Zebra
Aurora™ Vision

Aurora Vision Studio 5.3

Aurora Vision Studio Documentation

Created: 6/8/2023

Product version: 5.3.4.94078

Table of content:

1. Introduction

Table of content:

What's new in 5.0?

Worker Tasks

Before 5.0, you could only have one main loop in the program and everything happened there. Now, it became possible to perform many computations in parallel! Read more at:

- Worker Tasks section of the Macrofilters article.
- Creating a Worker Task in the Project Explorer.
- Parallel Image Saving program example.
- Parallel Enumeration program example.

HMI Events

Event-based programming is now possible in our HMI Designer. You can easily create separate subprograms that will be executed when something happens—for example, when the user clicks a button, logs in or changes a specific parameter. Read more at:

- Handling HMI Events article.
- HMI Handling Events program example.

New, powerful formulas

Formulas have been here for years, but with version 5.0 they can replace the vast majority of data analysis tasks. This is possible with many new functions for arrays, geometry and with new array execution of expressions (also known as broadcasting). Read more at:

- Formulas article.
- Formulas Migration Guide to version 5.0.
- General Calculations article.

Program Editor Sections and Minimal View

Program Editor is now divided into fours sections: **INITIALIZE, ACQUIRE, PROCESS, FINALIZE**. This unified program structure greatly simplifies creation of the main program loop. We have also completely re-designed the editor and added the new Minimal View mode for easier use in basic applications. Read more at:

- Main Window Overview article.
- Data Flow Programming article.
- Button Presence program example.

Results control

This new power control is used for easy definitions of Pass/Fail criteria. You just select a filter and set the range for its numeric outputs. What is more, the Results control also collects statistics automatically. Read more at:

- Running and Analysing Programs article.
- Main Window Overview article.
- Bottle Inspector Part 1 tutorial.
- Bottle Inspector Part 2 tutorial.

Module encryption

Sometimes you need to hide the contents of some macrofilters or protect them against unauthorized access. You can achieve this by using our new module encryption function. Read more at:

- Locking Modules section of the Project Explorer article.

Product Overview

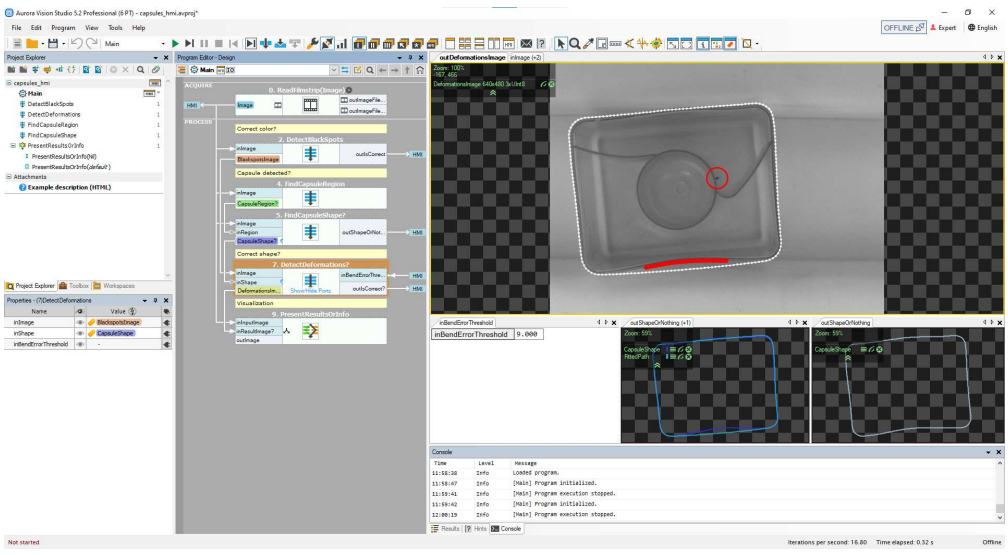
Welcome!

Thank you for choosing Aurora Vision Studio. What you have bought is not only a software package and support, but a comprehensive access to the image analysis technology. Please keep in mind that machine vision is a highly specialized domain of knowledge and mastering it takes time. However, whenever you encounter any problem, do not hesitate to contact our support team for assistance (av-support@zebra.com). We believe that working together we can solve any obstacles and make your projects highly successful. We wish you good time!

What is Aurora Vision Studio?

Aurora Vision Studio Professional is a dataflow-based visual programming environment designed for machine vision engineers. Together with its comprehensive library of highly optimized image analysis filters (tools) it allows the user to create both typical and highly customized algorithms for industrial vision systems. Furthermore, it is a complete solution as it also makes it possible to create custom graphical user interfaces (HMI).

The main design goal of Aurora Vision Studio has been clearly defined at the very beginning. It was to be the product that removes the dilemma whether to choose a tool that is powerful, or one that is easy to use, or one that is not limited to any particular hardware or application. Aurora Vision Studio has all of that – it is *intuitive*, and *powerful*, and *adaptable*.



The Main Window of Aurora Vision Studio.

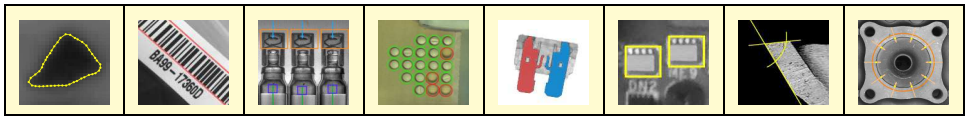
Range of Applications

The library of image analysis filters (tools) that come with Aurora Vision Studio can be described as composed of two sets:

1. a set of well-known general image processing and computer vision algorithms,
2. a set of ready-for-use tools specialized for industrial inspection systems.

In particular, these include:

- Image Processing
- Shape Fitting
- Barcode Reading
- 1D Measurements
- Histogram Analysis
- Blob Analysis
- Camera Calibration
- Data Code Reading
- 2D Measurements
- OCR
- Contour Analysis
- Fourier Analysis
- Corner Detection
- Gray-based Template Matching
- Machine Learning
- Planar Geometry
- Hough Transform
- 1D Profile Analysis
- Edge-based Template Matching
- GigE Vision and GenTL



Other Products

Aurora Vision Studio Runtime

Aurora Vision Studio is the application that is installed on the developer's computer. Programs created with Studio can be later executed (with the HMI) on other machines by using another product, Aurora Vision Studio Runtime, which includes the Aurora Vision Executor application. This lightweight application does not allow for making changes to the program and is intended for use in the production environment.

Aurora Vision Studio SMART

SMART is a simplified edition of Aurora Vision Studio, intended for use mainly by industrial end users and whenever the application is simple enough to fit into a single screen. Two main changes to the Professional edition are: (1) The program editor uses the Minimal view which hides all the connections and is better suited for small programs; (2) the Results control becomes the primary place where the user sets inspection limits (instead of formulas).

Aurora Vision Library (C++ and .NET)

Another product which is related to Aurora Vision Studio is Aurora Vision Library. While Studio is designed for rapid development, sometimes it is required to integrate the algorithms with bigger C++ or .NET projects. For that purpose the functionality of the filters of Aurora Vision Studio is also available in the form of a C++ and .NET library.

