

Creation of Plug-in Control Library

Version 1.4

2018-02-28

Application Note AN-IND-1-017

Author Vector Informatik GmbH

Restrictions Public Document

Abstract Instruction for the integration of customer's own plug-in controls in the Panel Designer

Table of Contents

1.0	Overview	2
2.0	Requirements	2
3.0	Instructions	2
3.1	Creating a Project in Visual Studio	2
3.2	Creating the Plug-in Control Library.....	4
3.3	Creating Plug-in Controls	4
3.3.1	Plug-in Control Properties	4
3.3.2	Special Plug-in Control Properties	5
3.3.3	Receiving Values	5
3.3.4	Sending Values	6
3.3.5	Serialization/deserialization	7
3.4	Bitmaps for Plug-in Controls	7
3.5	Use of Plug-in DLL	9
3.6	Exceptions.....	9
4.0	Attachment.....	10
4.1	Interfaces of Vector.PanelControlPlugin.dll	10
4.1.1	IPanelControlPluginLibrary	10
4.1.2	IPluginPanelControl	10
4.1.3	IProvidesSupportedDataTypes	11
4.1.4	IExchangeSymbolValue	12
4.2	Example of Implementation of a Plug-in Control Library	12
4.3	Example of Implementation of a Plug-in Control with User Control as Member	13
4.4	Example of Implementation of a Plug-in Control with Derivation from an Existing Control	13
4.5	Example of the Implementation of a Plug-in Control Interpreting a Struct Value	14
4.6	Example of Implementation of a Plug-in Control Using a WPF Control	14
5.0	Trademarks	15
6.0	Contacts	15

1.0 Overview

The use of Vector's own controls is described in the online help of the Panel Designer. This document expands the description to include the integration of the customer's own plug-in controls in the Panel Designer.

The following sections describe the integration of the customer's own plug-in controls in the Panel Designer.

Many examples are given as source text in the descriptions and in the appendix. The locations in these source texts that require adaptation (because they are user-specific) are marked with =====> and <=====.

2.0 Requirements

CANoe/CANalyzer since version 8.1 SP4 or CANape since version 14 is installed.

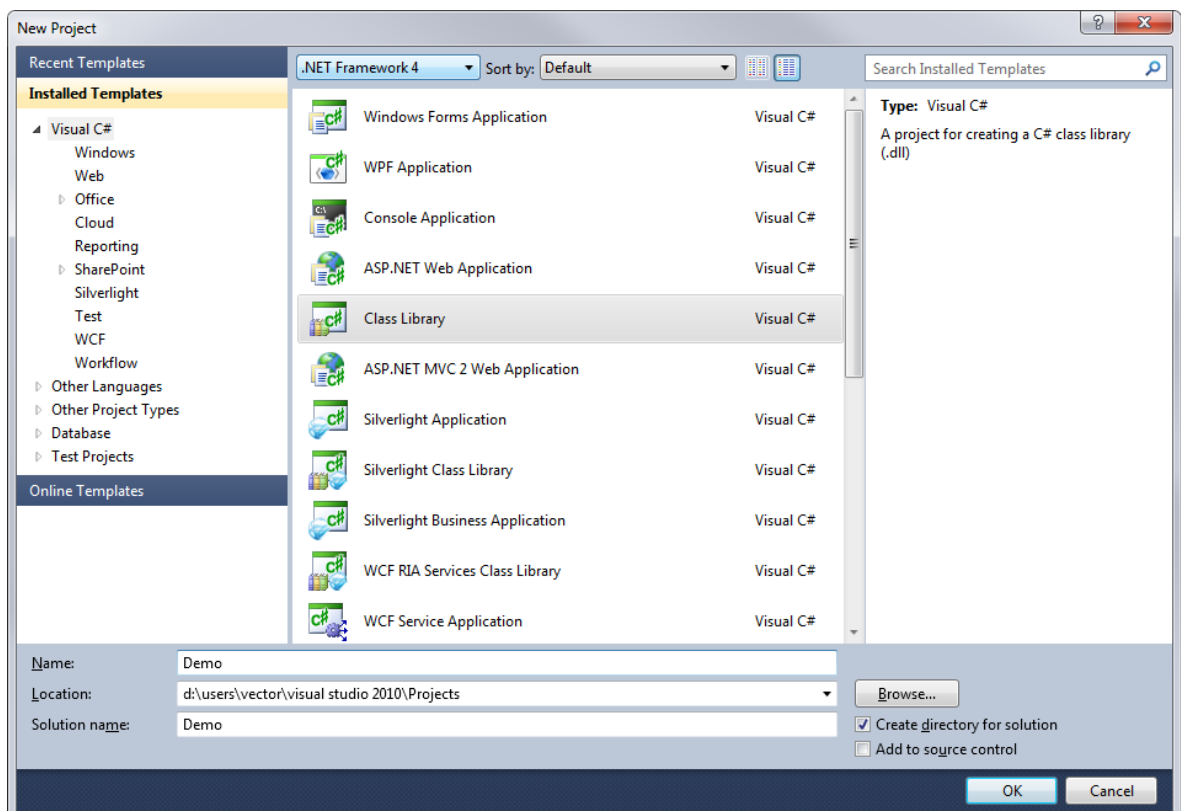
A development environment for .NET is present.

