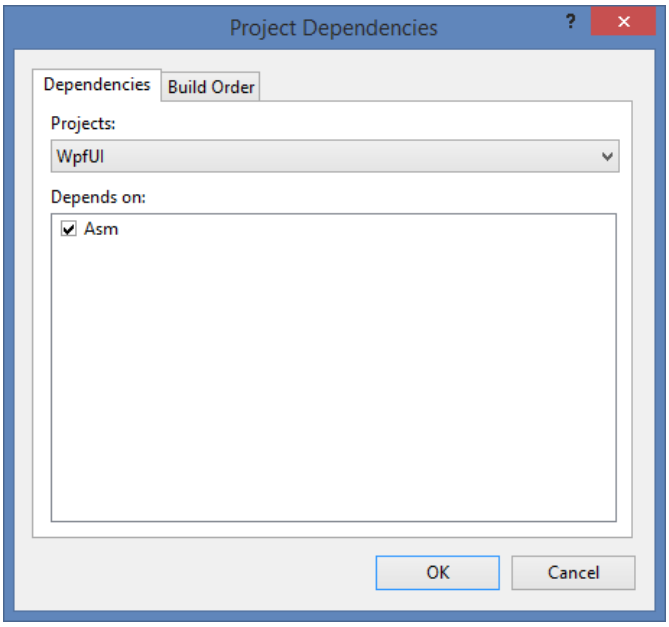
The active configuration should change to x64 (previously Win32) now.

Now one need to inform the compiler, that source assembler code (.asm file) should be handled by the Masm (assembler) compiler. To do so select the assembler source file in the solution explorer then right click the file (.asm) and then choose [Properties]. Within the [Configuration Properties->General] section change [Item Type] to “Microsoft Macro Assembler” then click [OK]:



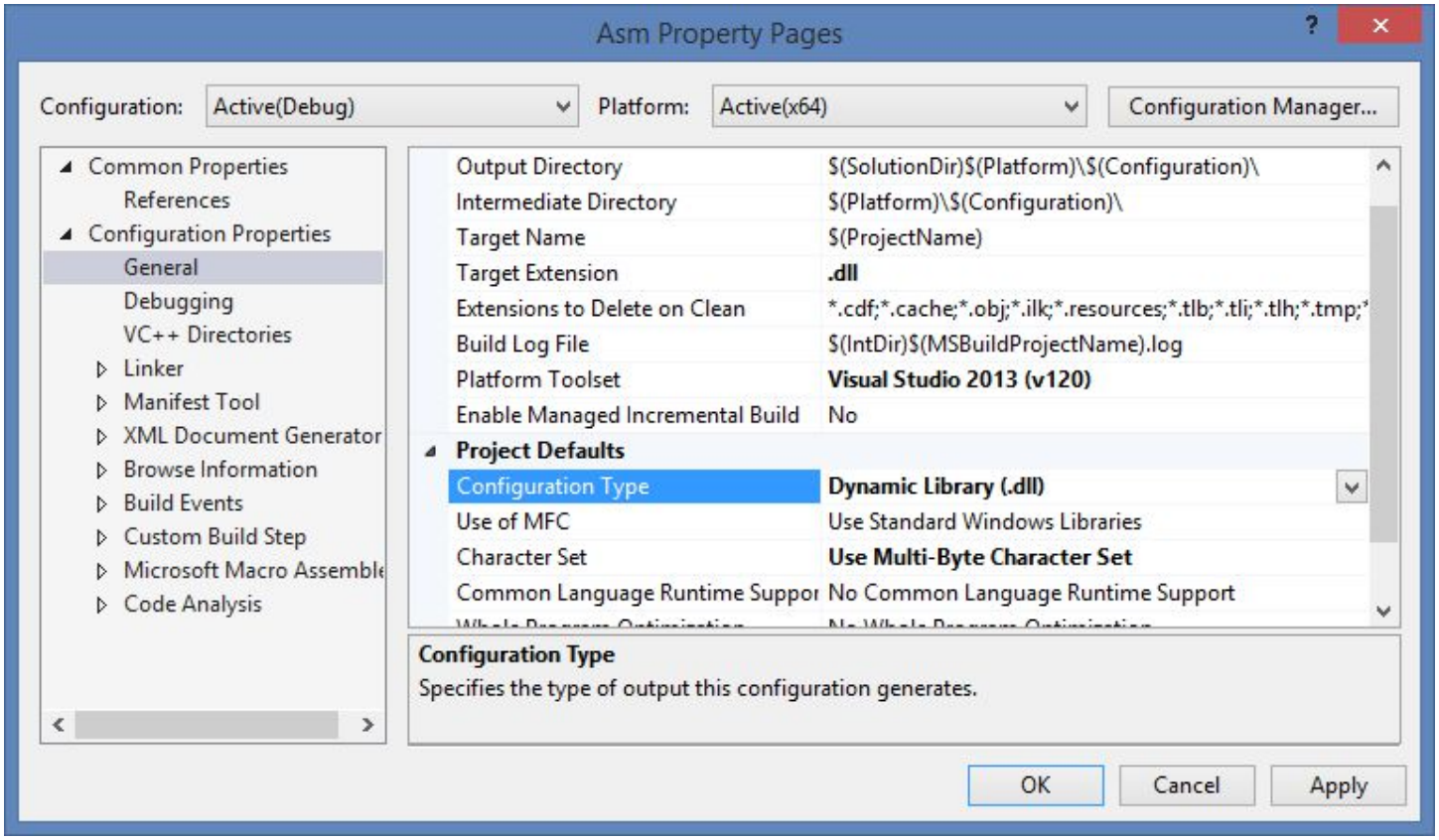
Following step is to inform the compiler on the building order and the building target. The model assumes dynamic DLL loading (during runtime) so the binary, compiled DLL library has to be located in the same directory as your WPF UI application.

