PosLib Instructions

Table of Contents

PosLib - About

PosLib is a library to save and to load Windows Forms and Windows Presentation Foundation application windows / forms and their controls positioning information into a .vnml file.

The library does not have any special handlers for each different control, only one which can be given type definitions and data cast bindings in a string form paired with a *System.Type*.

Basically the library saves a WinForms forms or a WPF windows positions into a single file and restores them from that file. There is of course a mechanism to prevent a form / window to be positioned out of screen(s), if system settings have changed so that such placement would be incorrect.

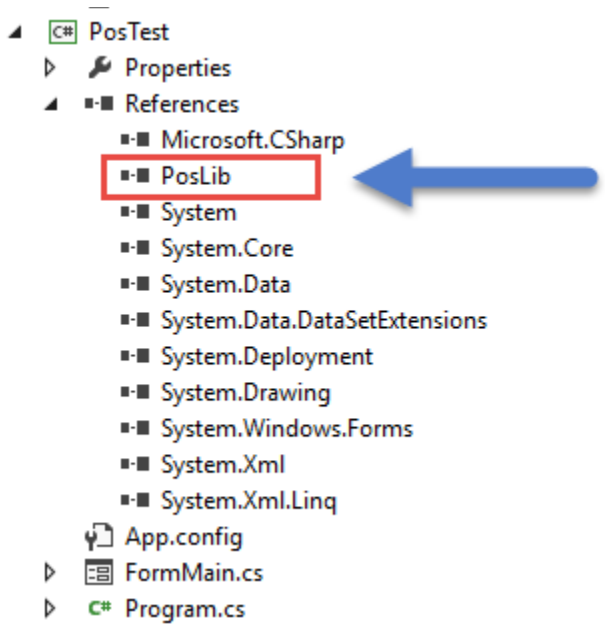
Controls and components of a form / window are given to the library in a static list so the library knows which additional information to save / restore.

1. Windows Forms

First a reference to the PosLib library is required for a Windows Forms application. The library supports any CPU setting so the processor architecture shouldn't matter.

So

- Open Visual Studio
- Create a Windows Forms application
- Add a reference to the PosLib library



1.1 Global binding

The global binding means that the PosLib saves a default data and attaches a `SystemEvents.DisplaySettingsChanged` event handler to react if display settings change during the time when the application is running. Checks are also made if the display settings have been changed since the last time the program was run to detect if changes to the loading procedure should be done.

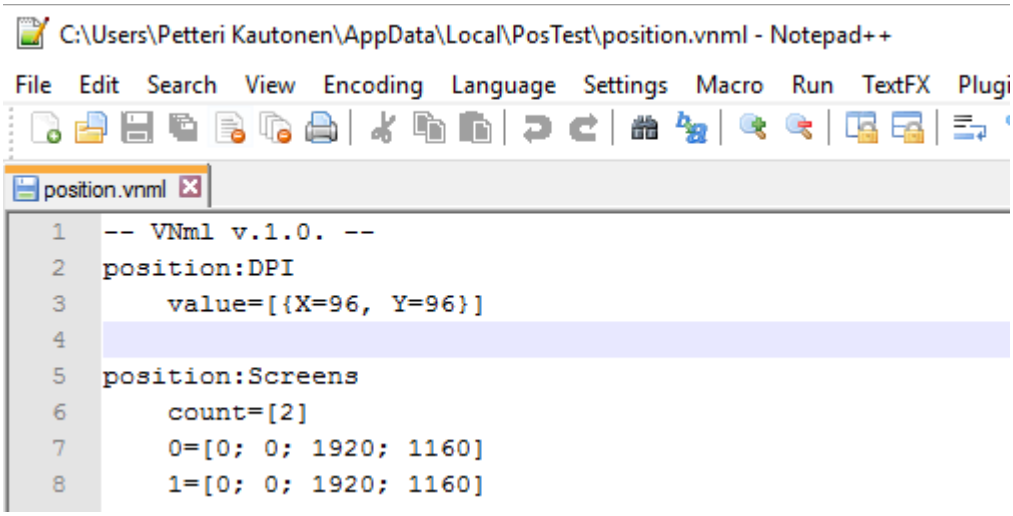
The make the binding open the *Program.cs* file and first add this line to the using section:

```
using VPKSoft.PosLib; // access the referenced namespace
```

After this we need to alter the *Main()* entry point by adding the following lines (bolded and underlined):

```
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            PositionCore.Bind(); // attach the PosLib to the application
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new FormMain());
            PositionCore.UnBind(); // release the event handlers used by the PosLib and save
the default data
        }
    }
```

Now the base thing is done. This would form a basic vnml file with some system settings stored:



1.2 Adding forms

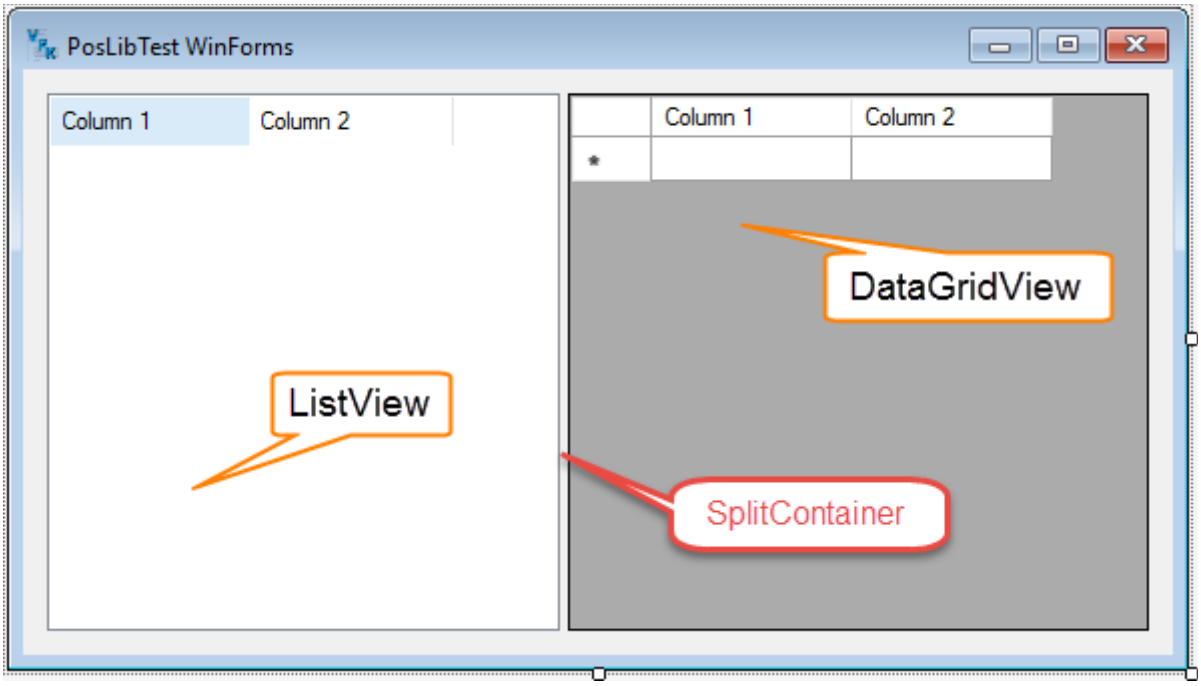
To have a form positioned a binding must also be created in form's constructor. To do so, just add the following lines (bobded and underlined):

```
using VPKSoft.PosLib;

namespace PosTest
{
    public partial class FormMain : Form
    {
        public FormMain()
        {
            // Add this form to be positioned..
            PositionForms.Add(this, PositionCore.SizeChangeMode.MoveTopLeft);

            InitializeComponent();
```

Nothing else needs to be done as the *Add(this, ..)* attaches handlers to save and to load the forms position.

1.3 A sample form

Now to create a very simple, useless and ugly sample form. We use *SplitContainer*, *DataGridView* and *ListView* for this sample form.

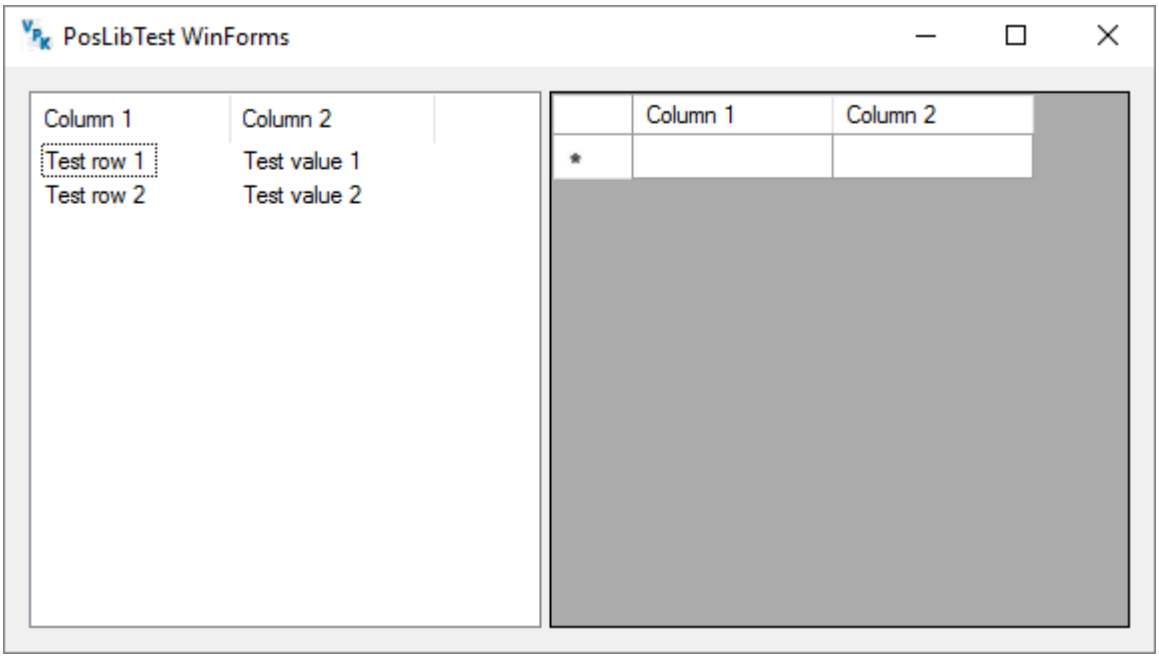


To add some content to the form:

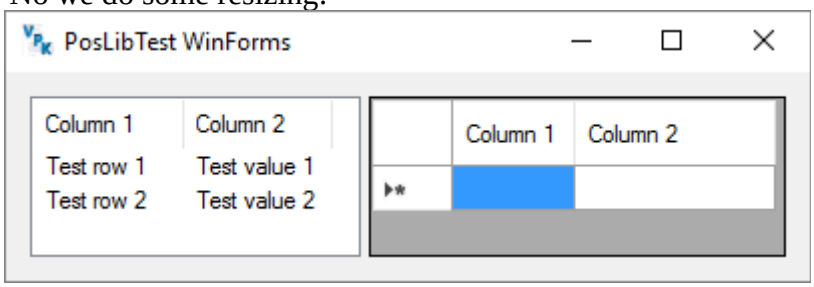
```
ListViewItem lvi = new ListViewItem("Test row 1");
lvi.SubItems.Add(new ListViewItem.ListViewSubItem(lvi, "Test value 1"));
listView1.Items.Add(lvi);
lvi = new ListViewItem("Test row 2");
lvi.SubItems.Add(new ListViewItem.ListViewSubItem(lvi, "Test value 2"));
listView1.Items.Add(lvi);
```

That's all the code we need.