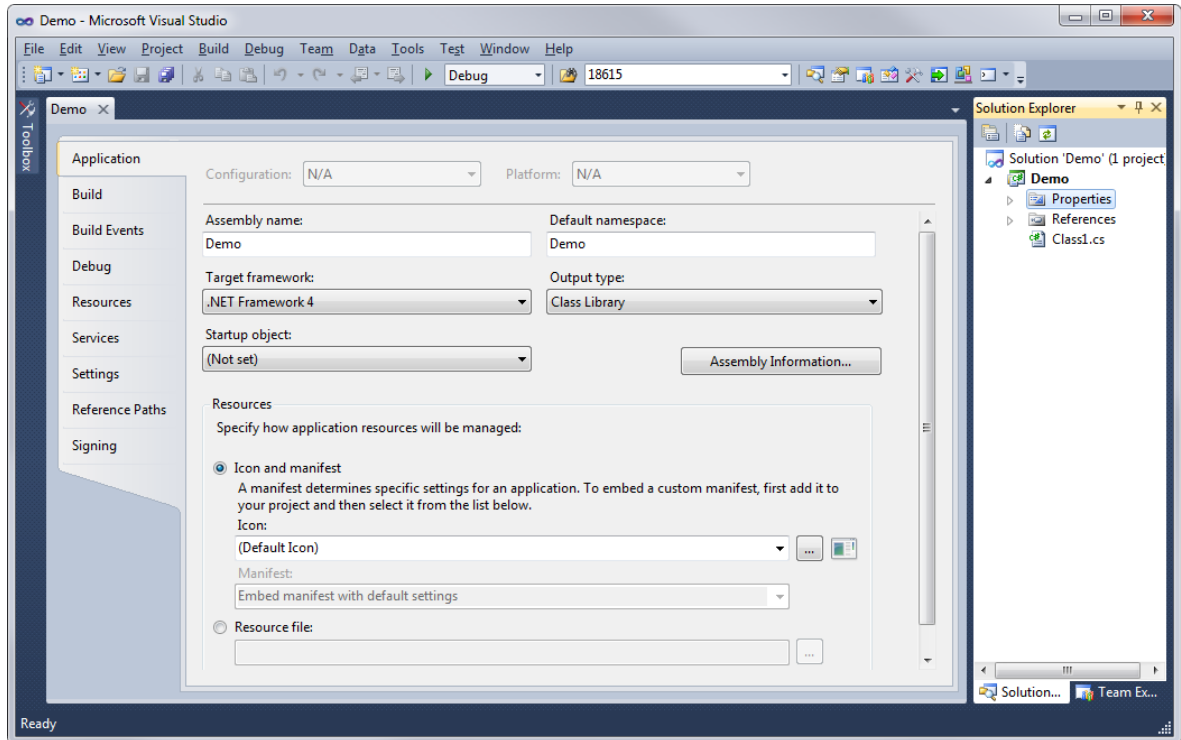
In this document, the procedure is described using Microsoft Visual Studio 2010® with C# programming language.