To do so, right-click the [Solution] in the Solution Explorer window then choose [Project Dependencies], choose your WPF UI project and mark that its building is depending on the assembler library (this way your assembler library is compiled first, before your WPF application):

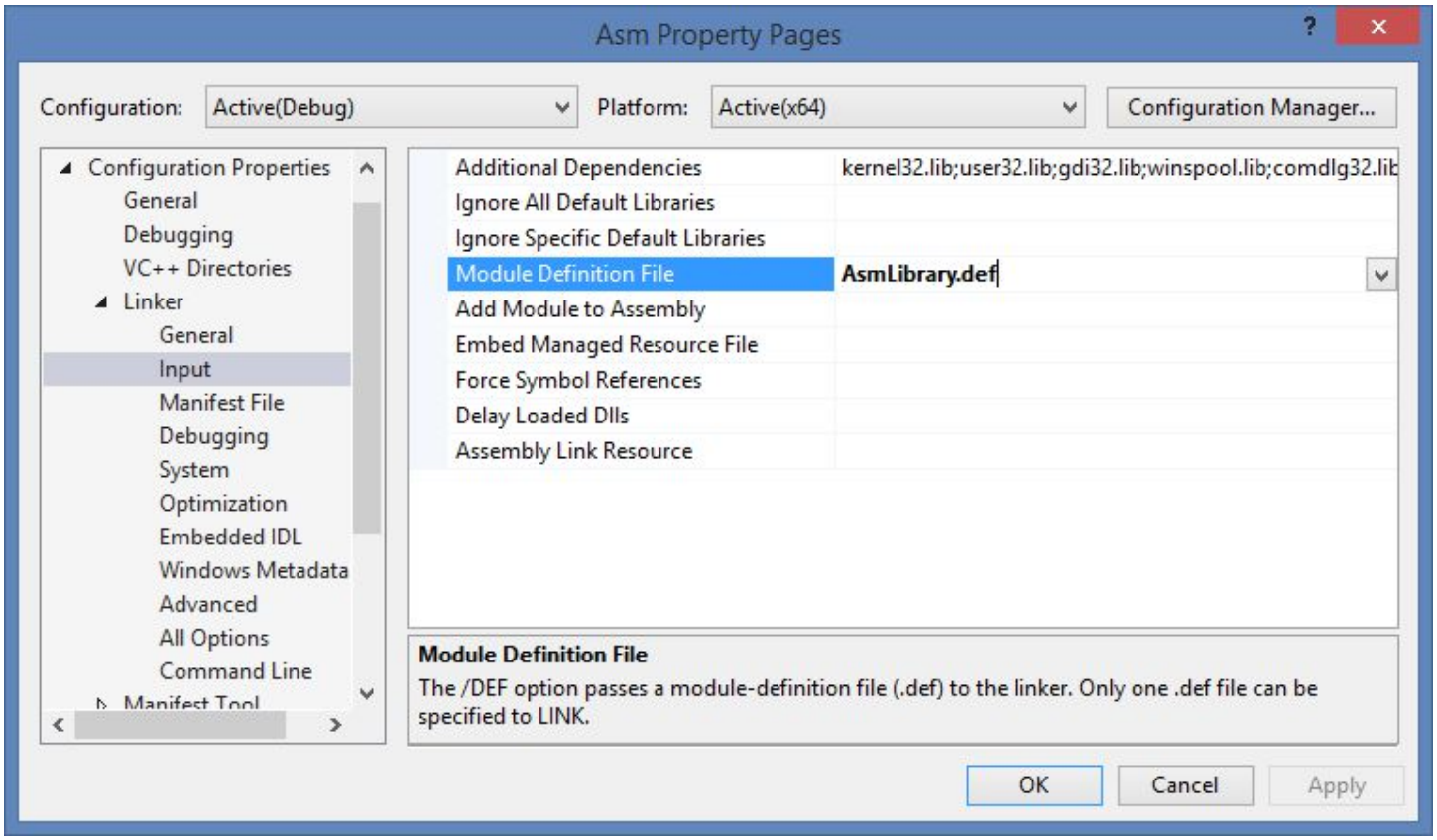


Now it is necessary to switch the project from the empty to let it create DLL file. Right click the assembler project, choose [Properties]. Then set the following set of configuration properties:

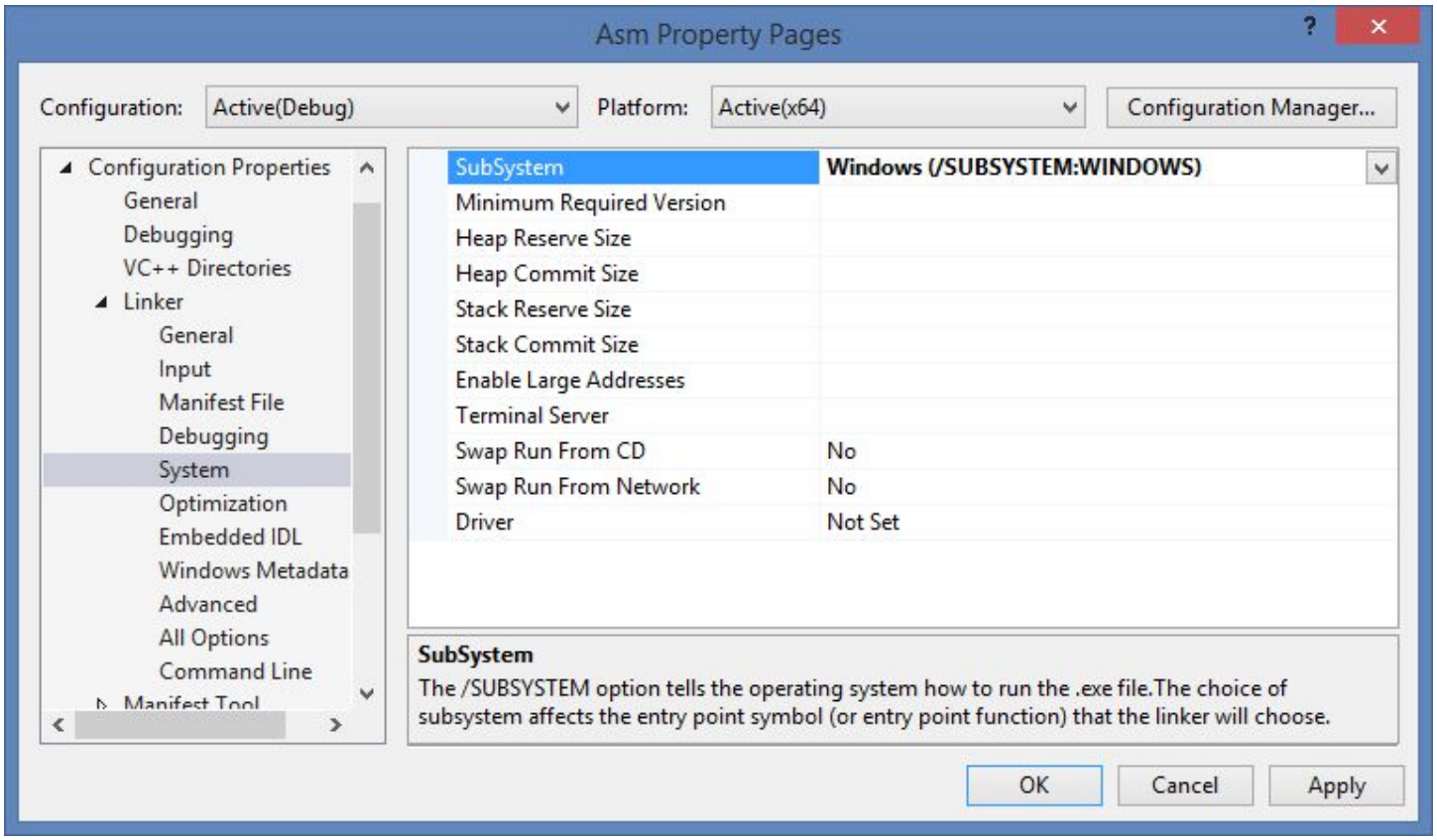
- in the [Configuration Properties->General] set:
  - "Target Extension" to .dll
  - "Configuration Type" to "Dynamic Library (.dll)"



