Zebra
Aurora™ Vision

Aurora Vision Studio 5.7

Human Machine Interface (HMI)

Created: 11/9/2026

Product version: 5.7.5.80526

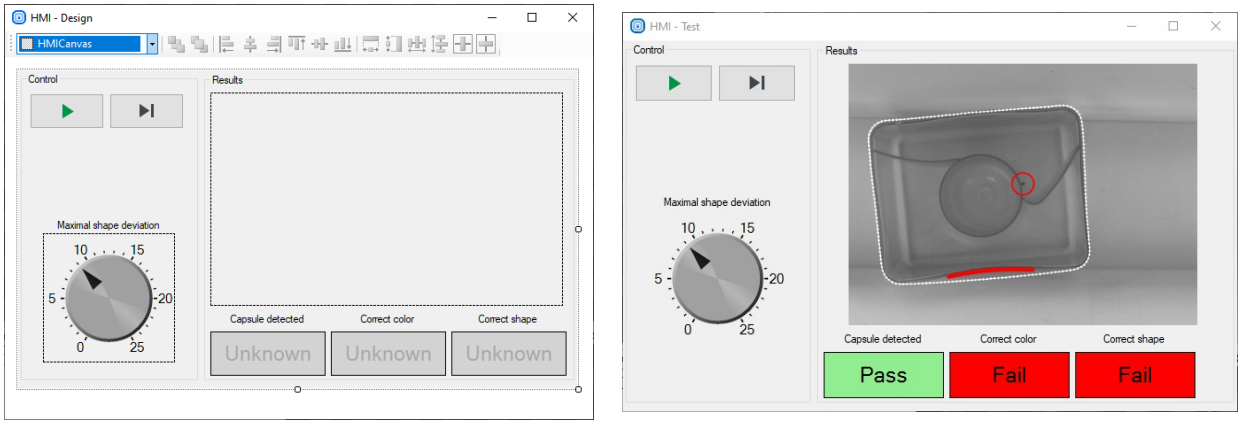
Table of content:

- Designing HMI
- Standard HMI Controls
- Handling HMI Events
- Saving a State of HMI Applications
- Protecting HMI with a Password
- Creating User Controls
- List of HMI Controls

Designing HMI

Introduction

Although image analysis algorithms are the central part of any machine vision application, a human-machine interface (HMI, end user's graphical environment) is usually also very important. Aurora Vision Studio, as a complete software environment, comes with an integrated graphical designer, which makes it possible to create end user's graphical interfaces in a quick and easy way.



A very simple HMI example: at design time (left) and at run time (right).

The building blocks of a user interface are called controls. These are graphical components such as buttons, check-boxes, image previews or numeric indicators. The user can design a layout of an HMI in an arbitrary way by selecting controls from the HMI Controls window and by placing them on the HMI Canvas. The Properties window will be used to customize such elements as color, font face or displayed text. There are also some controls, called containers, which can contain other controls in a hierarchical way. For example, a TabControl can have several tabs, each of which can contain a different set of other controls. It can be used to build highly sophisticated interfaces.

HMI controls are connected with filters in the standard data-flow way: through input and output ports. There are three possible ways of connecting controls with filters:

- From a filter's output to a control's input – e.g. for displaying an image or a text result.
- From a control's output to a filter's input, as data sources – e.g. for setting various parameters in a program with track-bars or check-boxes.
- From a control's output to a filter's input, as events – e.g. for signaling that a button has been clicked.

There are also a few properties in HMI controls that can be connected directly between two different control. See Label.**AutoValueSource** and the EnableManager control.

Overview of Capabilities

The HMI Designer in Aurora Vision Studio is designed for very easy creation of custom user interfaces which resemble physical control panels (a.k.a. front panels). With such an interface the end user will be able to **control execution process, set inspection parameters and observe visualization results**. There are also more advanced features for protecting the HMI with passwords, saving its state to a file, creating multi-screen interfaces or even for allowing the end user to create object models or measurement primitives.

There is, however, a level of complexity, at which other options of creating end user's graphical interfaces may become suitable. These are:

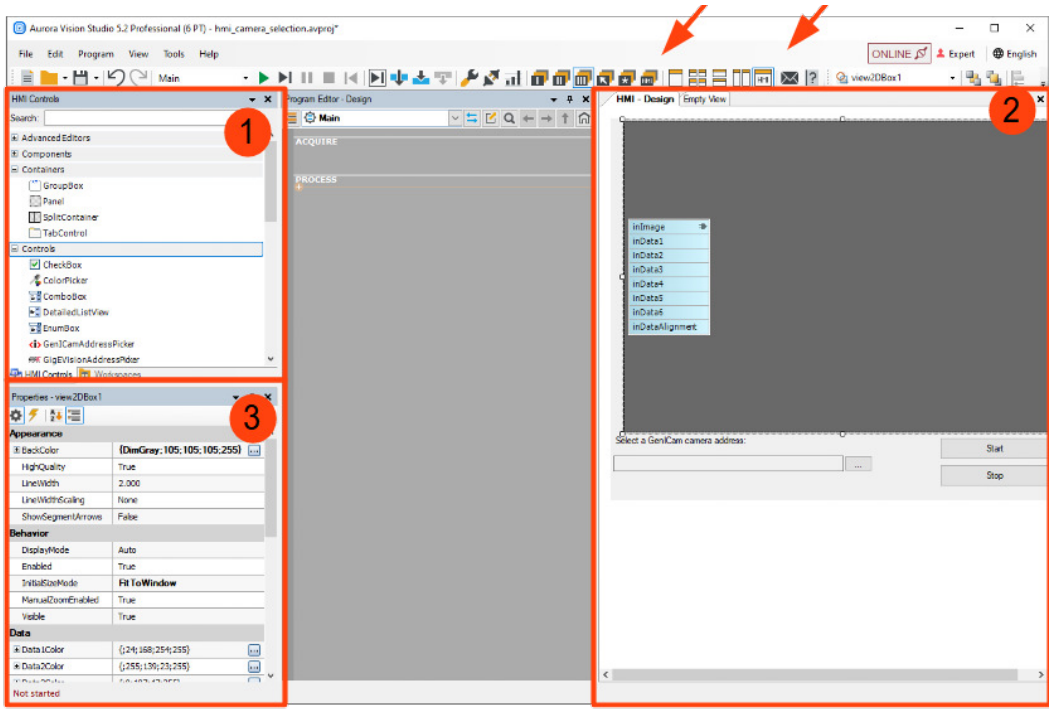
- Creating custom HMI controls in the C# programming language – especially for dynamic or highly interactive GUI elements, e.g. charts.
- Using .NET Macrofilter Interfaces and then creating entire HMI in the C# programming language.
- Using C++ Code Generator and then creating entire HMI in the C++ programming language.

Note: Aurora Vision Executor with HMI support is only available on Microsoft Windows operating system. For Linux we recommend generating C++ code and creating the user interface with Qt library.

Adding HMI to a Project

To open HMI Designer choose *View » HMI Designer* command in the Main Menu or click the *HMI* button on the Toolbar:

This adds "HMI – Design" special view (which can be undocked), and a new window, HMI Controls. The Properties window shows parameters of the selected control – here, of the main HMI Canvas.



Elements of the HMI Designer: (1) HMI Controls catalog, (2) HMI Editor, (3) Control's Properties + its context-sensitive help.

HMI Canvas is an initial element in the HMI Design view. It represents the entire window of the created application. At the beginning it is empty, but controls will be placed on it throughout HMI construction process.

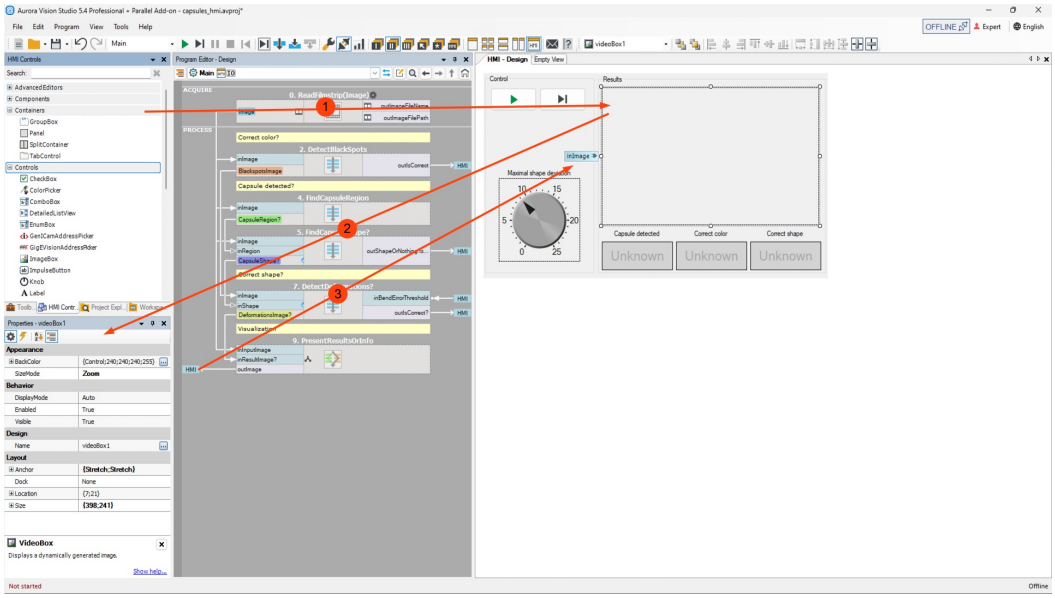
Removing HMI

If at any point you decide that the created HMI is not needed anymore, it can be removed with a the *Edit » Remove HMI* command available in the Main Menu.

Basic Workflow

The HMI design process consists in repeating the following three steps:

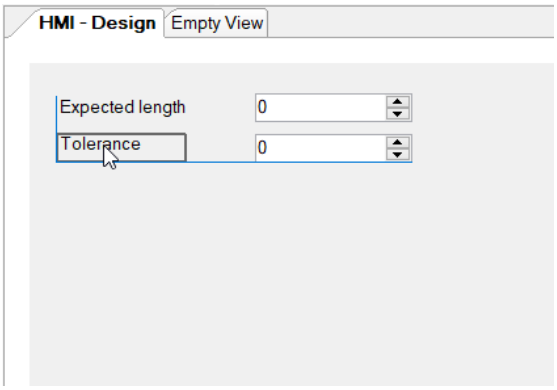
1. Drag & drop a control from HMI Controls to HMI Editor. Set its location and size.
2. Set properties of the selected control.
3. Drag & drop a connection between the control's input or output with an appropriate filter port in Program Editor.



The three steps of HMI design.

The stages of HMI design are explained below:

1. The controls can be dragged onto the HMI Canvas, just as the filters are dragged to the Program Editor panel. To align widgets relative to each other, snaplines can be used. The layout can be organized with the help of containers.



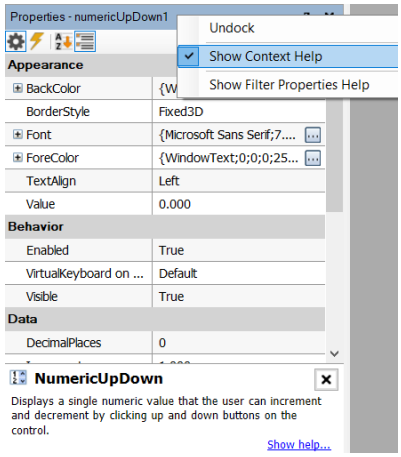
Snaplines assist in setting layout of the controls.

You can also use **Anchor** and **Dock** properties to let the control adapt its dimensions to the size of the application window:

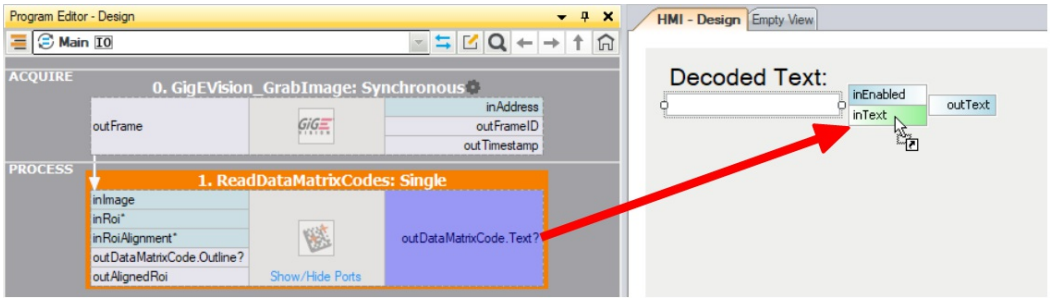
- **Anchor** set to *Right* or *Bottom* keeps the control in a fixed distance from the right or bottom edge of its parent window.
- **Anchor** set to *Stretch* makes the control resize with the parent window, keeping fixed distances from both edges.
- **Dock** property binds the control to one of the edges of its parent window, or to the entire free space, in such a way, that multiple controls with this property set automatically share the available window space. Please note, that in this case the order of putting the controls on the canvas does matter.

2. Properties of controls include graphical features, i.e. location, size, colors, fonts, borders and backgrounds. There are also properties affecting the control's behavior, like enabling or disabling, the initial contents (text), visibility, automatic size adjustment etc. Finally, specific controls have their own logical properties, usually the *value* and other configuration properties (minimal and maximal values, steps, defaults).

   Note: Make sure that Context Help is visible to quickly access a short description of each property:

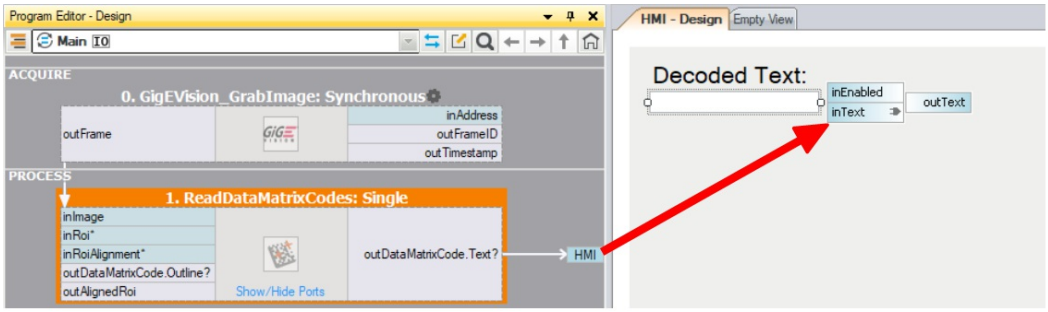


3. Connecting HMI controls is very similar to connecting filter inputs and outputs. First, the control we want to connect has to be selected. Labels of the control's inputs and outputs will appear next to it. Then drag an output of a control onto a compatible filter input, or an output of a filter onto a compatible input of a control.



Connecting a filter with a control.

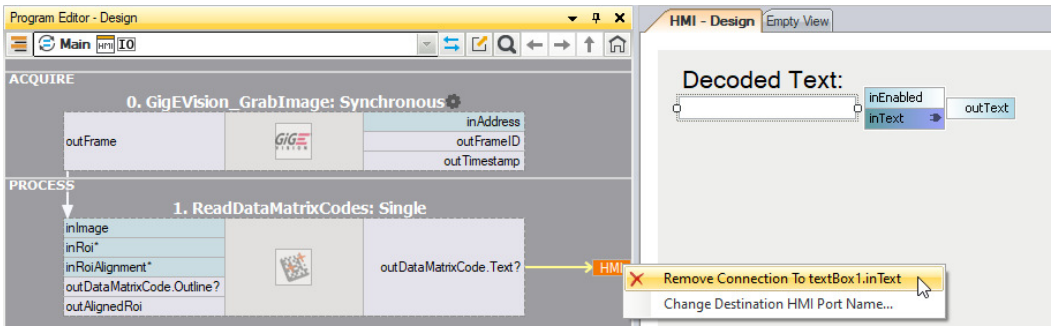
Aurora Vision Studio will assist this process, by disallowing connections between incompatible ports. After the connection is made, the input or output label will include a little plug icon, and the connected filter will have an "HMI" label next to its input or output.



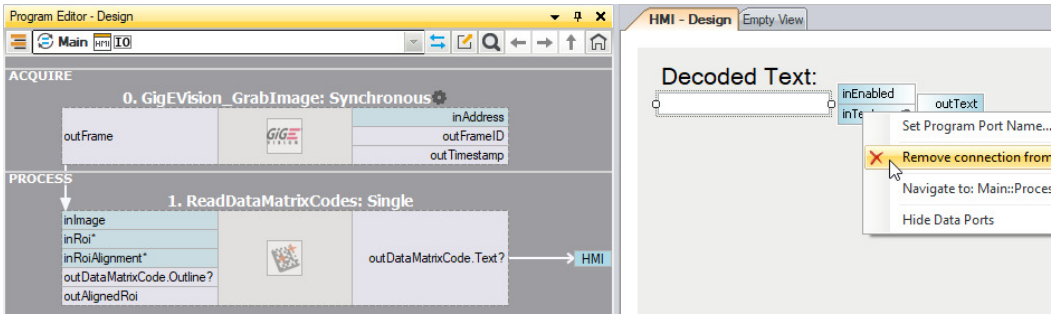
Connection between a filter and a control was established.

Note: If the HMI Editor is undocked, please make sure that it does not overlap with Program Editor before creating a connection.

If we need to remove a connection between a filter and an HMI control, we can right-click the label of either the input or the output. The context menu will include an option to remove the connection.



Disconnecting a control from a filter in Program Editor.



Disconnecting a filter from a control in HMI Designer.

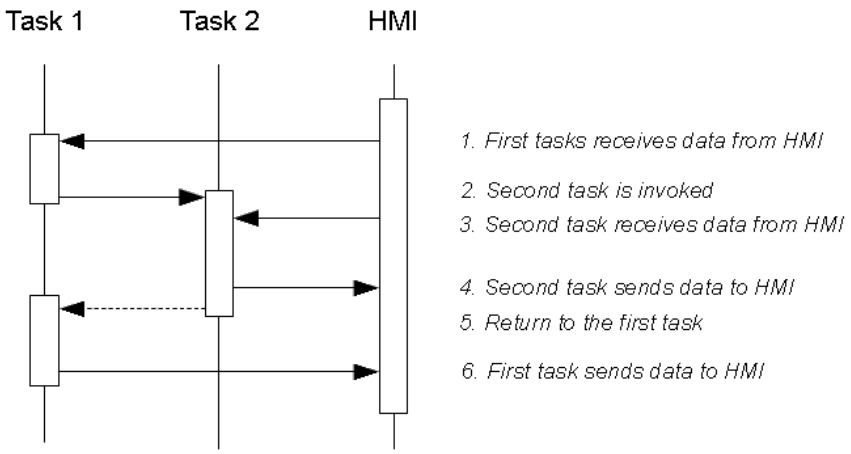
HMI Interactions with the Program

IMPORTANT: During program execution HMI controls exchange data with filters in precisely defined moments:

- Data is sent from HMI controls to filters at the beginning of each iteration of a macrofilter that has any connections with the controls.
- Data is sent in the opposite direction – from filters to HMI – at the end of each iteration.

This has some important implications:

- You CAN connect an output of an HMI control to several filters in a macrofilter and they will be guaranteed to receive exactly the same value in the same iteration.
- You cannot send a value to a control and expect this value to be immediately available when the next filter tries to read it. It will be available only in the next iteration. The next filter in the current iteration will still read the old value.
- If you have a task nested in another task the order of data transfers may become surprising – e.g. results of the outer tasks are not displayed until the inner task terminates.

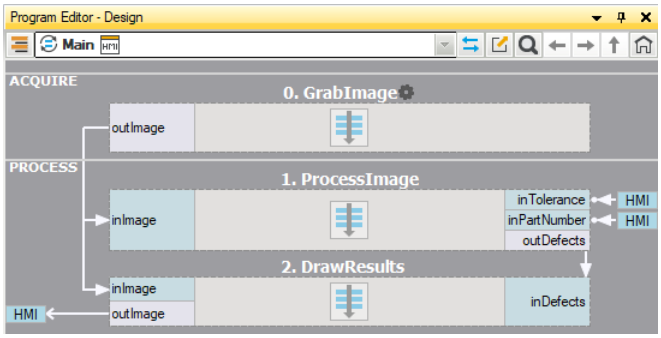


Sample communication structure when one task is nested in another. Please note, that Task 2 will often operate in a loop making it very long before Task 1 sends data to the HMI.

Recommendations

To handle communication between filters and controls efficiently, we recommend the following rules:

- Avoid nested loops (tasks) with HMI communication. Instead:
  1. Either create a single Finite State Machine per application. For example, instead of waiting in a nested loop for the user to click a button, create an application state named "Waiting" with a transition to another state on the button click event.
  2. Or create a "master" task without any HMI communication, with two or more nested tasks that do have HMI communication and represent different phases of the application. For example, the master may be the "Main" macrofilter with a loop and two sub-tasks named "WaitForStart" and "RunInspection".
- If one program iteration takes long time, but communication has to be performed at very specific moments, use a nested macrofilter just for sending or receiving data to or from HMI. This nested macrofilter will enforce communication moments. Use CopyObject filters when no direct connection from the Macrofilter Inputs block to HMI controls are possible. See also: Sending Values to HMI from Multiple Places below.
- Avoid complex dependencies between HMI controls (i.e. setting properties of one control using values set by the user in another control). Doing so would require sending values around through filters or registers in the program. An exception from this rule is the "Auto Value" feature of the Label control, which allows to link a label content with values set in such controls as TrackBar or Knob.
- Keep the program structure simple and clear. For most applications there should be a single loop consisting of three parts: (1) image acquisition, (2) image processing, (3) preparing data for the HMI. We recommend that all connections with HMI are created on the top level of the main loop, as can be seen in the picture below:



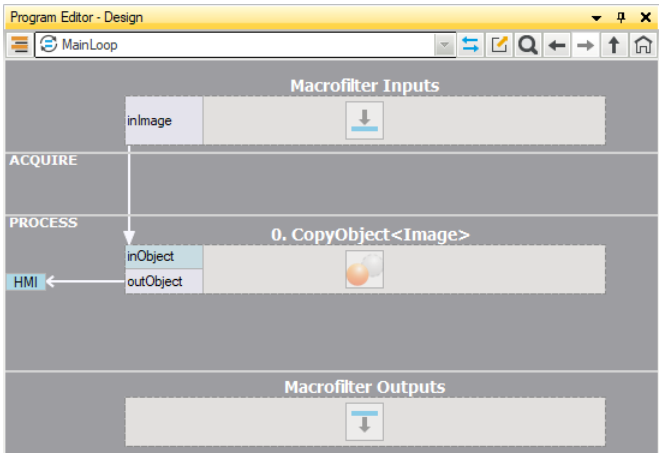
A typical simple structure of a program with HMI connections.

Sending NIL Values to HMI

It is also possible to send conditional values from filters to HMI controls. When a *Nil* value is sent then the control's property remains unchanged. This may be useful when both the program and the user modify the same property (e.g. the position of a track-bar). In such cases, data sent to the control from the program should be conditional, so that it does not override the value set by the user.

Sending Values to HMI from Multiple Places

Each input of an HMI control can only have one source of data in the program. However, sometimes it is required that values to that control are sent from several different places. For example, a VideoBox display can be connected with the output of a GigEVision_GrabImage filter, but we might also want to send an empty image to that control when an error occurs. This can be achieved by creating a macrofilter with one or several inputs and with the same number of CopyObject filters connected with the HMI (direct connections from macrofilter inputs to the HMI are not permitted). That macrofilter can then be then used in many times in the program, effectively allowing for explicit communication with the HMI from multiple places.