Aurora Vision Library is also worth considering for applications that require the highest possible performance. This is because there are some program optimization techniques, like in-place data processing, image memory re-use or performing initialization before the main loop, for which the complete control provided by the C++ or .NET programming languages is needed.

How to Learn?

Prerequisites

Aurora Vision Studio Professional is a drag and drop environment which makes it possible to create advanced vision inspection algorithms without writing a single line of code. Nevertheless, as a sophisticated tool and a fully-fledged visual programming language, it requires some effort before you become a power user.

Typically, the prerequisites required to start learning Aurora Vision Studio are:

- higher technical education,
- basic course in image processing and preferably also in computer vision,
- the ability to understand written technical documentation in English.

Learning Materials

The available materials for learning Aurora Vision Studio are:

- This User Manual
  – contains detailed instructions how to use the features of Aurora Vision Studio.
- The online video tutorials
  – provide a quick start for the most popular topics.
- The book "Image Analysis Techniques for Industrial Vision Systems"
  – explains the theory that lies behind the filters (tools) of Aurora Vision Studio.
- E-mail Support
  – do not hesitate to contact us if anything still remains unclear.

Additionally, it is recommended to go through the entire content of the Toolbox control and make sure that you understand when and how you can use each of the tools. This might take some time but it assures that in your projects you will use the right tools for the right tasks.

Terms Checklist

When you are ready to create your first real-life application, please review the following article to make sure that you have not missed any important detail: Summary: Common Terms that Everyone Should Understand

User Manual Conventions

There are a few formatting conventions used in the User Manual to make it clear and convenient for the reader. Here is the list of the types of terms and how they are formatted:

- **Names of filters** (tools) are written in Pascal Casing as links to filter help pages - e.g. ThresholdToRegion.
- **Names of categories** in the filter library (including the name of category such as "Region" and subcategory such as "Region Global Transforms" if needed) are written in Pascal Casing as links to the list of filters in the category - e.g. Region :: Region Global Transforms.
- **Names of Aurora Vision types** are written in Pascal Casing as links to filter help page - e.g. Point2D.
- **Values and other terms from Aurora Vision Library** are written using Italics - e.g. *False*, *NearestNeighbour*.
- **Names of Machine Vision techniques** are written in Pascal Casing - e.g. Template Matching, Blob Analysis.
- **Macrofilter names** are written using quotation marks - e.g. "Main", "DetectCapsule".
- **GUI elements of Aurora Vision Studio** are written in Pascal Casing - e.g. the Project Explorer, the Filter Catalog.
- **Paths** (to certain actions or files on disk) are written using Italics - e.g. *Start » All Programs » Aurora Vision » Aurora Vision Studio Professional » Uninstall* , *My Documents\Aurora Vision Studio Professional\Filters*.
- **Port names** are written using Bold - e.g. **inA**, **outImage**.

2. Getting Started

Table of content:

Installation

Aurora Vision Studio system requirements

Aurora Vision Studio is supported on Windows 10 64-bit operating system, including embedded editions. It requires .NET 4.8 environment to be installed on your computer (if you happen to encounter any problems with running Aurora Vision Studio on the mentioned systems, it is highly possible that .NET is not installed properly, so please reinstall or repair it then).

To be able to fully take advantage of the integration possibilities provided by Aurora Vision Studio (e.g. creating user filters(tools)) you will also need Microsoft Visual Studio development environment, version 2015 or higher (Express Editions are also supported).

Will Aurora Vision Studio work on my PC?

We intentionally do not indicate the minimum system requirements. Rather than "will it work?", we should ask about "how will it work?" instead. The performance of your application might depend on a lot of factors. The most important among them are:

- Hardware. CPU and GPU benchmarks can be found in under this link. We hope this will help you to evaluate the hardware impact on the performance of your algorithm.
- Complexity of your algorithm (Number of tools)
- Your programming skills (Appropriate choice of the tools and how they are parametrized - especially important when more advanced tools i.e. Template Matching are used)
- Speed requirements of your vision system
- Size of the input data
- Capacity and configuration of the network

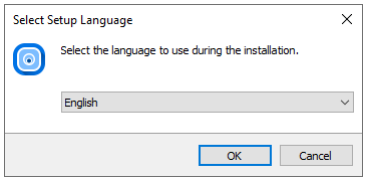
Installation Procedure

Important: Aurora Vision Studio setup program requires the user to have administrative privileges.

The installation procedure consists of several straightforward steps:

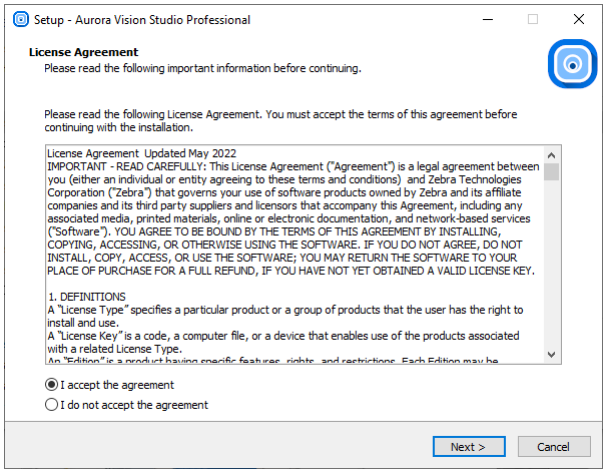
Setup language

Please choose the language for the installation process and click *OK* to proceed.



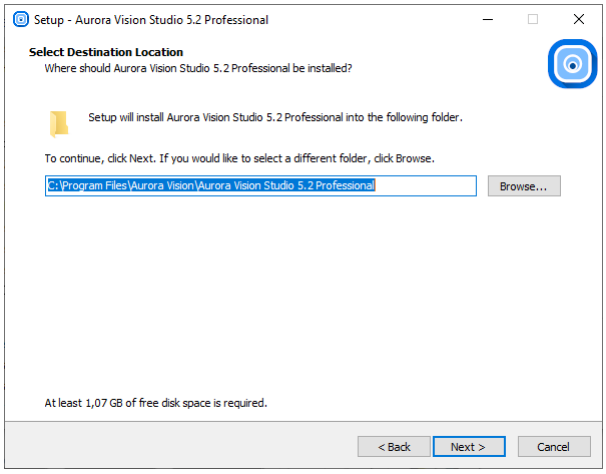
License agreement

Click *I accept the agreement* if you agree with the license agreement and then click *Next* to continue.



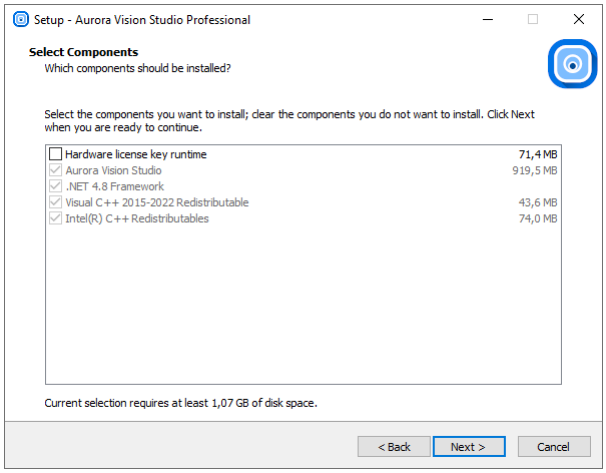
Localization on disk

Choose where Aurora Vision Studio should be installed on your disk.