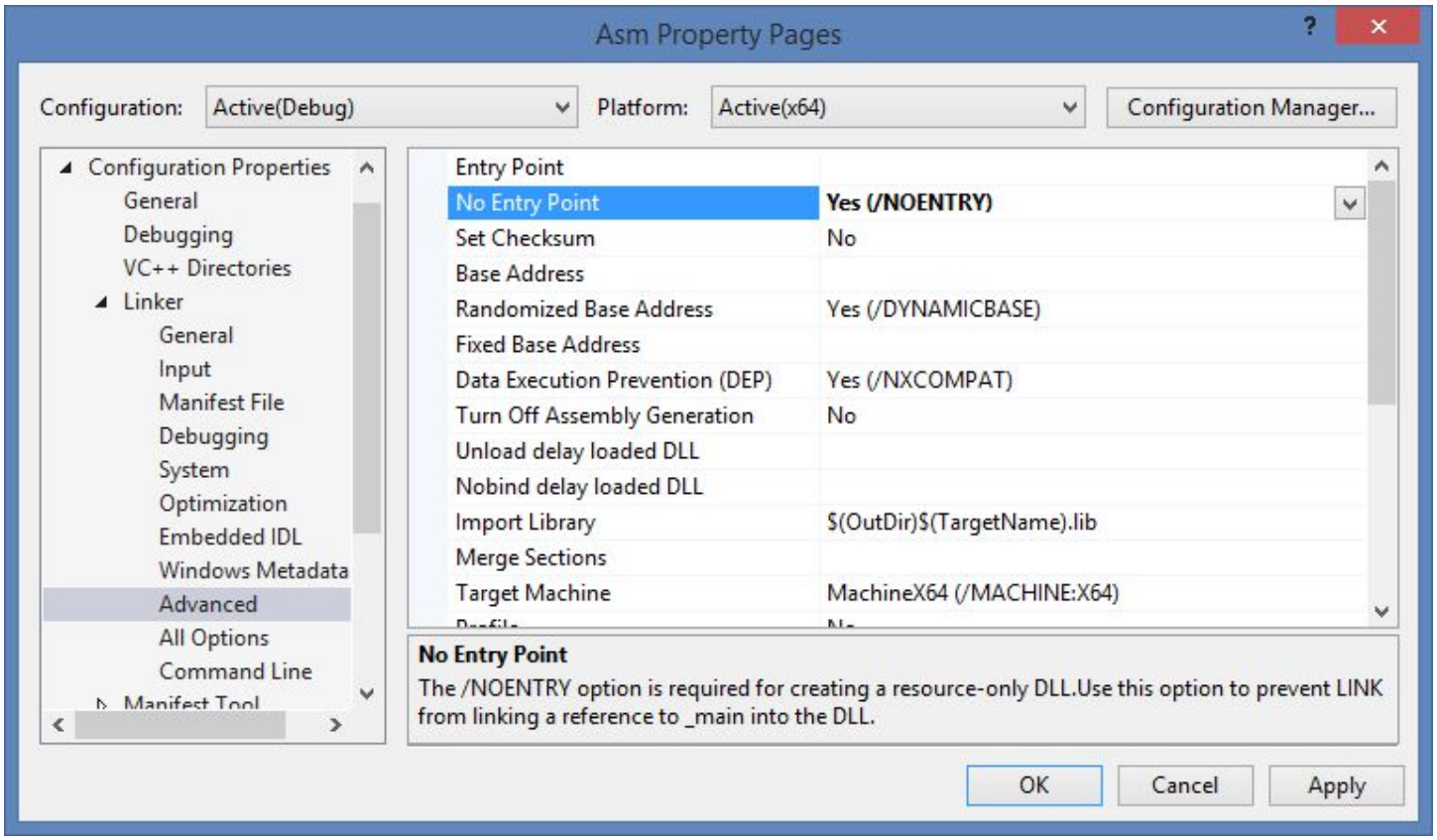
- in the [Configuration Properties->General->Linker->Input]:
  - provide your definition (.def) file in the “Module Definition File” property



- in the [Configuration Properties->General->Linker->System] set:
  - "Subsystem" to "Windows (/SUBSYSTEM:WINDOWS)"



- in the [Configuration Properties->General->Linker->Advanced] set:
  - "No Entry Point" to "Yes(/NOENTRY)"