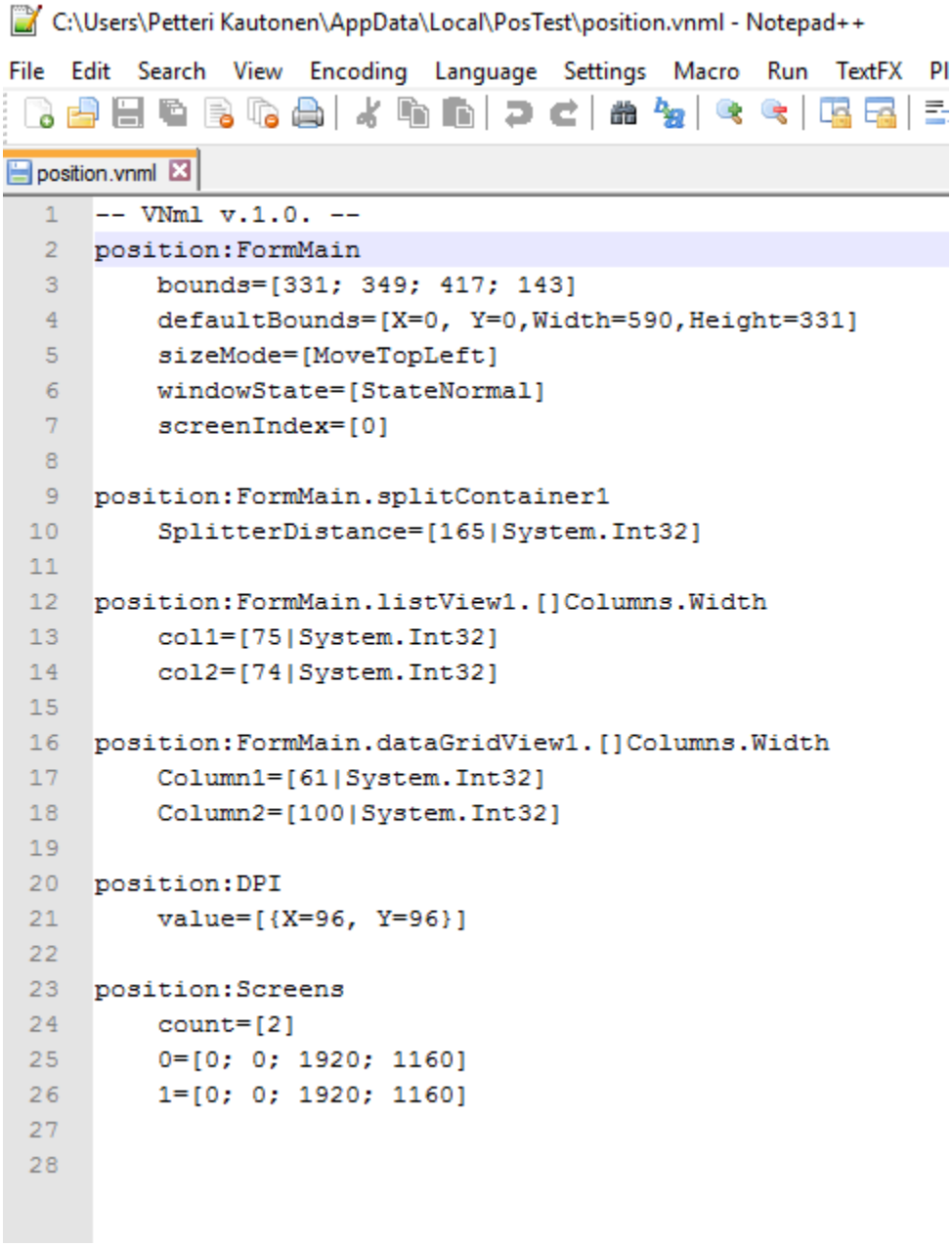
Lets run the test application. At first the main form looks quite like in the designer:



No we do some resizing:



Lets close the application and have a look at what it did save:



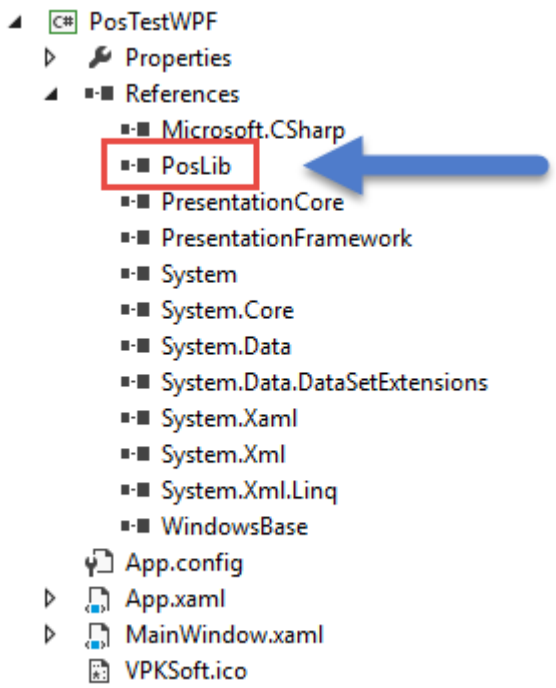
So there is some global data such as the systems DPI definition and the system screen count including their working area sizes in pixels. Then there is the FormMain's positioning information including the column widths. Now after we restart the application the size and outlook should be the same.

2. Windows Presentation Foundation

First a reference to the PosLib library is required for a WPF application. The library supports any CPU setting so the processor architecture shouldn't matter.

So

- Open Visual Studio
- Create a WPF application
- Add a reference to the PosLib library



2.1 Global binding

The global binding means that the PosLib saves a default data and attaches a `SystemEvents.DisplaySettingsChanged` event handler to react if display settings change during the time when the application is running. Checks are also made if the display settings have been changed since the last time the program was run to detect if changes to the loading procedure should be done.

