

**TS1 Allen Bradley Ethernet/IP Controls**

A PLC aware user interface control library

Table of Contents

[1 Introduction 3](#_Toc493068812)

[1.1 State of the Library 3](#_Toc493068813)

[2 PLC Communication 4](#_Toc493068814)

[3 Address Format 5](#_Toc493068815)

[4 Generic Data Binding to PLC Tag 6](#_Toc493068816)

[5 Controls 7](#_Toc493068817)

[5.1 CheckBox, IoCheckBox, ToggleButton, RadioButton 7](#_Toc493068818)

[5.2 Button 7](#_Toc493068819)

[5.3 ActionButton 8](#_Toc493068820)

[5.4 Slider and ProgressBar 8](#_Toc493068821)

[5.5 ComboBox and ListBox 8](#_Toc493068822)

[5.6 TextBox 9](#_Toc493068823)

[5.7 TextBlock 9](#_Toc493068824)

[5.8 Frame 9](#_Toc493068825)

[5.9 MessageControl 11](#_Toc493068826)

[5.10 IOControl 13](#_Toc493068827)

[5.11 Planned controls 15](#_Toc493068828)

[6 Custom Controls 16](#_Toc493068829)

[6.1 Example; custom popup window 17](#_Toc493068830)

# Introduction

The library aims to provide a set of controls that would enable rapid building of TS1 user interfaces that have Allen Bradley PLC backend. In a sense it competes with existing first and third party touch panel interfaces and third party softwares and provides similar functionality and usage. The added value comes from it’s foundation – it is based on state of the art user interface toolkit enabling modern features such as scalable interface, support for high resolution and high dpi panels, animations, multitouch, etc.

## State of the Library

As of September 2017 the library is alpha state. It has rich set of controls that have been tested in production but the design of the library has not been finalized yet. There’s continous stream of feedback, feature requestst and improvements coming in from deployment projects. There’s a *todo.txt* file in the project folder that enumerates currently open issues and planned features but be aware that the file may be outdated and may contain features that may not see implementation.

# PLC Communication

The controls library relies on EIPNET library to communicate with PLC-s. The EIPNET implements Ethernet/IP protocol and enables communication over TCP/IP socket. The consuming application is responsible of establishing the communication to PLC and assigning it for controls to use. It is also consuming applications responsibility to monitor if connection is still active and reconnect if neccessary. The controls don’t take exclucive ownership over the communication but rather use it in a polling manner for updating UI state, the connection can still be used by consuming application for additional communication.

A tcp connection can be established using following code snippet:

var client = new AllenBradleyTagClient();

client.Connect("192.168.250.1");

The AllenBradleyTagClient is included with controls library, it has to be assigned for the controls using following snippet:

CommInterface.AddClient("MAIN", client);

This will append a connection named *MAIN* to connectionlist enabling controls to use it. The interface is designed to support more than one connections to multiple PLCs. The connection can then be used in user interface in a following manner:

<plc:CheckBox Address="MAIN:vBool"/>

This will create a checkbox on screen and bind it’s IsChecked property to Allen Bradley Ethernet/IP tag *vBool* utlizing the connection named *MAIN*. When the CheckBox is clicked it will set or reset the adresssed bool in PLC. On the other hand when the bit is changed by PLC the checkbox will automatically update to show it’s new value achieveing two way binding.

The PLC variables are updated in a polling manner but clicking on the checkbox will issue a write command immediately. NB! The library utilizes CIP multiservice packet command to read multiple variables simoultanously.

The library is also optimized for data throughput. It only updates state of controls that are actually visible on screen, so when there are multiple tabs in application then the hidden ones will not be updated. Also if multiple controls point to the same tag (e.g. multiple checkboxes and radio buttons use bits from a same word) then this tag is only read once per update cycle. This enables achieving high data update rates even on relatively large applications.

# Address Format

The most common way to specify an address to a control is by typing it as a string into address property as in:

<plc:TextBox Address="MAIN:vBool"/>

* The address will always start with the connection specifier *MAIN:*; this allows to combine data elements from multiple PLCs in in UI e.g. **PLC1**:vBool and **PLC2**:vBool.
* The variables are accessed by tag name e.g. vBool
  + The tag name can address struct element e.g. vStruct1.vInt2
  + The tag name can contain array element e.g. vArray[1,2]
  + The tag name can contain a combination of both e.g. vStructArray[1].vInt2
* The data type and size of the tag is determined automatically by the library.