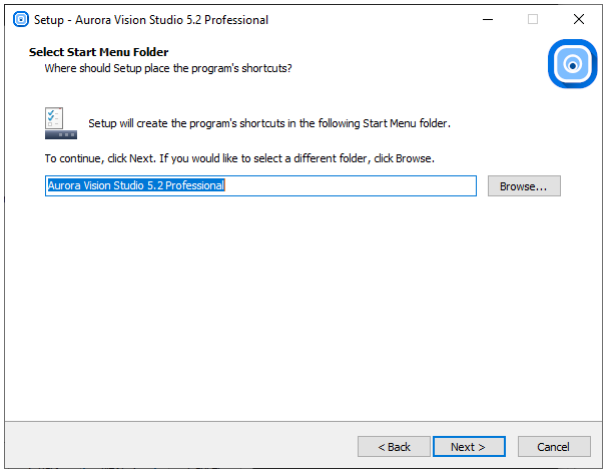
Components

Choose which additional components you wish to install.



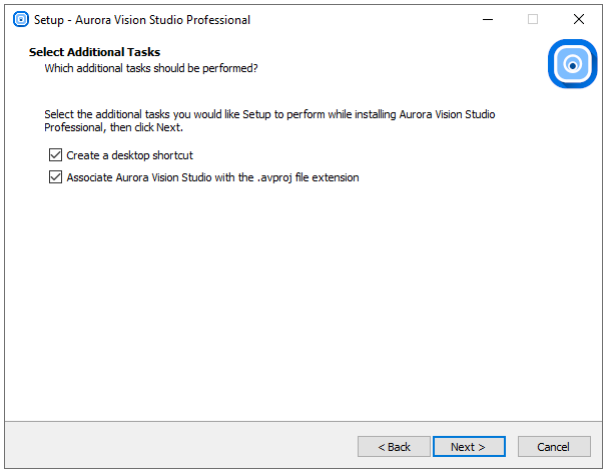
Start Menu shortcut

Choose if Aurora Vision Studio should create a Start Menu shortcut.



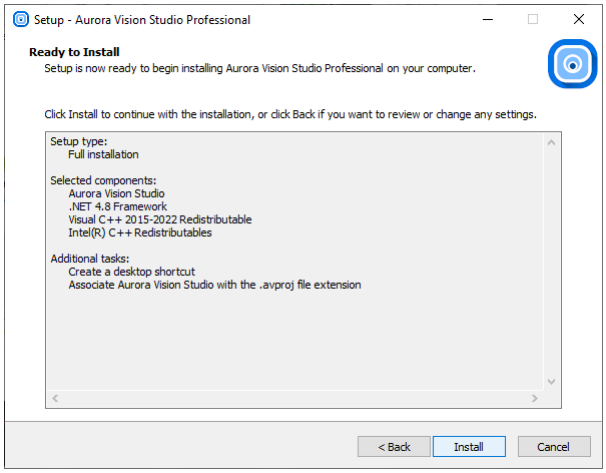
Additional Options

On this screen you can decide if the application should create a Desktop shortcut. You can also decide to associate Aurora Vision Studio with the .avproj files.

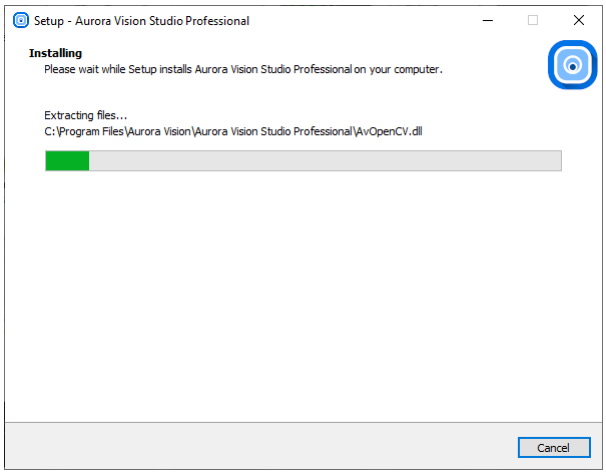


Installing

Click *Install* to continue with the installation or *Back* to review or change any settings.

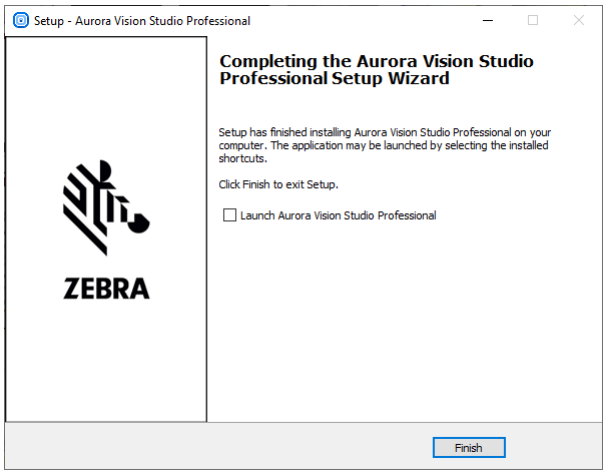


Please wait until all files are copied.



Final Options

At the end of the installation you are able to run the application.



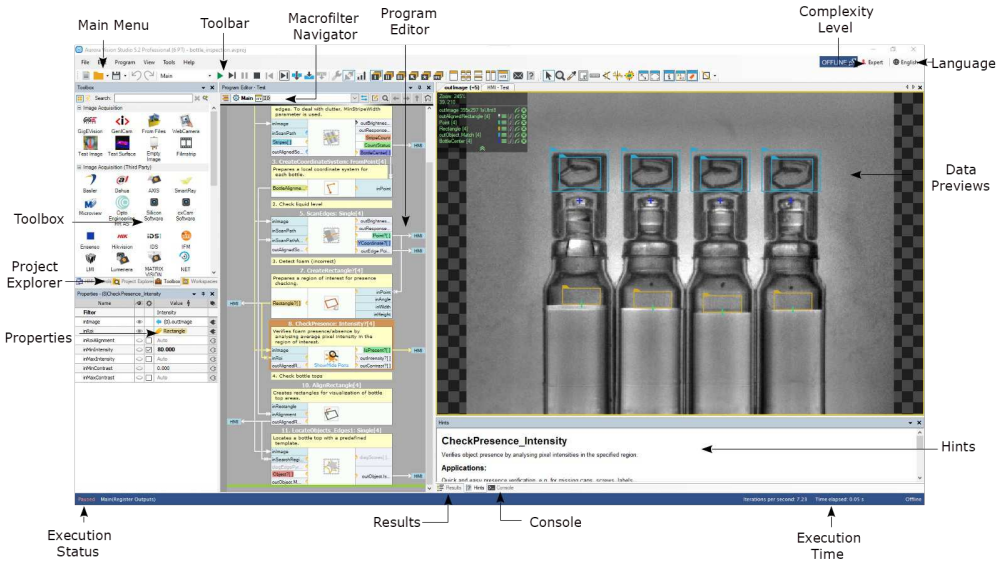
Deinstallation procedure

To remove Aurora Vision Studio from your computer please launch *Add or remove programs*, select *Aurora Vision Studio Professional » Uninstall* and follow the on-screen instructions.