Sample macrofilter for explicit communication with HMI from multiple places of a program.

Preparing Data for Display in HMI

There is a significant difference between how inspection results are visualized in the standard data preview windows (development environment) and in HMI (runtime environment). On data preview windows multiple data items are just dragged & dropped, and properties such as color or line thickness are selected automatically. This is highly suitable for quickly analyzing data in application development process. In HMI (runtime environment), however, we want to have full control over visualization style of each single element. Thus, before sending data to HMI controls such as VideoBox (for images) or Label (for text), the user has to prepare visualization by using appropriate filters, such as:

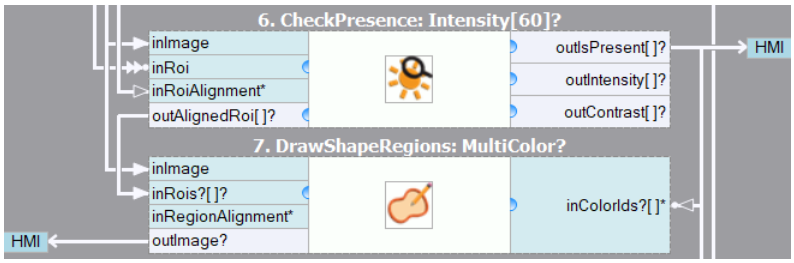
- Image Drawing – for preparing images overlaid with geometrical primitives, text labels etc.
- CropImage, ResizeImage, MirrorImage etc. – for transforming an image in an arbitrary way before display.

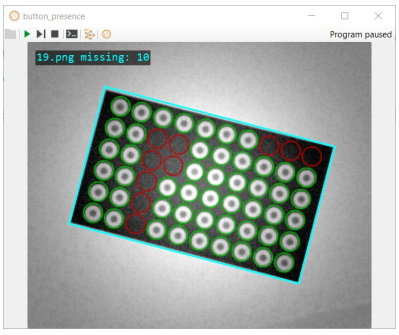
  Note: ZoomingVideoBox control has a built-it support for panning and zooming the displayed images, so no additional filters are required.
- FormatRealToString, FormatIntegerToString, FormatPoint2DToString etc. – for formatting numerical values into strings with properties such as the number of decimal digits or the fractional digits separator.
- ConcatenateStrings, StringToUpperCase, Substring etc. – for displaying textual values in a specific way.

Example 1:

In a button inspection application we detect buttons and check if each button is present or not. We want to display green or red circles over the buttons, indicating the inspection results.

The solution is to use DrawShapeRegions_MultiColor filters with its **inColorIds** input connected from the **outIsPresent** output of the CheckPresence_Intensity filter:





If more overlays are required on a single image, then we use multiple Image Drawing filters connected sequentially.

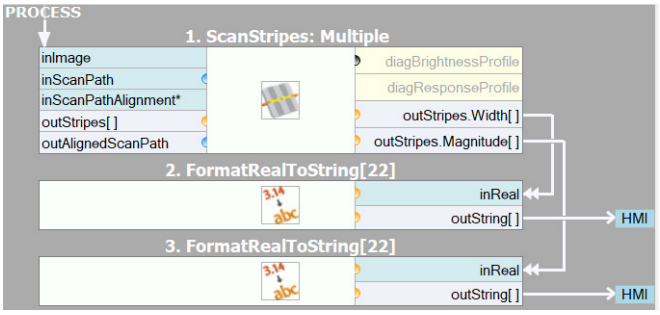
Note: If many drawing filters are connected in a sequence, then performance may suffer as Aurora Vision Studio keeps a copy of an image for each of the filters. A recommended work-around is to encapsulate the drawing routine in a very simple user filter – because on the C++ level (with AVL Lite library) it is

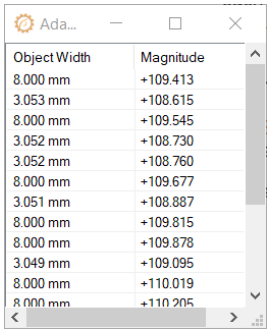
possible to do drawing "in place", i.e. without creating a new copy of the image. Please refer to the "User Filter Drawing Results" example in Aurora Vision Studio. Make sure you build it in the "Release" configuration and for the appropriate platform (Win32 / x64).

Example 2a:

The ScanMultipleStripes filter has an output named **outStripes.Width** and **outStripes.Magnitude**, which are of RealArray type. We want to display these numbers in the HMI with a DetailedListView control, which has several inputs of StringArray type.

The solution is to convert each Real number into a String using FormatRealToString filter. On the pictures below demonstrated is the visualization of two columns of numbers with different parameters used in the formatting filters:





Example 2b:

If a similar array of numbers is to be displayed in a Label control, which has a single String on the input (not an array), then some more preprocessing is still required – in the FormatRealToString filter we should add a new-line character to the **inSuffix** input and then use ConcatenateStrings_OfArray to join all individual strings into a single one.

See Also

- Standard HMI Controls
- Handling HMI Events
- Saving State of HMI Controls
- Protecting HMI with Password

Standard HMI Controls

Contents:

1. Introduction
2. HMI Canvas
3. Controls for Setting Layout of the Window
4. Controls for Displaying Images
5. Controls for Setting Parameters
6. Controls for Displaying Inspection Results
7. AnalogIndicator
8. Event Triggering Controls
9. The TabControl Control
10. The MultiPanelControl Control
11. Program Control Buttons
12. File and Directory Picking
13. Shape Editors
14. ProfileBox
15. View3DBox
16. Activity Indicator
17. TextSegmentationEditor and OcrModelEditor
18. KeyboardListener
19. VirtualKeyboard
20. ToolTip
21. ColorPicker
22. BoolAggregator
23. EnabledManager
24. ChangeLogger
25. EdgeModelEditor
26. GrayModelEditor
27. GenICamAddressPicker
28. GigEVisionAddressPicker
29. MatrixEditor
30. GrammarRulesPatternEditor
31. Deep Learning

Introduction

There are several groups of UI components in the HMI Controls catalog:

- Components – non-visual elements that extend the HMI window functionality.
- Containers – used for organizing the layout with panels, groups, splitters and tab controls.
- Controls – standard controls for setting parameters and controlling the application state.
- File System – used for selecting files and directories.
- Indicators – for displaying inspection results or statuses.
- Logic and Automation – for binding HMI properties together, including basic AND/OR conditions.
- Multiple Pages – for creating multi-screen applications.
- Password Protection – for restricting access to certain parts of an HMI to authorized personnel.
- Shape Editors – allow to define object models or measurement primitives.
- Shape Array Editors – allow to define an array of object models or measurement primitives.
- State Management – for loading and saving state of the controls to a file.
- Video Box – for high-performance image display.

In a basic machine vision application you will need one VideoBox control for displaying an image, possibly with some graphical overlays, and a few of TrackBars, CheckBoxes, TextBoxes etc., for setting parameters. You will also often use Panels, GroupBoxes, Labels and ImageBoxes to organize and decorate the window space.

Common Properties

Many properties are consistent across different standard controls. Here is a summary of the most important ones:

- **AutoSize** – specifies whether a control will automatically resize to fit its contents.
- **BackColor** – the background color of the component.

Three sample labels with different background colors.

- **BackgroundImage** – the background image used for the control.
- **BorderStyle** – controls the appearance of the control's border.

Three sample labels with different border styles.

- **Enabled** – can be used to make the control non-editable by the end user.

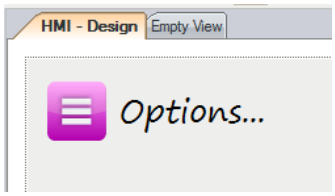
Sample buttons, first enabled, second disabled.

- **Font** – defines the size and style of the font used to display text in the control.

Three sample labels with different fonts.

- **ForeColor** – the foreground color of the control, used for displaying text.
- **Text** – the string displayed in the control.
- **Visible** – can be used to make the control invisible (hide it).

With the available properties, it is possible to create an application with virtually any look and feel. For example, to use a custom design for a button, we can use a bitmap while setting **FlatAppearance.BorderSize** = *0* and **FlatStyle** = *Flat*:



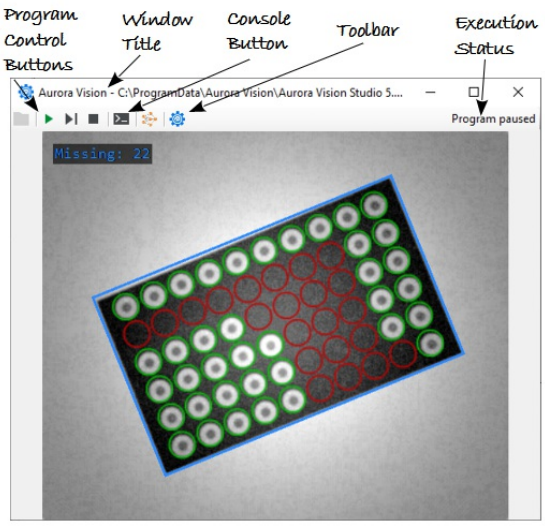
Sample custom style for a button and a label.

Note: Most properties of HMI controls can be also set dynamically during program execution, although only some of them are visible as ports when the control is selected. To expose additional ports, use the "Edit Port Visibility..." command from the control's context menu.

HMI Canvas

The HMICanvas control represents the entire window of the created application. Other controls are placed on it. Properties of HMICanvas define some global settings of the application, which are effective when it is run with Aurora Vision Executor (runtime environment):

- **ConsoleEnabled** – specifies whether diagnostic console can be shown in the runtime application.
- **PreventWindowClose** – enables preventing the user from closing the Aurora Vision Executor window when the program is running.
- **ProgramControllerVisible** – specifies whether Start / Pause / Stop buttons will be visible on the toolbar of Aurora Vision Executor.
- **ToolbarVisible** – specifies whether the toolbar of Aurora Vision Executor will be visible.
- **WindowIcon** – the icon of the Aurora Vision Executor window.
- **WindowMode** – specifies whether the application should be run in a fixed-size window, in a variable-size window or full-screen.
- **WindowTitle** – specifies the text displayed as the title of the runtime application.



Elements of the runtime application affected with properties of HMICanvas.

Remarks:

- If you are having trouble selecting the HMI Canvas because it is completely filled with other controls, right click on any control and choose the "Select: HMICanvas" command.
- Some parameters of HMI Canvas are automatically inherited by the contained controls. For example, if one changes the Font property, other controls will automatically have it set to the same value.
- Full-screen mode is especially useful, when it is required that the end user does not have access to elements of the underlying operating system. On Windows systems it is also possible to set Aurora Vision Executor as the "system shell", thus removing Desktop, Menu Start etc. completely.

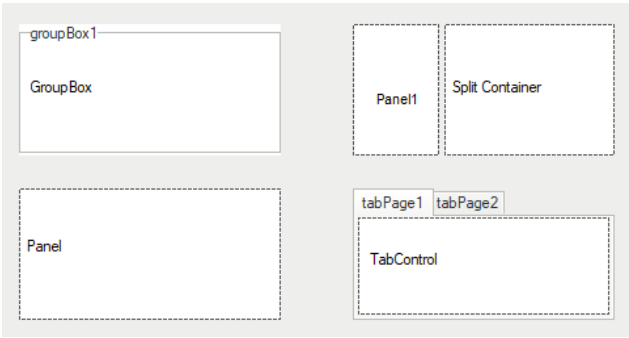
There is also a setting regarding scaling of HMI when opened on a computer with different DPI scale:

- **ScaleControlsOnDifferentDPI** – specifies whether scaling of controls is performed on project loading or not. Note that some elements (like font sizes) are always scaled, regardless of this option.

Controls for Setting Layout of the Window

Although it is possible to put all individual controls directly onto an HMI Canvas, it is recommended that related items are grouped together for both more convenient editing as well as for better end user's experience. Available controls in the "Containers" section are:

- GroupBox – can be used to group several controls within an enclosing frame with label.
- Panel – can be used to group several controls without additional graphical elements.
- SplitContainer – can be used to create two panels with a movable divider between them.
- TabControl – can be used to create multiple tabs in the user interface.



BackColor was changed from default to white in all containers.

To use any container drag it from the HMI Controls and drop it on the HMICanvas. Then put all the controls you want to group together onto it. Another method to do that is to click a desired container in the "Containers" section and then select the area on the HMI panel.

Other controls commonly used for making HMI look nice are: ImageBox (for displaying static images such as company logos) and Label (for any textual items).

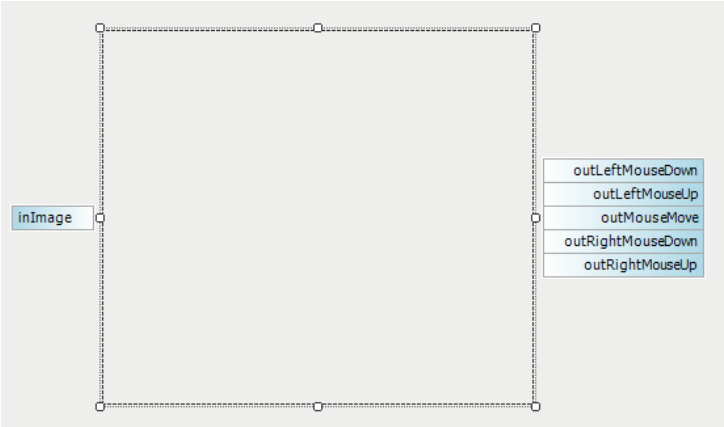
However, when you would like to create more sophisticated HMI Panel with multiple screens, go to The Multi Panel Control

See also: The TabControl Control.

Controls for Displaying Images

The "Video Box" section of the HMI Controls window contains several controls for high-performance display of image streams. These are:

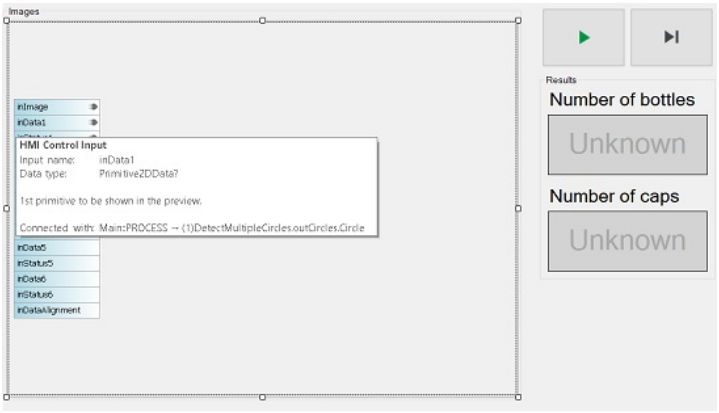
- VideoBox – the basic variant of the image display control.
- SelectingVideoBox – also allows for selecting a box region by the end user.
- ZoomingVideoBox – also allows for zooming and panning the image by the end user.
- FloatingVideoWindow – a component that allows for opening an additional window for displaying images.



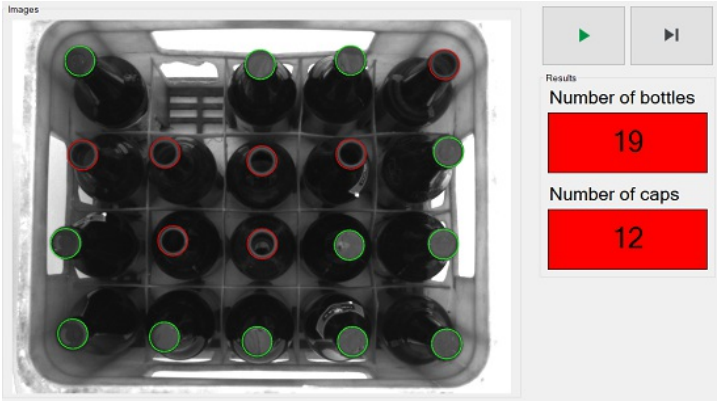
A VideoBox.

To display some additional information over images in a VideoBox it is necessary to modify the images using drawing filters before passing them to the VideoBox. This gives the user full control over the overlay design, but may expand the program. To keep the program simple, while still being able to display some overlay information, it is possible to use View2DBox controls, which can be found in the "Indicators" section. They accept images and 2D primitives produced by various filters as input, however, the possibility of changing their style is limited:

- View2DBox – allows for overlaying 2D primitives (Point 2D, Circle 2D, Rectangle 2D etc.) over an image.
- View2DBox_PassFail – the color of the overlaying 2D primitives may change depending on the Status inputs.



HmiView2DBox_PassFail used in the bottle crate example.



Executed bottle crate example program.

ImageBox from "Controls" section can be used to improve look of HMI for displaying static images such as company logos.

Common properties:

- **DisplayMode** – switches between GDI and DirectX implementations. The control initially attempts to launch in DirectX mode, but if it fails or if the graphics card does not meet certain technical requirements, it switches to GDI mode. Switching the Display Mode to DirectX bypasses the check for these technical requirements.

- **SizeMode** – specifies how the image should be fitted to the available window space.

Notes:

1. DO NOT use ImageBox control for displaying inspection results. Due to performance reasons, it should only be used for static images.
2. In case that View2DBox receives the *Nil* value as an input, the preview does not refresh itself. As a result the old image is being kept in the preview instead of an empty one.
3. The VideoBox controls provide mouse events, but ZoomingVideoBox and SelectingVideoBox make most of these events inactive when manual image manipulation is enabled. Only the Click event is always available.
4. In case of issues with image display during a remote connection, it is recommended to set the **DisplayMode** parameter to **GDI**.

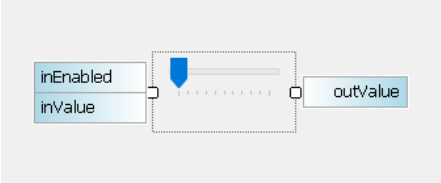
Controls for Setting Parameters

When some parameters of the application are to be set by the end user, the following controls can be used:

- TrackBar, Knob, NumericUpDown, RangeTrackBar – for setting numerical values. You can set **Minimum** and **Maximum** parameters and then the value entered by the end user will be available on the **outValue** output.
- CheckBox, ToggleButton, OnOffButton, RadioButton – for turning something on or off. The **outValue** or **outChecked** output will provide the state of the setting.
- ComboBox, EnumBox – for choosing one of several options.
- TextBox – for entering textual data. The entered text will be available on the **outText** output.

Example: Using TrackBar

TrackBar by default has two connectable inputs and one connectable output. The **inValue** input and the **outValue** output represent the same property – the value indicated by the current state of the control. The second input, **inEnabled**, is a boolean value and controls whether the TrackBar is enabled or disabled in the user interface.



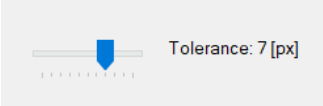
A track-bar.

Binding with a Label

Very often it is required that the current value of a TrackBar or a similar control is also visible in the HMI. This can be done with an additional Label control placed near the TrackBar, with the **AutoValueSource** property referring to the selected TrackBar.

Label showing an exact value of the TrackBar.

It is also possible to specify the format of the displayed text. The picture below shows a connected Label with the **AutoValueFormat** property set to "Tolerance: % [px]".

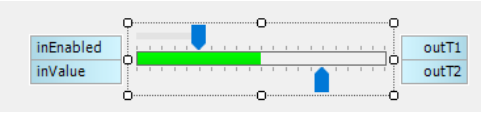


TrackBar with a connected Label using an additional text formatting.

Example: Using RangeTrackBar

RangeTrackBar by default has two connectable inputs and two connectable outputs. The **inValue** input represents the value indicated by the current state of the control and is displayed as a colored bar. The second input, **inEnabled**, is a boolean value and controls whether the RangeTrackBar is enabled or disabled in the user interface.

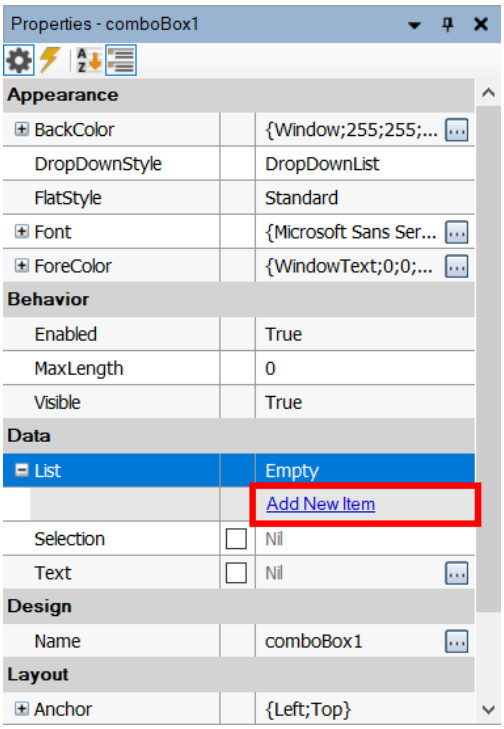
The outputs **outT1** and **outT2** represent the values of the lower and upper thresholds, respectively. These thresholds are visually indicated as colored thumbs on the axis. The control enforces that **outT1** cannot be greater than **outT2**, ensuring logical consistency while adjusting the range.



A range track-bar.

Example: Using ComboBox

ComboBox by default has one connectable input and two connectable outputs. To enable more ports e.g. a list of items to be displayed in ComboBox, right-click on it and select the *Edit Ports Visibility...*. You can connect an array of String values describing the names of items to **inList** input or simply add the items in the Properties window.



*Click **Add New Item** to increase list.*

By specifying the index in **Selection** parameter, the user defines initial selection which is chosen when the program is executed. Without this selection, the output will be Nil.

The **outSelection** output is of an Integer data type and returns the index of the selected item. The **outText** output is of a String data type and returns text associated with the selection.

Example: Using EnumBox

EnumBox is very similar to ComboBox. The main difference between them is that the latter allows creating your own list of items while EnumBox can display one of the Enum types available in Aurora Vision Studio e.g. ColorPalette, OCRModelType or RotationDirection.

Change Events

Controls that output a value adjustable by the user often also provide outputs informing about the moments when this value has been changed – so-called events. For example, the ComboBox control has an output named **outSelection** for indicating the index of the currently selected list item, but it also has **outSelectionChanged** output which is set to *True* for exactly one iteration when the selection has been changed.

Please note that most of the event triggering outputs e.g. outValueChanged, outSelectionChanged etc. are hidden by default and can be shown with the "Edit Port Visibility..." command in the context menu of the control.