The make the binding open the App.*xml* file and add event handlers for Startup and Exit:

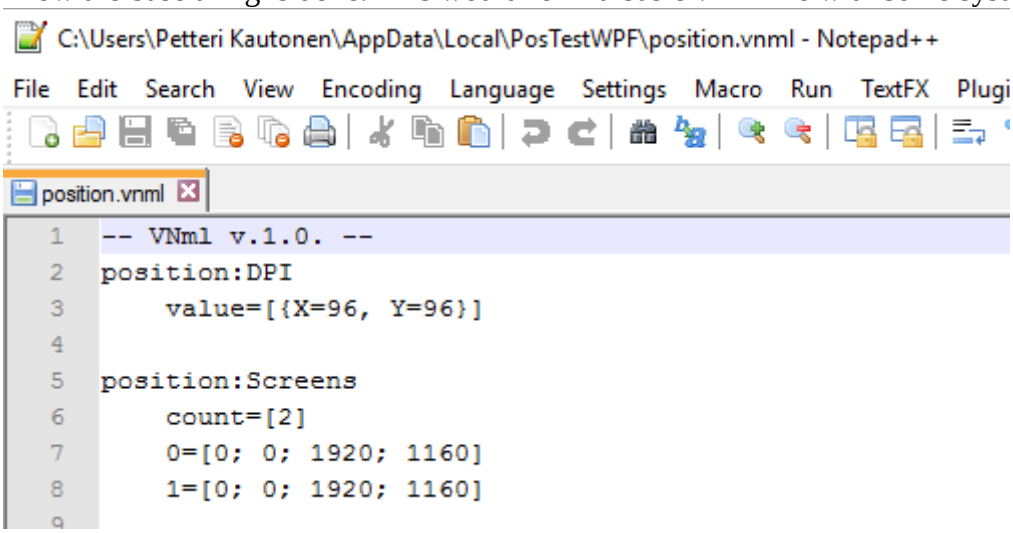
```
<Application x:Uid="Application_1" x:Class="PosTestWPF.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml" Startup="Application_Startup"
Exit="Application_Exit">
    <Application.Resources>

    </Application.Resources>
</Application>
```

The bolded and underlined parts are the required event handlers.

Next we edit the *App.xaml.cs* file and add required content to the event handlers (bolded and underlined):

```
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            PositionCore.Bind(); // attach the PosLib to the application
        }

        private void Application_Exit(object sender, ExitEventArgs e)
        {
            PositionCore.UnBind(); // release the event handlers used by the PosLib and save
the default data
        }
    }
```

Now the base thing is done. This would form a basic vnml file with some system settings stored:



2.2 Adding windows

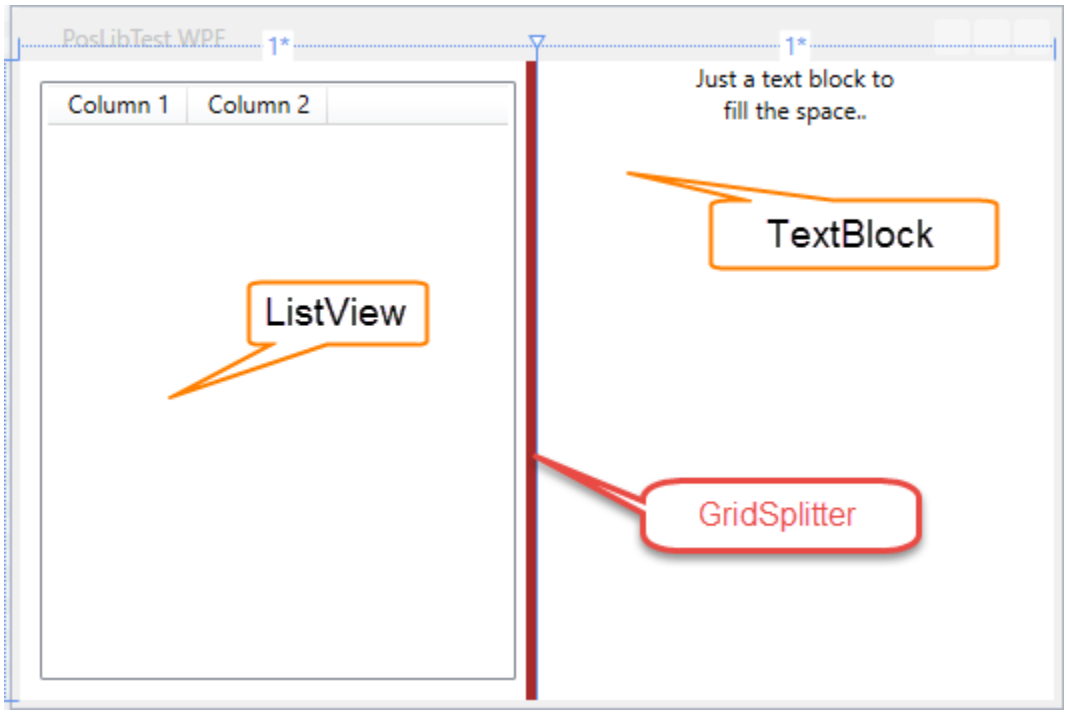
To have a form positioned a binding must also be created in window's constructor. To do so, just add the following lines (bobded and underlined):

```
using VPKSoft.PosLib;

namespace PosTestWPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            PositionsWindow.Add(this); // Add this window to be positioned..
```

Nothing else needs to be done as the *Add(this, ..)* attaches handlers to save and to load the window's position.

2.3 A sample window

Now to create a very simple, useless and ugly sample window. We use *ListView*, *GridSplitter* and *TextBlock* for this sample window.