# Generic Data Binding to PLC Tag

NB! This functionality is still very early in development and only supports one way binding of dependency properties of certain types (Bool, Byte, Int16, UInt16, Int32, UInt32, Float32 and String). The functionality and interface is subject to change. It is likely there will be mandatory *DataType* property in addition to Address once it’s ready.

The *AllenBradleyCipBinder* class enables data binding of any property to PLC tag e.g.

<CheckBox IsEnabled="{plc:AllenBradleyCipBinder Address=vBool}"/>

This is late addition to this library and possibly renders all other controls in this library obsolete if it pans out.

# Controls

To use the controls one should include the library in project references and add include the namespace in xaml (see below) as common with all third party controls in WPF. The library is inherited from *GuiControlLibrary.dll* thus it provides all the same functions that the referenced dll and sometimes it requires enumerations and controls defined in it. This manual does not stop describing GuiControlLibrary functions such as user management but focuses on the features introduced by AllenBradleyCipUiControls library.

xmlns:plc="clr-namespace: AllenBradleyCipUiControls;assembly= AllenBradleyCipUiControls"

The examples are written in XAML but all can also be done in code behind.

## CheckBox, IoCheckBox, ToggleButton, RadioButton

They serve similar purpose and they all bind IsChecked property to one bit in PLC tag. The binding is two-way. They also support an optional *AlternativeAddress* property that allows, when specified, to write different address than from where reading is done.

* CheckBox and ToggleButton are most similar in functionality but look different.
* IoCheckbox has has distinct style that is meant for displaying input or output values. It has IsOutput property that makes it clickable or just readonly.
* RadioButton is used for selecting distinct value

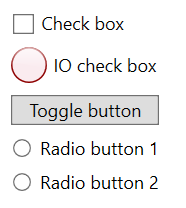
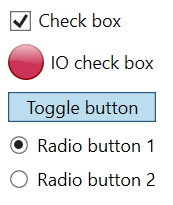
<plc:CheckBox Address="MAIN:vBool" Content="Check box" Margin="4"/>

<plc:IoCheckBox Address="MAIN:vBool" IsOutput="True" Content="IO check box" Margin="4"/>

<plc:ToggleButton Address="MAIN:vBool" Content="Toggle button" Margin="4"/>

<plc:RadioButton Address="MAIN:DB1.vBool " Content="Radio button 1" Margin="4" GroupName="1"/>

<plc:RadioButton Address="MAIN:DB1.vBool " Content="Radio button 2" Margin="4" GroupName="1"/>

## Button

Button is a command control; it utilizes one way binding to a single bit. It has supports *Address* and *ActionType* properties. The Address is bit address where it binds to and the action type can be one of the following:

* Momentary – the bit will be set to 1 when button is pressed and reset to 0 when released
* Alternate – the bit value will be inverted when button is clicked
* Set – the bit value will be set to 1 when clicked
* Reset – the bit value will be set to 0 when clicked

The button can be combined with a checkbox (or many) when it is neccessary display value:

<plc:Button Address="MAIN:vBool">

<StackPanel Orientation="Horizontal" IsHitTestVisible="False">

<plc:IoCheckBox Address="MAIN:vInput1"/>

<plc:IoCheckBox Address="MAIN:vInput2"/>

</StackPanel>

</plc:Button>



## ActionButton

The action button control combines functionalities of Button and ToggleButton. It inherits behaviour from Button and supports *CommandAddress* and *ActionType* enabling it but in addition it can display state like ToggleButton and this can be linked through *ToggleAddress* property. Like Button and ToggleButton it can be coupled with IoCheckBoxes to indicate sensor state result.

## Slider and ProgressBar

The ProgressBar allows to view and Slider also edit a value in PLC. The controls support an *Address* property that binds to the Value of control to the tag address on PLC. The type of the data on PLC can be any numeric type e.g. sint, int, word, lint, real, double, etc.

It is left up to application to specify correct value limits e.g. 16 bit unsigned number may have values between 0 and 65535.

Slider supports an optional *AlternativeAddress* property that allows, when specified, to write different tag than from where reading is done.

