A guide to wrapping a .NET Windows Framework Forms Control for PowerBuilder v1.0

Contents

Introduction

Currently the only way to use visual .NET controls in PowerBuilder is to build them as Active/X controls.

The is a guide on how to take a visual .NET Windows Framework Forms control and turn it into a control that can be placed on a window in a PowerBuilder application in the same way any Active/X control can.

Example

An example C# .NET Framework 4.8 solution is included, PBTextBox. It wraps the standard .NET TextBox so it can be used on a PowerBuilder window. It shows the mechanics required to wrap a .NET control, which when built will produce an Active/X control that PowerBuilder recognizes.

Steps

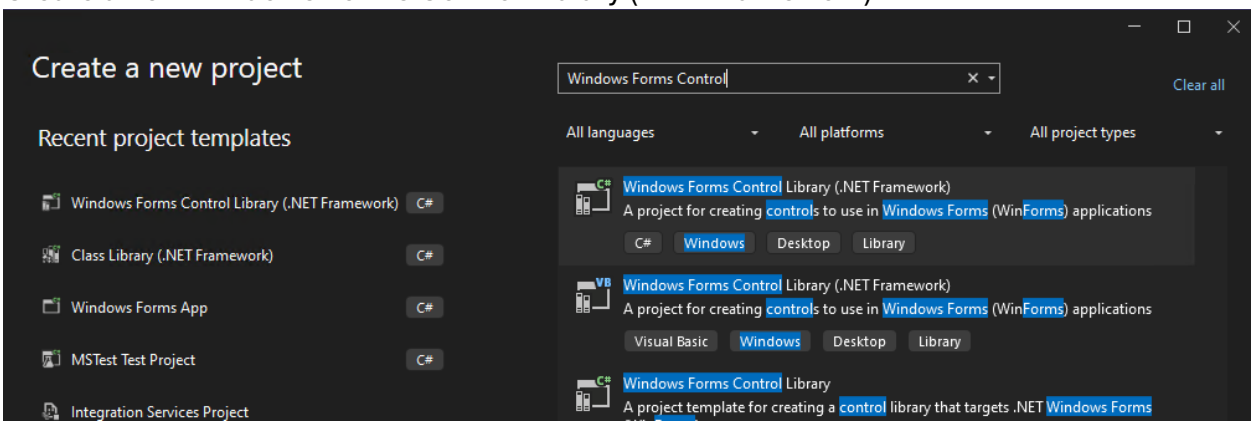
These are the steps requited:

- Create a new project
- Add your control
- Wrap the control
- Make it COM accessible
- Build the dll
- Register the dll
- Use it in PowerBuilder