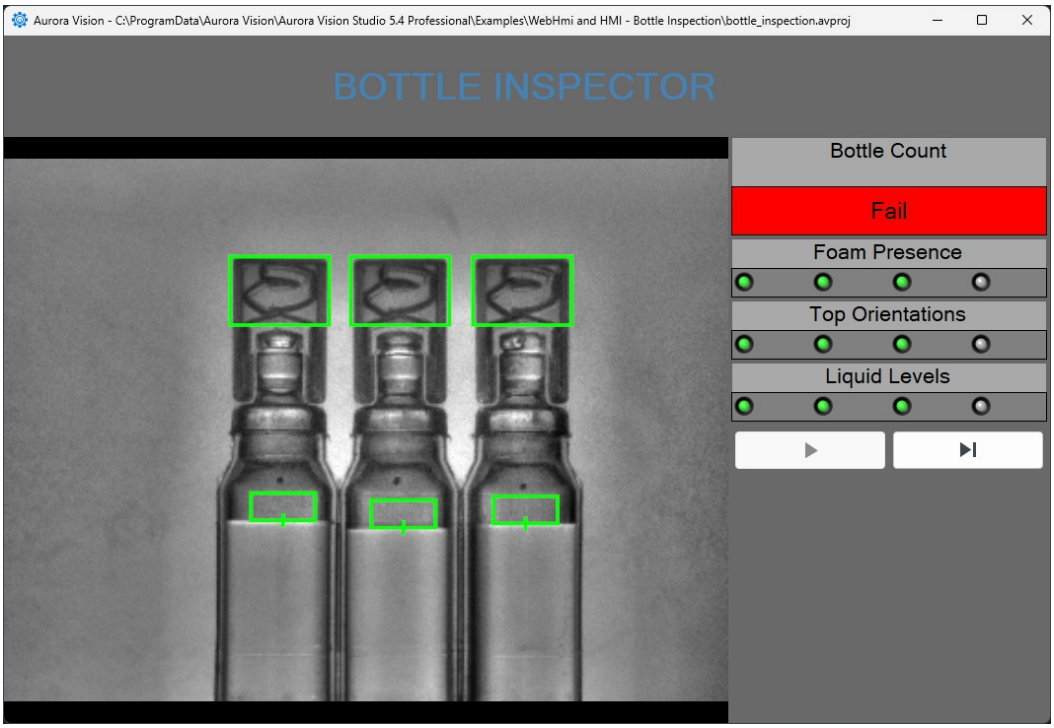
See also: Handling HMI Events.

Controls for Displaying Inspection Results

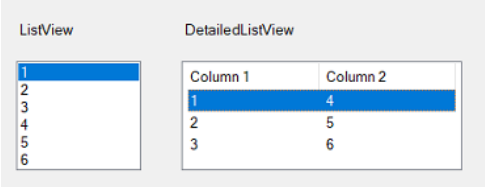
Non-graphical inspection results are most often presented with the following controls:

- Label – for displaying simple text results.
- ListView, DetailedListView – for displaying arrays or tables.
- PassFailIndicator – for displaying text on a colorful background depending on the inspection status.

- BoolIndicatorBoard – for displaying inspection status where multiple objects or features are checked.
- AnalogIndicator, AnalogIndicatorWithScales – for displaying numerical results in a nice-looking graphical way.



Example use of one PassFailIndicator and several BoolIndicatorBoards.

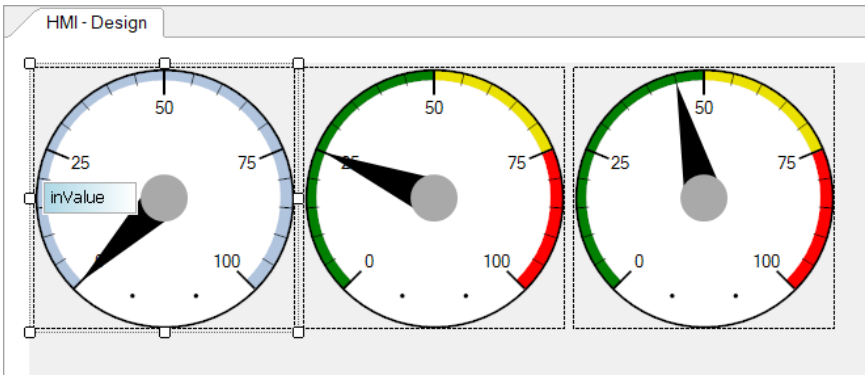


ListView and DetailedListView.

See also: Example with DetailedListView.

AnalogIndicator

AnalogIndicator and its variant AnalogIndicatorWithScales are used to display numerical results in a retro style – as analog gauges. The latter variant has green-orange-red ranges defined with properties like *GreenColorMinimum*, *GreenColorMaximum* etc. The ranges are usually disjoint, but there can also be nested ranges: e.g. the red color can span throughout the scale, and the green color can have a narrower range around the middle of the scale. The green sector will be displayed on top, producing an effect of a single green sector, with two red sectors outside.

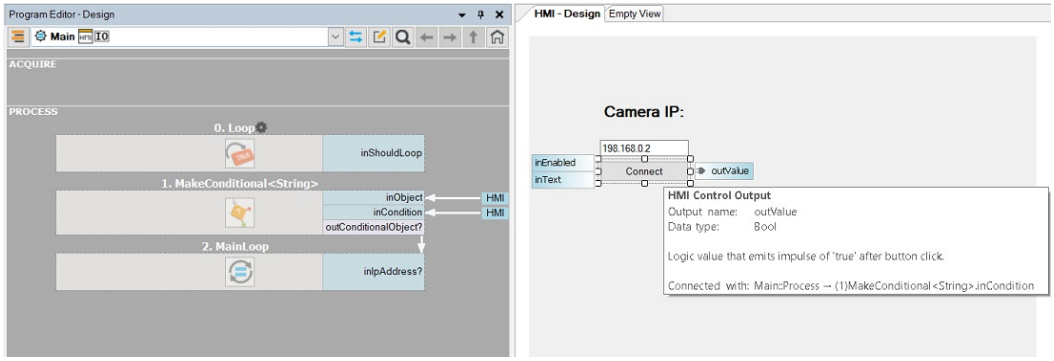


Analog indicators with and without scales

Event Triggering Controls

Most of the standard controls in Aurora Vision Studio fall into one of two categories: (1) data presenters and (2) parameter setters. There are, however, also some controls that can trigger events – most notably the *ImpulseButton* control.

The ImpulseButton control is an event triggering control, which has one output, **outValue**. The value returned is of the Bool type. As long the button is not clicked, it will return *False*. Once the button is clicked, the value becomes *True* for exactly one iteration of the program.

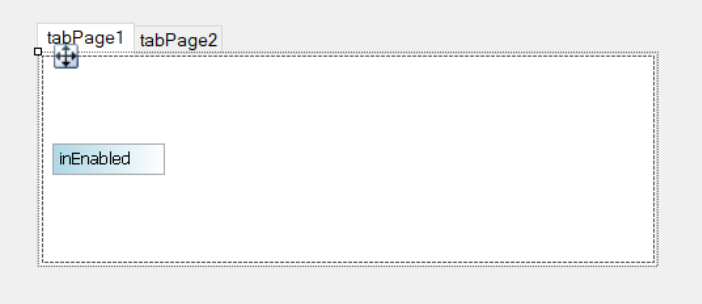


A simple use case of the ImpulseButton.

For more information about events see: Handling HMI Events.

The TabControl Control

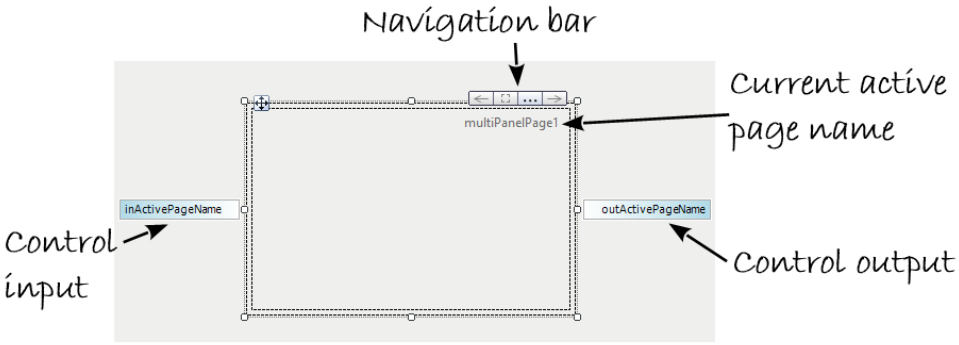
TabControl is the one of the methods to organize HMI panels within a limited space of a form. It is a container with several "tabs", where each page contains a different set of controls.



TabControl.

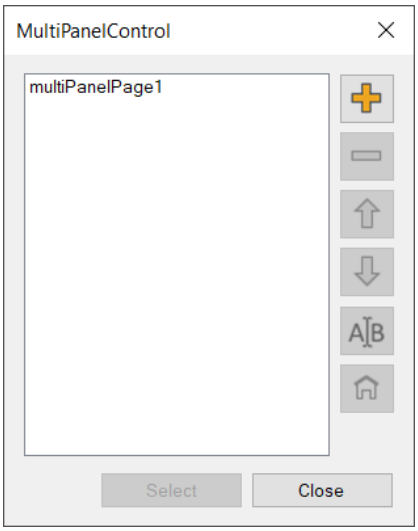
The MultiPanelControl Control

MultiPanelControl is an easy and convenient way to organize bigger HMI panels. It employs a multi-screen approach to provide more space and enables adding special purpose windows like e.g. a "Configuration" page.



MultiPanelControl.

To create a multi-screen HMI, put a MultiPanelControl control in your window and fit it to the window size. In the upper-right corner of this control, you will see a four-element Navigation Bar ( ). Using the left and right arrows you will be able to switch between consecutive screens. The frame button allows for selecting the parent control in the same way as the "Select: MultiPanelControl" command in the control's context menu. The triple-dot button opens a configuration window as on the picture below:



MultiPanelControl Property Box.

Also when you double-click on the area of a MultiPanelControl, the same configuration window appears. Using this window you are able to organize the pages. During work with pages there are at least two necessary things you have to remember. First – all pages names have to be unique globally. Second – one **MultiPanelControl** contains many **MultiPanelPages**. An important thing is the home icon – use it to set the **Initial Page** of your application. It will be the first page to appear when the program starts.

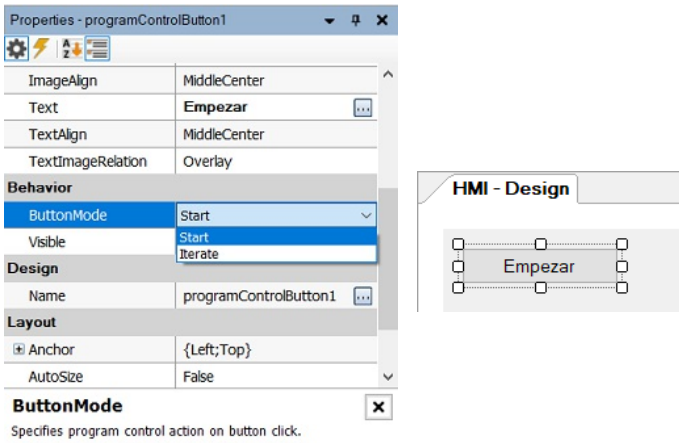
A MultiPanelControl control is usually accompanied by some **MultiPanelSwitchButton** controls that navigate between the pages. Usage of a button of this kind is very simple. Just place it on your window and set the **TargetPage** property (also available through double-clicking on the button). Page switching buttons are usually placed within individual pages, but this is not strictly required – they can also be placed outside of the MultiPanelControl control.

Every HMI Control has built-in input and output ports. In MultiPanelControl, by default, two ports are visible: "inActivePageName" and "outActivePageName", but others are hidden. To make them visible click the right mouse button on the control and select *Edit Ports Visibility...*. The "inActivePageName" input enables setting the active page directly from your program. It is another way to handle page changing. One is using the dedicated buttons by the operator and here comes another method. It might be useful e.g. when "something happens" and you would like to change the screen automatically, not by a button click. This can be e.g. information that a connection with remote host was lost and immediate reconfiguration is needed. The "outActivePageName" by definition is the output which allows you to check the current active page.

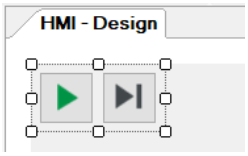
Note: There is a tutorial example HMI Multipanel Control.

Program Control Buttons

Controlling the program execution process, i.e. starting, iterating and pausing, by the end user is possible with a set of ProgramControlButton controls, or with the complete panel named ProgramControlBox. These controls are available in the "Controls" section of the HMI Controls window.



A Start button with a custom text.



ProgramControlBox, similar to the controls from Aurora Vision Studio.

Remarks:

- The *Stop* button is not available for the HMI, because the end user should not be able to stop the entire application at a random point of execution. If you have a good reason to make it possible, use a separate ImpulseButton control connected to an Exit filter.
- ProgramControlButton and ProgramControlBox controls are useful for the purpose of fast prototyping. In many applications, however, it might be advisable to design an application-specific state machine.

File and Directory Picking

There are two controls for selecting paths in the file system, which are FilePicker and DirectoryPicker. They look similar to a TextBox, but they have a button next to them, which brings up a standard system file dialog for browsing a file or directory.

What is worth mentioning about these two controls is the option *NilOnEmpty*, which can be useful for conditional execution. As the name suggests, it causes the control to return the special *NIL* value instead of an empty string, when no file or directory is chosen.

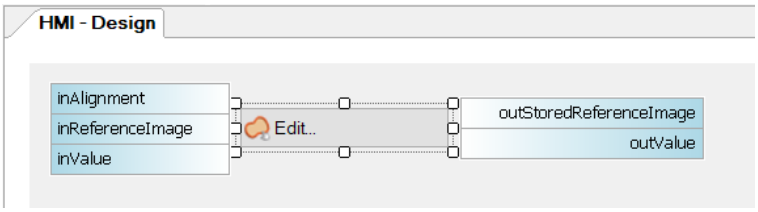
Shape Editors

Geometrical primitives, regions and fitting fields, that are used as parameters in image analysis algorithms, are usually created by the user of Aurora Vision Studio. In some cases, however, it is also required that the end user is able to adjust these elements in the runtime environment. For that purpose, in the "Shape Editors" section there are appropriate controls that bring graphical editors known from Aurora Vision Studio to the runtime environment.

Note: Shape Array Editor controls allow to create multiple primitives of the same data type and return the array of primitives e.g. Segment2DArray, Circle2DArray etc.

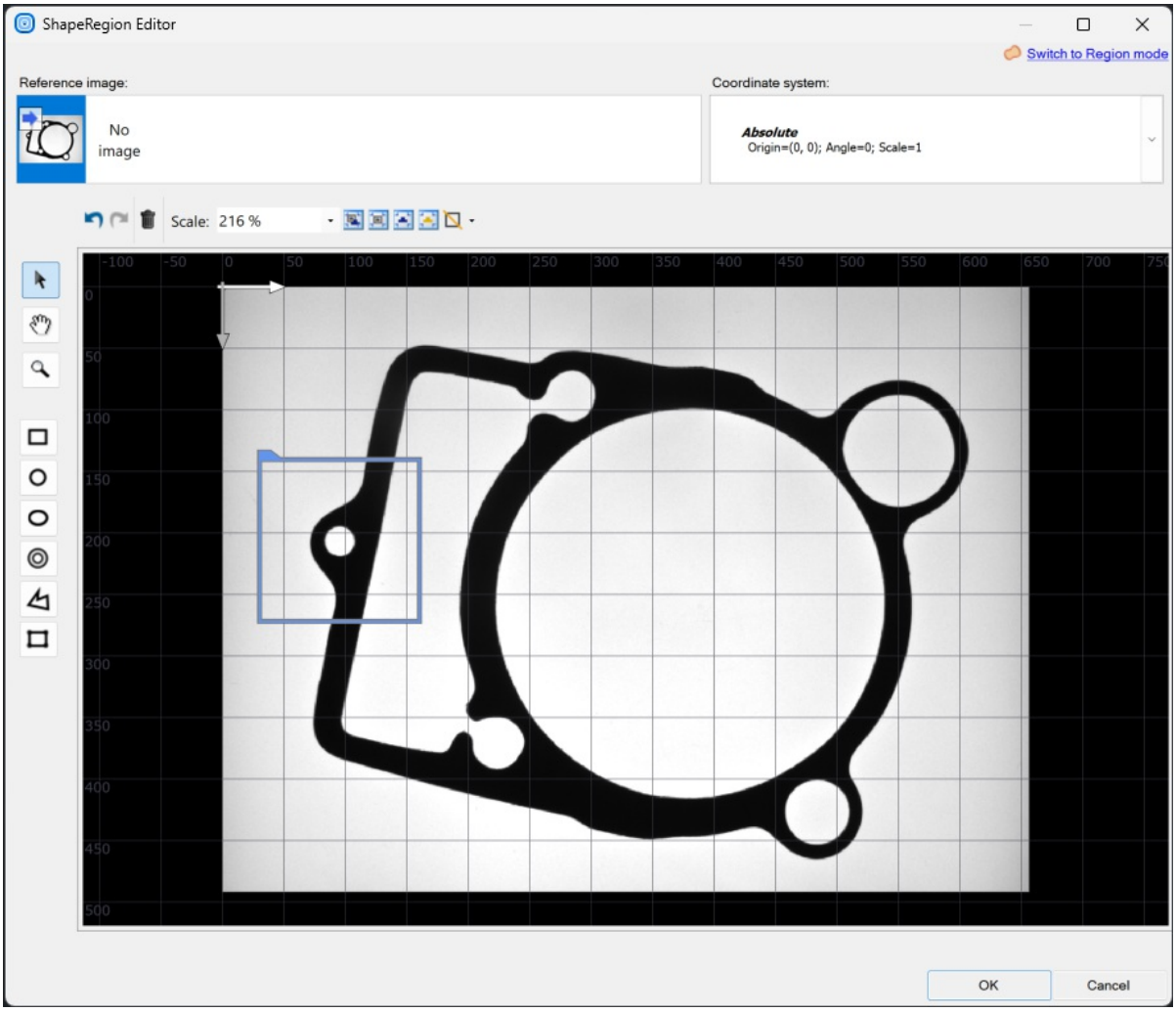
Each such editor has a form of a button with four ports visible by default:

- **inAlignment** – for specifying an appropriate coordinate system for the edited shape.
- **inReferenceImage** – for specifying a background images for the editor.
- **inValue** – for setting initial value of the edited shape.
- **outValue** – for getting the value of the shape after it is edited by the end user.
- **outStoredReferenceImage** – a reference image that was displayed in the editor during last editing approved with the OK button. Active when **inStoreReferenceImage** is set to *True*.



Sample ShapeRegion editor control in the HMI Designer.

When the end user clicks the button, an appropriate graphical editor is opened:



Sample ShapeRegion editor opened after the button is clicked in the runtime environment.

The following properties are specific for these controls allow for customizing appearance of an opened editor:

- **HideSelectors** – allows to simplify the editor by removing image and coordinate system selectors.
- **Message** – an additional text that can be displayed in an opened editor as a hint for the user.
- **WindowTitle** – text displayed on the title bar of the opened editor's window.

Shape editors can be used for creating input values for filter or modify existing values. Every shape editor has a proper data type assigned to **inValue** and **outValue**. The following list shows data types assigned to individual HMI controls and example filters that can be connected with them.

- Arc2DEditor – Arc2D – CreateArc
- ArcFittingFieldEditor – ArcFittingField – FitArcToEdges
- BoxEditor – Box – CreateBox
- Circle2DEditor – Circle2D – CreateCircle
- CircleFittingFieldEditor – CircleFittingField – FitCircleToEdges
- Ellipse2DEditor – Ellipse2D – MakeEllipse
- Line2DEditor – Line2D – DrawLines_SingleColor
- LocationEditor – Location – LocationsToRegion
- PathEditor – Path – OpenPath
- PathFittingFieldEditor – PathFittingField – FitPathToEdges
- Point2DEditor – Point2D – AlignPoint
- Rectangle2DEditor – Rectangle – CreateRectangle
- Ring2DEditor – Ring2D – CreateRing
- Segment2DEditor – Segment2D – CreateSegment
- SegmentFittingFieldEditor – SegmentFittingField – FitSegmentToEdges
- SegmentScanFieldEditor – SegmentScanField
- ShapeRegionEditor – ShapeRegion – DrawShapeRegions_SingleColor

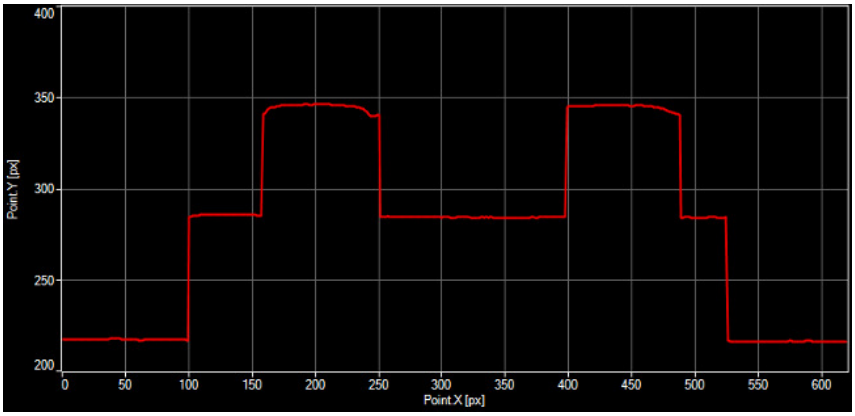
Array versions of shape editors are also available for creating multiple primitives:

- Arc2DArrayEditor – Arc2DArray
- ArcFittingFieldArrayEditor – ArcFittingFieldArray
- Circle2DArrayEditor – Circle2DArray
- CircleFittingFieldArrayEditor – CircleFittingFieldArray
- Ellipse2DArrayEditor – Ellipse2DArray
- Line2DArrayEditor – Line2DArray
- LocationArrayEditor – LocationArray
- PathArrayEditor – PathArray
- PathFittingFieldArrayEditor – PathFittingFieldArray
- Point2DArrayEditor – Point2DArray
- Rectangle2DArrayEditor – Rectangle2DArray
- Ring2DArrayEditor – Ring2DArray
- Segment2DArrayEditor – Segment2DArray
- SegmentFittingFieldArrayEditor – SegmentFittingFieldArray
- SegmentScanFieldArrayEditor – SegmentScanFieldArray
- ShapeRegionDeprecatedArrayEditor – ShapeRegionDeprecatedArray

ProfileBox

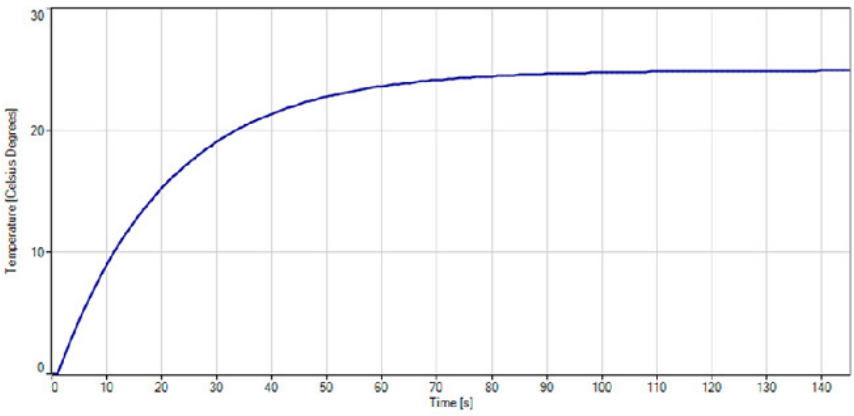
ProfileBox is the HMI control that displays a profile of evenly sampled measurements. ProfileBox is available in the "Indicators" section of the HMI Controls window.

ProfileBox may be useful for displaying an array of samples of a known length, i.e. a profile of an image column or a laser line.



Profile of a laser line.

Also, it is used for displaying dynamic data from sensors in the function of time.



Temperature chart.