<plc:Slider Address="MAIN:mybyte" Minimum="0" Maximum="7"/>

<plc:ProgressBar Address="MAIN:myword" Minimum="0" Maximum="65535"/>



## ComboBox and ListBox

The ComboBox and ListBox allow selection between items. The controls support Address property that binds selected index to signed 16 bit integer on PLC side. They also support an optional *AlternativeAddress* property that allows, when specified, to write different address than from where reading is done. If none of the items are selected then the index is -1. The items themselves have to be specified on application side through XAML or by loading from files.

<plc:ComboBox Address="MAIN:mybyte">

<TextBlock Text="Item 0"/>

<TextBlock Text="Item 1"/>

<TextBlock Text="Item 2"/>

<TextBlock Text="Item 3"/>

<TextBlock Text="Item 4"/>

<TextBlock Text="Item 5"/>

<TextBlock Text="Item 6"/>

<TextBlock Text="Item 7"/>

</plc:ComboBox>



## TextBox

TextBox is a control that is used for displaying and editing tag values. It supports *Address* property that points to the tag. The data type and size is loaded dyamically from PLC.

The TextBox validates the value that is inserted by user and does not allow to input more data than is suitable for selected data type.

It also supports an optional *AlternativeAddress* property that allows, when specified, to write different address than from where reading is done.

<plc:TextBox Address="MAIN:MyDate"/>



## TextBlock

A lightweight readonly version of TextBox, usefult when there is need to embed PLC variables within text.

<plc:TextBlock Address="MAIN:MyString"/>

Also useful in combobox/listbox datatemplate. E.g. combobox address will define selected element index and TextBlocks will provide operator the values for each item.

<plc:ListBox Address="MAIN:RecipeIndex">

<plc:TextBlock Address="MAIN:RecipeNames[0]"/>

<plc:TextBlock Address="MAIN:RecipeNames[1]"/>

<plc:TextBlock Address="MAIN:RecipeNames[2]"/>

</plc:ListBox>

## Frame

The frame control is the panel that provides TS1 application look. Relevant for this document are two properties *Address* and *PlcUserManagement*. The advanced functionalities of the parent GuiFrame control are not covered in this document.

The *Address* property binds to control ***word*** that is used for two way binding. The bits of this control word are defined as follows:

0 - start button active/enabled PLC->UI

1 - start button blinking PLC->UI

2 - start button clicked UI->PLC

3 - stop button active/enabled PLC->UI

4 - stop button blinking PLC->UI

5 - stop button clicked UI->PLC

6 - reset button active/enabled PLC->UI

7 - reset button blinking PLC->UI

8 - reset button clicked UI->PLC

9 - pause button active/enabled PLC->UI

10- pause button blinking PLC->UI

11- pause button clicked UI->PLC

12- alarm active/enabled PLC->UI

13- service mode active/enabled PLC->UI

14- user level bit 0 PLC->UI

15- user level bit 1 PLC->UI

The *PlcUserManagement* property allows to disable the user management that is inherited from GuiFrame control and load the active user level from PLC. In that case the two last bits in control word specify the user level. It can be one of the following:

0 = Operator

1 = Maintenance

2 = Supervisor

3 = Administrator

This example code should be root element of main window of application:

<plc:Frame>

<TabItem Header="Observe">

<TabControl>

<TabItem Header="Cell">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="1\*"/>