3.0 Instructions

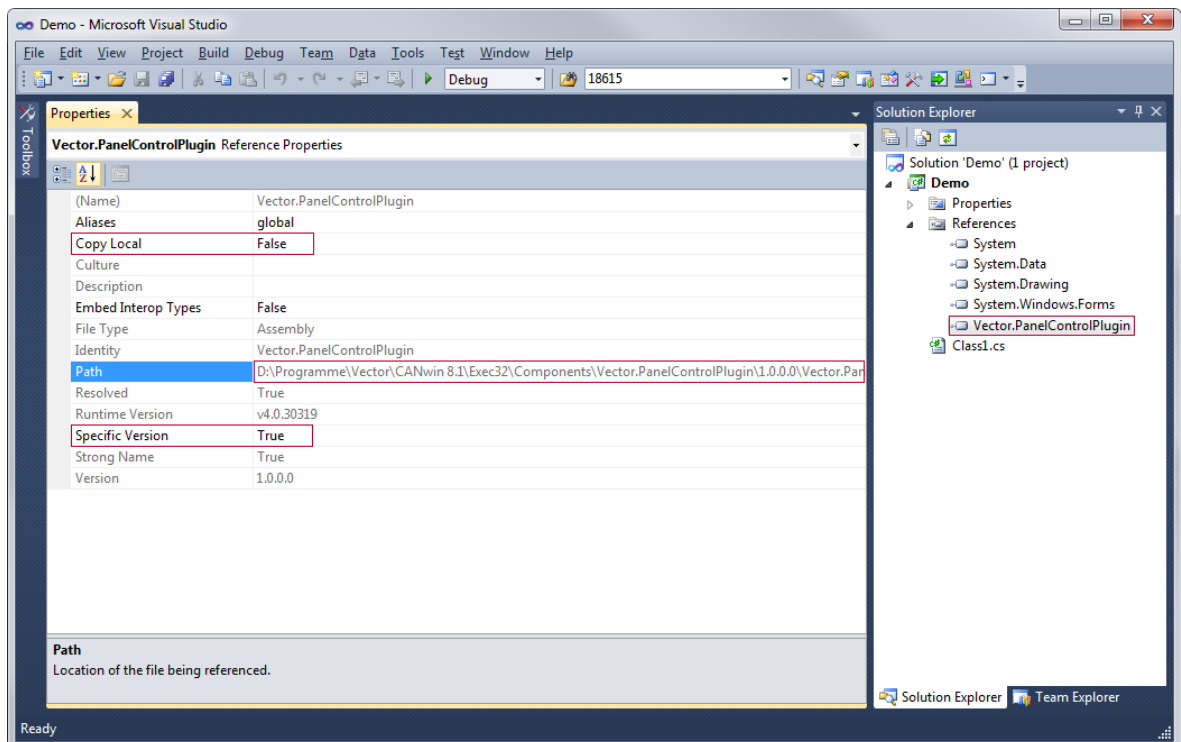
3.1 Creating a Project in Visual Studio

1. Create a new project of type **ClassLibrary**. Select **.NET-Framework 4** as the target framework.





2. Use the **Add Reference...** shortcut menu command in the Solution Explorer (References) to add a reference to the DLL **Vector.PanelControlPlugin.dll** in directory **Exec32\Components\Vector.PanelControlPlugin**. Select the subdirectory with the current version number, e.g., 1.0.0.0.
3. Configure **Copy Local** to **False** and **Specific Version** to **True**.



4. The following **References** are also needed for creating the plug-in control library and the plug-in controls:

- > System
- > System.Data
- > System.Drawing
- > System.Windows.Forms

3.2 Creating the Plug-in Control Library

In the created project, create a class that implements the **IPanelControlPluginLibrary** interface.

This class enables the plug-in control library that is to be integrated to be displayed in the Panel Designer with an icon and a name.

A separate page is created for each plug-in control library in the toolbox of the Panel Designer during runtime. This page lists all plug-in controls available in this library.

The specification of an image is optional. If zero is returned here, only the name of the library is displayed in the toolbox. Otherwise, the image is displayed and the name is displayed in the ScreenTip (see also the figure in section 3.5).

Ensure that the **public** class is declared.

Section 4.2 contains the source text of the example class.

3.3 Creating Plug-in Controls

In the created project, create a class that implements the **IPluginPanelControl** interface.

This class implements the interface of a plug-in control to the Panel Designer. A plug-in control library can have several of these classes that implement this interface. All plug-in controls of a library are displayed on the same page in the toolbox of the Panel Designer (see also figure in section 3.5).

Ensure that the **public** class is declared.

Sections 4.3 und 4.4 contain the source text of the example class.

The following sections provide a more detailed explanation of particular aspects of the class, especially the locations that require adaptation.

3.3.1 Plug-in Control Properties

Properties of a plug-in control can be configured in the Panel Designer. In order for the properties of your plug-in control to be available in the Panel Designer, they must be implemented as **public properties**. They can be provided with the attributes that influence the display in the property grid of the Panel Designer:

1. `[CategoryAttribute("MyControl Settings ")]`
This attribute specifies the category in which the property is displayed.
2. `[DisplayName("My Title")]`
This attribute specifies the text that is displayed for this property in the property grid.

There is additionally the **SupportedProperties** property, which returns a list of supported properties. The names of all properties that are to be displayed in the Panel Designer must be indicated here.

```
/// <summary>
/// Returns the list of property names, which are supported by the plugin
/// control,i.e. the list of properties, which can be configured in the
/// property grid of the Panel Designer.
/// </summary>
public IList<string> SupportedProperties
{
    get
    {
        List<string> supportedProperties = new List<string>();

        // =====>
        supportedProperties.Add("MyTitle");
    }
}
```

```
supportedProperties.Add("MyBorderStyle");  
// <=====  
  
return supportedProperties;  
}  
}
```

3.3.2 Special Plug-in Control Properties

There are a few default properties of plug-in controls that must be handled separately. This concerns the following properties that can be controlled via CAPL:

- > ControlBackColor
- > ControlForeColor

A plug-in control that supports one or both of these properties returns **true** in the associated properties **SupportsPropertyBackColor** and **SupportsPropertyForeColor** and implements the corresponding property for setting the background or foreground color.

```
/// <summary>  
/// Property to set or get the background color of the plugin control.  
/// This method can called via CAPL, to control the display of controls.  
/// </summary>  
public Color ControlBackColor  
{  
    // <=====  
    get  
    {  
        return mMyUserControl.BackColor;  
    }  
    set  
    {  
        mMyUserControl.BackColor = value;  
    }  
    // <=====  
}
```

A plug-in control that does not support one or both of these properties returns **false** in the associated properties **SupportsPropertyBackColor** and **SupportsPropertyForeColor** and makes available a dummy implementation of the properties in **ControlBackColor** and **ControlForeColor**:

```
public Color ControlBackColor  
{  
    // dummy implementation, if the plugin control does not support changing the  
    // background color:  
    get; set;  
}
```