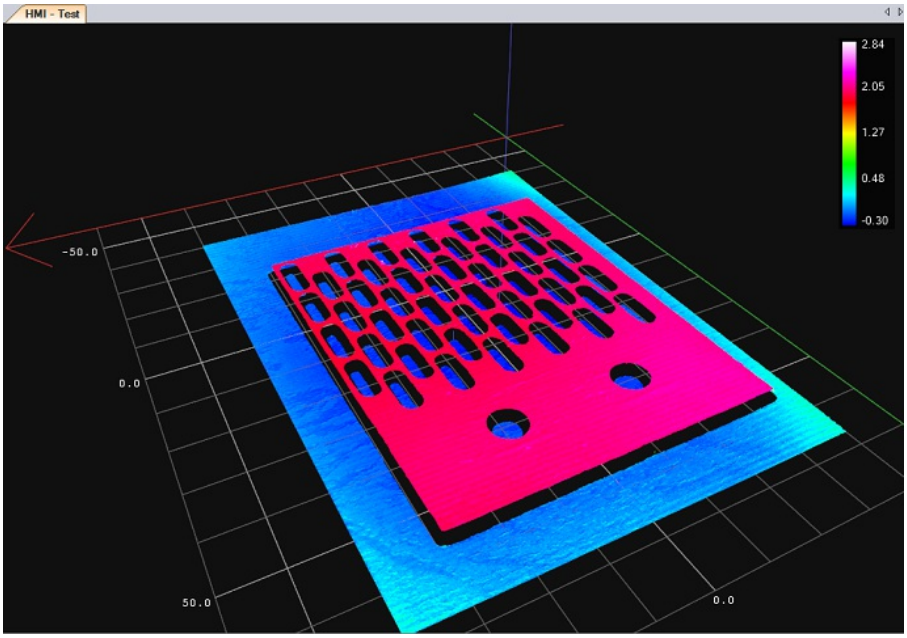
There are several properties specific to that control:

- **DomainDefaultLength, DomainDefaultSizeMode, DomainScale and DomainStart** – the set of parameters specifying the appearance and behavior of the X axis.
- **EnableManualZoomChange** – for specifying whether the end user is able to zoom the chart area.
- **ChartColor, GridColor, BackColor, AxisColor** – for specifying the colors of a ProfileBox content.
- **HighQualityMode** – for specifying whether the chart is drawn with high quality or with low quality, but faster.
- **HorizontalAxisName, VerticalAxisName** – for specifying the names of axes.

- **HorizontalAxisMin, HorizontalAxisMax, VerticalAxisMin, VerticalAxisMax** – for setting the limits of chart axes.

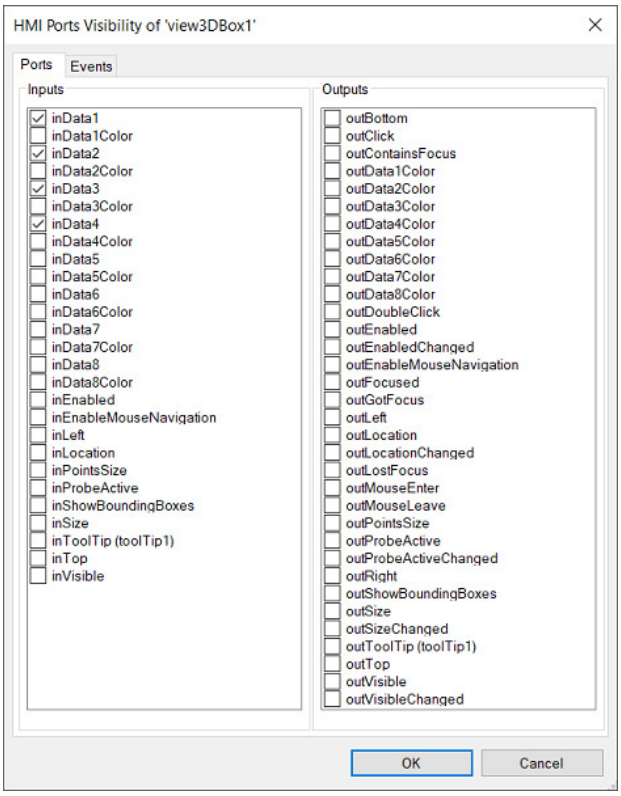
View3DBox

View3DBox is the HMI control that displays 3D primitives. It is available in the "Indicators" section of the HMI Controls window.



An example of using View3DBox control.

Default settings of that HMI control allow to connect four inData inputs. However, it is possible to use up to eight inputs. To make all inputs visible, right-click on the View3DBox, select **Edit Ports Visibility...** and select the inputs you want to use.



View3DBox ports visibility.

inData input accepts all 3D data types such as Point3D, Box3D, Circle3D, Plane3D, Segment3D, Line3D, Sphere3D and the data types for point cloud representation: Surface, Point3DGrid and Point3DArray.

The color of displayed primitives can be changed with **inDataColor** input.

There are also other properties that can be used to customize a data preview:

- **EnableAntiAliasing** – when set to true increases the render quality at the expense of performance.
- **GridMode** – specifies visibility and orientation of the measurement grid displayed in the scene.
- **PointSize** – specifies the display size of the cloud points relative to the size of the scene.

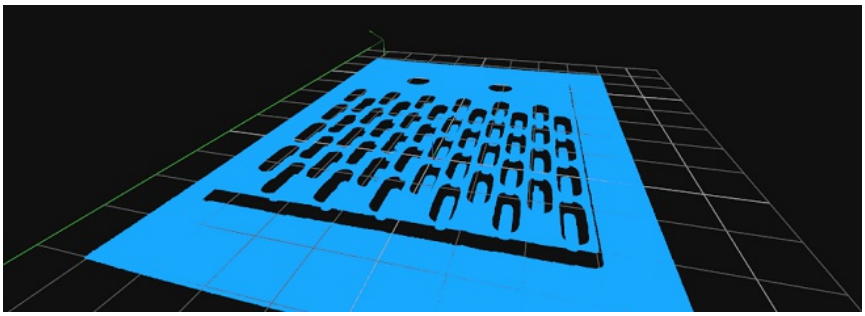
- **ProjectionMode** – selects the view/projection mode of 3D visualization.
- **ScaleColoringMode** – allows to enable automatic coloring of the cloud points according to the position along the specified axis.
- **ShowBoundingBoxes** – enables displaying range of bounding boxes around point cloud primitives in the preview.
- **WorldOrientation** – the type and orientation of a coordinate system used to display 3D primitives.

Interaction between the user and his program can be adjusted with parameters in **Behaviour** section. To customize this interaction the following parameters should be changed:

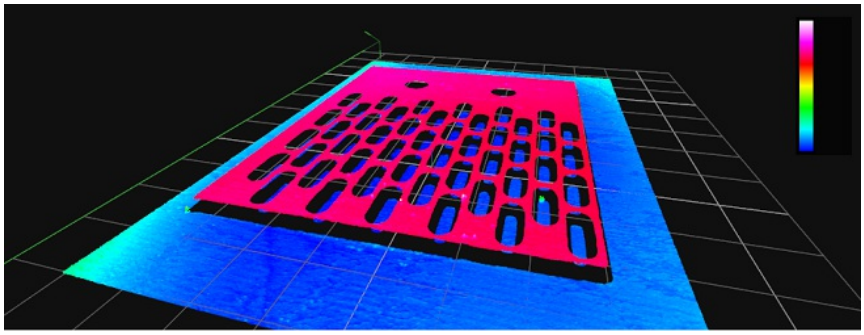
- **Enabled** – indicates whether the control is enabled.
- **EnableMouseNavigation** – when set to true enables the user to change observer position using mouse.
- **InitialViewState** – specifies the initial position of observer in the scene.
- **Visible** – determines whether the control is visible or hidden.

Commonly used features of View3DBox control

To make point cloud differences along Z axis more visible, change the ScaleColoringMode form Solid to ZAxis.



View3DBox with ScaleColoringMode set to Solid.



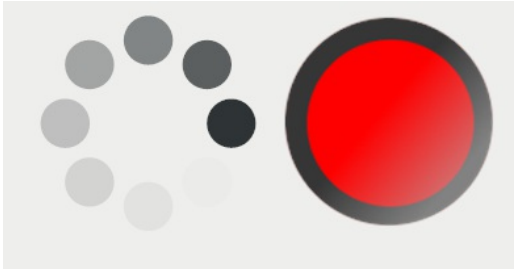
View3DBox with ScaleColoringMode set to ZAxis.

Point3DArray displaying

Point3DArray containing up to 300 points is displayed as an array of 3D marks (crosses). If there are more than 300 points, they are displayed as a standard point cloud.

Activity Indicator

ActivityIndicator is used to visualize a working or waiting state of User Interface. There are two appearance modes available: "Busy" and "Recording". Both are shown in the below picture. By setting its input **inActive** to *True* or *False* values, user can display the animation showing that the program is either calculating or recording something.



Busy and Recording mode of ActivityIndicator.

TextSegmentationEditor and OcrModelEditor

TextSegmentationEditor and OcrModelEditor HMI controls allow the user to edit a SegmentationModel and OCRModel respectively in a runtime environment. They allow to create an application which can be re-trained when new fonts are required to be read or when characters change their shape over time.

The concept of OCR and traditional OCR tools available in Aurora Vision Studio software are described in the following articles:

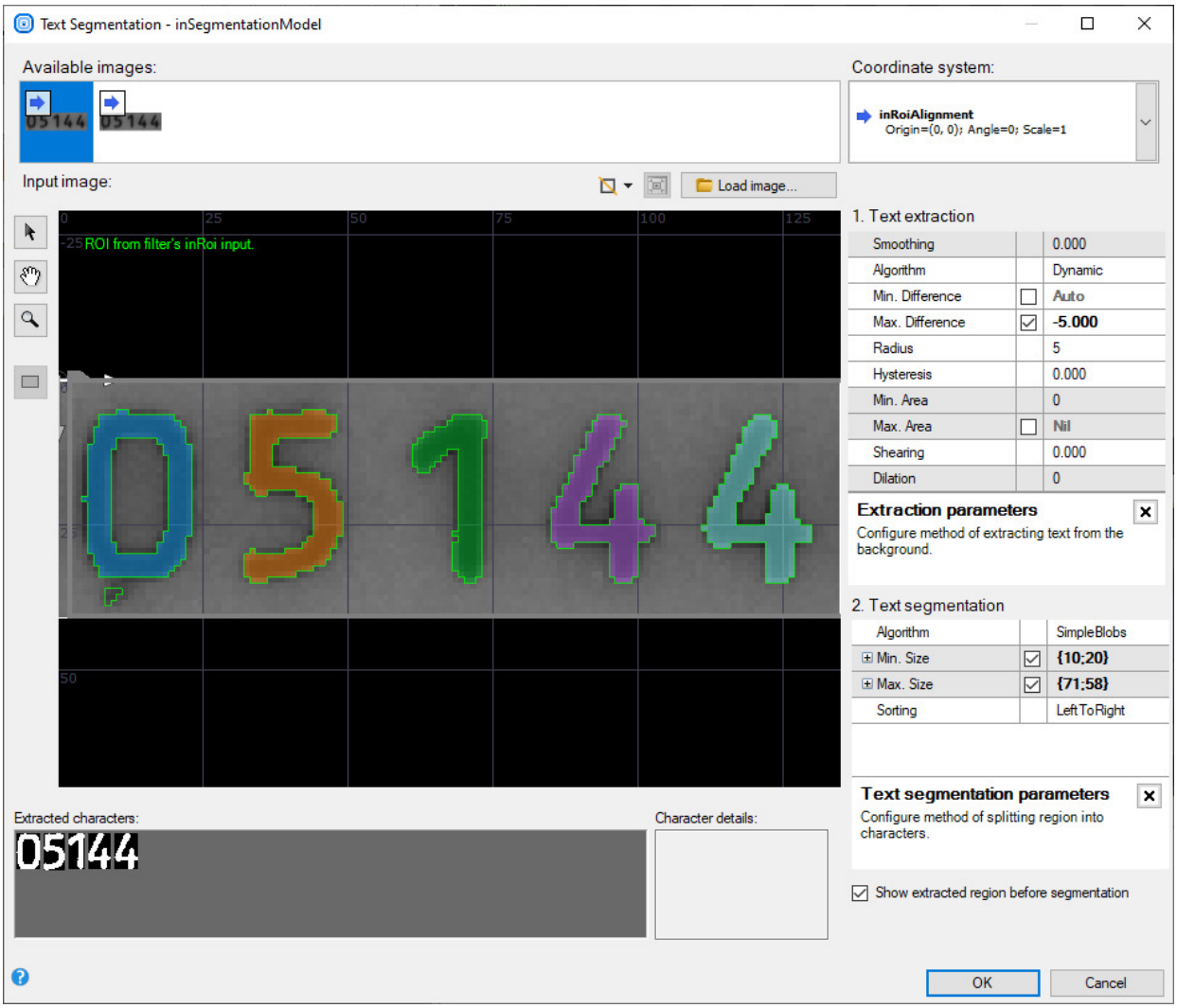
- Machine Vision Guide: Optical Character Recognition – traditional method,
- Creating Text Segmentation Models,
- Creating Text Recognition Models.

TextSegmentationEditor

Once a TextSegmentationEditor is added, the following ports are visible:

- **inReferenceImage** – an image set as the editor background.
- **inRoi** – a region of interest for the reference image.
- **inRoiAlignment** – an alignment for the region of interest. Its purpose is to adjust the ROI to the position of the inspected object.
- **outStoredReferenceImage** – the image set as the editor background that can be sent to the algorithm.
- **outValue** – a segmentation model. This value can be connected to **inSegmentationModel** input of the ExtractText filter.
- **outValueChanged** – an output of the bool data type that is set to *True* for one iteration in which the **outValue** is changed.

After clicking on TextSegmentationEditor button, the editor will be opened. However, to be able to open it, the program has to be executed.



Editor is the same as inSegmentationModel editor known from ExtractText filter.

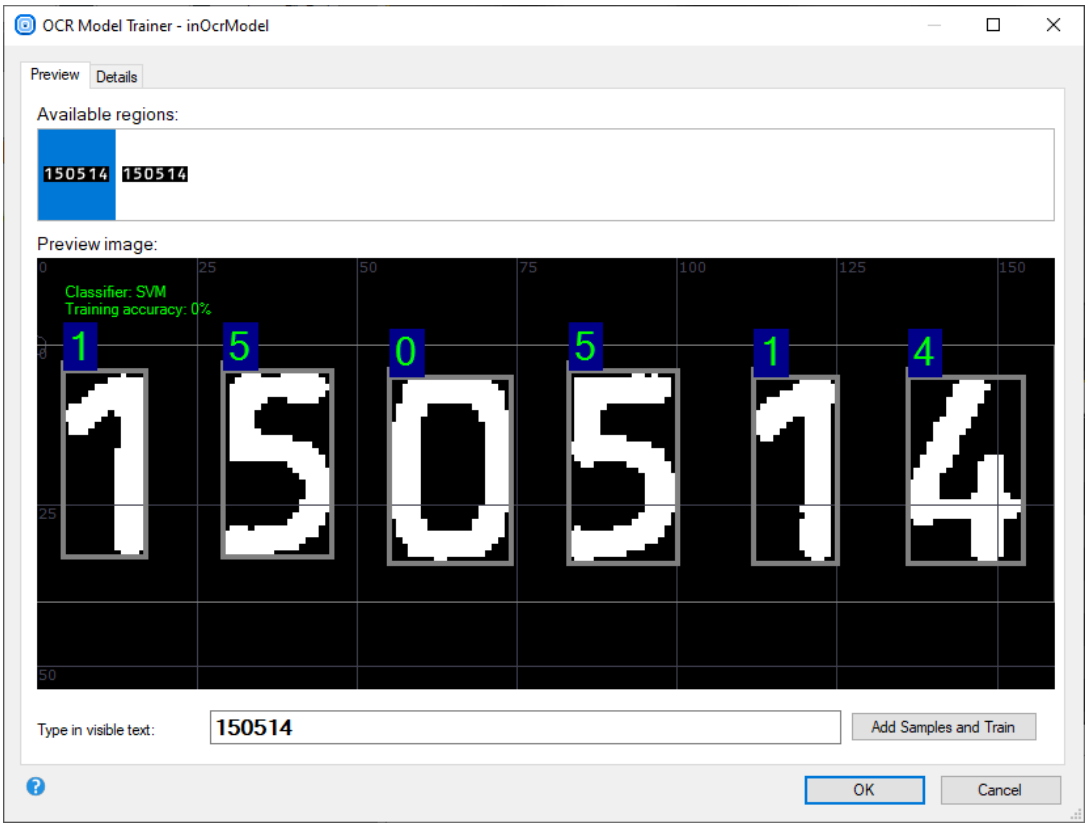
OCRModelEditor

Once an OCRModelEditor is added, the following ports are visible:

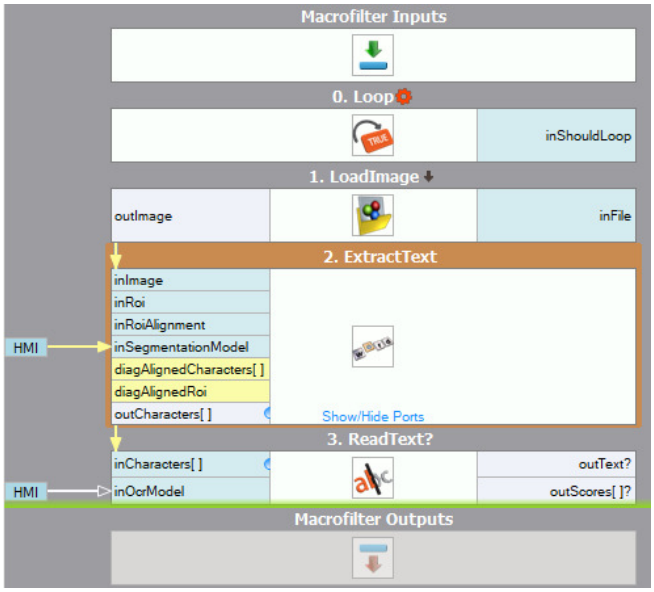
- **inReferenceCharacters** – is a value of RegionArray data type. Characters from ExtractText filter can be connected to that input. The characters will be used during a model training.
- **inReferenceImage** – an image set as the editor background.
- **outStoredReferenceImage** – the image set as the editor background that can be sent to the algorithm.
- **outValue** – is a result OcrModel. This value should be connected to **inOcrModel** input in ReadText filter.

- **outValueChanged** – an output of the bool data type that is set to *True* for one iteration in which the **outValue** is changed.

After clicking on OCRModelEditor button, the editor will be opened. However, to be able to open it, the program has to be executed.



This editor is the same as the OcrModel editor known from the ReadText filter.



A simple program using TextSegmentationEditor and OcrModelEditor.

KeyboardListener

KeyboardListener is used for getting information about keyboard events. This HMI control returns an IntegerArray where each value describes a number represented in the ASCII code.

To use KeyboardListener or any other tools from the "Components" category, a control must be dropped at any place on the HMICanvas.

All default output values allow checking which key is pressed, but there are a few differences between them:

- **outKeysDown** – emits array of keys that have been recently pressed.
- **outKeysPressed** – emits array of currently pressed keys. Using this output is recommended in most cases.
- **outKeysUP** – emits array of keys that have been recently released.

To use those values elsewhere in the program, the best solution is to copy them with the **CopyObject** filter

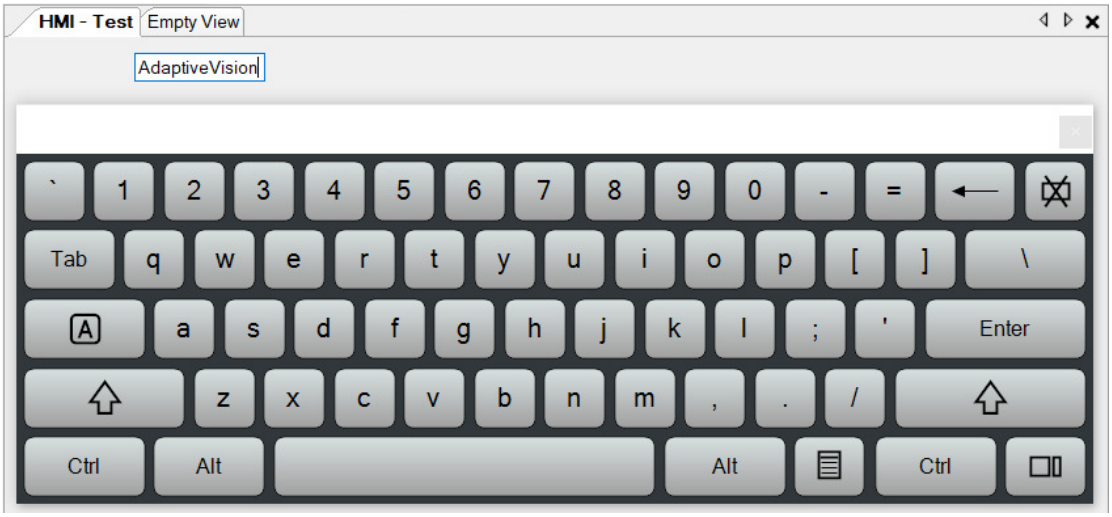
using the IntegerArray data type.

VirtualKeyboard

VirtualKeyboard is the HMI control which enables showing of automatic on-screen virtual keyboard for editing content of HMI controls.

This functionality can be used in a runtime environment when a physical keyboard is not available at the workstation.

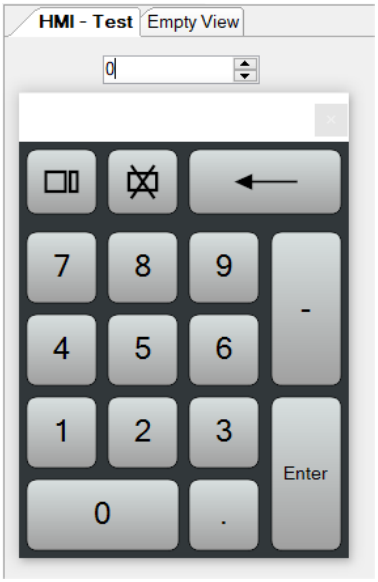
To use VirtualKeyboard or any other tools from the "Components" category, a control must be dropped at any place on the HMICanvas.



The example of using VirtualKeyboard for entering data on HMI panel into TextBox control.

User can set the initial preset size of keyboard window by setting proper **InitialKeyboardSize** value in the properties.

It is possible to display only a number pad keyboard for editing numerical values. To do that, **EnableNumPadOnlyKeyboard** must be set to *True* in the properties.



The example of using VirtualKeyboard for entering data on HMI panel into NumericUpDown control.

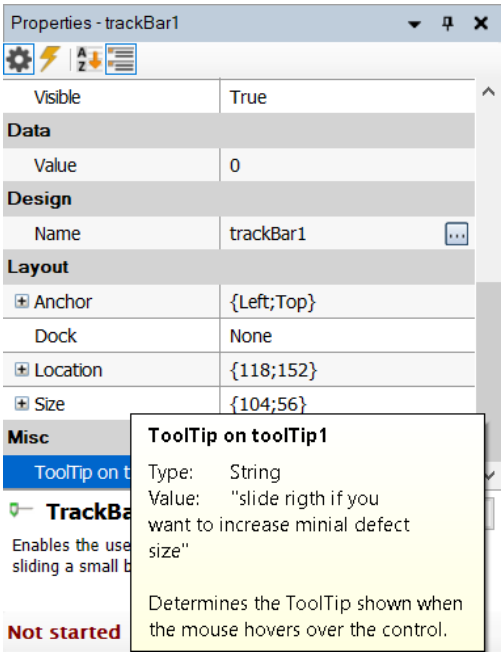
ToolTip

ToolTip HMI control is used to display messages created by the user. Those messages are displayed when user moves the pointer over an associated control.