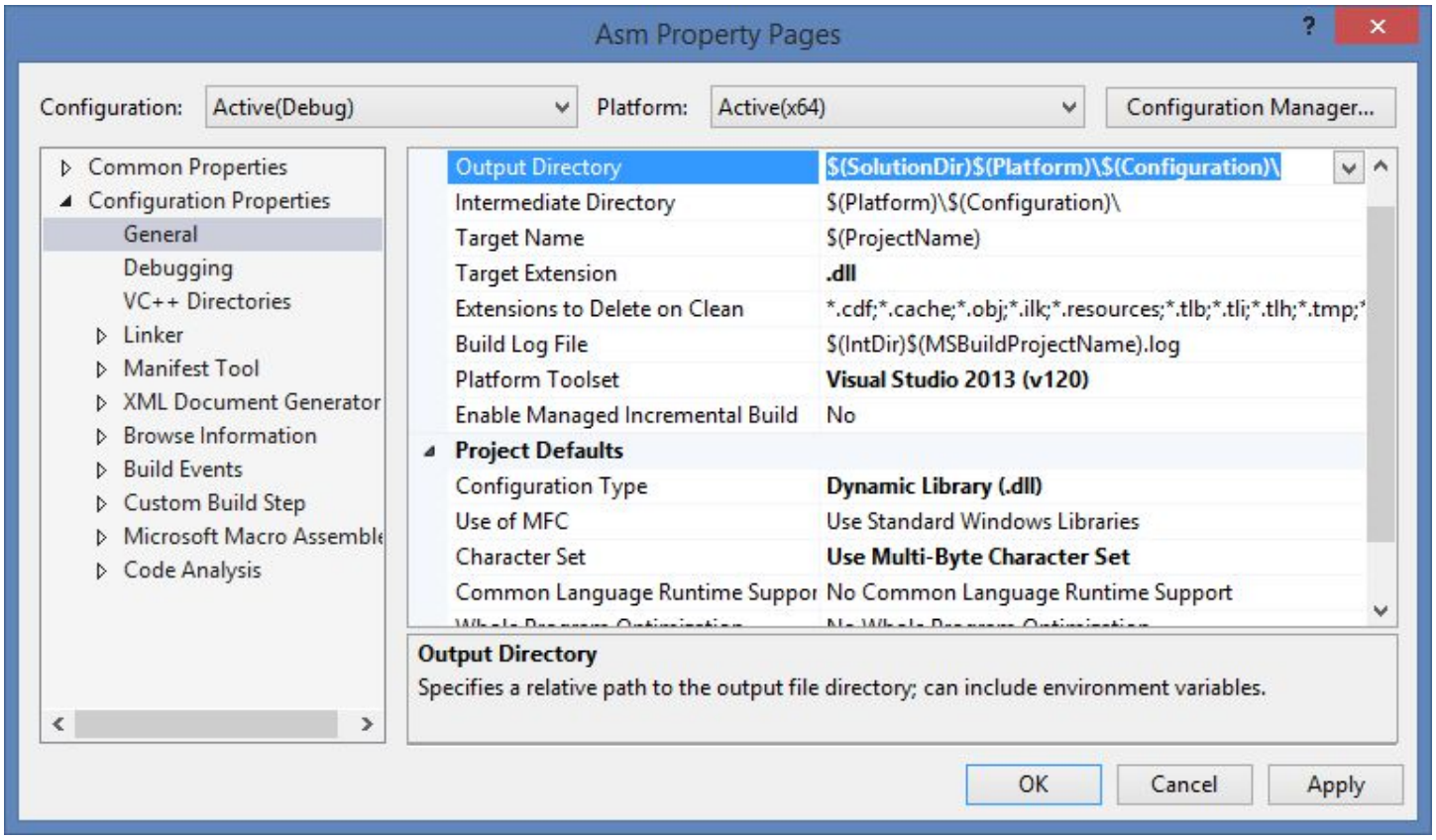


and then click [OK].

4.3. Common output directory

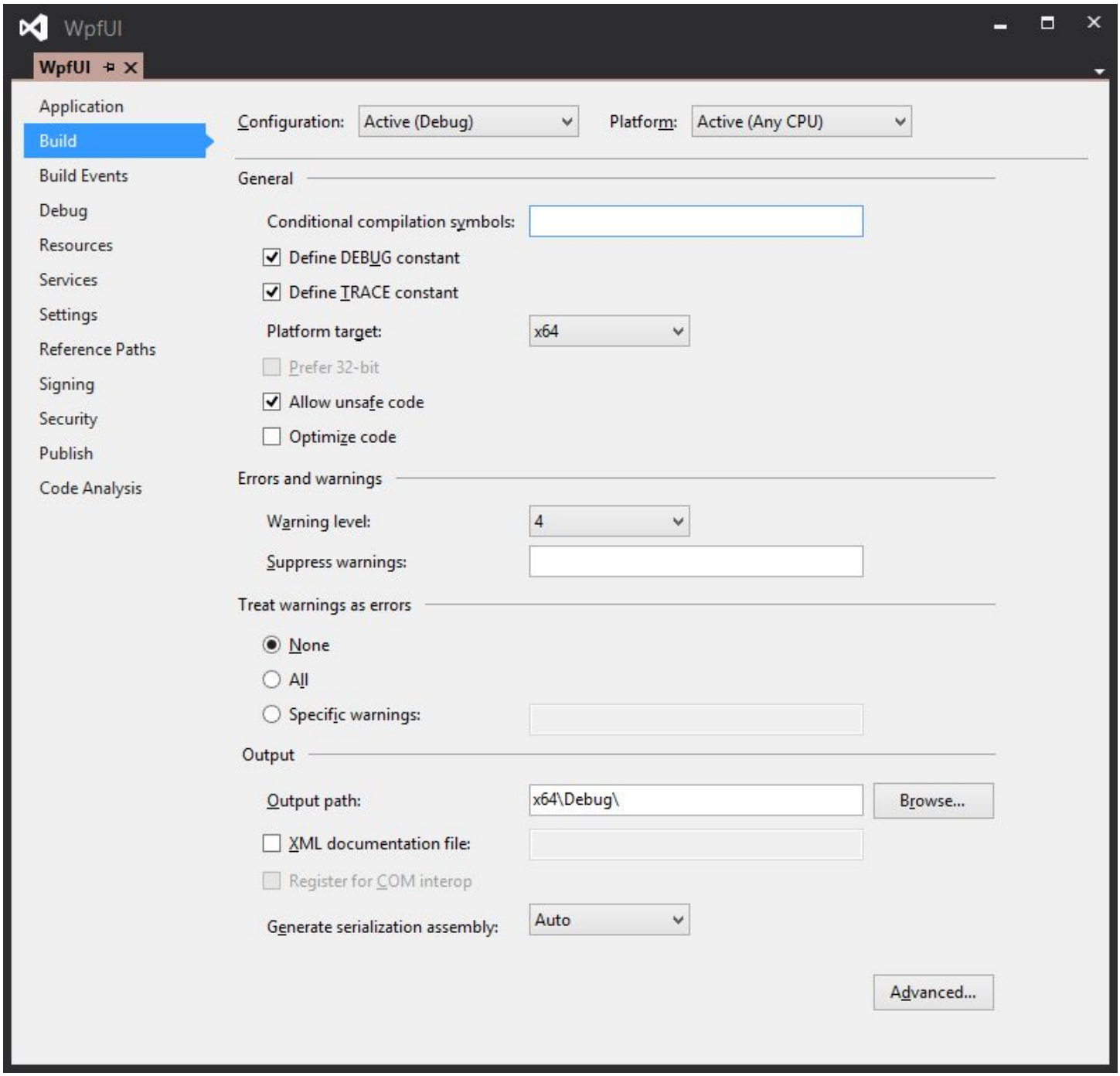
The solution is composed of two projects each of them is compiled independently. When compiling the DLL, the target (resulting, compiled) file should be injected into the user interface project to let the WPF UI be able to load it dynamically (it expects it in the current directory regarding the .exe location). VS does provide various macros with the appropriate directory names, unfortunately they are limited to the single project and the entire solution. The only reasonable idea is to create separate target folder on the solution level that can be referenced by both projects and thus integrate two outputs. Unfortunately the output directory is set different way in the different projects.