To add some content to the window, we first declare a class inside the MainWindow class definition:

```
public class Row
{
    public string Col1 { get; set; }
    public string Col2 { get; set; }
}
```

This Row-named class or actually its member names will be bound to the ListView's columns:

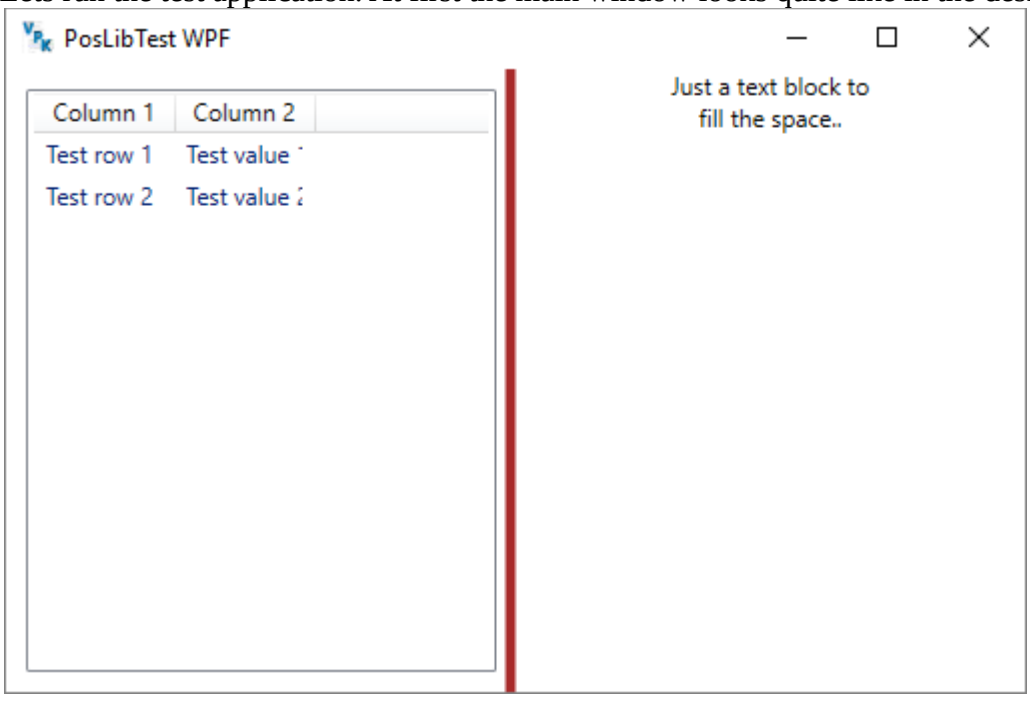
```
<ListView x:Name="listView1" x:Uid="ListView_1" Margin="10">
    <ListView.View>
        <GridView x:Uid="GridView_1">
            <GridViewColumn x:Uid="GridViewColumn_1" Header="Column 1"
DisplayMemberBinding="{Binding Col1}" Width="70" />
            <GridViewColumn x:Uid="GridViewColumn_2" Header="Column 2"
DisplayMemberBinding="{Binding Col2}" Width="70" />
        </GridView>
    </ListView.View>
</ListView>
```

No to add the actually useless content:

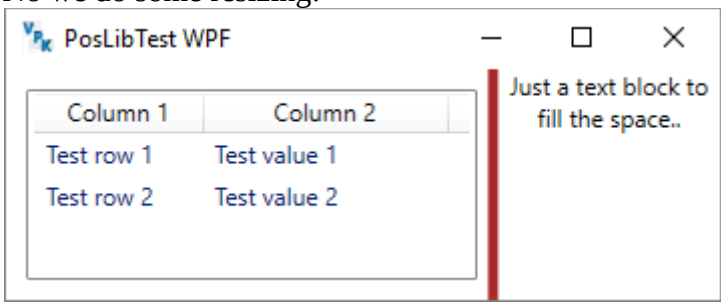
```
listView1.Items.Add(new Row() { Col1 = "Test row 1", Col2 = "Test value 1" });
listView1.Items.Add(new Row() { Col1 = "Test row 2", Col2 = "Test value 2" });
```

That's all the code we need.

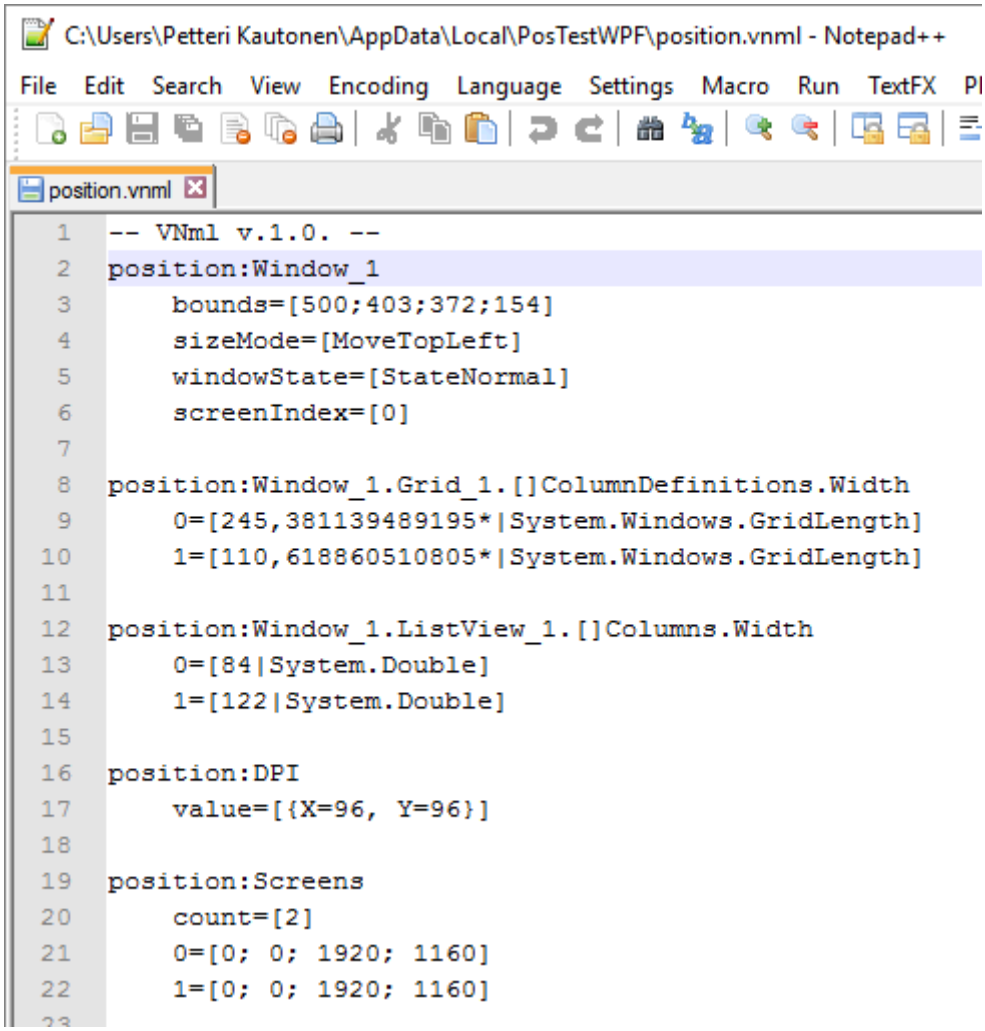
Lets run the test application. At first the main window looks quite like in the designer:



No we do some resizing:



Lets close the application and have a look at what it did save:



So there is some global data such as the systems DPI definition and the system screen count including their working area sizes in pixels. Then there is the MainWindos's positioning information including the ListView's column widths. Now after we restart the application the size and outlook should be the same.

2.4 Automatic Uid's

With WPF we need something to reference a Control by and not all controls have names, so with the application's build events come help in this matter.

We add two lines to the Build Events Pre-build event command line:

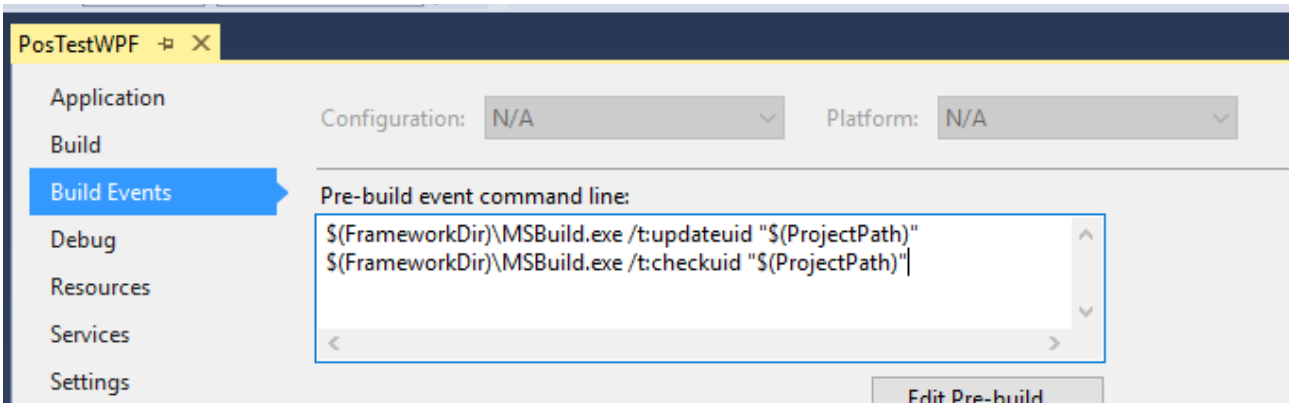
```
$(FrameworkDir)\MSBuild.exe /t:updateuid "$(ProjectPath)"
$(FrameworkDir)\MSBuild.exe /t:checkuid "$(ProjectPath)"
```

These lines will generate x:Uid definitions for the XAML, as seen in the sample window definition:

```
<Window x:Uid="Window 1" x:Class="PosTestWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PosLibTest WPF" Height="350" Width="525" Icon="VPKSoft.ico">
    <Grid x:Uid="Grid 1">
```

To comment the “Pre-build's” all that is required is to add two double dots before the command line like this:

```
::$(FrameworkDir)\MSBuild.exe /t:updateuid "$(ProjectPath)"
::$(FrameworkDir)\MSBuild.exe /t:checkuid "$(ProjectPath)"
```



3. Advanced functionality

There are some advanced properties in the library which weren't covered by the WinForms or the WPF instruction articles. We go over them in this chapter.

3.1 Adding properties to save

The PosLib library supports adding or removal of properties to the saved either by adding them in code or by modifying the PosLib.vnml file.

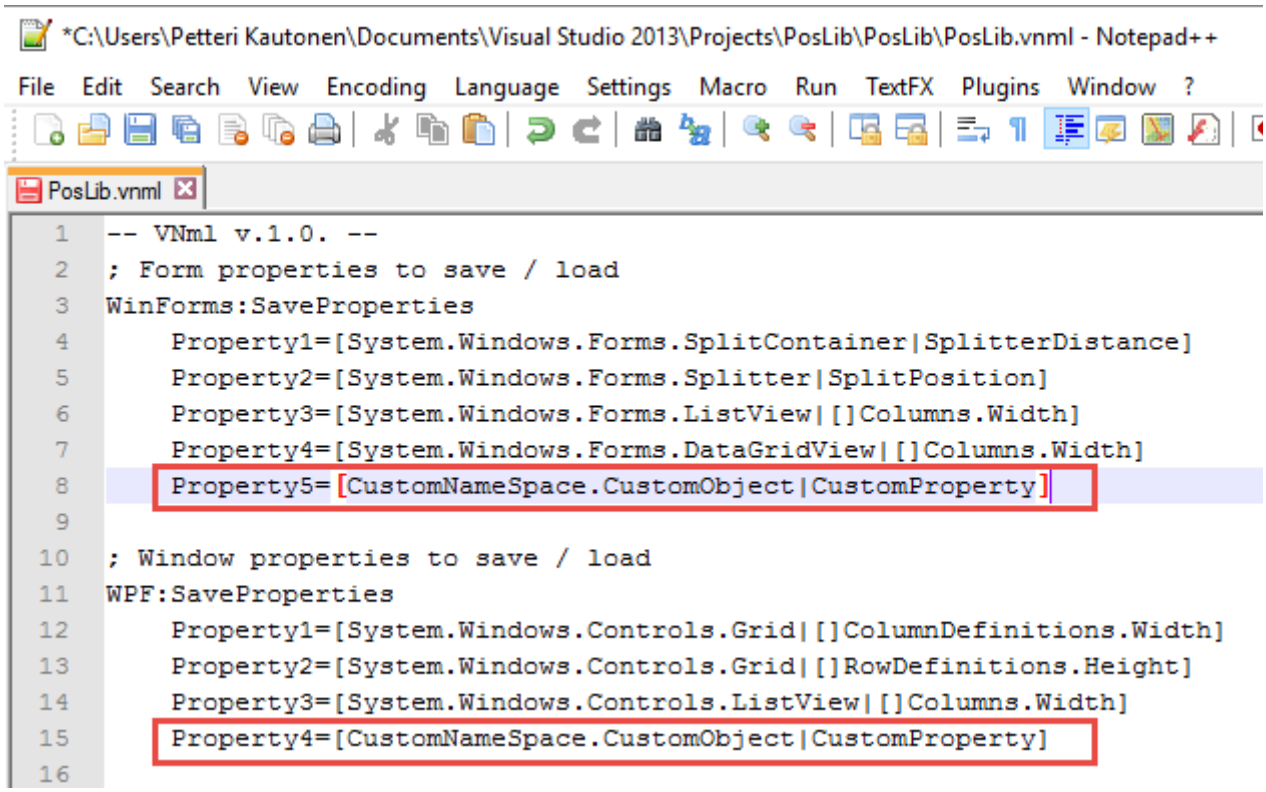
To add a custom property to be saved in WPF:

```
PositionsWindow.SaveWindowProperties.Add(
       new KeyValuePair<string, string>("CustomNameSpace.CustomObject", "CustomProperty"));
```