This functionality can be used when a developer wants to give helpful tips for a final user.

To use ToolTip or any other tools from the "Components" category, a control must be dropped at any place on the HMICanvas.

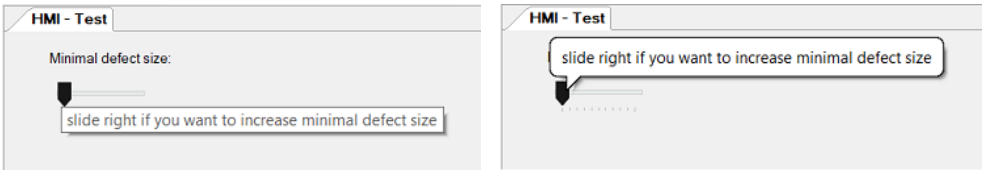
Once ToolTip control is added, each HMI Control has additional parameter available under the "Misc" category in the Properties window. It is possible to use multiple ToolTips with different parameters.



*Editable ToolTip message in **TrackBar** HMI control properties.*

ToolTip properties can be adjusted to the user's requirements by changing the following values:

- **Active** – determines if the ToolTip is active. A tip will only appear if the ToolTip has been activated.
- **AutomaticDelay** – sets the values of AutoPopDelay, InitialDelay and ReshowDelay to the appropriate values.
- **AutoPopDelay** – determines the length of time the ToolTip window remains visible if the pointer is stationary inside a ToolTip region.
- **BackColor** – the background color of the ToolTip control.
- **ForeColor** – the foreground of the ToolTip control.
- **InitialDelay** – determines the length of time the pointer must remain stationary within a ToolTip region before the ToolTip window appears.
- **IsBalloon** – indicates whether the ToolTip will take on a balloon form.
- **ReshowDelay** – determines the length of time it takes for subsegment ToolTip windows to appear as the pointer moves from one ToolTip region to another.
- **ToolTipTitle** – determines the title of the ToolTip.
- **UseAnimation** – when set to true, animations are used when the ToolTip is shown or hidden.
- **UseFading** – when set to true, a fade effect is used when ToolTips are shown or hidden.



Displayed information with the default ToolTip setting and with the active balloon form.

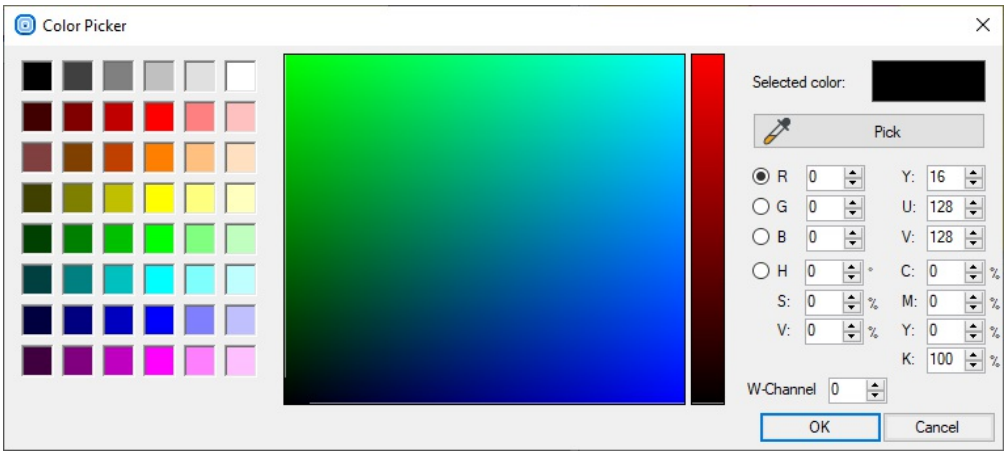
ColorPicker

ColorPicker is a tool which allows the user to pick the color components in the runtime environment.

This control can be used, for example, when it is necessary to know the reference color used by an algorithm.

The **outValue** port contains the value of selected color and can be connected to all tools containing an input of the Pixel data type.

In a Color Picker window the user can use the palette of colors or enter the values of color components manually.



Color Picker window.

BoolAggregator

BoolAggregator allows combining multiple boolean signal sources using logic functions for controls such as EnabledManager.

This function is useful when the user wants to aggregate boolean signals from at least two HMI controls, e.g. **OnOffButton** or **CheckBox**, without having to pass them through the program.

To use BoolAggregator or any other tools from the "Logic and Automation" category, a control must be dropped at any place on the HMICanvas. It then becomes available as an icon in the lower panel on the HMI designer. These controls have no graphical representation in the HMI application and only provide a data processing function.

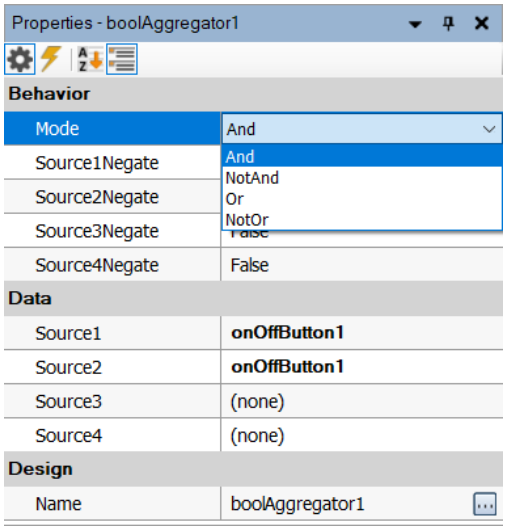
While working with that control, user needs to select boolean value sources to be aggregated. It can be done in the Properties window. Also, additional two values from the program can be aggregated by connecting them to the **inValue1** and **inValue2** inputs.

Using BoolAggregator's properties it is also possible to negate the value from a data source by setting **SourceNegate** to *True*.

Mode property specifies the type of aggregation function to be used to connect multiple data sources. The following aggregation functions are available:

- **And** – in1 and in2 and ... and inN (e.g.: {True, True, True, False} → False; {True, True, True, True} → True)
- **NotAnd** – not (in1 and in2 and ... and inN) (e.g.: {True, True, True, False} → True; {True, True, True, True} → False)
- **Or** – in1 or in2 or ... or inN (e.g.: {False, False, False, False} → False; {False, True, False, False} → True)
- **NotOr** – not (in1 or in2 or ... or inN) (e.g.: {False, False, False, False} → True; {False, True, False, False} → False)

The BoolAggregator control is a boolean value source control itself and its result can be used as a value source by subsequent controls, like EnabledManager or another BoolAggregator. The result value can also be used in the program by connecting to the controls **outValue** output port (hidden by default). The result value is computed asynchronously to the main program execution.



Bool Aggregator's properties.

EnabledManager

EnabledManager allows to automatically manage controls enable state (by writing to its **Enable** property) using other HMI controls as a boolean value source.

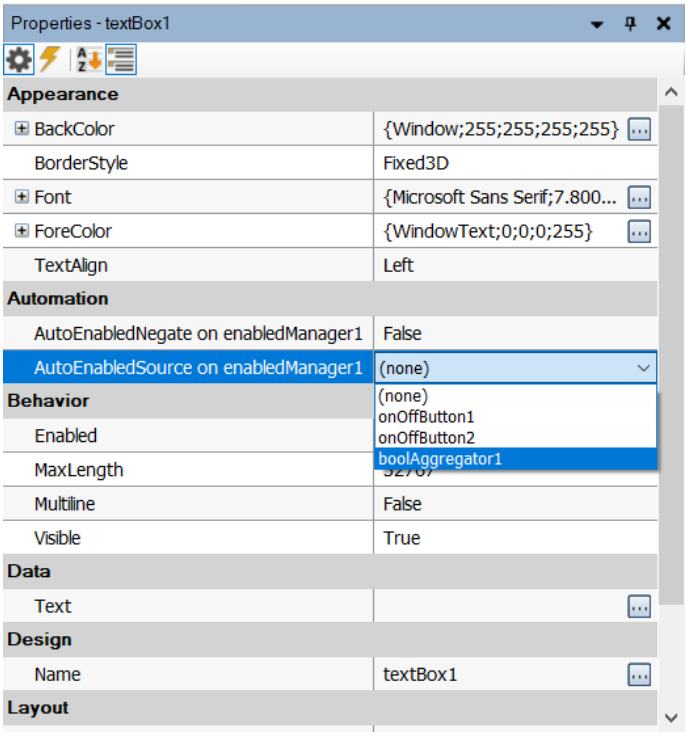
It can be used with BoolAggregator in order to manage controls' enable states in more complex cases.

To use EnabledManager or any other tools from the "Logic and Automation" category, a control must be dropped at any place on the HMICanvas. The control itself does not provide any properties or ports, but it activates the enable management functionality in the HMI design. Only one EnabledManager can be added to the design.

Once the EnabledManager is added to the design, it allows sending an enable signal to every HMI control. The user can specify the source of the enable signal by selecting it from the **AutoEnabledSource** property list available in the Properties window of arbitrary controls in the HMI design. It is also possible to negate the source data value by setting the **AutoEnabledNegate** to true.

Note that when the **AutoEnabledSource** property is in use for a given control the Enable property of that control cannot be changed in other ways (e.g. by making a connection to it from the program) as the mechanisms would interfere with each other.

In the example below **TextBox**'s enable state is controlled by **BoolAggregator** which gathers signals from two **onOffButton** HMI Controls.



*Automation category in **TextBox's** properties.*

ChangeLogger

ChangeLogger tracks changes in the HMI value editing controls and allows to log the user editing actions.

To use the ChangeLogger component, drop it at any place on the HMICanvas. Doing this will add additional properties in the Automation category for every HMI control that allows to define output data (such as TextBox, Knob or EdgeModelEditor). Next, in each editing control that should participate in the logging, set the property **LogChanges** to True. By default controls are not tracked.

User can specify for every editing control whether they would like to log its changes, as well as (optionally) what kind of comment they would like to include in a log file for it (**LogChanges** and **LogComments** properties respectively).

For simple applications, the ChangeLogger component can handle writing log entries to a log file completely automatically. To do so, a path to a log file must be specified in the **LogFile** property. The file will be created automatically if it does not exist, and new lines will be appended to existing files, but a subdirectory will not be created if it does not exist.

The log file name can be based on the current date. To do so, the following placeholder tags can be used:

- *%yyyy%* – current year
- *%mm%* – current month, always two digits
- *%m%* – current month, one or two digits
- *%dd%* – current day, always two digits
- *%d%* – current day, one or two digits

E.g. "HMILogs\Activity %yyyy%-%mm%-%dd%.log" on 11th July 2023 will be resolved to "HMILogs\Activity 2023-07-11.log".

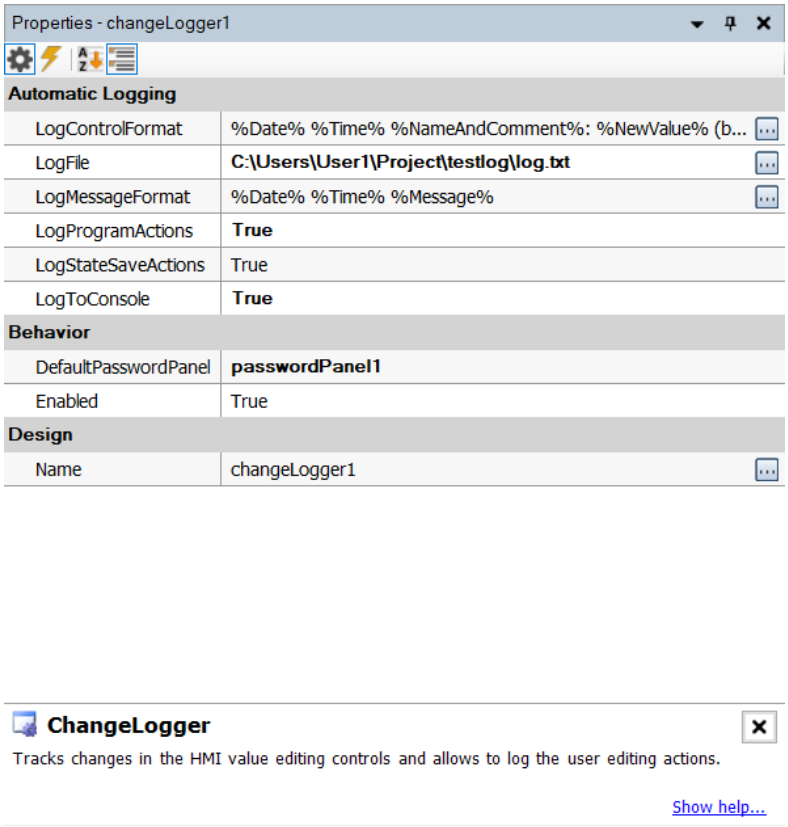
The format of a log entry line for a control value change action can be customized using the **LogControlFormat** property. The following placeholder tags can be used in the pattern:

- *%Date%* – date of the change in the local system format
- *%Time%* – time of the change in the local system format
- *%Name%* – name of the control that was changed (from its Name property)
- *%Comment%* – comment associated with the control that was changed (from its LogComment property)
- *%NameAndComment%* – name and comment automatically combined in the format "Name (Comment)", or just the name when the comment is not specified
- *%NewValue%* – textual representation of the new value set into the control (after the change action)
- *%OldValue%* – textual representation of the last known previous value in the control (from before the change action)
- *%UserName%* – name of the user currently logged in a PasswordPanel control (in a PasswordPanel that the changed control is nested in, or in a PasswordPanel specified in the **DefaultPasswordPanel** property, when the changed control is not nested in one)

There is also an option to enable logging to a file information about the program status changes (program loading, execution starting and stopping) by enabling the **LogProgramActions** property, as well as loading and saving of controls' state (by state management controls) by enabling the **LogStateSaveActions** property. The format of such log entries can be customized with the **LogMessageFormat** property.

For advanced applications and custom log writing scenarios the writing of log entries can be handled manually. To do so the **LogControl** event of the ChangeLogger component can be handled and a macrofilter created to perform the logging action. The event will be invoked after each tracked editing control change action, carrying in the parameters the same information about the editing control that are used in the LogControlFormat pattern.

The ChangeLogger component will group up the editing actions of a single control to not flood the log file. For example, when the user starts rotating the Knob control, instead of triggering the change action for every partial movement, only one change in the control is logged after the rotation of the Knob is finished. To do so a small delay is applied before logging the change.



*Properties available for **ChangeLogger** control.*

EdgeModelEditor

EdgeModelEditor is the HMI control that makes it possible to create a Template Matching model in the runtime environment by using an easy user interface called "GUI for Template Matching".

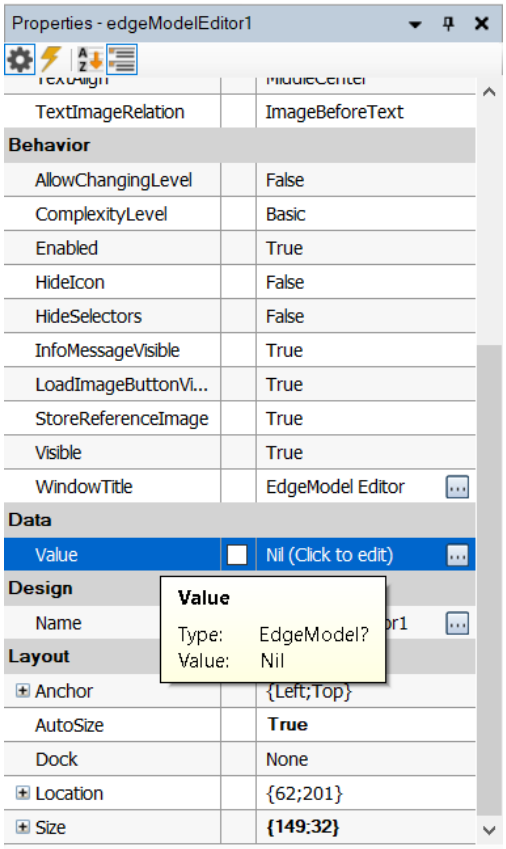
The process of creating a model representing the expected object's shape is described in Creating Models for Template Matching article.

Once an EdgeModelEditor is added, the user can specify a reference image by connecting it to **inReferenceImage** input. The reference image can also be loaded from a file by clicking *Load Image...* button available in the EdgeModel Editor's toolbar.

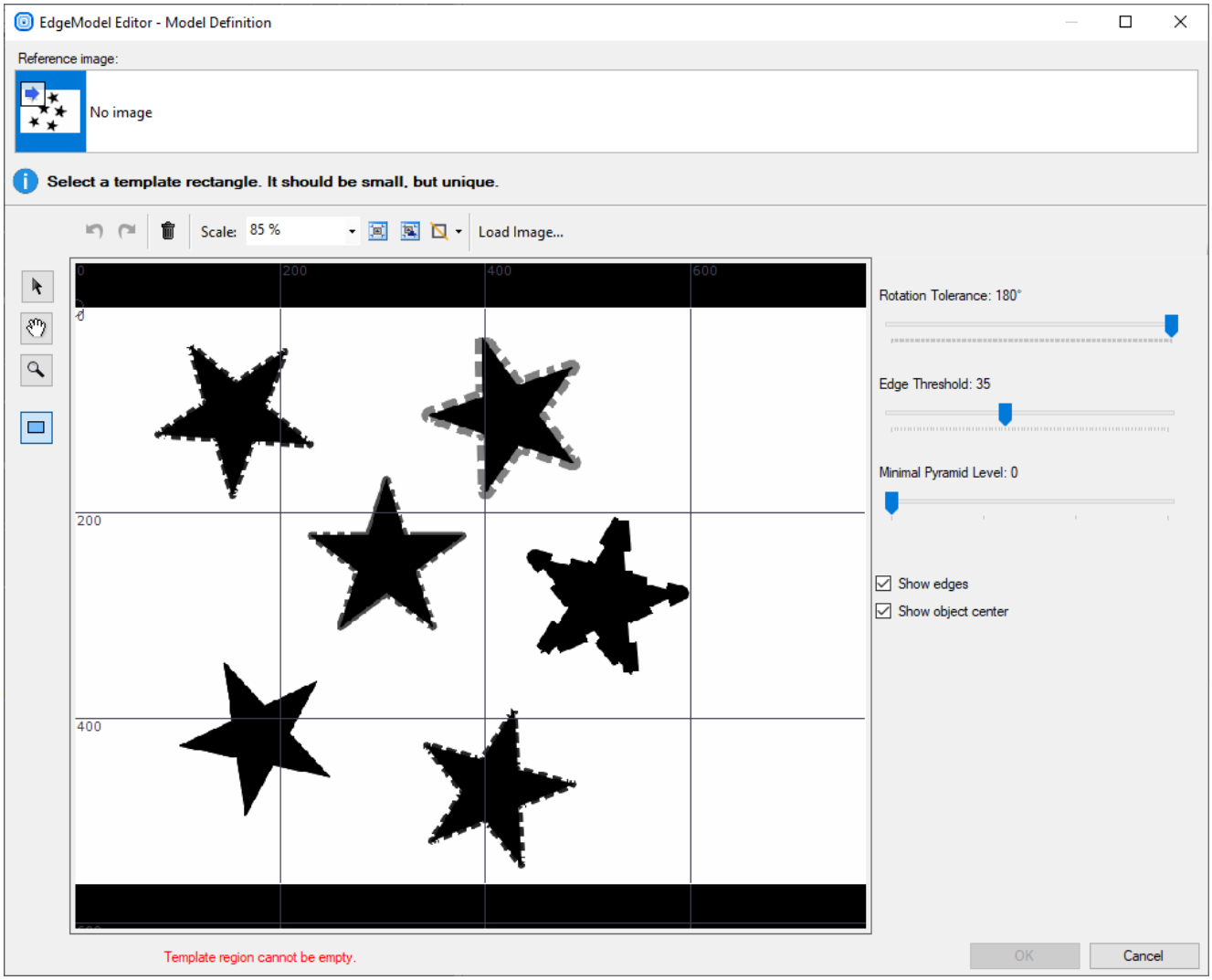
outValue output has EdgeModel data type. This output can be connected to **inEdgeModel** input in **LocateObjects_Edges1** filter.

Besides standard parameters available in the Appearance, Design and Layout sections of the Properties window, the EdgeModelEditor can be customized using the following parameters:

- AllowChangingLevel – allows changing the plugin complexity level.
- ComplexityLevel – default plugin complexity level.
- Enabled – determines if the control is enabled.
- HideIcon – determines if the editor icon is hidden.
- HideSelectors – determines if the image and alignment selectors of EdgeModel Editor are hidden.
- InfoMessageVisible – determines if plugin's hint messages are displayed under the *Reference image* bar.
- LoadImageButtonVisible – determines if a reference image can be loaded from the disc by using "Load Image..." button available in the editor's toolbar.
- StoreReferenceImage – if set to *True* the selected reference image is available at the **outStoredReferenceImage** output.
- WindowTitle – text displayed on the editor window bar.



First method to edit EdgeModel.



EdgeModelEditor window.

To use the model created by EdgeModelEditor, user needs to connect the **outValue** output to the **inEdgeModel** input of LocateSingleObject_Edges1 filter.

GrayModelEditor

GrayModelEditor is the HMI control that makes it possible to create a Template Matching model in the runtime environment by using an easy user interface called "GUI for Template Matching".

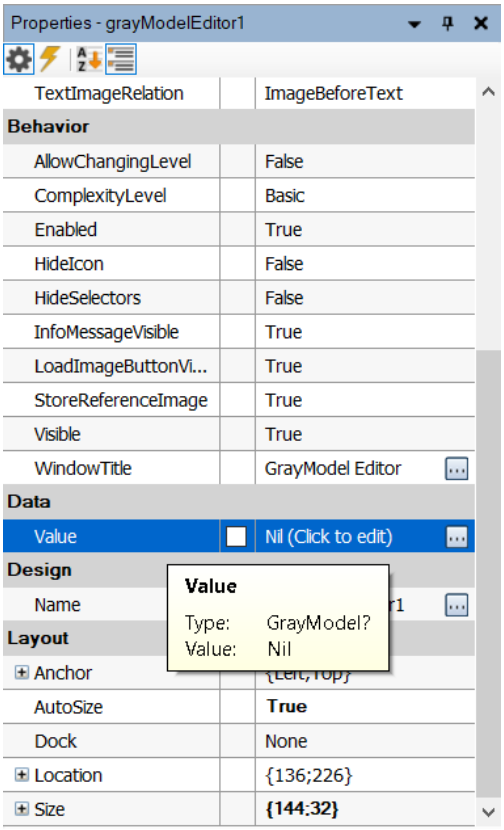
The process of creating a model representing the expected object's shape is described in Creating Models for Template Matching article.

Once a GrayModelEditor is added, the user can specify a reference image by connecting it to **inReferenceImage** input. The reference image can also be loaded from a file by clicking *Load Image...* button available in the GrayModel Editor's toolbar.