3.3.3 Receiving Values

The **IExchangeSymbolValue** interface is used for receiving and sending of symbol values, i.e., values of a linked signal or linked variable. When the plug-in control is loaded in CANoe, an object of this type is transferred (**Initialize** method). The data type of the symbol value is specified by the linked symbol and the underlying database. Symbols of Integer, Float, Long Array, Float Array, String, Byte Array or Struct type can be linked. Depending on the data type linked, the symbol value can be read via a specific property, e.g. the **LongValue** property (for Integer type) or the **DoubleValue** property (for Float type). If the value is set in CANoe, the **ValueChanged** event is triggered. In order to display the symbol value in the plug-in control, a corresponding event handler must be created for this event.

```
public void Initialize(IExchangeSymbolValue value)  
{  
    // Remember the symbol value object to be able to receive a value in the  
    // plugin control and send a value from the plugin control  
    mSymbolValue = value;  
  
    // Assign an event handler for receiving values from CANoe  
    mSymbolValue.ValueChanged += OnRxValue;
```

```

    ...
}

/// <summary>
/// Actions which are necessary, when a new value is received from CANoe.
/// The received value is in this-value and must be sent to the plugin control.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
void OnRxValue(object sender, EventArgs e)
{
    try
    {
        // Value.SymbolDataType depends on the symbol data type of the assigned
        // symbol. It is given from the CANoe database and must not be changed.
        switch (SymbolValue.SymbolDataType)
        {
            case ExchangeSymbolDataType.Long:
                // =====>
                // display Value.LongValue in the plugin control
                mMyControl.OnRxValue(SymbolValue.LongValue);
                // <=====
                break;
            case ExchangeSymbolDataType.Double:
                // =====>
                // display Value.DoubleValue in the plugin control
                mMyControl.OnRxValue(SymbolValue.DoubleValue);
                // <=====
                break;
            default:
                break;
        }
    }
    catch (System.Exception)
    {
    }
}

```

3.3.4 Sending Values

If the symbol value in the plug-in control can be changed, the value change is also transferred via the **IExchangeSymbolValue** interface. In the event handler that the plug-in control informs about the value change, the corresponding **LongValue**, **DoubleValue**,... property with the new value must be set.

```

/// <summary>
/// This method is called, when the panel is loaded in CANoe/CANalyzer.
/// </summary>
/// <param name="value">Object, which is used to transmit a symbol value during
/// the measurement. </param>
public void Initialize(IExchangeSymbolValue value)
{
    ...
    // =====> ONLY FOR TX: necessary only, if the plugin control shall send
    // values
    // Assign an event handler for sending a value from the plugin control.
    // If the plugin control only receives values, this is not necessary.
    mMyControl.ValueChanged += new
                                MyUserController.ValueChangedEventHandler (OnTxValue);
    // <=====
}

// =====> ONLY FOR TX: necessary only, if the plugin control shall send values
/// <summary>
/// Actions which are necessary, when a new value shall be sent to CANoe.
/// The sent value must be written to this.Value.
/// </summary>

```

```
/// <param name="sender"></param>
/// <param name="e">conts the value to be sent</param>
void OnTxValue(object sender, ValueChangedEventArgs e)
{
    try
    {
        if (mSymbolValue == null ||
            mSymbolValue.SymbolDataType == ExchangeSymbolDataType.Unknown)
            return;

        // Value.SymbolDataType depends on the symbol data type of the assigned
        // symbol. It is given from the CANoe database and must not be changed.
        switch (mSymbolValue.SymbolDataType)
        {
            case ExchangeSymbolDataType.Long:
                // =====>
                // put the value from the plugin control to Value.LongValue
                mSymbolValue.LongValue = Convert.ToInt32(e.mValue);
                // <=====
                break;
            case ExchangeSymbolDataType.Double:
                // =====>
                // put the value from the plugin control to Value.DoubleValue
                mSymbolValue.DoubleValue = e.mValue;
                // <=====
                break;
            default:
                break;
        }
    }
    catch (System.Exception)
    {
        // make sure, that no exceptions are passed
    }
}
// <=====
```

3.3.5 Serialization/deserialization

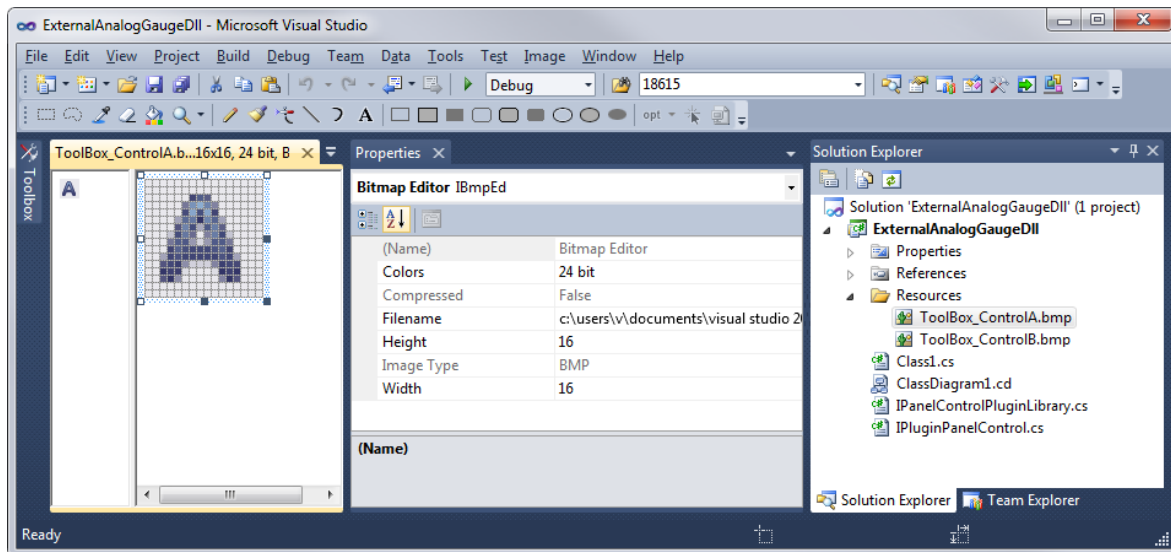
The properties of a plug-in control are serialized in the Panel file and are thereby available for loading in CANoe or in the Panel Designer. The serializing of the properties of a plug-in control must be implemented in the **SerializeSupportedProperties** method. The deserialization is implemented in the **DeserializeSupportedProperties** method.