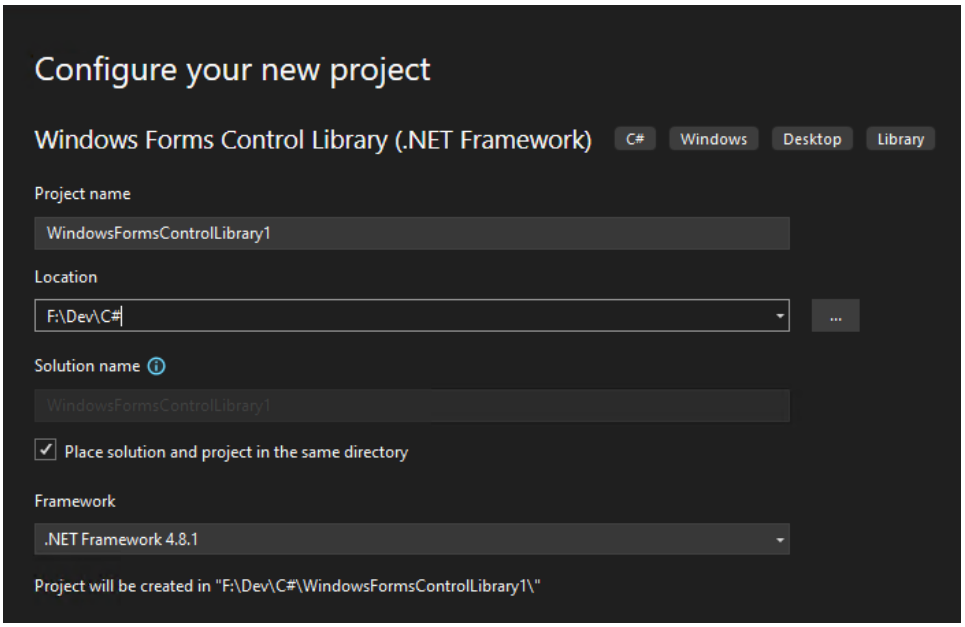
Getting started

Create a new project

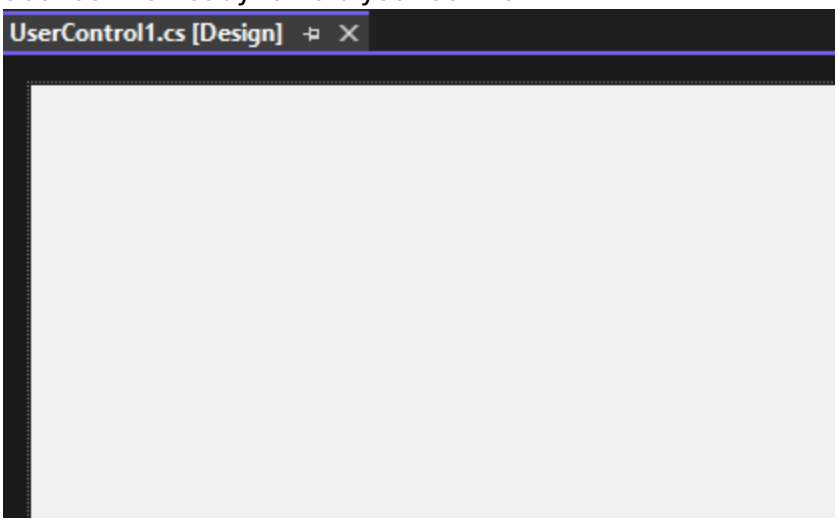
Create a new Windows Forms Control Library (.NET Framework)



Make sure it is using .NET Framework 4.8.x

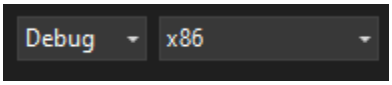


Once Visual Studio has finished creating the new project you'll be presented with an empty user control ready to hold your control.



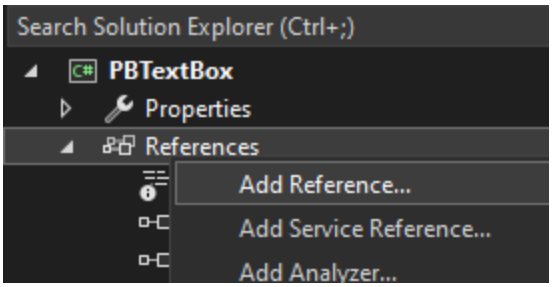
X86

Ensure the project is built as an x86 dll by setting the correct solution platform if you will be calling it from a 32-bit PowerBuilder application.