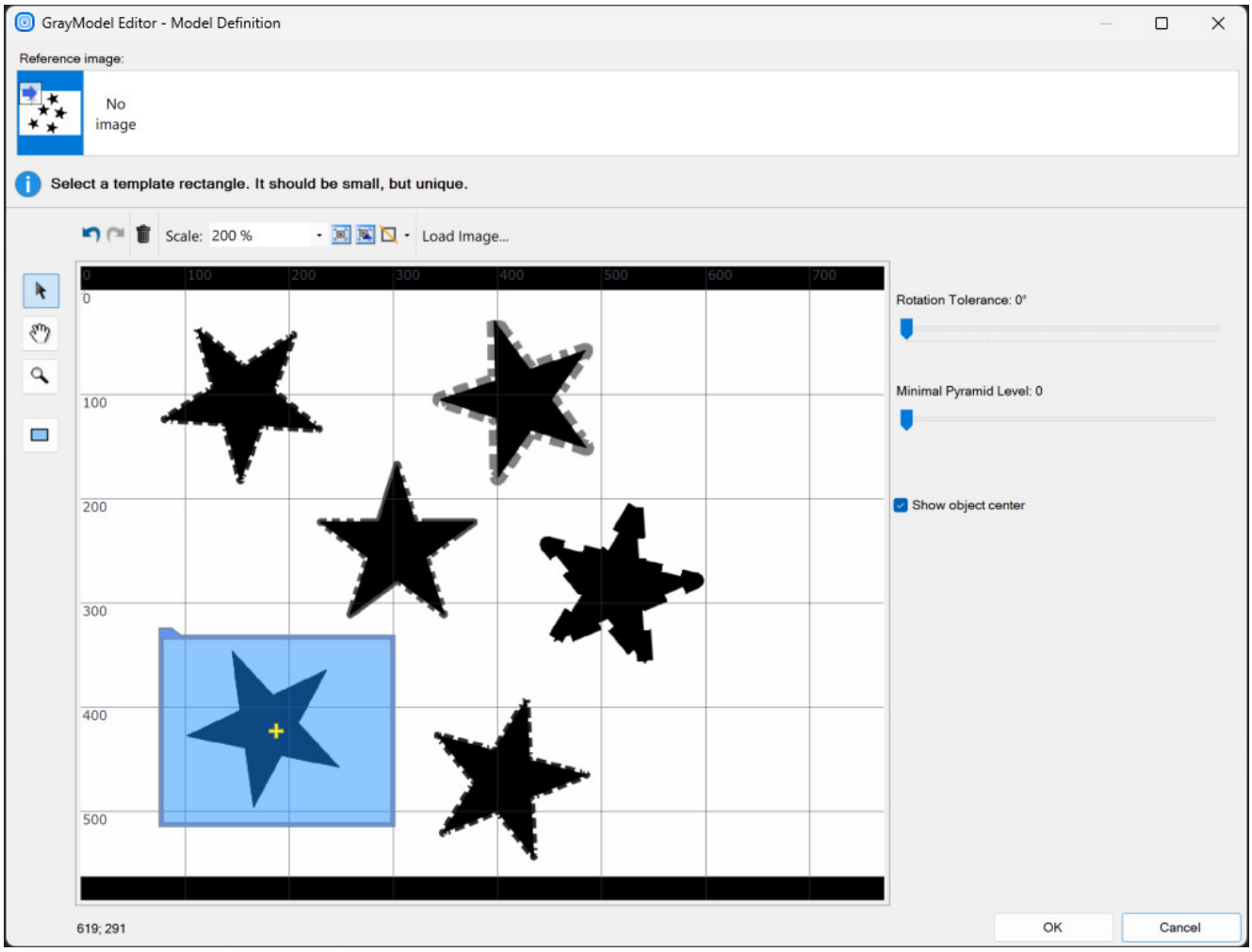
Besides standard parameters available in the Appearance, Design and Layout sections of the Properties window, the GrayModelEditor can be customized using the following parameters:

- AllowChangingLevel – allows changing the plugin complexity level.
- ComplexityLevel – default plugin complexity level.
- Enabled – determines if the control is enabled.
- HideIcon – determines if the editor icon is hidden.
- HideSelectors – determines if the image and alignment selectors of EdgeModel Editor are hidden.
- InfoMessageVisible – determines if plugin's hint messages are displayed under the *Reference image* bar.
- LoadImageButtonVisible – determines if a reference image can be loaded from the disc by using "Load Image..." button available in the editor's toolbar.
- StoreReferenceImage – if set to *True* the selected reference image is available at the **outStoredReferenceImage** output.
- WindowTitle – text displayed on the editor window bar.

To use the model created by GrayModelEditor, user needs to connect the **outValue** output to the **inGrayModel** input of LocateSingleObject_NCC filter.



First method to edit GrayModel.



GrayModelEditor window.

GenICamAddressPicker

GenICamAddressPicker is used to choose GenICam GenTL device address in runtime environment. This control can be used with GenICam filters.

See also: examples where a GenICamAddressPicker control is used: HMI Grab Single Image and HMI Image

Recorder.

GigEVisionAddressPicker

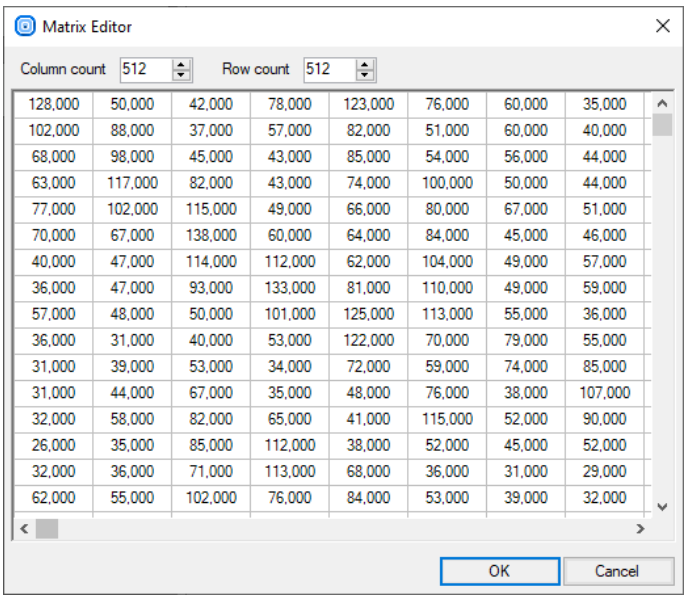
GigEVisionAddressPicker is used to choose GigE Vision device address in runtime environment. This control can be used with GigE Vision filters.

See also: an example where GigEVisionAddressPicker control is used: HMI Image Recorder.

MatrixEditor

MatrixEditor is used to modify existing Matrix or create a new one in runtime environment.

outValue returns matrix which was modified or created in MatrixEditor.



Matrix Editor window.

GrammarRulesPatternEditor

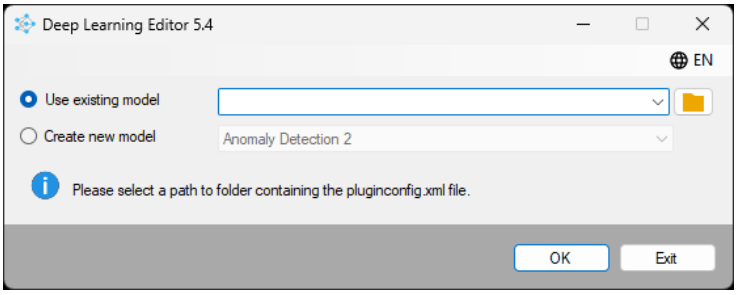
GrammarRulesPatternEditor is a tool which allows the user to create or edit grammar rules patterns in the runtime environment.

This control can be used when it is necessary to define or adjust grammar rules that are used for text recognition or pattern matching algorithms.

The **outValue** port contains the grammar rules pattern value and can be connected to all tools containing an input of the GrammarRulesPattern data type.

Deep Learning

Pressing the Deep Learning Start Button allows you to enter Deep Learning Editor from the level of the HMI.



- **ModelPath** – path of a saved model. If empty or invalid the Model Path Selection dialog box will be opened.
- **AlwaysOnTop** – editor will always be visible as a top window.
- **DisableChangingLocation** – prevents the user from changing model path.
- **Language** – selects language of the editor.

Handling HMI Events

Introduction

The HMI model in Aurora Vision Studio is based on the program loop. A macrofilter reads data from the HMI controls each time before its first filter is executed. Then, after all filters are executed, the macrofilter sends results to HMI. This is repeated for each iteration. This communication model is very easy to use in typical machine vision applications, which are based on a fast image acquisition loop and where the HMI is mainly used for setting parameters and observing inspection results. In many applications, however, it is not only required to use parameters set by the end user, but also to handle events such as button clicks.

To perform certain actions immediately after an event is raised, Aurora Vision Studio also comes with the HMI subsystem that handles events independently of the program execution process. This allows for creating user interfaces that respond almost in real-time to the user input e.g. clicking button, moving mouse cursor over an object or changing its value. It would not be possible if events were handled solely by the program execution process.

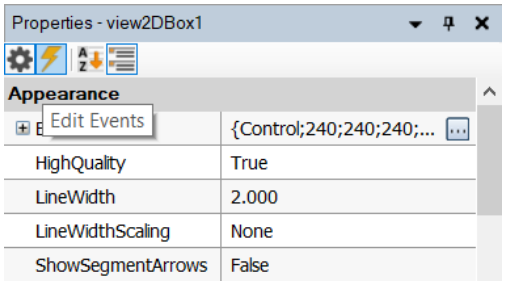
Event Handlers

An event handler is a subprogram in Aurora Vision Studio that contains the actions to be performed when an associated event occurs. In fact, it is a macrofilter that is executed once for each received event. If there are several HMI controls whose events should trigger the same procedure, one event-handling macrofilter can be used for all of them.

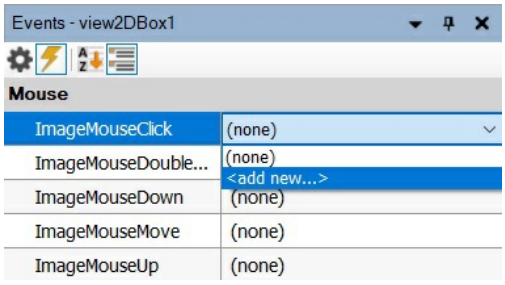
Creating Event Handlers

To create an event handler:

1. Left-click the HMI control that will be the source of an event and go to its Properties.
2. Switch to the Events window by clicking the icon:

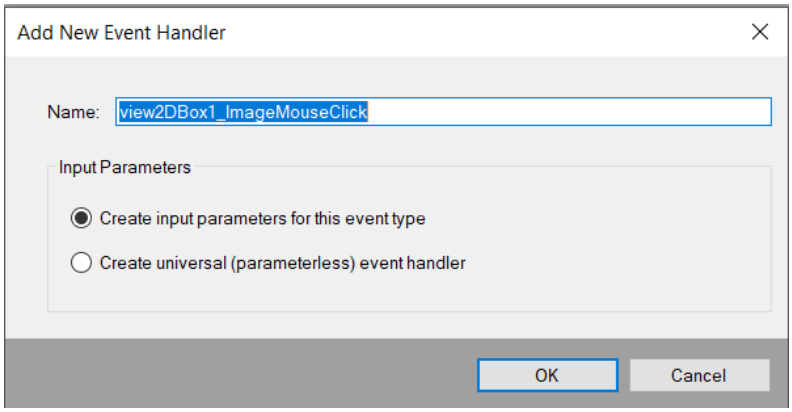


3. This will open the Events window with most commonly used events that can be handled. Click the drop-down list next to the event name and select <add new...> as shown below:

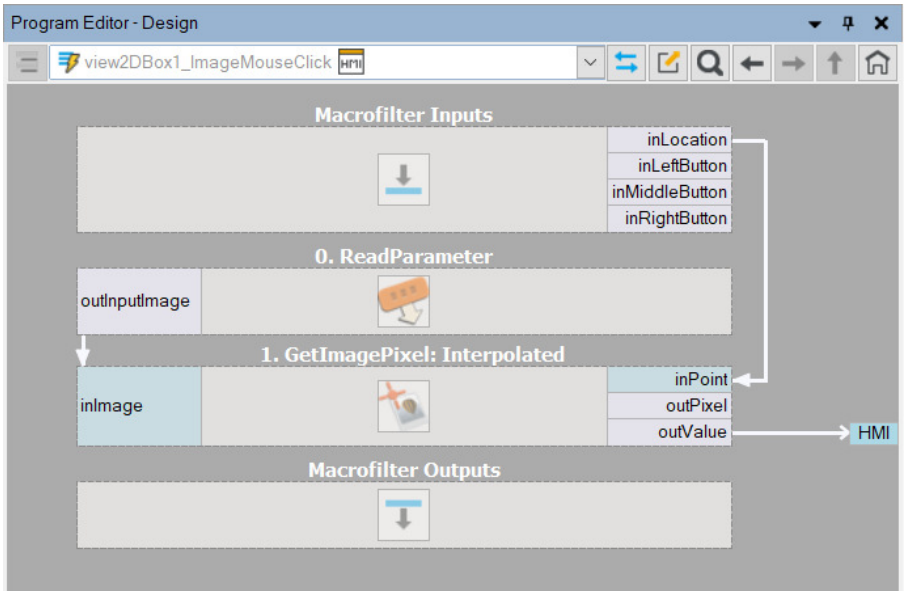


NOTE: There are many more events available. If you want to see the complete list, right-click on the HMI Control, select Edit Ports and Events Visibility... and go to the Events tab.

4. Define the name of the event handler in the pop-up window:

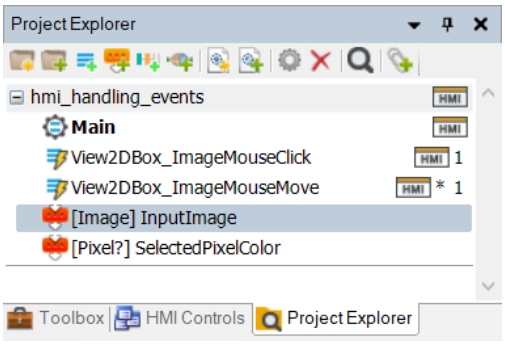


5. Some types of events, e.g. mouse move, can send additional information about the performed action. When creating an event handler you need to specify whether input parameters (containing additional information) should be created. If you select this option, the created event handler will have some inputs that can be used in the event-handling algorithm. A practical example of using input parameters is the mouse click event handler on View2DBox HMI control:



NOTE: When input parameters are used, the event handler is assigned only to the HMI control it was created for, and cannot be used for other controls. To create a universal event-handling macrofilter, which can contain an algorithm executed for several different HMI controls, choose the second option in "Add New Event Handler" pop-up window.

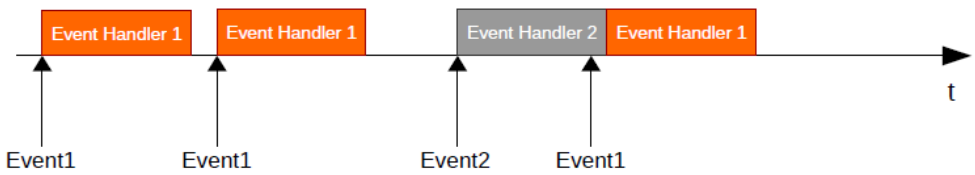
6. Once the event handler is created, it is available in the Project Explorer:



Queuing Event Handlers

To create applications that handle events efficiently, it is important to understand the mechanism of event-handling macrofilters execution. Especially the way how the HMI subsystem, mentioned earlier in this article, queues them when several events come at short intervals.

When an HMI Control sends an event to signal the occurrence of an action, the HMI subsystem retrieves the event and stores it in a FIFO queue. This queue is then processed in a separate thread dedicated for handling HMI events.



Sample events sequence and a timeline showing the order in which they are handled

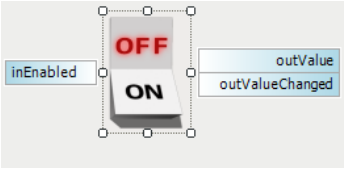
This has important implications:

- Event handlers should not contain any time-consuming algorithms that could delay the execution of subsequent events.
- Events are always handled one by one. Therefore it is essential to make event-handling procedures as short as possible.
- All event handlers are executed in parallel to the main program loop, but no two event handlers are executed at the same time.

- If you use non-free data analysis functions in event handlers, they will get counted for the ThreadLimit license restriction. This will be indicated by asterisk next to Event Handler name in Project Explorer. Without any restrictions users can use filter from following modules: FoundationLite, Genicam, ThirdParty and OpenCV.

Iteration-based Events

Apart from event handlers, it is still possible to create applications in the iteration-based model, where events are available as a standard HMI control's output. Such events are detected by the HMI subsystem independently of the program execution process but have to be handled by this process. Consequently, even though the event is detected instantly, it has to wait until the program enters any macrofilter with an incoming HMI connection related to that event. The value on this connection changes for exactly one iteration and if this particular HMI output is connected in multiple macrofilters, the first read also resets the HMI control's value.



Example of HMI control containing an event signal (outValueChanged) at its output

Until version 5.0, this approach was the only way to handle HMI events. Although event handlers ensure that an event is handled right after the action occurrence, the "old" approach remains in use. Especially in applications where:

- HMI is mainly used for displaying results and setting the parameters of an algorithm.
- HMI controls do not change the state and value of each other. For instance, the inEnabled state of one HMI control does not depend on the output of other HMI control.

Iteration-based events are typically used to fulfill the following types of requirements:

1. **Actions** – for example, when the user wants to save the current input image to a file after clicking a button.
2. **State Transitions** – when the user expects the application to switch to another mode of operation after clicking a button.

Actions

When a button (ImpulseButton) is clicked, it will change its **outValue** output from *False* to *True* for exactly one iteration. This output is typically used in one of the following ways:

1. As the forking input of a variant macrofilter, where the *True* variant contains filters that will be executed when the event occurs.
2. As the input of a formula block with a conditional operator (if-then-else or ?:) used within the formula to compute some value in a response to the event.

State Transitions

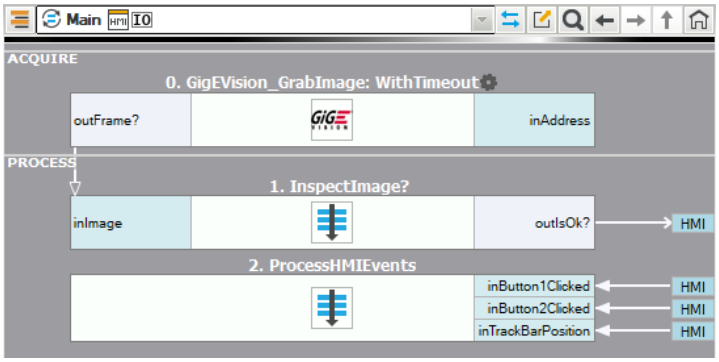
Handling events that switch the application to another mode of operation is described in the Programming Finite State Machines article.

Handling Events in Low Frame-Rate Applications

The HMI model of Aurora Vision Studio assumes that the program runs in a fast, continuous loop, with cycles not longer than 100 ms (10 frames per second). If the main loop of the application is slower, e.g. when the camera is triggered and is not receiving any trigger signal for some time, then the HMI will also be slow or halted.

To solve the problem we need to handle the image processing and the HMI events at different frequencies. The recommended approach is to use an image acquisition filter with the timeout setting (100 ms) and process the images conditionally only in some iterations, while handling the HMI events continuously, at least 10 times per second. This can be done with GigEVision_GrabImage_WithTimeout, GenICam_GrabImage_WithTimeout or with any other image acquisition filter of this kind.

Below is a sample program structure with two separate *step* macrofilters handling the image processing part and the HMI event processing part at different frequencies. Note the conditional mode of execution of the "InspectImage" macrofilter.



An example program structure with a fast cycle of HMI processing and a slow cycle of image analysis.

Saving a State of HMI Applications

The function of saving HMI applications' state allows to save the configuration, which is currently chosen in respective application controls, and to bring it back at a later time. This means, it gives the possibility of bringing back chosen settings after closing and restarting the application, but also of storing many configuration sets and to quickly restore one of them, e.g. after changing the type of inspected object.

Values stored in respective controls that are related to the parameter being edited by given control, which may influence the application execution through the connections to the program, are all considered in the saved controls' configuration. It does not necessarily have to be the visual state of an application, such as control size on the application window. One configuration set can include all controls in the application, but also only a chosen part of the controls narrowed down to the interior of proper containers.

A single application configuration is saved to a single configuration file. In other words, one file always corresponds to one saved HMI state. The location for storing configuration files can be set individually for each application being created. The format of a saved configuration file is strictly related to the controls' arrangement in the HMI application. Configuration files are not interchangeable between different applications and significant changes in the application project (like deleting existing controls and, at the same time, adding new ones of the same type) may preclude from correct loading of already existing configuration files.

In order to use the function of saving HMI applications' state, you should add to your project one (or more) controls from the State Management category. The function's behavior is parametrized by properties of such controls and the process of saving and loading application state is controlled by them. In case a state of only some of the controls present in an application should be saved, they should be placed together inside a separate container (a control from the Containers category) together with the control, which controls saving a state, and properly adjust the Range property of this control. It is possible to create many groups of controls like this and nest many containers one inside of another.

Available controls for managing the function of saving HMI applications' states are described below:

StateControlBox control

This control is an integrated application state manager. It allows the end-user to fully control the list of saved configurations, including: creating new configurations with given names, saving changes in existing configurations, loading configurations chosen from the list and removing existing configurations. There is no limitation regarding the number of saved application states.

This control consists of the following elements (from left to right):

- combo box with saved configurations,
- button for loading the configuration chosen in the combo box,
- button for saving current application state to the configuration chosen in the combo box,
- button for saving current application state to a newly created configuration with given name,
- button for removing the configuration selected in the combo box.

Each of configurations is saved as a separate file in the folder, which is associated with this control. The list of configurations visible in the control corresponds to the list of files in this folder. The location of such folder in a file system is defined by the *BaseDirectory* and *BaseSubdirectory* properties. The combo box with configurations is filled with names of all files with required extension, which are found in given folder (therefore, it is advised to use a separate folder for configurations, which does not contain any other elements). **You should not link more than one StateControlBox control with the same folder in the same application** because it may cause incorrect overwriting of configuration files, suggesting of loading configurations meant for other application parts and the fact, that controls will not be able to synchronize with the changes that they introduce.

This control has the following properties, which are essential in relation to saving an HMI state:

- **BaseDirectory** – it defines the starting location of a base folder of configuration files in a local file system. It accepts one of the predefined values:
  - **ProjectDir** – the folder where the project of currently executed application (.avproj file) or the application executable file (.avexe) is saved. In order for the control to work properly with such setting, it is necessary to save a project (**the control will not be able to work properly in an unsaved project in Aurora Vision Studio**).
  - **MyDocuments** – the system folder of user's documents (e.g.: *C:\Users\John\Documents*).
  - **UserLocalSettings** – the system folder of local settings of currently logged user (e.g.: *C:\Users\John\AppData\Local*)
  - **None** – no starting folder, the BaseSubdirectory property will be pointing the full, absolute path to the folder.

  In order to put configuration files in one of system folders, you should always use predefined values mentioned above (instead of inputting a fixed absolute path) because on different operating systems these folders may have different locations.