Main Window Overview

Elements of the User Interface

The user interface of Aurora Vision Studio has been carefully designed for optimal user experience. All the main elements of the application are available on a single screen, as depicted below. The standard layout of windows can be changed in an arbitrary way – use this feature to adapt the application to your preferences. In particular, it is advisable to work with two HD monitors (or one curved ultra-wide, 34") and use the second monitor to display undocked data previews or the HMI window.



Elements of the User Interface.

The purpose of the individual controls, going clockwise starting from the top left corner, is as follows:

- **Main Menu** and **Toolbar**
  This is a standard application menu that contains all major actions and options, as well as a control bar which provides convenient access to the most common actions.

  See also: Main Menu and Application Toolbar
- **Macrofilter Navigator**
  This control displays the tree of the macrofilter instances contained within the selected program. By default it is attached to the top of Program Editor, but can also be undocked and placed anywhere in the main window.
- **Program Editor**
  The Program Editor is the area within Aurora Vision Studio in which filters (tools) are placed and connected to create programs. To insert a filter there, just drag and drop one from the Toolbox or double-click on it.

  See also: Program Editor
- **Complexity Level**
  This option controls the amount of features that are available according to the user's personal experience and needs.
  See also: Complexity Levels
- **Language**
  This option controls the language of the user interface (English, German, Japanese, Polish, Simplified Chinese and Traditional Chinese).

  See also: Optimizing Image Analysis for Speed
- **Data Previews**
  These panels display the data computed by filters. If you drag and drop inputs and outputs of various filters, this is where you will see the corresponding data. Extra options can be found in the Toolbar.

  See also: Preview and Data Presenting
- **Hints**
  This window provides hints as for the use and function of various filters as well as possible solutions to the most common problems.
- **Execution Time**
  It displays the total execution time of the ongoing execution process.
- **Results**
  The Results window has two functions. First, it displays numerical and textual outputs of the selected filter as well as the relevant statistics. Second, it allows for setting limits upon output values, thus providing a quick and easy way for telling OK objects from NOK ones during inspection.
- **Console**
  The Console window informs the user about events related to project editing and execution. It is particularly important for finding errors.
- **Execution Status**
  It informs the user whether the program is running, paused or stopped. During execution it also displays the current program location (the call-stack).
- **Properties**
  The Properties window is where the parameters of filters and HMI controls are set.
- **Project Explorer**
  It displays a list of modules, macrofilters, global parameters and attachments contained within the currently edited project. Please note that you only see a list of definitions of macrofilters here, not their instances. One macrofilter (design) can have multiple instances (uses) and individual instances can be browsed in the Macrofilter Navigator control.

  See also: Browsing Macrofilters
- **Toolbox**
  It is a task-oriented catalog of filters commonly used in machine vision applications.

  See also: Toolbox

Program Display

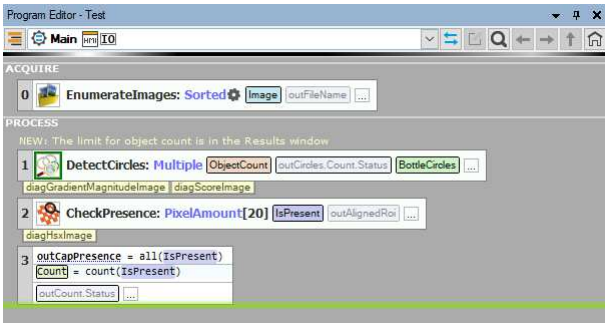
The Program Editor in Aurora Vision Studio is available with three display modes featuring different level of details. To change the display mode, expand the *View* bar and select **Minimal**, **Compact** or **Full**; each of them will suit various users to different degrees.

- **Minimal**

  Minimal program view is recommended for small-scale applications, typically ones that fit into one screen. In this mode, connections between filters are hidden and named output labels are used instead. The main benefit here is that basic applications can be easily presented in a single view. On the other hand when the program is getting bigger, dependencies between different program elements may become less clear and the program becomes more difficult to analyze.

  In Minimal program view you create connections by naming outputs and selecting them from a drop-down menu in the Properties window. The drag & drop action between filters is also possible, but the connection still remains hidden.

  This view is the only one available in the SMART edition.



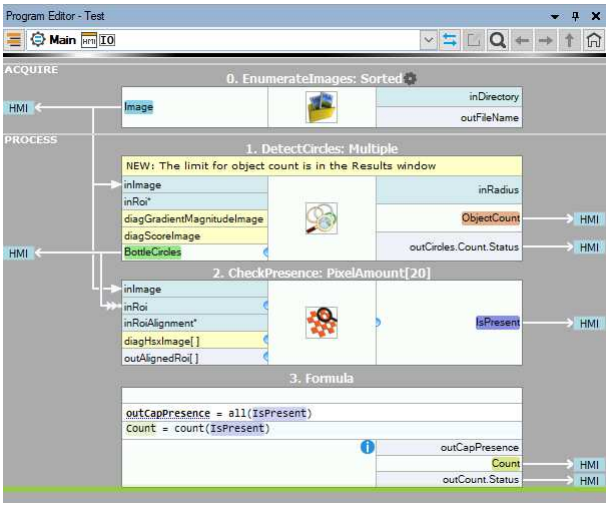
The Minimal view in the Bottle Crate example.

- **Compact**

  Compact program view is the default one in the Professional edition. It aims to provide the optimal level of detail by displaying explicit connections between filters while still hiding some inputs and outputs that are usually not being

connected (you can use the Show/Hide action to change that for any port).

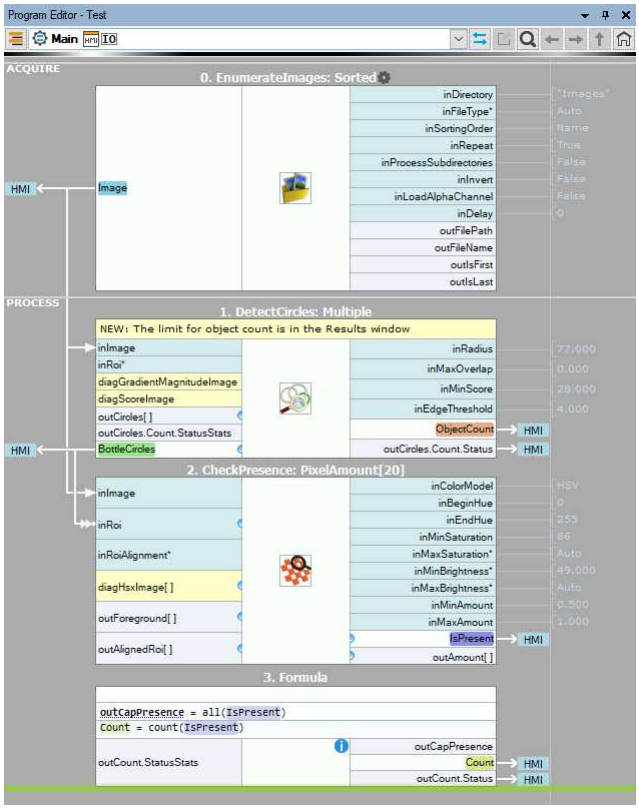
This mode suits well both simple and complex applications alike.



The Compact view in the Bottle Crate example.