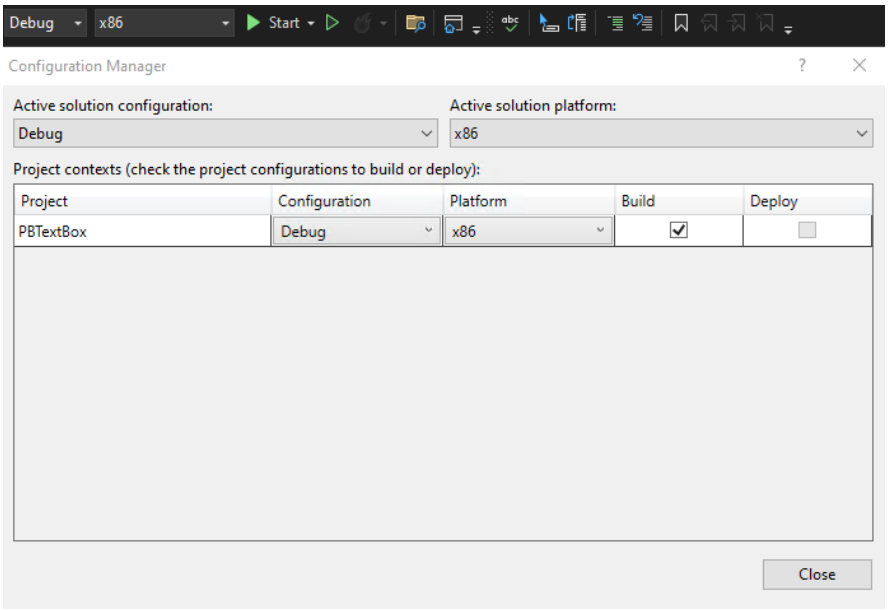
Reference Visual Basic

Right-click on References in the Solution Explorer and select Add Reference...



Click on Framework and check Microsoft.VisualBasic

This is used in some of the attributes we add to classes later



Add your control

The example uses a standard Windows Forms control, you may be using a control from a third-party. You need to follow their instructions to make sure it is available in the project, and you can place it on top of the UserControl1 or add it in code, whichever you prefer. As a PowerBuilder programmer I tend to prefer using the designers rather than just writing code.

Make sure the third-party control you want to use is available in the Toolbox and drag it onto UserControl1 (rename everything as you would like). Set the properties as you wish so it auto sizes etc.

Wrap the control

The control will come with properties, methods and events but you won't be able to directly access these from PowerBuilder. You'll need to add your own properties, methods and events that call the ones on the control. That's the wrapping part.

Add a few methods you want to call. You can directly map your methods to the third-party control methods, or just add your own more complex, and more useful methods for your project.

Add an Interface

To make those methods, properties and events available to PowerBuilder you need to hide the methods properties and events on the third-party control that can't be accessed. Otherwise the assembly won't register in a COM accessible way. You do this by adding an interface that details the properties, methods and events you added in your wrapper that you want to be available in PowerBuilder.

In the example I have one Property, Text, and two methods, GetText, ClearText. So my interface, IPBTextBoxControl, looks like this

```
public interface IPBTextBoxControl
{
   string Text { get; set; }
   string GetText();
   void ClearText();
 }
```

Inherit from this call in the main class, in the example:

```
public partial class PBTextBoxControl : UserControl, IPBTextBoxControl
```

Make it COM accessible

Now comes the tricky part, I don't really understand what all this stuff does, but it seems to work.

ActiveXControlHelper

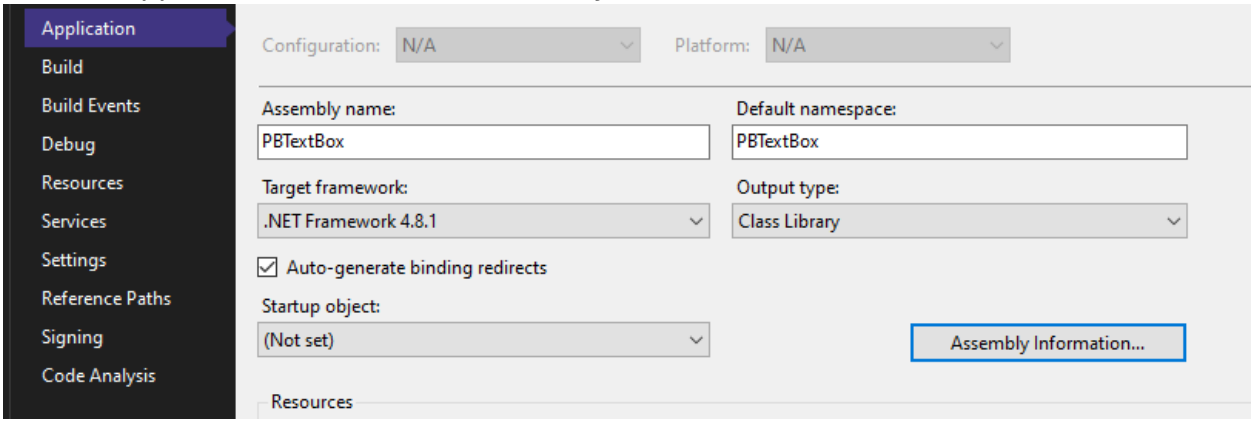
Add the ActiveXControlHelper cs file from the sample project to your project, set the correct namespace when you do.