- **BaseSubdirectory** – it defines the path to a folder with configuration files. In case of entering a relative path (e.g.: *MyDataFiles\MyConfigurationFiles*) such location defines a subfolder in the starting folder chosen by BaseDirectory property. In case of entering an absolute path (e.g.: *D:\MyConfigurationFile\MyAppConfiguration* or *\\MyServer\\MyShare\MyConfigurations*) such value defines a full absolute folder path.
- **CreateSubdirectory** – setting it to *True* will cause folder defined by the BaseSubdirectory property to be fully created (when it does not exist yet) before saving a new configuration to it. In case of setting it to *False* (a default value), an attempt to save a configuration will fail when such folder does not exist yet.
- **ControlRange** – it defines which of the controls present in an HMI application take part in saving and restoring their configurations. It accepts the following values:
  - **Global** – all controls in an entire HMI application save and load their state as part of an action of such controller,
  - **InContainer** – only those controls in an HMI application which are located in the same container as a state controller control take part in saving and loading their states (e.g. only controls which are located in the same GroupBox control as the StateControlBox control will save their states).
- **AutoLoad** – when set to *True*, it causes chosen configuration to be automatically loaded to an HMI application after switching position on the list of saved configurations. In case of setting it to *False* (a default value), such configuration will be loaded only after clicking the "Load Configuration" button.
- **PromptBeforeSave** – when set to *True*, it causes displaying an additional message asking for action confirmation (in order to prevent from overwriting the currently chosen file accidentally) after pressing the "Save Configuration" button. When set to *False* (a default value), the configuration currently chosen on the list will be overwritten immediately after pressing the "Save Configuration" button. This property does not influence the behavior of the "Save Configuration As..." button.

Additional properties of the StateControlBox control:

- **ButtonsSize** – it allows to change the sizes of the control buttons located on the right side of the combo box. It accepts one of the following values: Small (a default value), Medium, Large.

StateControlButton control

This control resembles in appearance an ordinary button (ImpulseButton control) and provides the same properties defining its appearance. On the contrary to the ImpulseButton control, the reaction for clicking this control is performing one of the chosen actions of saving or loading application state.

On the design level this control becomes associated with one specific configuration file located in a constant path (described by the BaseDirectory and FilePath properties). It performs one of the saving or loading operations of application states on such file. On the contrary to the StateControlBox control, it does not provide an end-user with full control over the configuration list and allows to build an individual, non-standard user interface for controlling of application state saving.

This control has the following properties, crucial from the point of view of saving an HMI state:

- **BaseDirectory** – it defines the starting location of the path to a configuration file, just like the BaseDirectory property of the StateControlBox control.
- **FilePath** – it defines the path to a configuration file. It can point to a location which is relative to that one set by BaseDirectory, or to a full absolute path just like the BaseSubdirectory property of the StateControlBox control. The file name should be entered without extension (it will be added by the control automatically).
- **CreateSubdirectory** – setting it to *True* will cause the folder, which should contain the file described by the FilePath property, to be fully created (when it does not exist yet) before saving a new configuration to it. In case of setting it to *False* (a default value), an attempt to save a configuration will fail when such folder does not exist yet.
- **ButtonMode** – it defines the action, which the button should perform after clicking it. It can accept one of the following values:
  - **Save** – the current HMI application state is to be saved to the associated file.
  - **Load** – an HMI application configuration is to be loaded from the associated file.
- **ControlRange** – it defines which of the controls present in an HMI application take part in saving and loading their configurations, just like the ControlRange property in the StateControlBox control.

StateAutoLoader control

This control can be used to automatically load chosen HMI configuration on application start or to automatically save current HMI state to a configuration file on application close. StateAutoLoader should be placed inside a container control which defines its range. This control is visible only in the design mode, it becomes invisible during runtime.

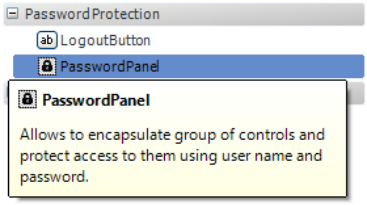
StateAutoLoader enables to create an non volatile HMI thanks to automatic loading and saving of a configuration file. Basically, it is recommended to use this control to automatically load HMI configuration on start and the StateControlButton control to manually save current HMI configuration to a file.

This control has the following properties, crucial from the point of view of saving and loading an HMI state (most of the properties are the same as those in the StateControlBox control):

- **AutoLoadOnOpen** – when set to *True*, it causes that the configuration from the file defined in the FilePath property will be loaded on application start.
- **AutoSaveOnClose** – when set to *True*, it causes that the current state of HMI will be saved to the configuration file defined in the FilePath property on application close. Please note that a configuration will not be saved if the application is closed in an unexpected way (e.g. on a power loss).
- **BaseDirectory** – it defines the starting location of the path to a configuration file, just like the BaseDirectory property of the StateControlBox control.
- **FilePath** – it defines the path to a configuration file. It can point to a location which is relative to that one set by BaseDirectory, or to a full absolute path just like the BaseSubdirectory property of the StateControlBox control. The file name should be entered without extension (it will be added by the control automatically). If no file is found in the given location and AutoLoadOnOpen is set to True, the loading on application start will be ignored and controls state will remain unaffected.
- **CreateSubdirectory** – setting it to *True* will cause that the folder which should contain the file described by the FilePath property will be fully created (when it does not exist yet) before saving a new configuration to it. In case of setting it to *False* (a default value), an attempt to save a configuration will fail when such folder does not exist.
- **ControlRange** – it defines which of the controls present in an HMI application take part in saving and loading their configurations, just like the ControlRange property in the StateControlBox control.

Protecting HMI with a Password

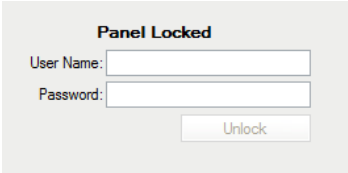
It is possible to lock the whole HMI or certain parts of it with a password in order to make sure that only authorized users will be able to modify a running program in the production environment.



PasswordPanel control in the HMI Controls catalog

Creating Password Protected Panels

In order to make some area of HMI locked for unauthorized users you need to add the PasswordPanel control from the *Password Protection* category to your HMI. Then, you should add all HMI controls that you wish to be password protected into the PasswordPanel (in exactly the same way as you would do with a regular panel). When you run such a program you will see a login screen instead of the content of the PasswordPanel (the controls inside the panel will not be visible) and only when you log in the content of the panel will be unlocked (it will become visible). Please note that you can create several password protected areas in your HMI, each with its own users list, e.g. to distinguish the administration panel from a regular worker's panel. The controls which should be available for everyone without providing a password should be placed in HMI outside of any PasswordPanels.



PasswordPanel's login page

Logging Out

After you log in to the PasswordPanel, you can log out of it (lock its content) in the following ways:

- by clicking the LogoutButton control placed inside PasswordPanel (you can add it from the *Password Protection* category),
- by clicking the *Esc* key when keyboard focus is on a control inside PasswordPanel (and when **UseEscToLock** property of PasswordPanel control is set to True),
- by clicking the *Ctrl+L* key combination when keyboard focus is on a control inside PasswordPanel.

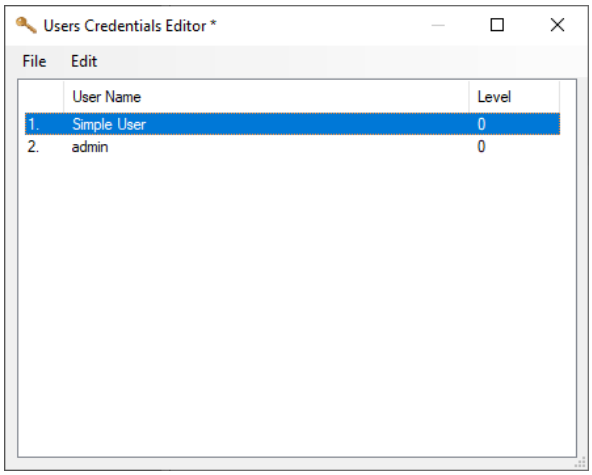
You will be also automatically logged out after the period of time defined in the **AutoLockTimeout** property (in seconds) of the PasswordPanel control if the system input has been idle for that period of time (no keystrokes or mouse clicks/moves).

Managing Users and Passwords

Users with access to HMI PasswordPanel are identified by user name and password pairs. User names must be unique and are case insensitive. User password must be at least 4 letters long and is case sensitive. Multiple users identified by different user names can be given access to a single PasswordPanel. Credentials are stored in a separate ***.avuser** files (stored outside of .avproj project files or .avexe executable files). You need to assign such a file to each PasswordPanel control to determine which users should have access to a given area. You do this by setting the **CredentialsFileName** property of the PasswordPanel control to the path pointing to the .avuser file. An ***.avuser** file can be created and edited in the following ways:

- by right clicking the PasswordPanel control in the HMI editor and choosing *Edit Credentials...* (this will open an existing file assigned to the control, if any, and set the CredentialsFileName property to the newly saved file path),
- by choosing *Tools » Edit HMI User Credentials File...* in Aurora Vision Studio's main menu,
- by running the *AuroraVisionUserCredentialsEditor.exe* application from the Aurora Vision Studio Runtime's catalog.

Please note that user passwords are stored in a hashed form and thus it is not possible to read the current password of a user (it is only possible to set new password for an existing or new user).



Aurora Vision User Credentials Editor

When moving your project to a different computer, please remember to copy also the ***.avuser** files and check if their paths are correct (absolute or relative, as set in the **CredentialsFileName** property). You need to copy these files even when you generate a runtime executable file (***.avexe**) from your project because they are not exported to ***.avexe** files. **Ensuring that is crucial as when *.avuser files are not found you will not be able to unlock the password protected area**.

It is also possible to share your ***.avuser** file through network using file sharing (in such case be sure to share files as read-only). This gives ability to quickly edit user access right inside production environment in a single location on a server by a system administrator. To use credentials file shared in a network CredentialsFileName property must be set to an absolute network share path (e.g. "\\MyServer\MyPath\MyCredentialsFile.avuser").

Password Protection in the Production Environment

Please note that it is not enough to just copy your project files (or an ***.avexe** file) and ***.avuser** files to the production environment machine to ensure that your program cannot be modified by unauthorized users. This is because ***.avuser** files and application project files can be modified or overwritten, modifying user credentials or removing password protection. In order to prevent that, you should properly configure the production machine:

- system administrator must have a password,
- a normal non-administrative user (logged in during vision system runtime) should not have permissions to modify the project catalog and the files it contains, including .avuser file.

A simple solution to fulfill these requirements on Microsoft Windows is to place application project files in a folder created with default inherited permissions inside Program Files directory (administrator rights will be required) and to work on production machine using non-administrative user account.

In such configuration it is still possible to edit your ***.avuser** file by authorized staff (for example to periodically change passwords) when you run *AuroraVisionUserCredentialsEditor.exe* with the system administrator privileges.

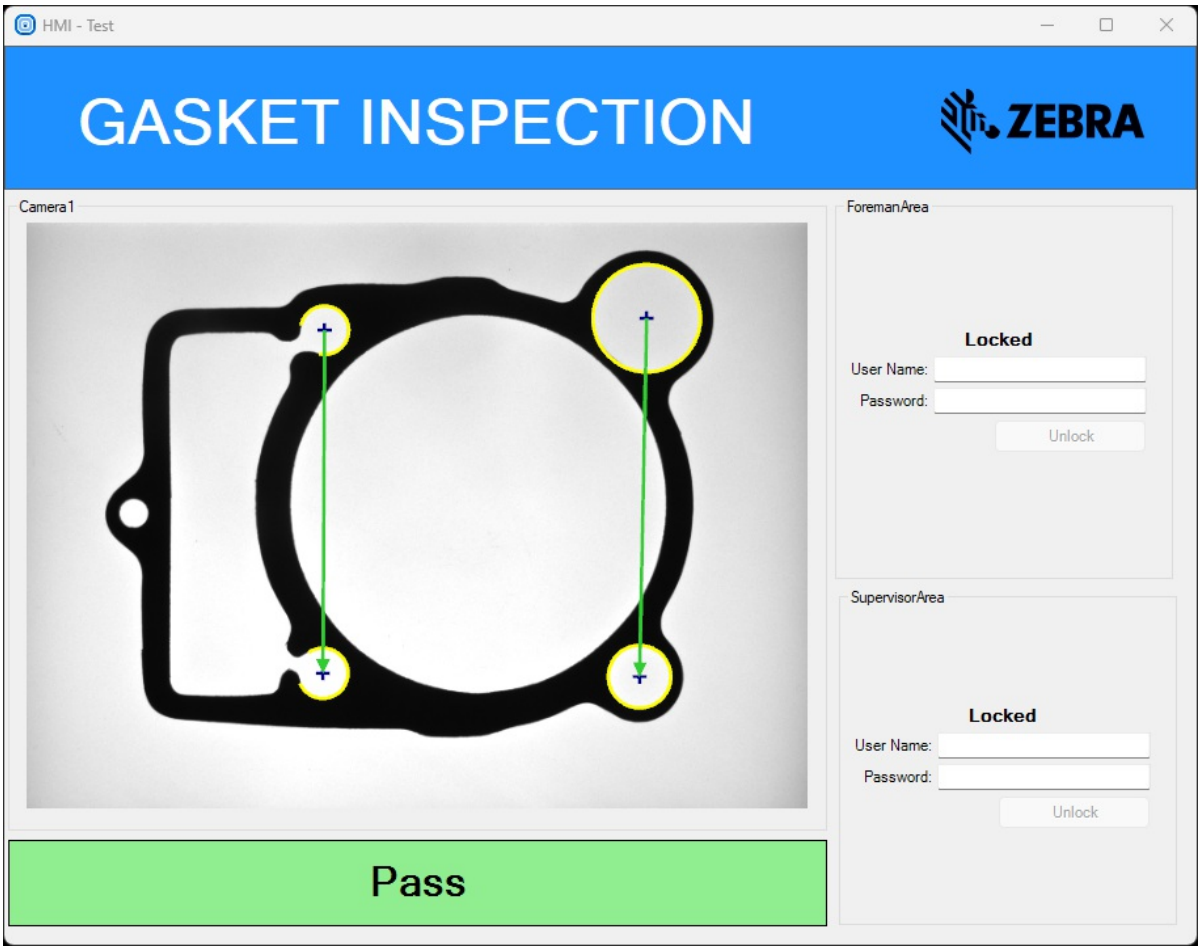
Multi-level password protection

Aurora Vision Studio allows for creating HMI with multi-level password protection. Besides the user name and password a ***.avuser** file also stores user's access level, which is an integer number between 0 and 255. When creating a multi-level protected HMI you can determine the minimum access level that is required to log in to each protected area. What is worth mentioning, you can create several PasswordPanels or even one inside another, and assign to them the same ***.avuser** file.

Example:

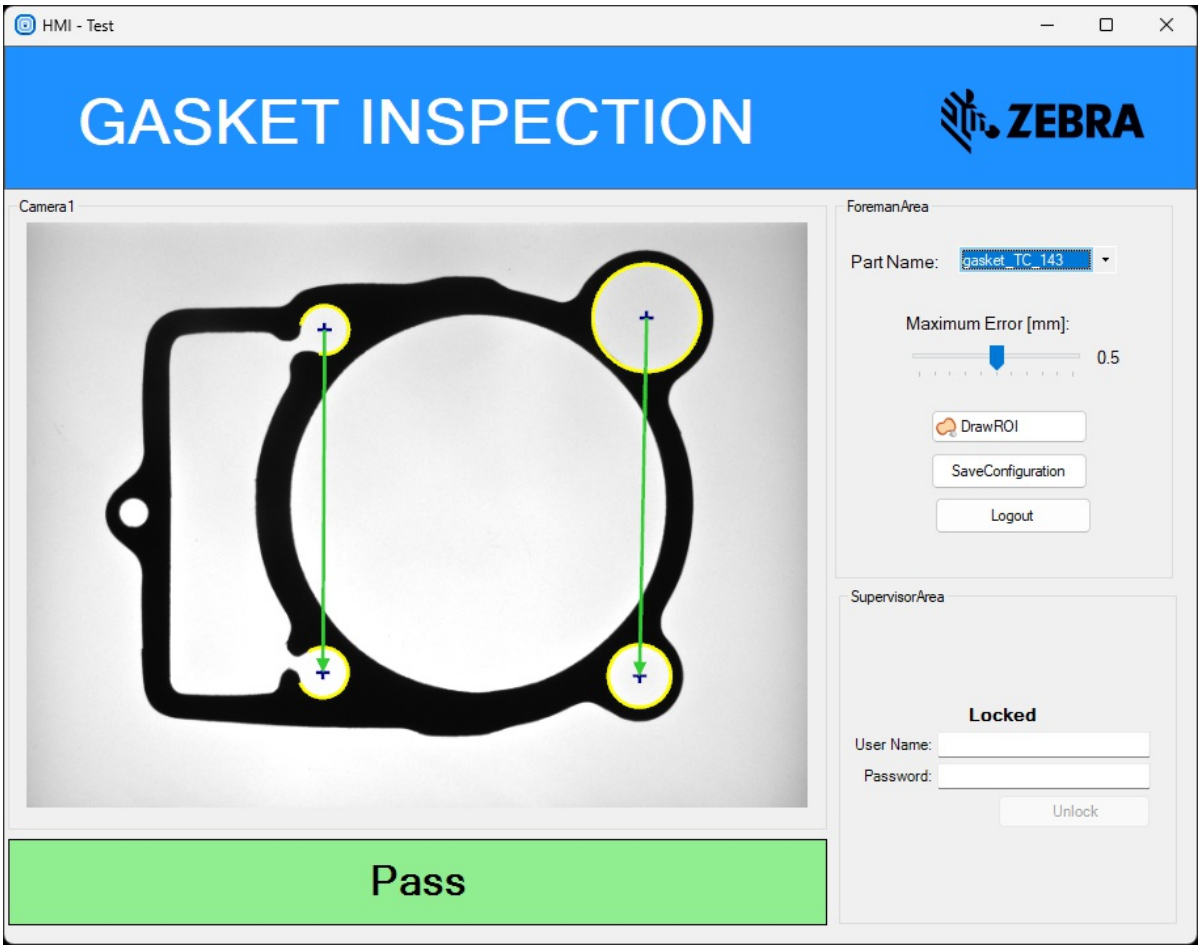
The multi-level password protection is typically used in applications with at least two PasswordPanels and at least two access levels, as in the example below.

The unlocked area for the unauthorized users is only used for displaying the result of inspection.



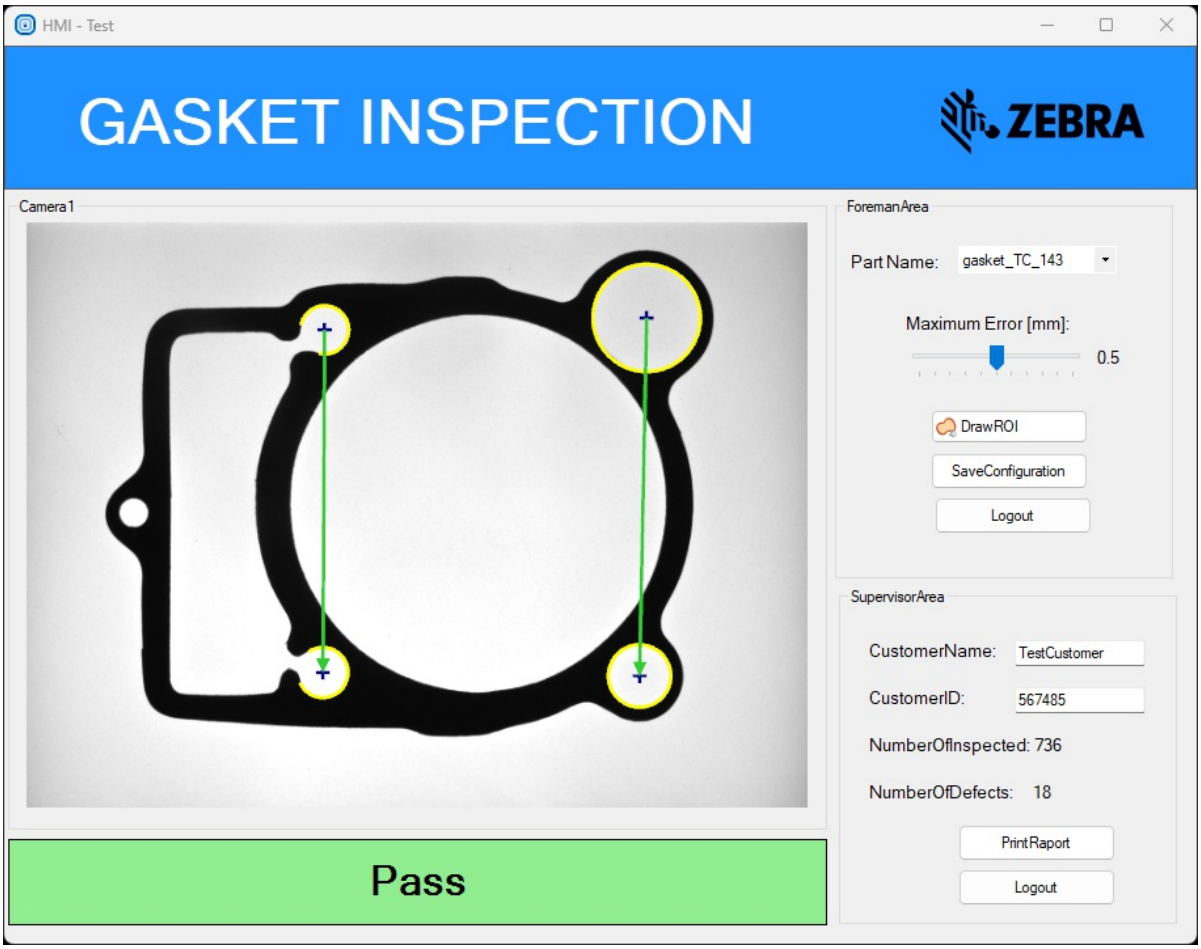
HMI with Locked ForemanArea and SupervisorArea.

The first protected area has the **RequiredAccessLevel** set to 0. It contains controls for setting parameters such as the name of an inspected element, the region of interest and the maximum error of a measurement. The values of these controls can be set by a factory foreman, having access level 0.



HMI with Unlocked ForemanArea.

The second protected area is locked even for a Foreman user, since its **RequiredAccessLevel** is set to 1, thus only users with higher access level are able to see its content. In this particular example such user is a Supervisor. Moreover, a Supervisor user is able to log into both the ForemanArea and the SupervisorArea at the same time.



HMI with Unlocked ForemanArea and SupervisorArea.

Per-control multi-level protection

Besides putting the protected controls inside a PasswordPanel control, it is also possible to define the minimum required user access level for each control individually. After adding a PasswordPanel control to the HMI project (e.g. named "passwordPanel1") all other controls will receive a new extended property **UserAccessLevel** (e.g "UserAccessLevel on passwordPanel1", one property for each PasswordPanel in the project). By default the value of this property is empty (equal to *Nil*), which means that the control does not have a user level requirement and its Enable state is not affected. By setting the value equal or greater than 0, the control becomes password protected and will be disabled (its Enable property will be set to False) until a user with an equal or greater access level will log in on the associated PasswordPanel.

Please note that when using the UserAccessLevel property on a control, the control's Enable state cannot be changed in any other way. This includes associating **EnabledManager** with it or connecting to its **Enable** property, as the methods would interfere with each other.