<ColumnDefinition Width="1\*"/>

<ColumnDefinition Width="1\*"/>

</Grid.ColumnDefinitions>

<plc:MessageControl/>

</Grid>

</TabItem>

<TabItem Header="Log"/>

</TabControl>

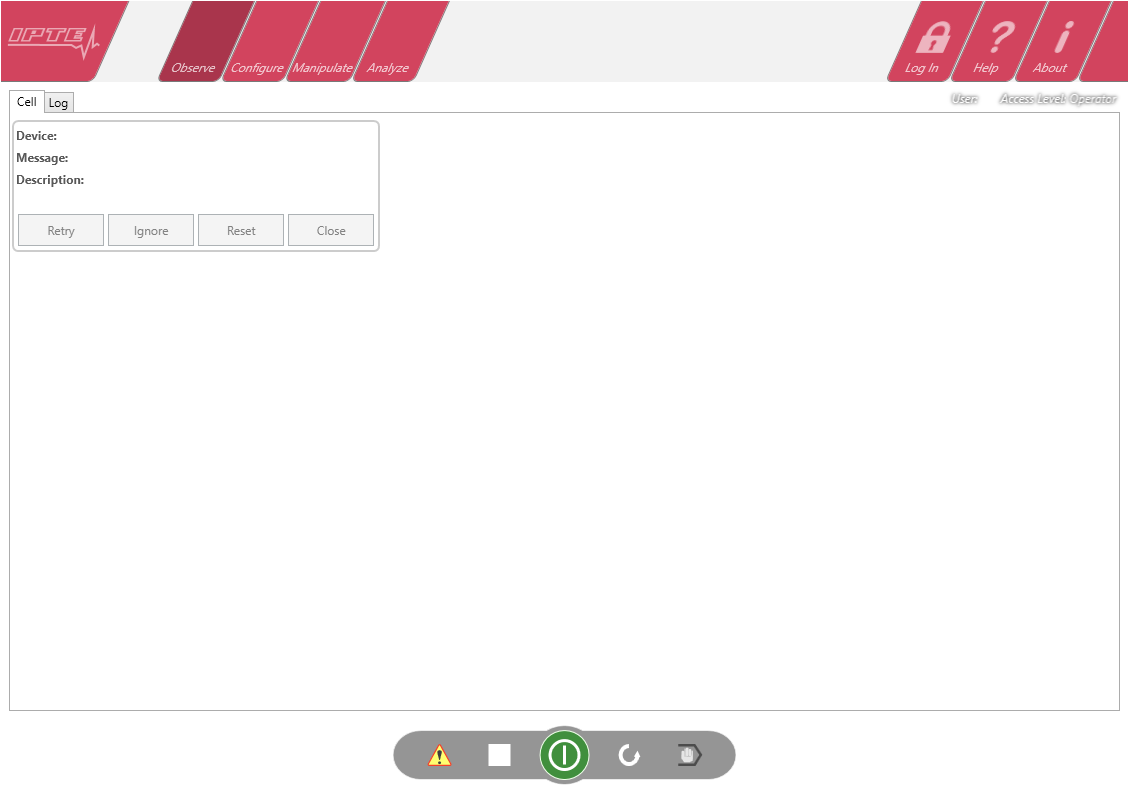
</TabItem>

<TabItem Header="Configure"/>

<TabItem Header="Manipulate"/>

<TabItem Header="Analyze"/>

</plc:Frame>



## MessageControl

This control is used for displaying error and warning messages on the welcome page of application (see illustration at Frame control).

The control is not connected to PLC directly but it supports a list of *MessageItems* that each are bindable to two addresses. The control has Messages property that has to be filled with messages; typically the messages are to be loaded from file but for demonstration purposes they are initialized in XAML.

<plc:MessageControl>

<plc:MessageControl.Messages>

<plc:MessageItem IsActiveAddress="MAIN:Errors[1]" Device="Station1"

Caption="Short Message" Description="Long Message" Severity="Error"/>

<plc:MessageItem IsActiveAddress="MAIN:Errors[2]" ButtonControlAddress="MAIN:ErrorButtons"

Device="Station2" Caption="Short Message" Description="Long Message"

Severity="Warning"/>

</plc:MessageControl.Messages>

</plc:MessageControl>

Each MessageItem has two addresses: *IsActiveAddress* and *ButtonControlAddress*, first is mandatory and second optional.

The *IsActiveAddress* specifies a two way binding to a boolean tag that will show the panel if set. If second binding is missing then there will be a Reset button enabled for errors and Close button enabled for warings. Clicking this button will clear the bit specified by IsActiveAddress.

The *ButtonControlAddress* specifies an extra word that is used for controlling the four buttons (Retry, Ignore, Reset, Close) on the MessageItem. This provides options to operator about how to proceed in a problem situation. The control address has following bits:

0 - retry button enabled PLC->UI

1 - retry button was clicked UI->PLC

2 - ignore button enabled PLC->UI

3 - ignore button was clicked UI->PLC

4 - reset button enabled PLC->UI

5 - reset button was clicked UI->PLC

6 - close button enabled PLC->UI

7 - close button was clicked UI->PLC

There are multiple use cases for this control. In a simple case the PLC has array of bits that specify active errors and warnings. The message items are loaded from file and when a bit is set the connected item will be displayed. When user clears the message the bit is reset from UI. The bit can also be cleared by PLC which will hide the message.