Ensure that these methods are compatible with each other and that additions are downward compatible, i.e., that new properties are always added at the end and, if necessary, ignored.

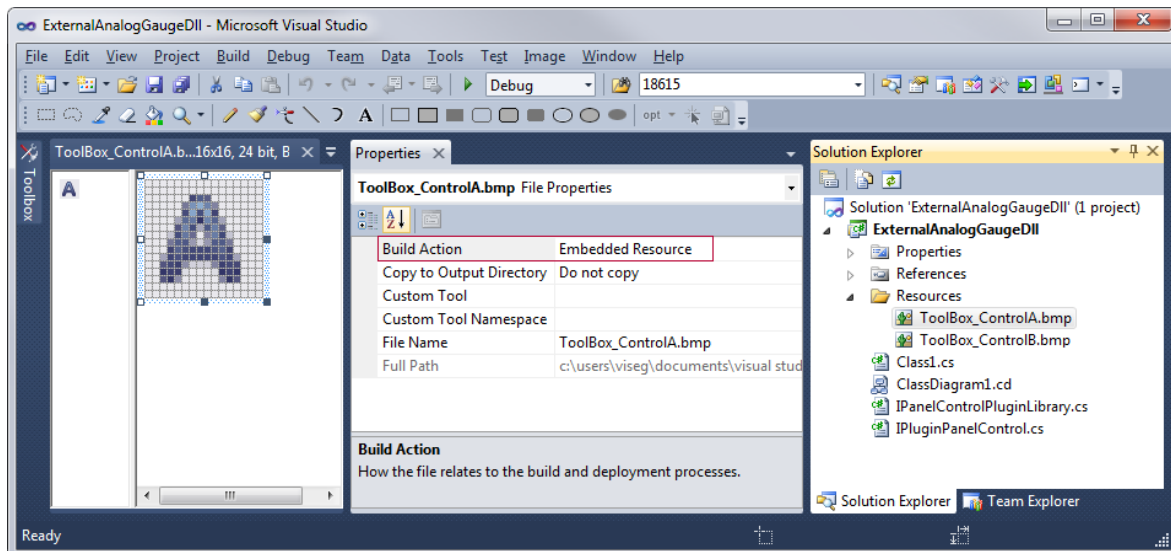
3.4 Bitmaps for Plug-in Controls

If you want to support icons for the plug-in control library and the toolbox items of the plug-in controls, create a 16x16 bitmap in each of the resources.

Note that you set the transparent color for the entire bitmap via the color in the lower left corner.



Set the **Build Action** to **Embedded Resource**:



The toolbox bitmap of the plug-in control is set via the **ToolboxBitmap** attribute of the class. A very specific syntax is required for this so that the bitmap is also found later in the resource. Integrate the bitmap in the following way:

```
// The file must be found in the subdirectory "Resources" of the project
// Set Build Action of the bmp-file to "Embedded Resource"
[ToolboxBitmap(typeof(resxfinder), "Demo.Resources.ToolBox_ControlA.bmp")]
public class MyControlImplA : IPluginPanelControl
```

The **resxfinder** class that is used in the **ToolboxBitmap** attribute must exist once outside any namespace in the project:

```
public class resxfinder
{
    // This is a dirty workaround for a bug in 'GetImageFromResource'.
    // Use the type of this this class in the 'ToolboxBitmap' Attribute and
    // NOT the Control class. The second parameter is the bmp name with
    // the DEFAULT(!) namespace of the assembly (see project settings) and the
    // relative directory
    // where the bitmap can be found.
    // Class 'resxfinder' must be within this assembly.
}
```

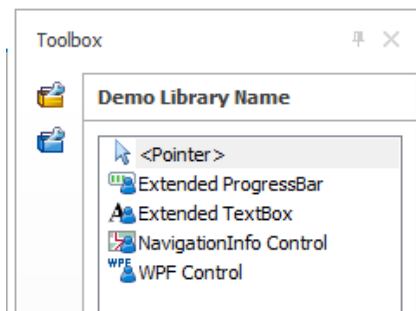



```
// See also: http://www.bobpowell.net/toolboxbitmap.htm
// Example:
// Default namespace: "Demo"
// directory: Resources
// Bitmap name: "ToolBox_ControlA.bmp"
// Attribute:
// [ToolboxBitmap(typeof(resxfinder), "Demo.Resources.ToolBox_ControlA.bmp")]
}
```

3.5 Use of Plug-in DLL

Compile the DLL and copy it to the **Exec32\ControlLibraries** directory of the installed CANoe version.