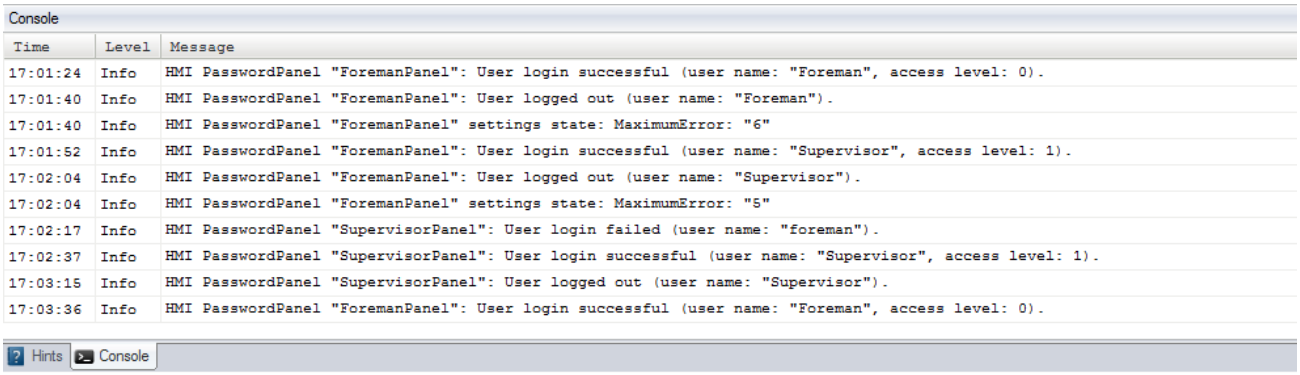
Also note, that the control is not required to be placed inside a PasswordPanel container to be able using the UserAccessLevel property. This allows for separating the protected controls from the password panels and allows for creating more flexible HMI designs.

Event Logging

All events related to each PasswordPanel are displayed in the console window and saved to the application log file. Description of the following events is saved:

- user logged in,
- user logged out,
- username or password was invalid,
- user access level was lower than the **RequiredAccessLevel**.

Moreover, at the moment the user is logging out, the values of all HMI controls within the protected area are also saved. This makes it possible to conduct audit trials about who changed particular parameters and at what time.



Console with events logged for audit trial purposes.

Creating User Controls

- Introduction
- Serialization of a Control to a File
- Descriptions Added to a Control
- Editing Properties in the HMI Designer
- Creating Modules of User Controls
  - Prerequisites
  - Creating Modules of Controls
  - Creating Projects of User Controls in Microsoft Visual Studio
- Defining Control Ports
  - Conditional Data Types of Ports
  - Data Ports out of Control Events
- Conversion to AMR
- Saving Controls State
- Interoperability with Extended HMI Services
  - Controlling Program Execution and Reactions to Changes in Program Execution
  - Controls Managing Saving of the HMI State
  - Controlling On-Screen Virtual Keyboard

Introduction

The HMI system of Aurora Vision Studio makes it possible to design user interfaces by composing it from ready-made components, which are called controls. It is possible to extend this functionality by adding new, external controls by using a mechanism of user controls.

HMI is based on the Windows Forms library from the .NET Framework 4.0. The included controls are compatible with Windows Forms controls, appropriately extended for use in Aurora Vision Studio.

First of all designing new HMI controls mainly consists of creating correct controls for the Windows Forms system, according with the requirements specified in the document Developing Custom Windows Forms Controls with the .NET Framework. There are three possible approaches to creating user controls:

- composite controls (composition of several controls into a new one),
- extending controls (extending an existing control with new functionality),
- custom control (a control created from scratch).

Details of implementation of those methods are presented in the document: Varieties of Custom Controls.

A single HMI control is represented by a single, complete and public class derived from System.Windows.Forms.Control or from one of its derived classes. Despite the created components are called user controls, it is not necessary to derive it from System.Windows.Forms.UserControl. This class is used only when implementing composite controls.

Serialization of a Control to a File

HMI layout together with controls settings are saved to a .avhmi file. A control is saved to a file in a way consistent with the Windows Forms principles. Saving of a control is realized by saving values from its public properties. Only properties designed for that purpose are saved and only when their values differ from default values.

To control if property is supposed to be subject of serialization System.ComponentModel.DesignerSerializationVisibilityAttribute attribute should be used.

A Control can signalize if properties has been changed against their default values in one of the following two ways:

- by marking given property with System.ComponentModel.DefaultValueAttribute attribute which specify constant default value, or
- by implementation of ShouldSerialize*XXX*, Reset*XXX* mechanism.

Only one from the above mechanisms should be used. Not using any of them will cause a control property to be always serialized and its value may not be reset in editor.

During loading of HMI from a file, control properties are initialized in unspecified order. If a control can enter improper state when properties are set with erroneous values or controls property values depends against other properties (e.g. it limits the value to a given range between minimum and maximum), then it should implement System.ComponentModel.ISupportInitialize interface with help of which HMI subsystem will initialize properties in a single transaction.

Property values are serialized to a file in an XML format and every value must be converted to text so it can be saved in an XML node. To ensure this is possible, data types used in public properties must support conversion from and to text independent of localization. Therefore, they need an assigned type converter which is capable of conversion in both directions with System.String type. Most simple

built-in types and types provided by .NET Framework (such as System.Integer or System.Drawing.Size) already support such conversion and do not require additional attention.

Descriptions Added to a Control

Control class as well as every public property of a control can be marked with System.ComponentModel.DescriptionAttribute attribute with description for an end user. Description will be shown in the editor while designing HMI.

Control class can be marked with System.Drawing.ToolboxBitmapAttribute attribute which specifies an icon image to be shown in the editor catalog.

Editing Properties in the HMI Designer

In addition to control layout (placement and size) the editor allows editing public properties of a control. Values of public properties specify configuration and initial state of controls in HMI application.

Not every property can be modified in the HMI editor. To enable property editing, the property must be public and its data type must allow conversion to AMR data. The property value is edited inside the Aurora Vision Studio environment through tools common for HMI and program editor. This data is edited in AMR representation and by controlling the way of conversion to AMR data one can affect the way of editing (e.g. by setting a permitted range). A more detailed description regarding conversion to AMR can be found further in this article.

HMI editor uses its own rules for determining property visibility in the editor. To control if property should be visible for edition or hidden it should be marked with HMI.HMIBrowsableAttribute attribute with bool type value specifying its visibility in the editor. If public property of a control is not marked with this attribute then the editor automatically determines its visibility.

Creating Modules of User Controls

Prerequisites

Following prerequisites must be met for creating modules of user controls:

- Installed Aurora Vision Studio environment (in version 4.3 or higher).
- Installed Microsoft Visual Studio environment (in version 2015 or higher) with installed tools for chosen language of a .NET platform (or other environment enabling creating applications for Microsoft .NET platform based on .NET Framework 4.0).

Creating Modules of Controls

To create a module of controls one should create DLL library (Class Library) for .NET environment, based on **.NET Framework 4**. Library must make control classes available as public types. Moreover, library must make available one public class of arbitrary name which implements HMI.IUserHMIControlsProvider interface. This class has to implement GetCustomControlTypes method which is returning an array with a list of types (class instance System.Type) indicating class types of controls available through the library. Aurora Vision Studio environment will create an instance of this class to obtain a list of controls available through the module.

Required data types from HMI namespace are available through **HMI.dll** library included in an Aurora Vision Studio installation directory. User HMI controls project should add a reference to this file.

A DLL file prepared in such a way should be placed in the HMIControls folder in the installation folder of the appropriate Aurora Vision product or in the HMIControls folder in a sub-folder of the Aurora Vision product folder in user documents. Loading of the module requires restarting the Aurora Vision Studio/Executor environment.

It is recommended to compile the user controls project using "any CPU" platform thanks to which it will be feasible to use in Aurora Vision Studio editions for various processor platforms. In case of necessity to use specifically defined processor architecture it should be ensured to use modules compiled for 32-bit processors with Aurora Vision 32-bit environment and modules compiled for 64-bit processors with 64-bit environment.

Creating Projects of User Controls in Microsoft Visual Studio

To create a project of a user control module:

- In Microsoft Visual Studio create new project of **Windows Forms Class Library** type for chosen .NET platform programming language (e.g. C#), based on **.NET Framework 4**.
- While creating new project, Microsoft Visual Studio environment will add to it first control named "UserControl1" created as a composite control. If controls will be created utilizing other method this class should be removed and replaced with classes of adequate realization type.
- Add to the project reference to HMI.dll module placed in Aurora Vision Studio installation directory.
- In the project create control class definition according to your needs. Each control class must be of a public access, derived from System.Windows.Forms.Control or its derivatives.
- Add to the project one public class implementing HMI.IUserHMIControlsProvider interface and implement in it GetCustomControlTypes method.
- Compile project to a DLL library. Resulting DLL file should be placed in HMIControls folder of adequate Aurora Vision product.

If a project using additional HMI controls is created in the Aurora Vision Studio environment it should be remembered that, in order to execute such project in the Aurora Vision Executor environment, all required user control modules must also be added to adequate folders of the environments on the destination computers.

Defining Control Ports

Control ports allow creating connections and perform data synchronization between HMI and vision program. Control ports in HMI are based on control's public properties. One control property can create one input and/or output port. For instance, MyProperty property in a control can create input port inMyProperty and output port outMyProperty. Data synchronization between vision program is realized by writing a program value to a control property (using a set accessor) by the HMI subsystem. Data synchronization between HMI and vision program is realized by reading a value out of control property (using a get accessor) and passing it to the program.

In order to create an output port of a control property, it must allow reading (it must implement a public get accessor). In order to create an input port, it must allow both reading as well as writing (it must implement both get and set public accessors).

In order to create a control port based on a given property, a HMI.HMIPortPropertyAttribute attribute should be specified on the property, passing port visibility type as attribute argument. The attribute argument is a value of HMI.HMIPortDirection enumeration of bit flags, which accepts the following values:

- HMI.HMIPortDirection.Input – property is visible in the editor as a control's input port ready to connect with a program. By default, port is visible and can be hidden by a user through the port visibility editor.
- HMI.HMIPortDirection.Output – property is visible in the editor as a control's output port ready to connect with a program. By default, port is visible and can be hidden by a user through the port visibility editor.
- HMI.HMIPortDirection.HiddenInput – property is available in the editor as a control's input port, but the port is hidden by default. Port visibility can be set by a user through the port visibility editor.
- HMI.HMIPortDirection.HiddenOutput – property is available in the editor as a control's output port, but the port is hidden by default. Port visibility can be set by a user through the port visibility editor.
- HMIPortDirection.None – property is not a port (using property as a port is forbidden). Property is neither available in the port visibility editor.

It is also possible to use logic sums of above values to create bidirectional ports.

If public property is not marked with HMI.HMIPortPropertyAttribute attribute then in some cases, when the property type is a basic type, the property can be automatically shown in the port visibility editor. This means that there is a possibility for the end user to promote some properties to ports capable to be connected with the vision application. If a public property of a control should never be used as a HMI port, such property should be marked with a HMI.HMIPortPropertyAttribute attribute set to HMIPortDirection.None value.

If a value of a property used as a port can change as a result of control's internal action (e.g. as a result of editing control's content by a user) then the control should implement property value change notification mechanism. For each such property a public event of a EventHandler type (or a derived type) and with the name consistent with a property name extended with a "Changed" suffix should be created. For instance, for a property named MyProperty, MyPropertyChanged event should be created, which will be invoked by a user control implementation after each change of the property value. This mechanism is particularly necessary when given property realizes both input and output ports, and can be changed by a control itself as well by a write operation from the vision application.

In order for a control's property to be able to create a port, capable to be connected with a vision application, its data type must support conversion to an AMR representation. A port can be connected only to the data types to which conversion is possible (when it is supported by an AMR converter assigned to the property data type). You will find more on AMR conversion below.

Conditional Data Types of Ports

System of vision program types includes concept of conditional and optional types. Such types can take additional special symbol *Nil* meaning no value (empty value). Inside HMI control, based on .NET data types, *Nil* symbol is represented as null value. In order for property to be able to explicitly accept connection with conditional and optional types, it must be of data types that enables assigning null value (reference type or constructed on System.Nullable<T>).

By default, properties of basic value types and reference types cannot accept or generate conditional data (by default types are not conditional). In order for a property to be able to accept conditional types, and to be able to generate conditional types, causing conditional execution of connected filters (its data type, from vision program's point of view, was conditional) it must be explicitly marked as conditional. In order to mark property as a conditional one:

- In case of using value types (simple types or structures), it is necessary to define property of a type that can hold null value (System.Nullable<T>). For instance, to attain *Integer?* port type, a property of int? (System.Nullable<System.Int32>) type should be declared.
- In case of using reference types (e.g. System.String), which already can hold null value, property should be marked with a HMI.AmrTypeAttribute attribute, with basic (non-conditional) name of required AMR type given as an argument and with IsNillable argument set to true.

Data Ports out of Control Events

It is also possible to create a HMI output data port based on a .NET control event. This allows for easier interoperability with a standard control event notification schema known from Windows Forms. In order to create an event port, a public event must be implemented by the control class, with delegate type EventHandler (or a type derived from it). Similarly to property ports, a HMI.HMIPortPropertyAttribute attribute should be specified on the event (but only output port direction is allowed). For some kinds of events it is also possible that the event will be automatically show in the port visibility editor.

The following code will create a outMyEvent port in the HMI control:

```
[HMI.HMIPortProperty(HMI.HMIPortDirection.Output)]
public event EventHandler MyEvent;
```

Event ports are of Bool type in the AVS environment. During normal operation False value will be returned from it. After the underlying event has been invoked by the control, the port will return True value for a single read operation, effectively creating an impulse of True value for a time of a single vision program iteration. This can be used for example to control the flow in a Variant Step macrofilter.

Conversion to AMR

Data in vision application in Aurora Vision environment is represented in internal AMR format. Data in HMI controls based on .NET execution environment is represented in the form of statically typed data of .NET types. To exchange data with vision program, as well as to edit control's property data in edition tools, data must be converted between .NET types and AMR representation. This task is performed by a system of AMR converters. For a single .NET type it assigns a single converter (similarly to converters from Components system), which allows to translate data to a single or more AMR types.

In order to be able to connect control's property to a vision program, as well as to be able to edit it during development, data types of the property must posses AMR converter. HMI System posses set of built-in converters which handle the following standard types:

.NET type	Default AMR type	AMR types feasible for conversion
int (System.Int32)	Integer	Integer, Real
float (System.Single)	Real	Real, Integer
decimal (System.Decimal)	Double	Real, Double, Integer, Long
bool (System.Boolean)	Bool	Bool
string (System.String)	String	String, Integer, Long, Real, Double
System.Drawing.Size	Size	Size
System.Drawing.Point	Point2D	Point2D, Location
System.Drawing.Rectangle	Box	Box
System.Drawing.Color	Pixel	Pixel
List<int> (System.Collections.Generic.List<System.Int32>)	IntegerArray	IntegerArray
List<bool> (System.Collections.Generic.List<System.Boolean>)	BoolArray	BoolArray
List<string> (System.Collections.Generic.List<System.String>)	StringArray	StringArray, IntegerArray, LongArray, RealArray, DoubleArray, BoolArray, String?Array, Integer?Array, Long?Array, Real?Array, Double?Array, Bool?Array

Conversion of all value types in their counterparts allowing null (System.Nullable<T>) value is also possible.

Additionally, when editing in the Aurora Vision Studio environment, the editor enables edition in property grid of arbitrary enumeration type as well as types required to define appearance and behavior of controls such as System.Drawing.Image, System.Windows.Forms.AnchorStyles or System.Drawing.Font.

In case of establishing a connection between vision program and HMI control's port, AMR data type of control's port does not have to be strictly specified. Instead of that AMR converter assigned to control's property is choosing best possible conversion between port in vision program and control's property. In this way it is possible to, for instance, more precisely convert data between property of System.Decimal type and Integer and Real types, where Decimal type enables universal representation of ranges and precisions required for editing both of those types.

During edition in property grid a default AMR data type of control's property is suggested by an AMR converter. It is possible to change default AMR data type of control's property. In order to do that, HMI.AmrTypeAttribute attribute should be specified on the property and a name of new data type in AMR convention should be given as an argument. Only types supported by an AMR converter or name aliases of those types are supported (e.g. for property of System.String type assigning "File" or "Directory" AMR data types, where both are aliases of String type). HMI.AmrTypeAttribute attribute also allows to:

- assign permitted range (min...max) for numerical types (using Min and Max parameters),
- define that specified type should be represented in its conditional (T?) form or can accept *Nil* special value (property data type must allow for assigning null value).

Saving Controls State

HMI subsystem provides functionality to save an application state (settings chosen by application's end user). Saving of the state is accomplished by saving settings of each of the controls. Saving the state of a single control is realized by the same serialization mechanism as in serialization of control's properties in saving to an avhmi file. Control's state is defined by values visible on chosen control properties. In order for a control to be able to save elements of its state (its settings), the state must be fully represented in public bidirectional properties, which data types have possibility to be converted into a text (type converter handles System.String type conversion). For control's property value to be saved together with the application state, such property must be marked with the HMI.HMIStatePortAttribute attribute.

Properties are serialized in unspecified order and in context of various states of other properties. Controls must then implement properties representing state in such a way that no dependencies will appear between them which can cause errors during execution or lead a control into incorrect state. It is recommended that a control be able to correct input value of such properties without causing errors (e.g. in case of errors in state storage file or HMI application changes between saving and reading of the application state). If given control's state is visible on more than one property (e.g. numeric state can be read/written also in text form on Text property) then only one property should be subject to state saving. State saving properties can, but do not have to, be control's ports enabling connection with vision application. Similarly, properties realizing control's ports can, but do not have to, save its state.

Interoperability with Extended HMI Services

Control might need to cooperate with HMI system elements which extends beyond capability to connect its ports with a vision program, or interpreting type attributes. Such cooperation comes down to communicating with particular extended services of the HMI system. Examples of such services include program execution control or application state saving systems.

Each extended service is available through its interface instance, which is provided by the HMI system, and which is passed to controls. In order for a control to be able to obtain a reference to a service interface, it must implement HMI.IServiceConsumer interface. This interface requires control to implement one method: void SetHMIServiceProvider(System.IServiceProvider provider). HMI subsystem will call this method once during control initialization (outside design mode) and will pass extended HMI service provider instance in form of the System.IServiceProvider interface. Control can then invoke GetService method of the provider, giving as an argument an interface type of a service to which it wants to gain access. Provider will return a reference of requested interface or null if requested service is not available.

The following example shows how to obtain access to a program execution control service interface:

```
public class MyControl : Control, HMI.IServiceConsumer
{
    private HMI.IProgramControlService programControlProvider;

    public void SetHMIServiceProvider( IServiceProvider provider )
    {
        programControlProvider = (IProgramControlService)provider.GetService(typeof(IProgramControlService));
    }
}
```

Controlling Program Execution and Reactions to Changes in Program Execution

Program execution controlling consists of, among others, program starting, pausing, starting/resuming, stopping and iterating, as well as events of program transition to particular states after execution of the operations.

Any HMI control can influence program control, as well as follow changes of the program execution state. In order to do that, control must obtain access to a program control service, by acquiring HMI.IProgramControlService interface. This interface provides following elements:

- **ProgramStateChanged** event, with a help of which control will be notified about changes of the program execution state. As a part of a call, a program execution flag IsExecuting is passed (i.e. if the program is in state of program execution, regardless of whether filters are currently processed, e.g. program is in executing state in active work or pause state, but is not executing after it was stopped or when it's loading was not completed). Remaining event parameters inform about capability of switching to appropriate execution states.
- **ProgramEvent** event, informing about interesting, from control's point of view, application life cycle events.
- **StartProgram**, **IterateProgram**, **PauseProgram**, **StopProgram** methods, with a help of which a control can force adequate change in program execution state.

Controls Managing Saving of the HMI State

It is possible to create a control which, through a mechanism of saving HMI application state, will save or load state of whole application. In order to do so, control must acquire HMI.IStateManager interface. This interface provides methods for saving whole application to an indicated file, loading whole application from an indicated file and to find project localization in local file system.

Controlling On-Screen Virtual Keyboard

HMI subsystem allows displaying a virtual on-screen keyboard, useful on systems with touch panels. The keyboard is, to a limited extent, shown and hidden automatically by the HMI environment. If required, an HMI control can explicitly control showing and hiding of the virtual keyboard. To do so, the control must obtain the HMI.IVirtualKeyboardService interface. This interface provides methods for displaying keyboards of specified types, avoiding obstruction of the visibility of a specified control or specified part of the screen, as well as methods for hiding the keyboard.

Explicit control of on-screen keyboard will work only if the HMI application designing user activated the keyboard for given HMI application. When the keyboard is not active in an application, all requests of displaying and hiding the keyboard are ignored without errors.

List of HMI Controls

Introduction

In the following article you can find all HMI controls to make it easier to find their documentation on the page.

Index

- General
  - HMICanvas
- Advanced Editors
  - Edge Model Editor
  - Gray Model Editor
  - Matrix Editor
  - OCR Model Editor
  - Text Segmentation Editor
- Components
  - Keyboard Listener
  - ToolTip
  - Virtual Keyboard
- Containers
  - Group Box
  - Panel
  - Split Container
  - Tab Control
- Controls
  - CheckBox
  - ColorPicker
  - ComboBox
  - Detailed List View
  - EnumBox
  - GenICam Address Picker
  - GigEVision Address Picker
  - ImageBox
  - ImpulseButton
  - Knob
  - Label
  - List View
  - NumericUpDown
  - OnOffButton
  - Program Control Box
  - ProgramControlButton
  - RadioButton
  - RangeTrackBar
  - TextBox
  - ToggleButton
  - TrackBar
- Deep Learning
  - StartButton
- File System
  - DirectoryPicker
  - FilePicker

- Indicators
  - ActivityIndicator
  - AnalogIndicator
  - AnalogIndicatorWithScales
  - BoolIndicatorBoard
  - PassFailIndicator
  - ProfileBox
  - View2DBox
  - View2DBox_PassFail
  - View3DBox
- Logic and Automation
  - BoolAggregator
  - EnabledManager
  - ChangeLogger
- Multiple Pages
  - MultiPanelControl
  - MultiPanelSwitchButton
- Password Protection
  - LogoutButton
  - PasswordPanel
- Shape Array Editors
  - Arc2DArrayEditor
  - ArcFittingFieldArrayEditor
  - BoxArrayEditor
  - Circle2DArrayEditor
  - CircleFittingFieldArrayEditor
  - Line2DArrayEditor
  - LocationArrayEditor
  - PathArrayEditor
  - PathFittingFieldArrayEditor
  - Point2DArrayEditor
  - Rectangle2DArrayEditor
  - Segment2DArrayEditor
  - SegmentFittingFieldArrayEditor
  - ShapeRegionArrayEditor
- Shape Editors
  - Arc2DEditor
  - ArcFittingFieldEditor
  - BoxEditor
  - Circle2DEditor
  - CircleFittingFieldEditor
  - Line2DEditor
  - LocationEditor
  - PathEditor
  - PathFittingFieldEditor
  - Point2DEditor
  - Rectangle2DEditor
  - RegionEditor
  - Segment2DEditor
  - SegmentFittingFieldEditor
  - SegmentScanFieldEditor
  - ShapeRegionEditor
  - ShapeRegionDeprecatedEditor

- State Management
  - StateAutoLoader
  - StateControlBox
  - StateControlButton
- Video Box
  - FloatingVideoWindow
  - SelectingVideoBox
  - VideoBox
  - ZoomingVideoBox

Zebra
Aurora™ Vision

This article is valid for version 5.7.5

Legal Terms of Use Privacy Policy

ZEBRA and the stylized Zebra head are trademarks of Zebra Technologies Corp., registered in many jurisdictions worldwide. All other trademarks are the property of their respective owners. ©2026 Zebra Technologies Corp. and/or its affiliates.