To Add a custom property to be saved with WinForms:

```
PositionForms.SaveFormProperties.Add(
       new KeyValuePair<string, string>("CustomNameSpace.CustomObject", "CustomProperty"));
```

By editing the PosLib.vnml file directly:



3.1.1 Custom type casts in WPF

With WPF some types need to be cast to another type (usually upper type in the inheritance tree) before saving or loading it's property.

To understand this lets have a look at the code directly:

```
/// <summary>
/// A List of KeyValuePair class instances which hold the
/// <para/>predefined casts from Type to Type instructed by the TypeCast class.
/// </summary>
public static List<KeyValuePair<string, TypeCast>> Casts =
new List<KeyValuePair<string, TypeCast>>(new KeyValuePair<string, TypeCast>[]
{
    new KeyValuePair<string, TypeCast>("System.Windows.Controls.ListView",
    new TypeCast(Assembly.GetAssembly(typeof(GridView)),
    "System.Windows.Controls.GridView",
    "View",
    "[]Columns.Width"))
});
```

So an assembly reference is definitely required as well as the source and destination types.

Another piece of code is also required for cast definition to pair with the PosLib library:

```
/// <summary>
/// A list of type names and their value to be loaded/saved to a position.vnml file.
/// <remarks>If a string describing a property value starts
/// <para/>with [] (square brackets) the value is considered as a collection,
/// <para/>which can be cast to IList interface for enumeration.</remarks>
/// </summary>
public static List<KeyValuePair<string, string>> SaveWindowProperties =
new List<KeyValuePair<string, string>>(new KeyValuePair<string, string>[]
{
       new KeyValuePair<string, string>("System.Windows.Controls.Grid",
              "[]ColumnDefinitions.Width"),
       new KeyValuePair<string, string>("System.Windows.Controls.Grid",
              "[]RowDefinitions.Height"),
       new KeyValuePair<string, string>("System.Windows.Controls.ListView",
              "[]Columns.Width")
});
```

The bolded parts will make a *ListView.View* property to be cast to GridView and then to be handled as *ListView.Columns:*

```
12  position:Window_1.ListView_1.[]Columns.Width
13      0=[84|System.Double]
14      1=[122|System.Double]
```

Such casts are required when a required type value to be saved is not visible in the logical or in the visual tree, although seen in the xaml (bold):

```
        <ListView x:Name="listView1" x:Uid="ListView_1" Margin="10">
            <ListView.View>
                <GridView x:Uid="GridView_1">
                    <GridViewColumn x:Uid="GridViewColumn_1" Header="Column 1"
DisplayMemberBinding="{Binding Col1}" Width="70" />
                    <GridViewColumn x:Uid="GridViewColumn_2" Header="Column 2"
DisplayMemberBinding="{Binding Col2}" Width="70" />
                </GridView>
            </ListView.View>
        </ListView>
```

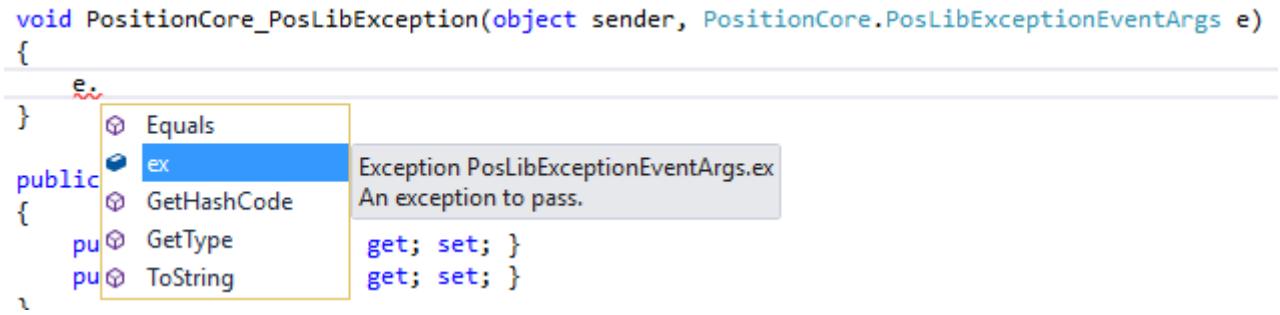
3.2 PosLib exception handling

The library supports two methods of exception handling. The default is that an event is fired on an exception giving the exception information with the event's arguments.