Start CANoe with the desired configuration and start the Panel Designer. In the toolbox, under the icon for the Vector standard controls, you now see the bitmap or the name of the created plug-in control library. If you select this, the plug-in controls contained in this library appear in the toolbox.



You can then use the plug-in controls for the design of your panel. If you want to edit the properties of the plug-in controls in the Properties Window, you may not see everything at first. This is because only the most important properties of a plug-in control are displayed at first. If you want to see all properties, select  in the toolbar of the Properties Window.

You can link the plug-in control with a signal, variable, or diagnostic parameter, as usual. Integer- and Float-type symbols are supported at present.

3.6 Exceptions

An unclean implementation of the plug-in control can cause CANoe to crash. This is the case if uncaught exceptions occur in event handlers (e.g., **OnLoad** or **OnPaint**) or in the properties. For this reason, it must be ensured that all exceptions are caught within the plug-in control DLL with a try-catch block.

4.0 Attachment

4.1 Interfaces of Vector.PanelControlPlugin.dll

4.1.1 IPanelControlPluginLibrary

```
public interface IPanelControlPluginLibrary
{
    /// <summary>
    /// Returns the name of the plugin control library as it shall be shown in the
    /// toolbox of the Panel Designer
    /// </summary>
    string LibraryName { get; }

    /// <summary>
    /// Returns a bitmap (16x16), which shall be shown in the toolbox of the
    /// Panel Designer.
    /// Return null, if no image shall be shown. In this case, the name of the
    /// library is shown.
    /// </summary>
    Image LibraryImage { get; }
}
```

4.1.2 IPluginPanelControl

```
public interface IPluginPanelControl
{
    /// <summary>
    /// This method is called, when the panel is loaded in CANoe/CANalyzer.
    /// </summary>
    /// <param name="value">Object, which is used to transmit a symbol value during
    /// the measurement. </param>
    void Initialize(IExchangeSymbolValue value);

    /// <summary>
    /// Returns the name of the plugin control, which is shown in the toolbox of
    /// the Panel Designer.
    /// </summary>
    string ControlName { get; }

    /// <summary>
    /// Returns the control, which actually shall be shown in the panel.
    /// </summary>
    Control ExternalControl { get; }

    /// <summary>
    /// This property is for exchanging the symbol value.
    /// </summary>
    IExchangeSymbolValue SymbolValue { get; set; }

    /// <summary>
    /// Returns the list of property names, which are supported by the plugin
    /// control, i.e. the list of properties, which can be configured in the
    /// property grid of the Panel Designer.
    /// </summary>
    IList<string> SupportedProperties { get; }

    /// <summary>
    /// Returns true, if the plugin control can change the background color,
    /// otherwise false.
    /// </summary>
    bool SupportsPropertyBackColor { get; }

    /// <summary>
    /// Returns true, if the plugin control can change the foreground color,
    /// otherwise false.
    /// </summary>
    bool SupportsPropertyForeColor { get; }
}
```

```
// if the following properties are not needed, they may have a dummy
// implementation (doing nothing)

/// <summary>
/// Property to set or get the background color of the plugin control.
/// This method can called via CAPL, to control the display of controls.
/// Provide a dummy implementation (do nothing), if the plugin control cannot
/// change its background color.
/// </summary>
Color ControlBackColor { get; set; }

/// <summary>
/// Property to set or get the foreground color of the plugin control.
/// This method can called via CAPL, to control the display of controls.
/// Provide a dummy implementation (do nothing), if the plugin control cannot
/// change its foreground color.
/// </summary>
Color ControlForeColor { get; set; }

/// <summary>
/// Property to enable or disable the plugin control.
/// This property is called, when the DisplayOnly property is set in the
/// property grid of the Panel Designer.
/// This method can called via CAPL, to control the display/behavior of
/// controls.
/// Provide a dummy implementation (do nothing), if the plugin control cannot
/// be disabled/enabled, e.g, when the plugin control is a display-only
/// control.
/// </summary>
bool Enabled { get; set; }

/// <summary>
/// Property to set the plugin control visible or invisible.
/// This method can called via CAPL, to control the display of controls.
/// Provide a dummy implementation (do nothing), if the plugin control cannot
/// be hidden.
/// </summary>
bool Visible { get; set; }

/// <summary>
/// Serializes all supported properties and returns the serialization string in
/// the out parameter.
/// </summary>
/// <param name="serializationString">serialized properties</param>
/// <returns>true, if serialization was successful, otherwise false</returns>
bool SerializeSupportedProperties(out string serializationString);

/// <summary>
/// Deserializes the supported properties from the given string.
/// </summary>
/// <param name="serializationString">serialized properties</param>
/// <returns>true, if deserialization was successful, otherwise false</returns>
bool DeserializeSupportedProperties(string serializationString);
}
```

4.1.3 IProvidesSupportedDataTypes

```
public interface IProvidesSupportedDataTypes
{
    /// <summary>
    /// Returns the data types, which are supported by the plugin control
    /// </summary>
    ExchangeSymbolDataType SupportedDataTypes { get; }
}
```