- **Full**

  Unlike in the Compact mode, here all the inputs and outputs of the given filter are always visible. Most importantly, this mode provides a handy preview of the filter's properties directly in the Program Editor – whenever possible they are displayed on the right margin of the editor. This, however, comes at the cost of higher verbosity which may impact readability.



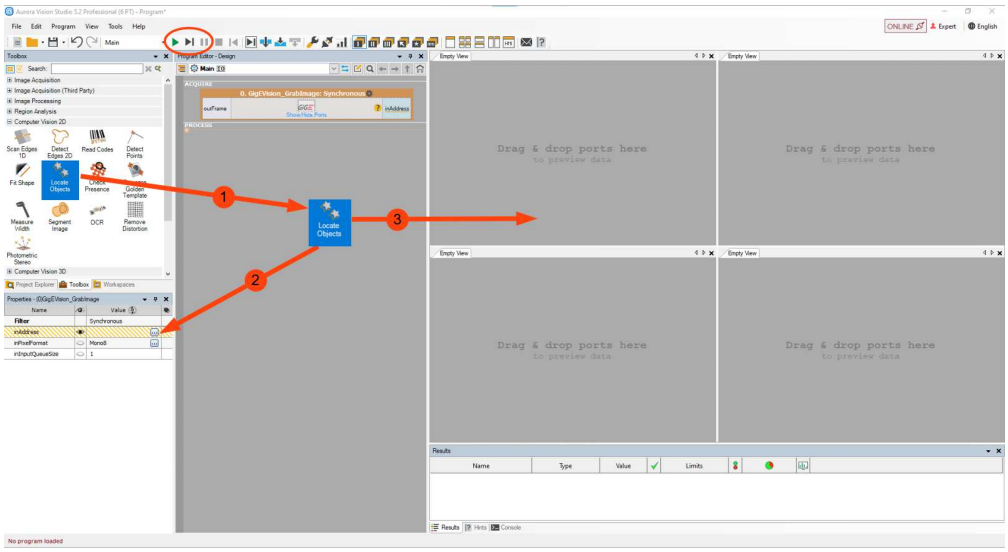
The Full view in the Bottle Crate example.

The Basic Workflow

Creating vision algorithms in Aurora Vision Studio comes down to three intuitive steps:

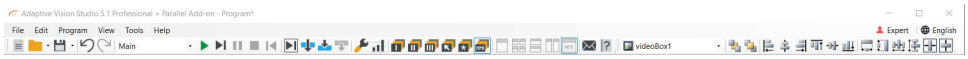
1. Drag & drop (or double-click) filters from the Toolbox to the Program Editor.
2. Drag & drop connections between filters or set constant input values in the Properties window.
3. Drag & drop filter outputs to the Data Previews panels.

Whenever a part of a program is ready, click *Run* or *Iterate* buttons in order to test it. The Console window at the bottom will also display some important information concerning the execution. Keep an eye on it – especially when something goes wrong.



The workflow of Aurora Vision Studio.

Main Menu and Application Toolbar

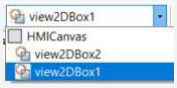


The Application Toolbar contains buttons for most commonly used actions and settings.

Here is the list of the most important menu commands:

File			
	New	Opens a new project.	Ctrl+N
	Open	Opens an existing project.	Ctrl+O
	Open Example	Opens an existing example project.	
	Open Tutorial	Opens an existing tutorial project.	
	Save	Saves the current project.	Ctrl+S
	Save As	Saves the current project in a specified location.	
	Connect to Remote Executor	Opens a window that provides connection with a remote Aurora Vision Executor, allowing application deployment of the current project. See also Remote Executor for details.	
	Export to Runtime Executable	Exports the current project to an executable file which can be run (but not edited) with Aurora Vision Executor.	
	Generate C++ Code	Creates a C++ program for the currently opened Aurora Vision program. See also C++ Code Generator for details.	
	Generate .NET Macrofilter Interface	Generates a .NET assembly which provides the chosen macrofilters as class methods. Generated assembly may be referenced in managed languages: C# or Visual Basic .NET. See .NET Macrofilter Interface Generator for more information.	
Edit			
	Undo	Reverts the last performed operation.	Ctrl+Z
	Redo	Re-performs the operation reverted with Undo command.	Ctrl+Y
	Rename Current Macrofilter	Renames current macrofilter.	F2
	Remove HMI	Unbinds HMI from the current project and clears all the HMI controls.	
Program			
Main Main CreateModel	Startup Program (combo box)	Selects a worker task from which the execution process will start. When the execution is paused, shows active worker tasks an allows the user to switch to them. See Testing and Debugging for more details.	
	Run	Executes the program until all iterations are finished, the user presses *Pause* or *Stop* buttons or something else pauses program. See Testing and Debugging for more details.	F5
	Run Single Worker	Executes the program running only the primary worker task until all iterations are finished, the user presses *Pause* or *Stop* buttons or something else pauses program.	F5
	Iterate	Executes the program to the end of a single iteration of the outermost Task. See also Execution Process and Testing and Debugging for more information about iterations.	F6
	Pause	Suspends the current program execution immediately after the last invoked filter (tool) is finished.	Ctrl+Alt+Pause
	Stop	Stops the program immediately after the last invoked filter is finished.	Shift+F5

	Iterate Back	Executes the program to the end of a single iteration, reversing direction of enumerating filters. See Testing and Debugging for more details.	Shift+F6
	Iterate Current Macrofilter	Executes the current program to the end of the currently selected macrofilter. See Testing and Debugging for more details.	Ctrl+F10
	Step Over	Executes the next single instance of a filter or a macrofilter, without entering into the macrofilter.	F10
	Step Into	Executes the next single filter instance. If it is an instance of a macrofilter it enters inside of it.	F11
	Step Out	Executes all filters till the end of the current macrofilter, and exits to the parent macrofilter.	Shift+F11
	Run with Aurora Vision Executor	Executes the program using Aurora Vision Executor application installed on the local machine.	Ctrl+F5
	Diagnostic Mode	Turns on or off Diagnostic Mode, which controls whether filters compute additional information helpful for program debugging, but resulting in a slower program execution.	
	Program Statistics	Shows information about execution time of each filter in the selected macrofilter.	F8
View			
	Program Editor	Switches the Program Editor window visibility.	
	Filter Catalog	Switches the Filter Catalog window visibility.	
	Console	Switches the Console window visibility.	
	Filter Properties	Switches the Filter Properties window visibility.	
	Project Explorer	Switches the Project Explorer window visibility.	
	HMI Controls	Switches the HMI Controls window visibility.	
	Toolbox	Switches the Toolbox window visibility.	
	HMI Designer	Switches the HMI Designer window visibility. See Designing HMI for details.	
	Hints	Switches the Hints window visibility.	
	Dock Open Windows	Docks all undocked data preview windows.	
	Program Display: Minimal	Switches Program Editor into a mode where no inputs or outputs are displayed, and connections are created by selecting data sources in the Properties window.	
	Program Display: Compact	Switches Program Editor into a mode where primary inputs and outputs are visible and can be connected in a visual way.	
	Program Display: Full	Switches Program Editor into a mode where all inputs and outputs are visible together with input values preview on the editor's margin.	
	1x1 Preview	Arranges data previews in a single tabbed window.	
	2x2 Preview	Arranges data previews in the 2 x 2 layout.	
	Arrange Horizontally	Arranges data previews horizontally.	
	Arrange Vertically	Arranges data previews vertically.	
	User-defined Preview Layout 1	Activates the first preview layout defined by the user.	
	User-defined Preview Layout 2	Activates the second preview layout defined by the user.	
	User-defined Preview Layout 3	Activates the third preview layout defined by the user.	
	Auto Preview Layout	Activates a preview layout which will be automatically filled in accordance with the currently selected filter.	
	HMI Design Layout	Activates a preview layout containing only the HMI window.	
Tools			
	Check Project for Issues	Checks if current project has any issues.	
	Manage GenICam Devices	Opens a manager for enumerating and configuring GenICam/GenTL compatible cameras.	
	Manage GigE Vision Devices	Opens a manager for enumerating and configuring GigE Vision compatible cameras.	
	Macrofilters Preview Generator	Saves a graphical overview of selected macrofilters.	
	Edit HMI User Credentials File	Opens a window which allows to configure the credentials of password-protected HMI. See Protecting HMI with a Password for details.	
	Settings	Opens a window which allows to customize the settings of Aurora Vision Studio.	
Help			