To set the output project for the assembler party, right click the assembler project in the Solution Explorer window then select [Properties]. Browse to [Configuration Properties->General] and verify if "Output Directory" contains: $(SolutionDir)$(Platform)\$(Configuration)\ . Correct if necessary.

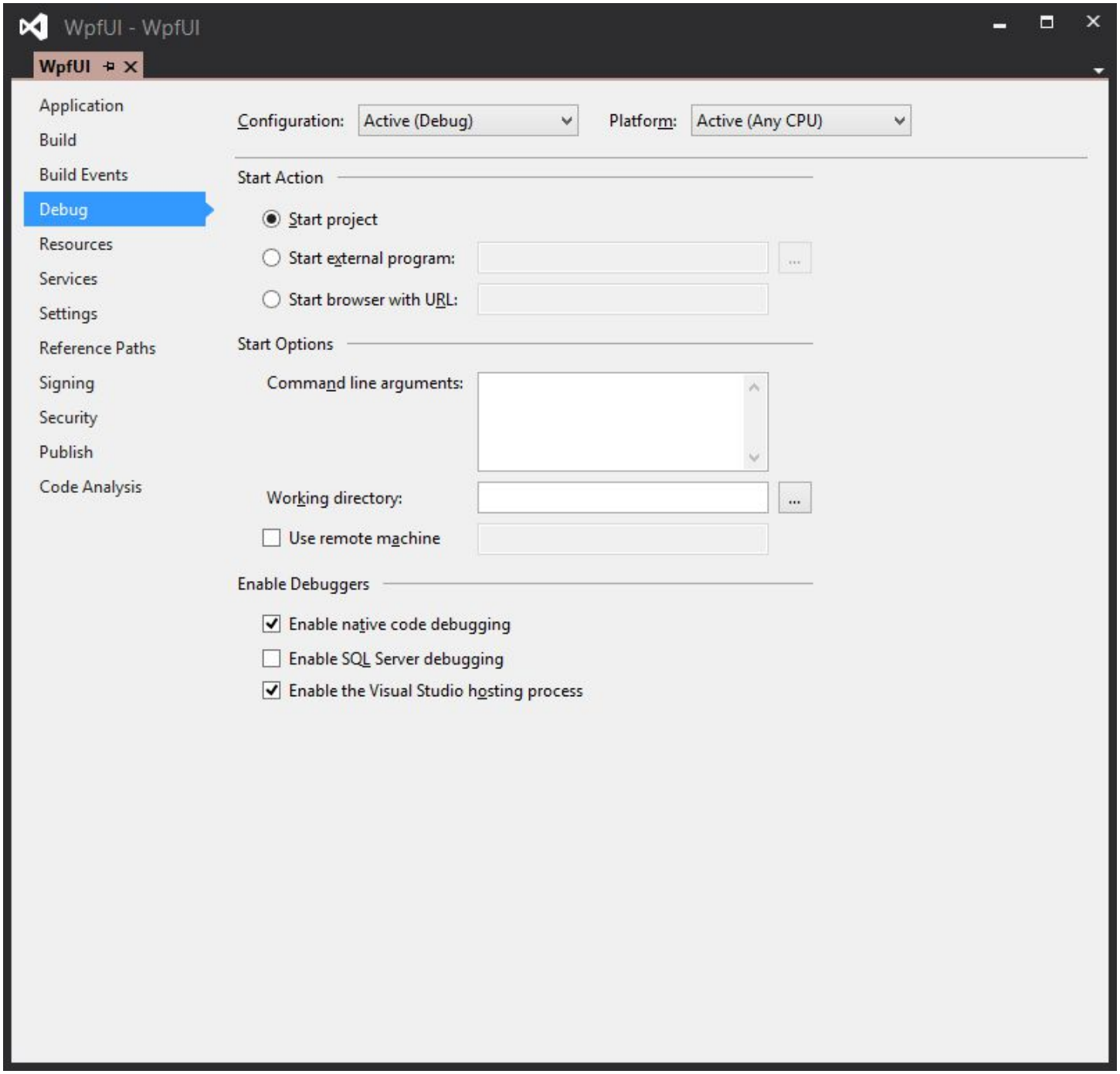


While compiling, the above setting will compile the assembler DLL in to the following directory: <solution>\x64\Debug (or <solution>\x64\Release, regarding your current build mode).