4.1.4 IExchangeSymbolValue

```
/// <summary>
/// Enumeration contains the data types, which are supported for plugin
/// controls
/// </summary>
[Flags]
public enum ExchangeSymbolDataType
{
    ByteArray = 8,
    Double = 2,
    DoubleArray = 0x20,
    DoubleArray2D = 0x40,
    DoubleArray3D = 0x80,
    Long = 1,
    LongArray = 0x10,
    LongLong = 0x200,
    String = 4,
    Struct = 0x100,
    Unknown = 0
}

/// <summary>
/// Class for the exchange of a symbol value from and to CANoe/CANalyzer.
/// </summary>
public interface IExchangeSymbolValue
{
    /// <summary>
    /// data type of the assigned symbol (i.e. of the signal, variable,...)
    /// </summary>
    ExchangeSymbolDataType SymbolDataType { get; }
    long LongValue { get; set; }
    double DoubleValue { get; set; }
    byte[] ByteArray { get; set; }
    string StringValue { get; set; }
    int[] LongArray { get; set; }
    double[] DoubleArray { get; set; }

    event EventHandler ValueChanged;
}
```

4.2 Example of Implementation of a Plug-in Control Library

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Vector.PanelControlPlugin;

namespace Demo
{
    public class ControlLibraryImpl : IPanelControlPluginLibrary
    {
        #region IPanelControlPluginLibrary Members

        /// <summary>
        /// Returns the name of the plugin control library as it shall be shown in
        /// the toolbox of the Panel Designer.
        /// </summary>
        public string LibraryName
        {
            get
            {
                // =====>
                return "Demo Library Name";
                // <=====
            }
        }
    }
}
```

```

/// <summary>
/// Returns a bitmap (16x16), which shall be shown in the toolbox of the
/// Panel Designer.
/// Return null, if no image shall be shown. In this case, the name of the
/// library is shown.
/// </summary>
public System.Drawing.Image LibraryImage
{
    get
    {
        // =====>
        return Properties.Resources.LibraryImage;
        // <=====
    }
}