	View Help	Opens up the documentation of Aurora Vision Studio.	F1
	Message to Support	Sends an e-mail to support with attached Aurora Vision Studio screenshot, log and optional user supplied message.	
	Download Remote Support Client	Downloads TeamViewer application that allows for remote support.	
	License Manager	Allows to view and manage available licenses for Aurora Vision products.	
	Check for Updates	Checks if there is newer version of Aurora Vision Studio.	
	About Aurora Vision Studio	Displays information about your copy of Aurora Vision Studio, e.g. the application version, loaded assemblies and plugins.	
HMI Designer			
	HMI elements list	Selects a HMI element which will be edited.	
	Send To Back	Sends the HMI control to back.	
	Bring To Front	Brings the HMI control to front.	
	Align Lefts	Aligns all marked HMI controls' lefts.	
	Align Centers	Aligns all marked HMI controls' centers.	
	Align Centers	Aligns all marked HMI controls' rights.	
	Align Tops	Aligns all marked HMI controls' tops.	
	Align Middles	Aligns all marked HMI controls' middles.	
	Align Bottoms	Aligns all marked HMI controls' bottoms.	
	Make Same Width	Makes all marked HMI controls' same width.	
	Make Same Height	Makes all marked HMI controls' same height.	
	Make Horizontal Spacing Equal	Makes all marked HMI controls' horizontal spacing equal.	
	Make Vertical Spacing Equal	Makes all marked HMI controls' vertical spacing equal.	
	Center horizontally	Centers the HMI control horizontally.	
	Center vertically	Centers the HMI control vertically.	

Application Settings

Aurora Vision Studio is a customizable environment and all its settings can be adjusted in the Settings window, located in the *Tools » Settings* menu. Falling back to defaults is possible with the *Reset Environment* button located at the bottom of the window. Here is the list of the Application Settings:

1. Environment
   - General
   - Previews
   - Library
   - Filter Properties
   - Macrofilter Navigator
   - View
   - Project Explorer
   - Show these messages:
   - Startup
   - General
   - Filter Catalog
   - Project Explorer
   - Program Editor
   - Console
   - HMI
   - Toolbox
   - Default Program
   - Program Analysis
2. Program Execution
3. Filters
4. File Associations
5. Program Edition
6. Results
7. Project Files
   - Project Binary Files
8. Messages

1. Environment

General

Complexity level:
Selecting tools level complexity.

Language:
User interface language.

Theme:
User interface color theme.

Display short project name in the Title Bar
If checked, only name of the current project is displayed in the Main Window title. Otherwise, full path to the project file is displayed.

Floating point significant digits:
Sets the number of digits after floating point separator.

Startup

Load last project on application startup
Loads previously opened program at AVStudio startup.

Console

Show date column
Adds or removes 'Date' column in the Console.

Show time column
Adds or removes 'Time' column in the Console.

Show message level column
Adds or removes 'Level' column in the Console.

Previews

Display region border in image previews
Displays region bounding rectangle.

2. Program Execution

General

Break before undefined mandatory inputs
If checked, automatically breaks the program when the filter with undefined mandatory inputs is about to be executed.

Edit undefined mandatory inputs on break
If checked, tries to edit missing input data when program execution breaks before undefined mandatory inputs.

Break when an exception occurred
Automatically breaks the program when exception occurred.

Break when a warning occurred
Automatically breaks the program when warning occurred.

Break on assertion failed
Automatically breaks the program when assertion failed.

Diagnostic mode
Enables or disables computation of diagnostic output values.

Offline mode
Enables or disables the offline mode, in which data for filter outputs within the ACQUIRE section is acquired from local storage.

Display performance statistics automatically when execution is finished
Automatically opens the Statistics window after program execution.

Save changes before execution
Automatically saves program before execution start.

Previews update mode:
Sets how program will be visualized in context of previews refreshing.

HMI

Display HMI in Standalone Window
Displays HMI in a separate window as if it was run in Aurora Vision Executor.

3. Filters

Library

Auto reload filters
Tracks loaded filters directories for changes and reloads filters if any has been detected.

Filter Catalog

Close open groups
Allows only one opened group at a time.

Merge filters by group
Allows AVStudio to combine several filters items into single convenient tool.

Search bar enabled
Enables searching in the Filter Catalog.

Group categories by library
Filters in Filters Catalog will be split by library origin.

Toolbox

Show small icons in toolbox
Showing small icons in the Toolbox allows to unlock Toolbox window as a small convenient set of tools.

Use library view as staring page
Showing small icons in the Toolbox allows to unlock Toolbox window as a small convenient set of tools.

Filter Properties

Show context help
Shows help for current selected property at the bottom of the Filter Properties window.

Project Explorer

Show macrofilter usage
Shows usage count of each macrofilter.

4. File Associations

Default Program

Check .avproj file association on startup
Check if current program is the default for opening .avproj Files on startup.

5. Program Edition

Macrofilter Navigator

Expanded tree view
Shows parent-child relations between macrofilters. If not checked, macrofilters will be listed without any marked relations.

Show macrofilters preview in tooltips
Shows the graphical previews of macrofilters as tooltips.

Auto hide macrofilter list
Hides the expanded list of macrofilters on selection.

Program Editor

Show warning when connecting diagnostic outputs
Enables warning on creating diagnostic connections.

Smooth variant switch
Enables animated variants switch.

Show comments
Enables filter instances comments.

Wrap comments
Wraps long comments or, when unchecked, clips comments to single line.

Wrap formulas
Expands formula blocks to display entire formulas.

Tab size
Number of spaces the tab is equal to in the inline editors, e.g. formula editor.

Block quick commands visibility:
Defines the visibility of block quick commands (for adding new inputs and outputs).

Editor Zoom [%]:
Defines a program editor font and image size.

Filter icon size:
Defines the size of the filter icon in Program Editor.

Use larger snap size for editor points
Allows easier dragging of points in editors, useful on touchscreens.

Editor view type:
Enables choosing the information amount displayed in the program editor.

Highlight compatible ports while creating connections
Highlights compatible ports when either dragging global parameter or filter port.

Tooltip delay [ms]:
Defines a delay in milliseconds of the tooltip appearance for hovered program elements.