I believe this is used to make sure the dll registers in such a way it is accessible to PowerBuilder. If you just follow the examples on the internet of creating an Active/X control, this bit is missing and PowerBuilder won't display your control in it's list of Active/X controls.

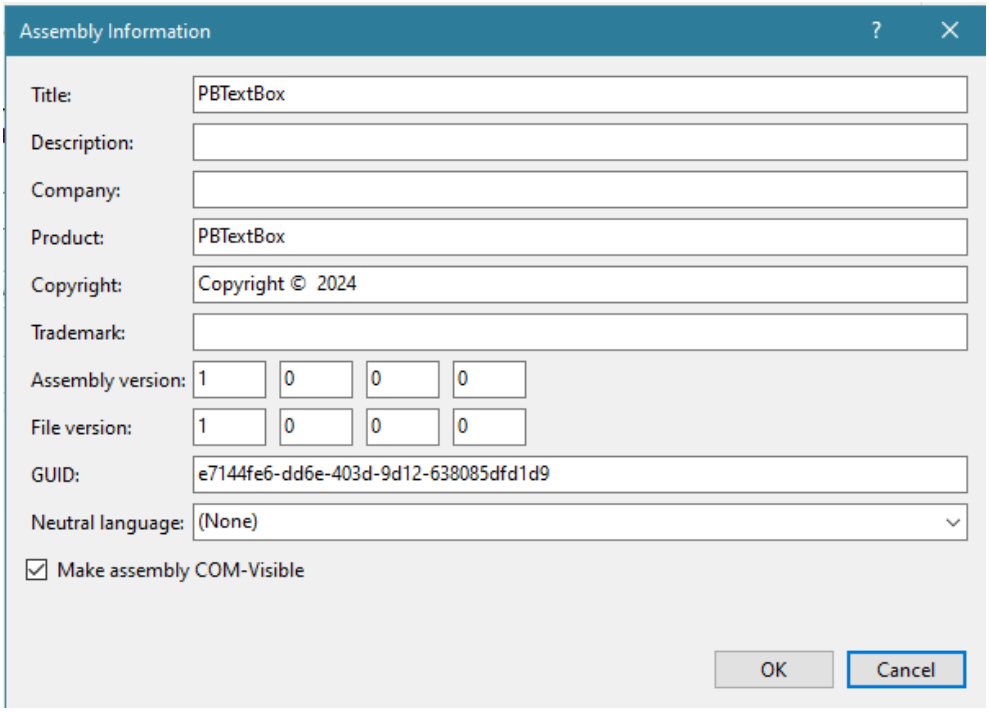
Project Properties

Right-click on you project in Solution Explorer and select Properties.