Now it is time to let WPF UI application compile into the same directory as assembler project does. This is done for Debug mode and for Release mode independently. Right click the WPF UI project in the Solution Explorer then select [Properties]. Choose “Build” on the left then switch “Configuration:” to “Active (Debug)” or “Debug” when necessary, choose “Platform target:” to “x64”, provide “Output path:” equal “x64\Debug”.
Remember to check “Allow unsafe code” (otherwise unsafe clause will cause compile errors).
The configuration for the “Debug” is presented below:



By default, the debugging is limited to the managed code only. To enable debugging of both kinds of code (even in one debug session!) it is necessary to right click WPF UI project then select [Properties], navigate to the Debug tab and check “Enable native code debugging”:



4.4. Compile the project

Empty assembler project won't compile successfully. It is necessary to provide at least minimum assembler code into the assembler (.asm) file:

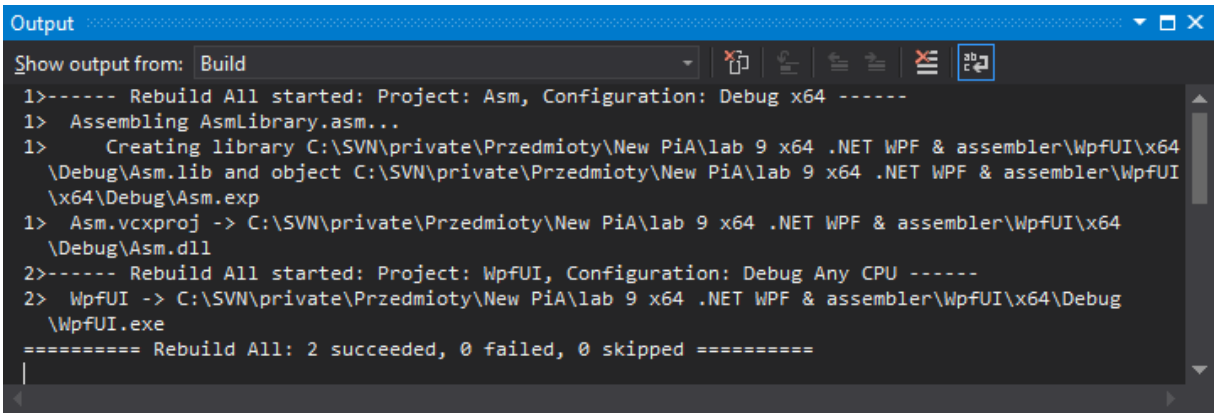
```
.data
.code

end
```

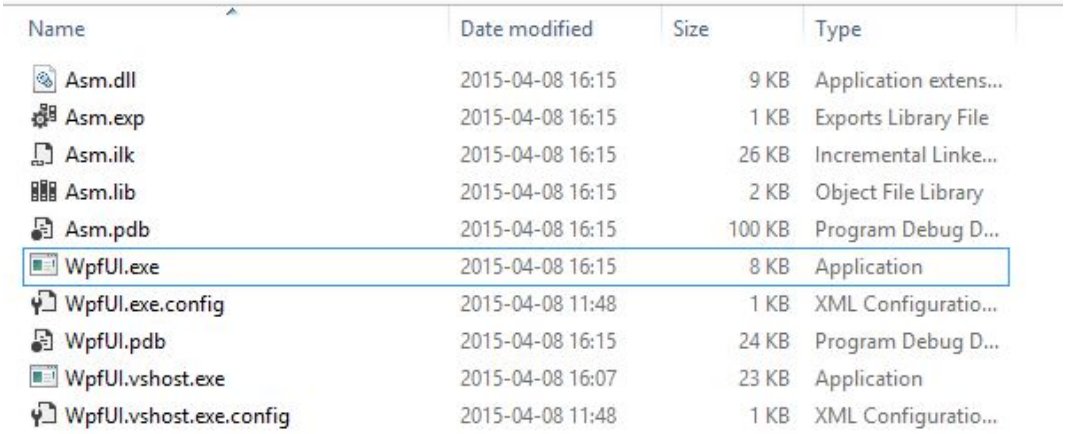
and minimum information to the definition (.def) file:

```
LIBRARY "Asm"
EXPORTS
```

The project should create the appropriate output directory and generate both assembler DLL and WPF UI executable in the single directory. The Output window shall contain the information on successfully compiled two projects:

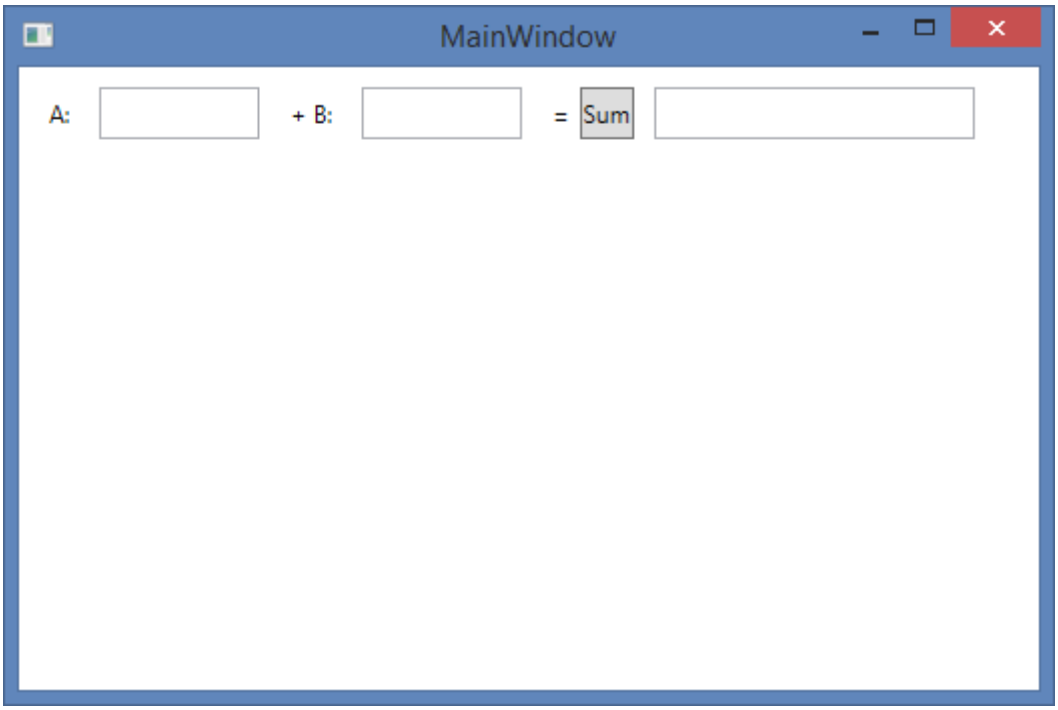


The target (build) folder should contain:



5. Implementation

Implementing simple technology sampler that performs adding operation on two floats require two input boxes for input, one for output and a button triggering operation:



The XAML code of the UI may look this:

```
<Window x:Class="WpfUI.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <StackPanel Margin="10" HorizontalAlignment="Left" Orientation="Horizontal"
VerticalAlignment="Top">
            <Label>A:</Label>
            <TextBox Width="80" Name="txtA" Margin="10,0"/>
            <Label>+ B:</Label>
            <TextBox Width="80" Name="txtB" Margin="10,0"/>
            <Label>=</Label>
            <Button Content="Sum" Click="Button_Click"/>
            <TextBox Width="160" Name="Result" Margin="10,0"/>
        </StackPanel>
    </Grid>
</Window>
```

Observe the button on-click event (Button_Click) that will handle operation. This function appears automatically once you click the button within the UI designer window. The empty, Button_Click function is added to the "code behind" in C# language (the file with the name of the dialog windo and extension .xaml.cs:

```
namespace WpfUI
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void Button_Click(object sender, RoutedEventArgs e)
        {
        }
    }
}
```

To importing the DLL into the C# it is necessary to provide using directive on `System.Runtime.InteropServices` namespace and provide function prototype with arguments identical to the assembler implementation. Moreover it is essential to mark this code as unsafe to let the Garbage Collector do not move the pointers and do not modify the memory.
Sample implementation may look similar to the:

```
using System.Runtime.InteropServices;
```

```
    public class AsmProxy
    {
        [DllImport("Asm.dll")]
        private static extern double asmAddTwoDoubles(double a, double b);

        public double executeAsmAddTwoDoubles(double a, double b)
        {
            return asmAddTwoDoubles(a, b);
        }
    }
```

Once the code is ready the click handler can be implemented:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            AsmProxy asmP = new AsmProxy();
            double a, b, r;
            a = Double.Parse(txtA.Text);           //convert argument 1 from text (WPF UI) to double
            b = Double.Parse(txtB.Text);           //convert argument 2 from text (WPF UI) to double
            r = asmP.executeAsmAddTwoDoubles(a, b);//execute assembler code
            Result.Text = r.ToString();            //return value to the WPF UI as text
        }
```

Before final compiling, the assembler implementation requires providing the assembler code in the assembler source (.asm) file, i.e.:

```
.data
.code
asmAddTwoDoubles proc
	vaddpd ymm0, ymm0, ymm1
	ret
asmAddTwoDoubles endp
end
```

and corresponding export definition (.def) file:

```
LIBRARY "Asm"
EXPORTS

asmAddTwoDoubles
```

The function above (`asmAddTwoDoubles`) is a leaf function thus does not require stack modification on return (RSP).