Another often used case is that machine is divided into autonomous stations; each station can have multiple active warnings and one active error. The warnings only have IsActiveAddress specified but for errors there’s an extra control word per station. When a station has an error it will show the panel and enable a set of buttons that user can press to solve it; the station will resume when operator has responded.

To show a message that user can not clear (no buttons, PLC has to remove) bind the ButtonControlAddress to 0 word.

NB! The library does not make any assumptations about the file format and loading the data is left to the user. Here’s a code sample for filling the list with 16 sample messages:

for (int i = 0; i < 16; i++)

{

MessageItem msg = new MessageItem();

msg.Caption = "Message #" + i;

msg.Description = "A longer description about how to resume.";

msg.Device = "Station name";

msg.Severity = GuiControlLibrary.Severity.Error;

msg.IsActiveAddress = new AllenBradleyCipAddress("MAIN","ErrorBools[1]");

msg.ButtonControlAddress = AllenBradleyCipAddress.Parse("MAIN:ErrorButtonsWord");

messages.Messages.Add(msg);

}

## IOControl

The IO control is used in TS1 for displaying status of digital inputs and outputs. The control itself can not be bind to PLC but it supports *IoGroups* that in turn supports individual *IoItems* that can be bind to tag. IoItems support an *Address* property and support an optional *AlternativeAddress* property that allows, when specified, to write different address than from where reading is done.

The lists of inputs and outputs are typically not initialized in xaml but rather loaded from file but for demo purposes here’s an example XAML:

<plc:IoControl>

<plc:IoControl.Groups>

<plc:IoGroup Caption="Group 1">

<plc:IoGroup.Items>

<plc:IoItem Address="MAIN:Inputs[0]" DisplayAddress="I1.1" Caption="Input1" IsOutput="False"/>

<plc:IoItem Address="MAIN:Inputs[1]" DisplayAddress="I1.2" Caption="Input2" IsOutput="False"/>

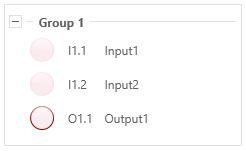
<plc:IoItem Address="MAIN:Outputs[0]" DisplayAddress="O1.1" Caption="Output1" IsOutput="True"/>

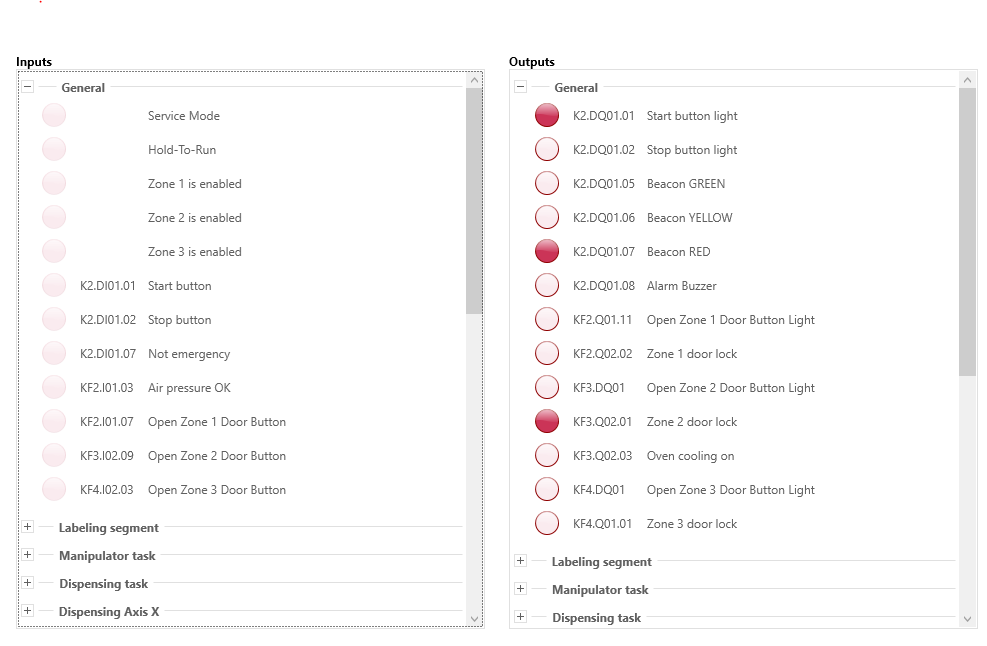
</plc:IoGroup.Items>

</plc:IoGroup>

</plc:IoControl.Groups>

</plc:IoControl>



In TS1 applications the IO page will have two separate controls; one for inputs and other for outputs but the control does not restrict mixing inputs and outputs.