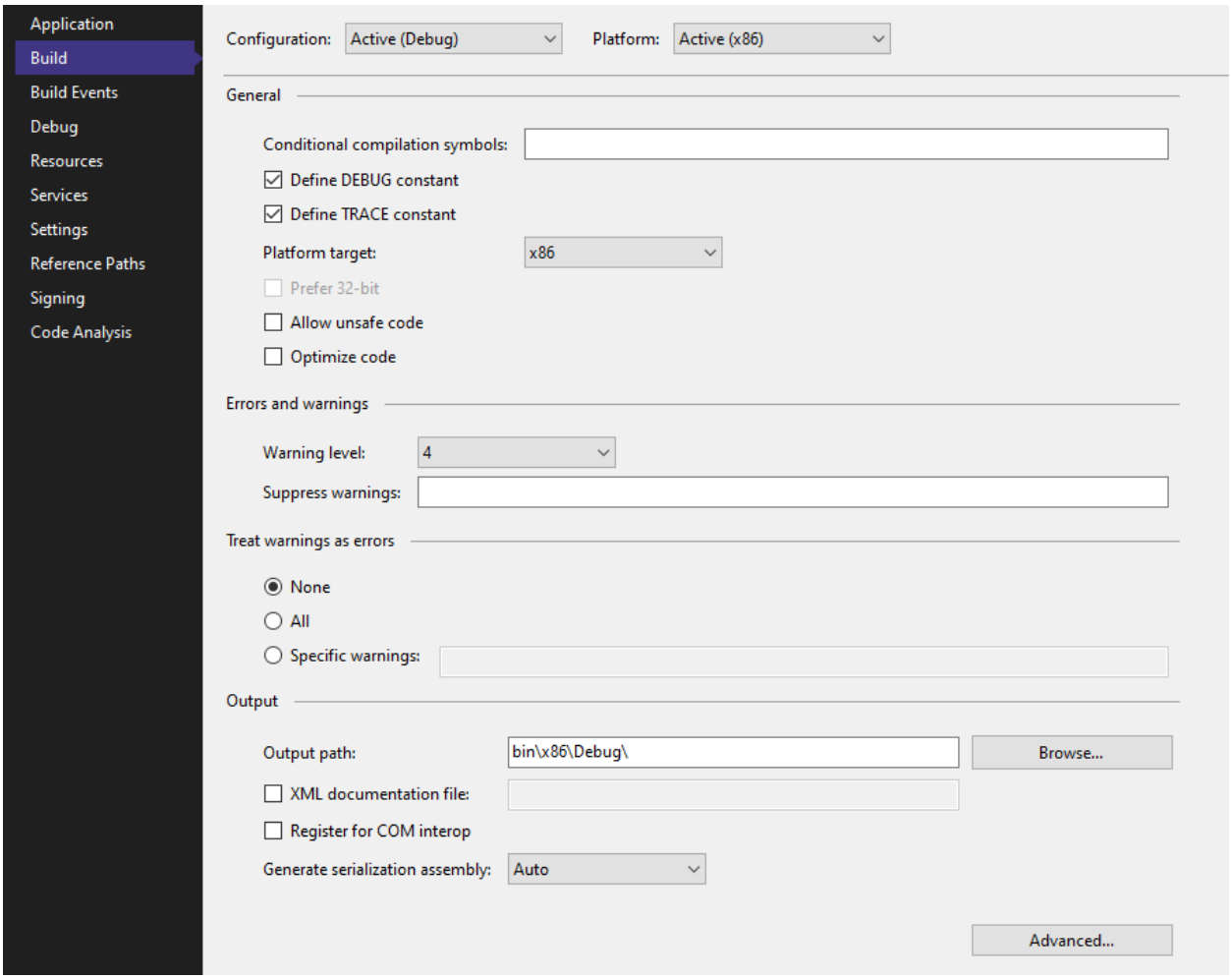
Click on Application, and then the Assembly Information button.



Check “make assembly COM visible”. No idea what this does, but it is required.



Click on Build and make sure “Register for COM interop” is not checked.



I believe this is because we need to register the dll ourselves in the way it requires rather than using the in-built register.

Making the code registry ready

Add these using to your class if any are missing

```
using Microsoft.VisualBasic;
using System;
using System.ComponentModel;
using System.Runtime.InteropServices;
using System.Windows.Forms;
```

Add this one to the Interface class

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

GUIDs are used to identify the dll in the registry so we need to specify the GUIDs ourselves. You can generate a new GUID in Visual Studio using Tools | Create GUID.