Show indices of filter instances
If checked, indices of filter instances will be visible.

Show array and conditional markers on filter ports
If checked, ports that accepts or introduce array ([]), conditional (?) or optional (*) data will include appropriate information in their names.

Add generic filters as uninstantiated
If checked, generic filters will be added to program as uninstantiated, without prompting for choosing data type.

Autoconnect filters on insertion (experimental)
If checked, automatically connects filter image and coordinate system inputs with deduced outputs.

Open the default editor when adding the filters to the program
If checked, automatically opens the default data editor for inserting filter.

Close filter variant selection dialog after filter insertion
If checked, filter variant selection dialog will close after inserting one filter. Uncheck to insert multiple filters.

Preserve port previews when extracting macrofilter
If checked, port previews of filters extracted to new macrofilter will be preserved.

Program Analysis

Check array synchronization problems during program edition
If checked, potential array synchronization problems caused by user action are detected and require confirmation.

Show array synchronization in Program Editor
If checked, array synchronization is presented in Program Edition.

Check array synchronization during program loading
If checked, array synchronization is verified during program loading.

Check array synchronization before program start
If checked, array synchronization is verified before program start.

Ignore warnings in disabled macrofilters during program execution
If checked, warnings in disabled macrofilters will be ignored during program execution.

6. Results

View

Flat view
If checked, outputs are arranged in a flat list in the results. Otherwise, outputs are only displayed in a tree layout according to their logical relationship to their parent outputs.

Show statistics columns
If checked, statistics-related columns are displayed.

Use output labels
If checked, labeled outputs are presented as their Data Source Labels. Otherwise, original output name is used.

Contents
Defines the source for the results control contents. Either the selected filters or all filters in the current macrofilter (variant).

Show only visible outputs
If checked, hidden outputs are also hidden in the results.

Show only textual outputs
If checked, only outputs of textual data type (e.g. String, Integer, Real, etc.) are shown in the results.

Show only checked outputs
If checked, only outputs which are checked.

Show only labeled outputs
If checked, only outputs which have the label defined.

7. Project Files

Project Explorer

Watch for project file changes
Notifies, when project file has been changed i.e. by external program.

Project Binary Files

Binary Files Compression Level:
Configures project avdata files compression level.

8. Messages

Show these messages:

Warn when trying to modify running program
If checked, warning will be displayed before stopping program to make modification in it.

Ask about opening an example documentation on startup
If checked, application will notify about an example's description in documentation.

Warn when trying to replace existing preview
When checked, a warning is shown to prevent from accidentally removing carefully prepared preview.

Show warning when reconnecting inputs
Enables warning on creating a connection which would replace existing connection.

Show message after extracting macrofilter
If checked, additional help message is shown each time a macrofilter is extracted from selection.

Warn about using themes at high resolution and DPI
If checked, application will notify about the recommendation not using themes at high resolutions and DPI.

Ask for refactoring older project to use sections on loading project
Enables warning on creating a connection which would replace existing connection.

Show message after Workspace change when executing the program
If checked, the application will notify about program stop when the Workspace has changed.

Introduction to Data Flow Programming

Important: Aurora Vision Studio does not require the user to have any experience in low-level programming. Nevertheless, it is a highly specialized tool for professional engineers and a fully-fledged visual programming language. You will need to understand its four core concepts: Data, Filters (Tools), Connections, Macrofilters and the general program structure defined by four Sections.

Data

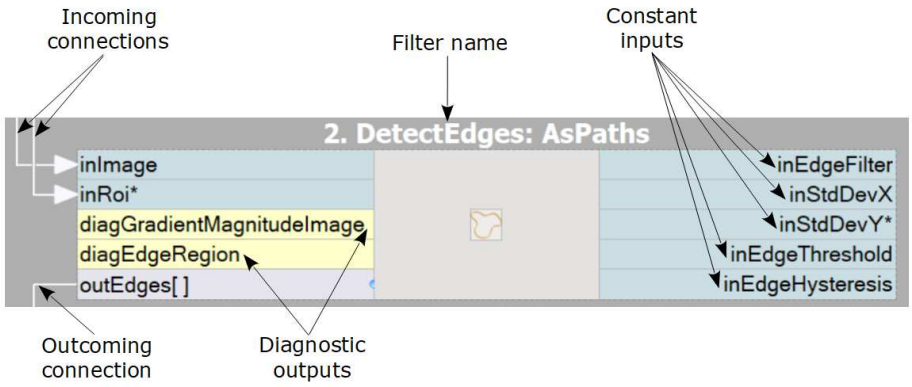
Aurora Vision Studio is a data processing environment so data is one of its central concepts. The most important fact about data that has to be understood is the distinction between *types* (e.g. Point2D) and *values* (e.g. the coordinates *(15.7, 4.1)*). Types define the protocol and guide the program construction, whereas values appear during program execution and represent information that is processed. Examples of common types of data are: Integer, Rectangle2D, Image.

Aurora Vision Studio also supports *arrays*, i.e. variable-sized collections of data items that can be processed together. For each data type there is a corresponding array type. For example, just as *4* is a value of the Integer type, the collection *{1, 5, 4}* is a value of the IntegerArray type. Nested arrays (arrays of arrays) are also possible.

Filters (Tools)

Filters are the basic data processing elements in data flow programming. In a typical machine vision application there is an image acquisition filter at the beginning followed by a sequence of filters that extract information about regions, contours, geometrical primitives and then produce a final result such as a pass/fail indication.

A filter usually has several inputs and one or more outputs. Each of the ports has a specific type (e.g. Image, Point2D etc.) and only connections between ports with compatible types can be created. Values of unconnected inputs can be set in the Properties window, which also provides graphical editors for convenient defining of geometrical data. When a filter is invoked (executed), its output data can be displayed and analyzed in the Data Preview panels.



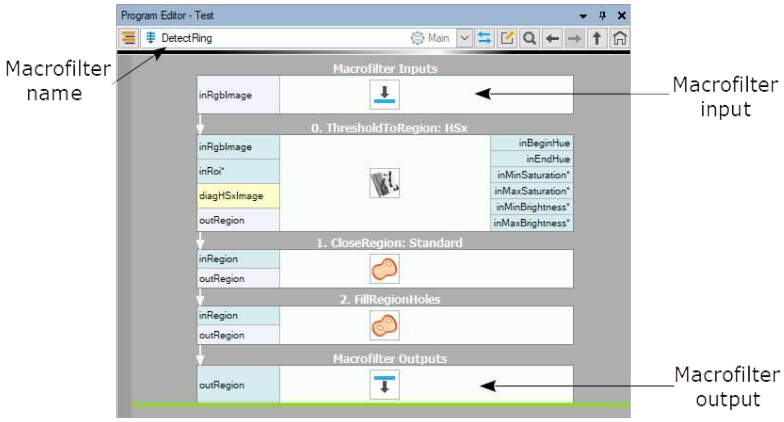
Connections

Connections transmit data between filters, but they also play an important role in encapsulating much of the complexity typical for low-level programming constructs like loops and conditions. Different types of connections support: basic flow of data , automatic conversions , array (*for-each*) processing and conditional processing . You do not define the connection types explicitly – they are inferred automatically on the *do what I mean* basis. For example, if an array of regions is connected to an input accepting only a single region, then an array connection is created and the individual regions are processed in a loop.