To add an event handler:

```
PositionCore.PosLibException += PositionCore_PosLibException;
```

Then you just need to add some code to react to the exception information:



Or if you just don't care no event handler shouldn't be added.

The second method is much more aggressive and will lead to application crash if an error occurs in the PosLib library. This is indicated by setting the *PositionCore.ExcludeTypeConversionExceptions* to false.

3.3 Values of unknown type

The basic list of type values to save contains only basic integer and floating point types:

```
public static List<Type> EnumerableTypes = new List<Type>(new Type[] {
                                                          typeof(int),
                                                          typeof(double),
                                                          typeof(long),
                                                          typeof(float),
                                                          typeof(uint),
                                                          typeof(ulong)});
```

Whether to get converters for other types is defined by the property *PositionCore.GetTypeDescriptors*, which default value is set to true.

There is also possibility to add custom *TypeConverter – Type* bindings using the *PositionCore.CustomTypeConverters*.

```
/// <summary>
/// A list of TypeConverter class instances paired to a type.
/// </summary>
public static List<KeyValuePair<Type, TypeConverter>> CustomTypeConverters = new
List<KeyValuePair<Type, TypeConverter>>();
```

3.4 Reset positioning data

The are some circumstances when the positioning data for a WinForm form or a WPF window shouldn't be loaded such as screen settings were changed after a last application session or a command line argument was given to prevent loading the positioning data.

To prevent the PosLib from loading the positioning data with a command line argument, just give argument -- *skipPos* to the application. Do note that the *SkipLoad* property is read only so you won't have any affect on this one.

```
/// <summary>
/// Gets a value if the PosLib should skip loading the position data.
/// <para/>This is indicated with giving a command line argument --skipPos
/// <para/>for the application's command line.
/// </summary>
public static bool SkipLoad
```

For the other property you have affect on. The property is called *SkipLoadOnScreenChange*, which indicates to the library if positioning data should be reset if screen settings were changed after a last application session:

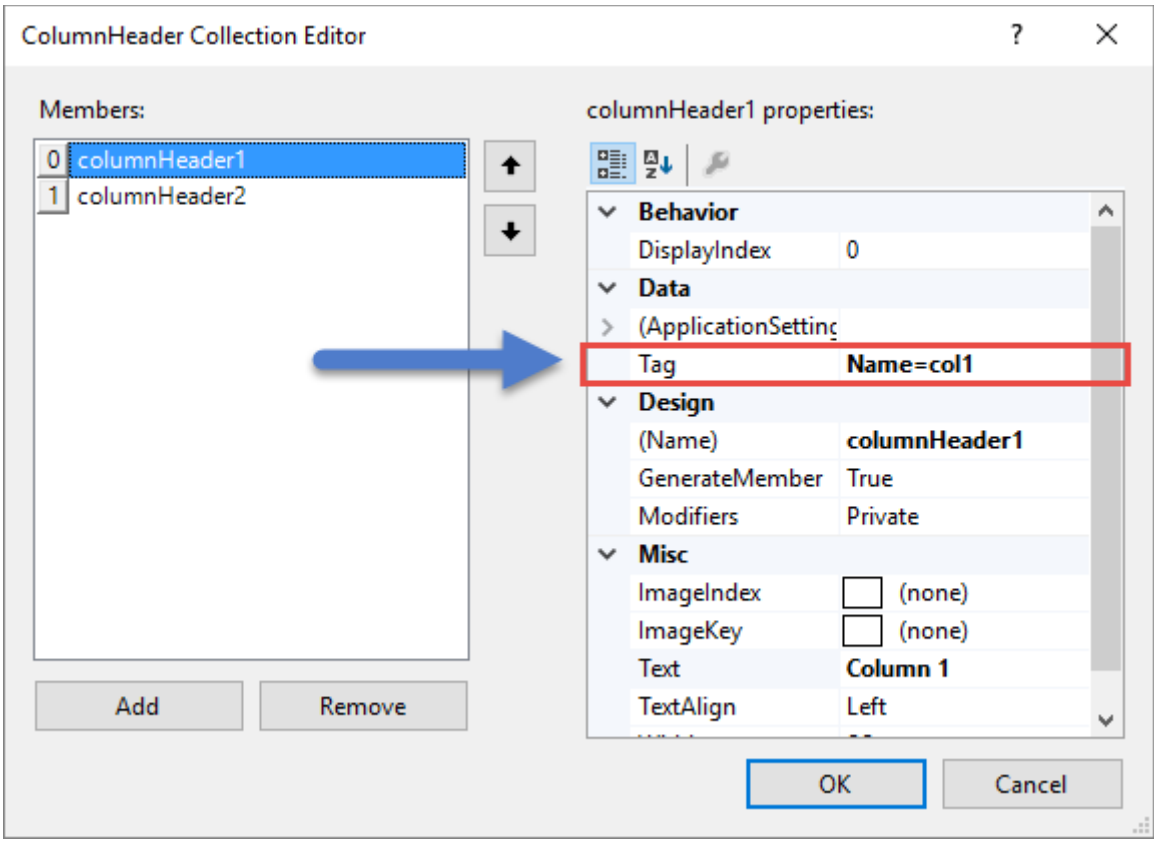
```
/// <summary>
/// Gets or sets a value indicating if a form/window position load
/// <para/>should be skipped if screen setting changed after the last application start.
/// </summary>
public bool SkipLoadOnScreenChange
```

3.5 Referencing controls by “name”

The PosLib gets a control “name” by trying to access few properties of the control.
These are in following order:

1. Uid (WPF only)
2. Name
3. Tag=Name=”Name”
4. An internal fallback name used in collection item indexers.

This Tag=Name=”Name” is used with for example with *System.Windows.Forms.ListView.Colums* collection as a the Name assigned to *System.Windows.Forms.ColumnHeader* appears to be a desing time-only property:



4. Thank you

Thanks for the patience to read this document. Happy programming sessions for everyone.

Petteri Kautonen / VPKSoft

By the way you are **allowed** to **ask questions**. vpksoft@vpksoft.net.