Add three constants do your class, PBTextControl in the sample. Like this:

```
public const string ClassId = "DE8F83A2-480D-4BC4-AB96-4BC1E8A159D3";
public const string InterfaceId = "A1B2C3D4-E5F6-7890-1234-56789ABCDEF0";
public const string EventsId = "97127976-5374-444F-8748-3153463A288D";
```

Create new GUIDs and replace the ones here, there should be three different GUIDs.

These identify the various parts of the class and must be changed if you want to create a new version and keep the old one. I also suggest changing the namespace or class name to add a version number, if you think you will need multiple versions on the control on a given PC at the same time.

Add the COM Register Functions to the class under the GUID declarations

```
[EditorBrowsable(EditorBrowsableState.Never)]
[ComRegisterFunction]
private static void Register(System.Type t)
{
  ComRegistration.RegisterControl(t);
}

[EditorBrowsable(EditorBrowsableState.Never)]
[ComUnregisterFunction]
private static void Unregister(System.Type t)
{
  ComRegistration.UnregisterControl(t);
}
```

Add the three attributes to the top of the class to set how the class should be registered. In the example:

```
[Guid(PBTextBoxControl.ClassId), ClassInterface(ClassInterfaceType.None)]
[ComSourceInterfaces("PBTextBox.IPBTextBoxControl")]
[ComClass(PBTextBoxControl.ClassId, PBTextBoxControl.InterfaceId, PBTextBoxControl.EventsId)]
public partial class PBTextBoxControl : UserControl, IPBTextBoxControl
```

Ensure you replace the namespace, class name, and interface name in these with yours.

Add the two attributes to the Interface class to get is ready for the registry, in the example:

```
[ComVisible(true)]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface IPBTextBoxControl
```

Build the dll

Click Build | Build Solution – check the build was successful and

Register the dll

Open a command prompt as administrator.

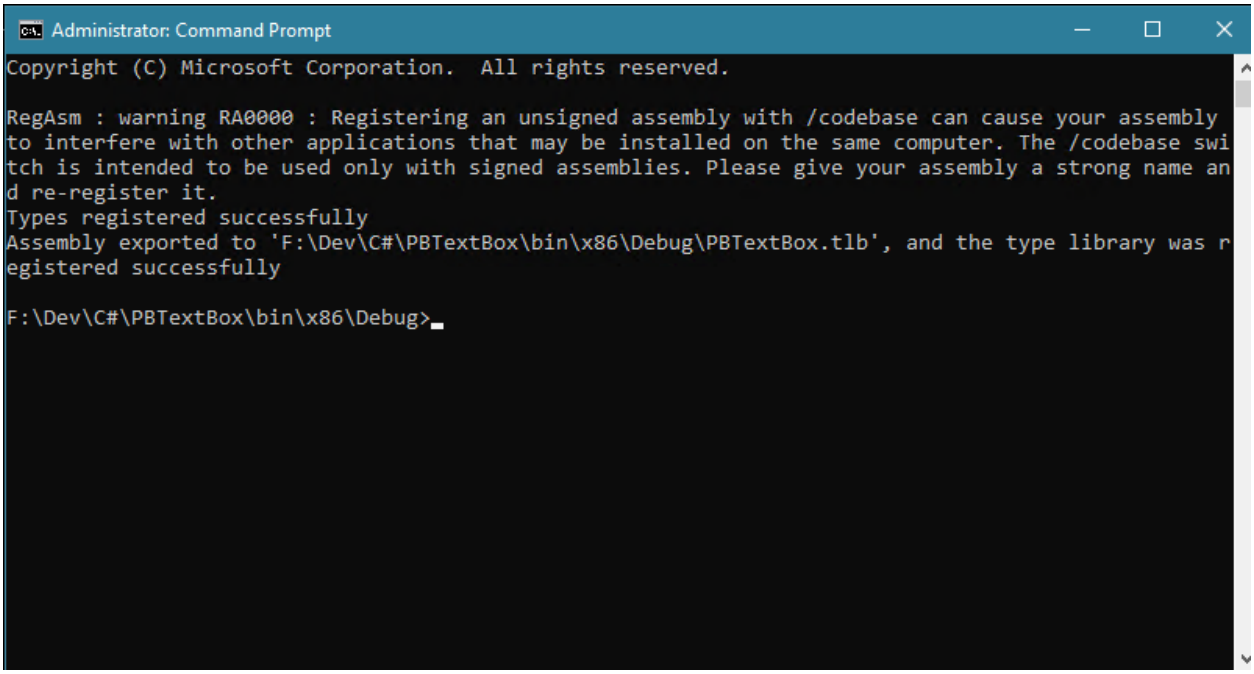
Change to the drive your dll was built to, if different from C:\