An example code to initialize list of IOs:

for (int i = 0; i < 32; i++)

{

var group = new IoGroup();

group.IsExpanded = i == 0;

group.Caption = "I" + (100 + i).ToString("d3");

Outputs.Groups.Add(group);

for (byte j = 0; j < 16; j++)

{

IoItem item = new IoItem();

item.IsOutput = true;

item.Address = new AllenBradleyCipAddress("MAIN", "Inputs[" + i + "]”);

item.Caption = group.Caption + "." + j;

item.DisplayAddress = item.Address.ToString();

group.Items.Add(item);

}

}

## Planned controls

* A popup window whose visiblity can be linked to PLC tag area. The contents of the window are user defineable.

# Custom Controls

In addition to existing controls in this library it is easy to create application specific controls and bind them to PLC tags. In order to create a custom control that is updated together with the rest of UI elements in this library it must implement IAllenBradleyCipUpdatable interface and register itself in communication interface.

The interface defines following properties and methods:

* bool NeedsUpdating { get; } is used by library detect if the component needs to be updated. This mechanism allows to suppress updating of nonvisible components to save bandwidth and improve overall UI responsiveness.  
  NB! This function is called from background thread and thus may not access any dependency properties.
* IEnumerable<AllenBradleyCipAddress> GetAddresses(); if the control needs updating then this property is used by the library to fetch PLC addresses that the control is bound to. Each address may specify data area with range from single bit to multiple words.  
  NB! This function is called from background thread and thus may not access any dependency properties.
* void Update(AllenBradleyCipAddress address, byte[] data); this function is used by library to return data from PLC. The array contains the raw word values that were read from given address.  
  NB! If address specifies only a single bit then the data will still contain whole word. The bit must be extracted from it.  
  PS! This function is dispatched to application threads, it is safe to update UI properties directly from this function body.

In addition to implementing the interface the control must register itself as listener. This should be done in object constructor using following code snippet:

CommInterface.ListenAddress(this);

## Example; custom popup window

public class PopupWindow : GuiControlLibrary.GuiDialogWindow, IAllenBradleyCipUpdatable

{

//dependency property that is exposed to ui and normal field backend for threadsafe access

private AllenBradleyCipAddress[] \_threadsafeaddresses = new AllenBradleyCipAddress[1];

public static readonly DependencyProperty AddressProperty = DependencyProperty.Register(

"Address",

typeof(AllenBradleyCipAddress),

typeof(PopupWindow),

new PropertyMetadata((d, e) =>

(d as PopupWindow).\_threadsafeaddresses[0] = e.NewValue as AllenBradleyCipAddress)

);

//constructor. Will register this window as listener

public PopupWindow()

{

if (System.ComponentModel.DesignerProperties.GetIsInDesignMode(this)) return;

CommInterface.ListenAddress(this);

Topmost = true;

}

//don’t allow alt+f4 to close the window. We'll hide the window when bit is deactivated.

protected override void OnClosing(CancelEventArgs e)

{

if (IsVisible) e.Cancel = true;

base.OnClosing(e);

}

[Category("PLC")]

public AllenBradleyCipAddress Address

{

get { return (AllenBradleyCipAddress)GetValue(AddressProperty); }

set { SetValue(AddressProperty, value); }

}

//IAllenBradleyCipUpdatable member

public IEnumerable<AllenBradleyCipAddress> GetAddresses()

{

return \_threadsafeaddresses;

}

//IAllenBradleyCipUpdatable member

public void Update(AllenBradleyCipAddress address, byte[] data)

{

try

{

Visibility = new BitArray(data)[0] ? Visibility.Visible : Visibility.Collapsed;

}

catch

{

//disable updating

Address = null;

}

}

//IAllenBradleyCipUpdatable member update always, even when hidden

public bool NeedsUpdating { get { return true; } }

}