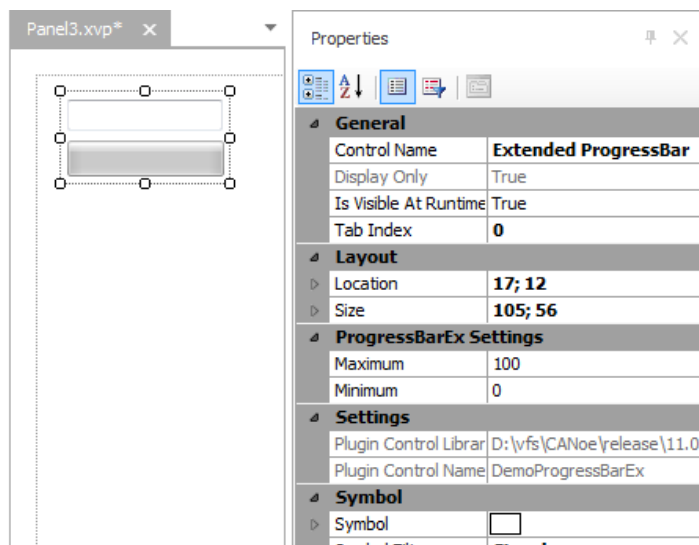
#endregion
}
}

```

4.3 Example of Implementation of a Plug-in Control with User Control as Member

The example **DemoProgressBarEx** contains the class **DemoProgressBarEx**, which implements **IPluginPanelControl** and the user control **ProgressBarInternal**, which is a member of **DemoProgressBarEx**.

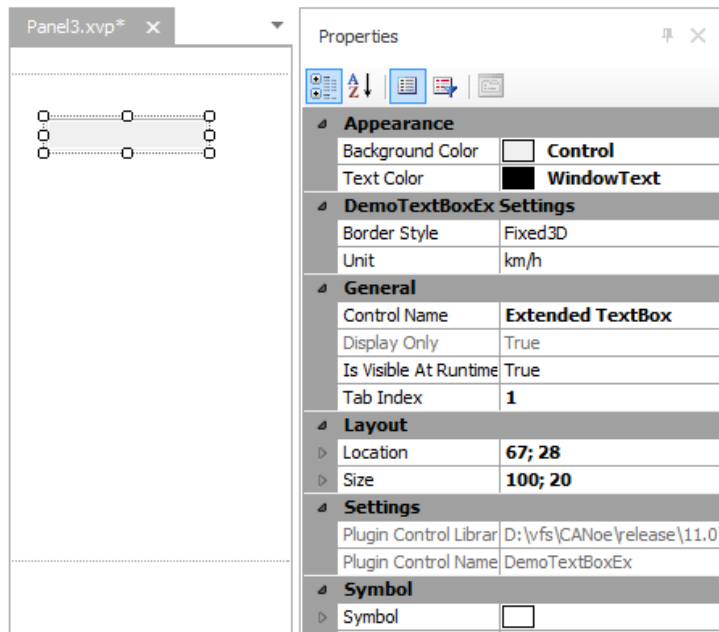
The user control shows the use of an aggregated control, which can receive and send long or double values. The value is displayed in a text box as well in a progress bar. The value can be changed in the text box. The control has two specific properties **Min** and **Max** for specifying the range of the progress bar. It does not support the default properties **BackColor** and **ForeColor**.



See the source files **DemoProgressBarEx.cs** and **DemoProgressBarInternal.cs** for implementation details.

4.4 Example of Implementation of a Plug-in Control with Derivation from an Existing Control

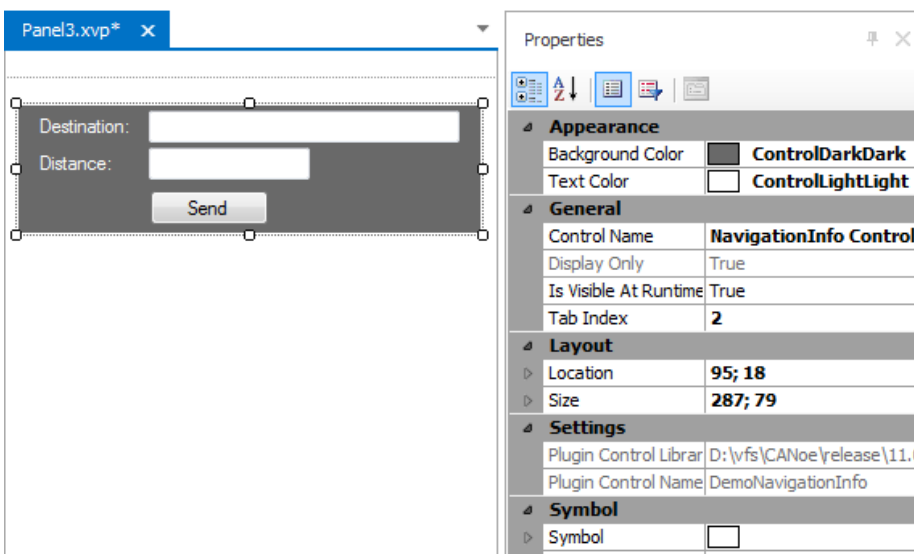
The example **DemoTextBoxEx** shows the use of a user control which can receive long, double or string values and which is derived directly from a **TextBox** control. The control supports the default properties **BackColor** and **ForeColor** as well as two specific properties **Border Style** and **Unit**. At runtime the unit is always displayed behind the received value.



See the source file DemoTextBoxEx.cs for implementation details.

4.5 Example of the Implementation of a Plug-in Control Interpreting a Struct Value

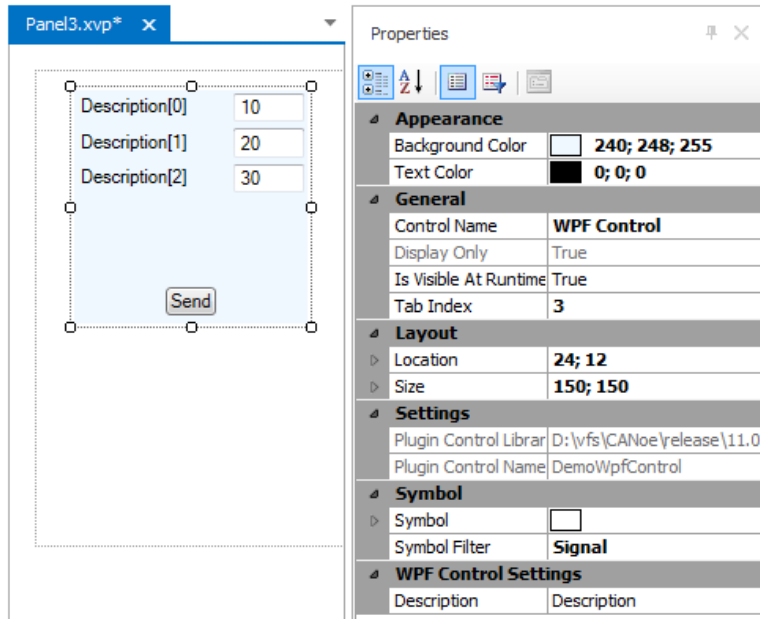
The example **DemoNavigationInfo** shows the use of a user control which can receive and send a byte array, which is interpreted as a data structure. The control supports the default properties **BackColor** and **ForeColor**.



See the source file DemoNavigationInfo.cs and DemoNavigationInfoInternal.cs for implementation details. Take care, that the implementation of the methods `DemoNavigationInfoInternal.OnRxValue()` and `DemoNavigationInfoInternal.Send()` must exactly match the underlying data structure.

4.6 Example of Implementation of a Plug-in Control Using a WPF Control

The example **DemoWpfControl** shows the use of a user control which can receive and send an integer array and which is developed as a WPF control. The control supports the default properties **BackColor** and **ForeColor** and one specific property **Description**. At design time three array elements are shown, at runtime all array elements are shown as a combination of the value of the **Description** property + array index combined with a text box, which shows the value of the integer array element. The value can be edited, the whole array is sent, when the **[Send]** button is clicked,



See the source file DemoWpfControl.xaml, DemoWpfControl.xaml.cs, and DataRow.cs for implementation details.

5.0 Trademarks

All mentioned names are either registered or unregistered trademarks of their respective owners.

6.0 Contacts

For a full list with all Vector locations and addresses worldwide, please visit <http://vector.com/contact/>.