CD to the folder the dll was built to

Run

```
Regasm <yourdllname>.dll /tlb:<yourdllname>.tlb /codebase
```

You should get a result like this:



Ignore the warning and look for “Types registered successfully” and “type library was registered successfully”.

NOTE: You must be using the correct REGASM.EXE, you want the 32-bit one and the one for .NET 4.x. I usually locate these on the disk and copy them to the folder with the dll in to be sure the correct one is being used.

Feel free to automate this with a bat file or a powershell script to make life easier.

I’m not sure how often you need to actually do it, if just the once is enough, or any time you make code changes. I suspect the once may be enough if the GUIDs aren’t changed.

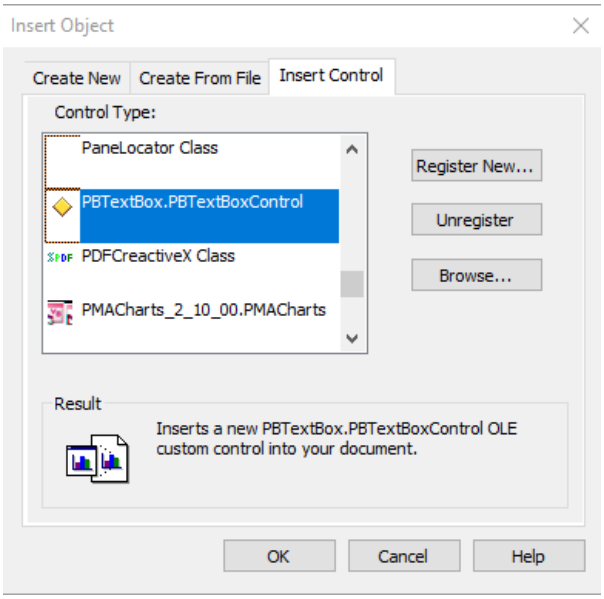
NOTE: PowerBuilder holds on to the dll. So if you run an application from the PowerBuilder IDE that uses this dll, close the application but keep the IDE open, the dll is locked, and builds will fail. You have to close the PowerBuilder IDE every time, before you can build the dll again. It’s very annoying.

Use it in PowerBuilder

Open the window you wish to place the control in in PowerBuilder.

Select Insert | Control | Ole... and then the Insert Control Tab.

Find your control in the list, it will be called <namespace>.<class>, for the example PBTextBox.PBTextBoxControl



Click on OK and place the control on your window.

Name the control as you wish, for example ole_PBTextBox

You can access the properties and methods like so:

```
Ole_PBTextBox.object.ClearText()
ole_PBTextBox.object.Text = “Hello World”
ls_text = ole_PBTextBox.GetText()
```

Arguments can be passed as long as there are maychign datatypes in .NET, and results can be returned. For complex result sets you can put them into an olo object and parse out the result:

```
ole_result = ole_control.object.Method1(param1, param2)
If IsNull(ole_result) = FALSE Then
        ls_value1 = ole_result.object.Value1
        ls_value2 = ole_result.object.Value2
End If
```

NOTE: The method and property names are case sensitive and must exactly match the one on the C# class.