Macrofilters

Macrofilters provide a means for building bigger real-life projects. They are reusable subprograms with their own inputs and outputs. Once a macrofilter is created, it appears in the Project Explorer window and since then can be used in exactly the same drag and drop way as any regular filter.

Most macrofilters (we call them Steps) are just substitutions of several filters that help to keep the program clean and organized. Some other, however, can create nested data processing loops (Tasks) or direct the program execution into one of several clearly defined conditional paths (Variant Steps). These constructs provide an elegant way to create data flow programs of any complexity.



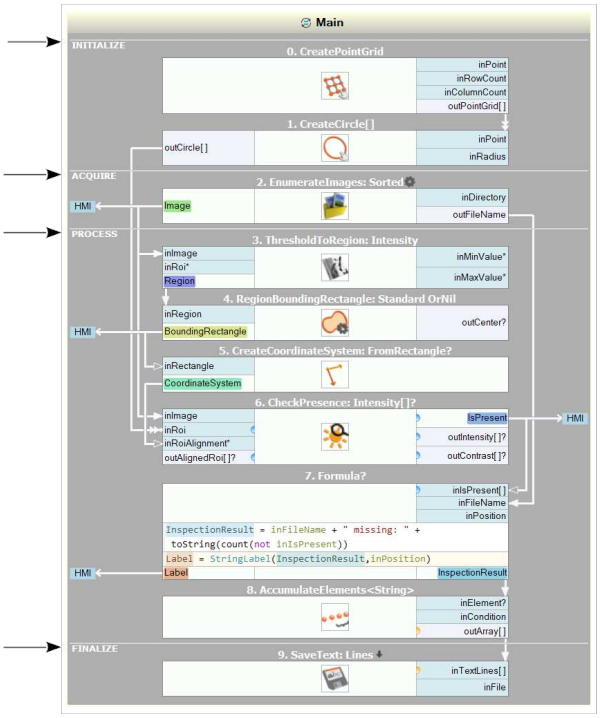
c++

Data and types are very similar to what you know from C++. We also have a generic collection type – *array* – which is very similar to *std::vector*. Filters and macrofilters are just equivalents of functions. But, instead of a single returned value they often have several output parameters. Connections correspond to local variables, which do not have to be named. On the other hand loops and conditions in Aurora Vision Studio are a bit different to C++ – the former are done with array connections or with Task macrofilters, for the latter there are conditional connections and Variant Step macrofilters. See also: Quick Start Guide for the C/C++ Programmers.

Sections

In order to improve clarity of applications and make it easier for the user to manage the data flow, starting from Aurora Vision Studio 5.0 we have introduced a new feature called "Sections". These special areas visible in the Program Editor are responsible for dividing the application code into four consecutive stages. Placing filters in the right section makes the application more readable and reduces necessity of using macrofilters in simple applications.

Currently, there are four sections available: **INITIALIZE**, **ACQUIRE**, **PROCESS** and **FINALIZE**.



INITIALIZE – this section should consist of filters that have to be executed only once, before the loop is started. The filters in this section will not be repeated during the loop. Such filters are usually responsible for initiation of a connection (e.g. GigEVision_StartAcquisition, TcpIp_Accept) or setting values of constant parameters for application execution.

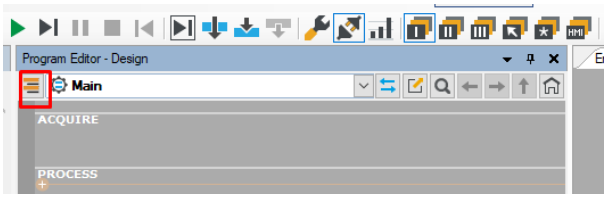
ACQUIRE – this section is intended to include filters from loop generation category like EnumerateImages or GigEVision_GrabImage that generate stream of data that will be processed or analyzed in the next section. Filters in this section will be executed in every iteration.

PROCESS – this is the main section that contains filters responsible for analyzing, processing and calculation of data. In most applications the **PROCESS** section will be largest one because its main purpose is to perform all the key tasks of the application. Filters in this section will be executed in every iteration.

FINALIZE – filters placed in this section are executed only once, after the loop. In most cases this section is used to close all the connections and save or show the inspection results (e.g. GigEVision_StopAcquisition, TcpIp_Close, SaveText). This section is executed only if the task macrofilter finishes without exceptions - it is not executed if an error occurs, even if it is handled by the error handler.

By default, sections are not visible inside Step and Variant Step macrofilters. However, the view in these macrofilters can be extended with **INITIALIZE** and **PROCESS** sections. In Worker Task and Task macrofilters the default view consist of **ACQUIRE** and **PROCESS** sections and can be extended to all four sections.

To change the default view of the sections in the Program Editor click on the button located to the left of the macrofilter name in the top bar of the Program Editor



Running and Analysing Programs

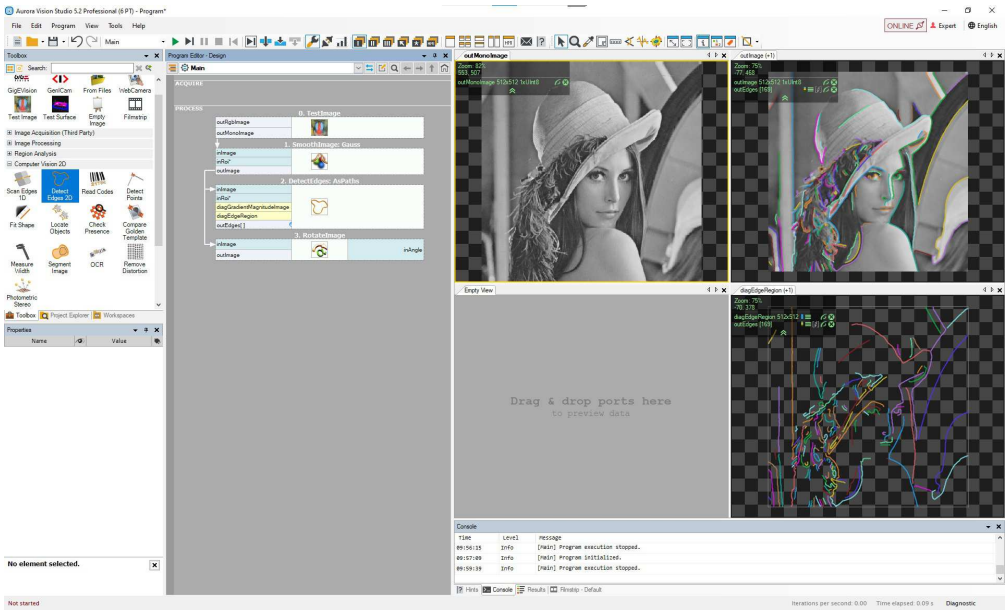
Example Projects

One of the best ways to learn Aurora Vision Studio quickly is to study the example projects that come with the application. The *File » Open Example...* command is a shortcut to the location they are stored in. The list of available projects is also available in the Program Examples section.

Executing Programs

When a project is loaded you can run it simply by clicking the *Run* button on the Application Toolbar or by pressing F5. This will start continuous program execution; results will be visible in the Data Preview panels. You can also run the program iteration by iteration by clicking Iterate or pressing F6.

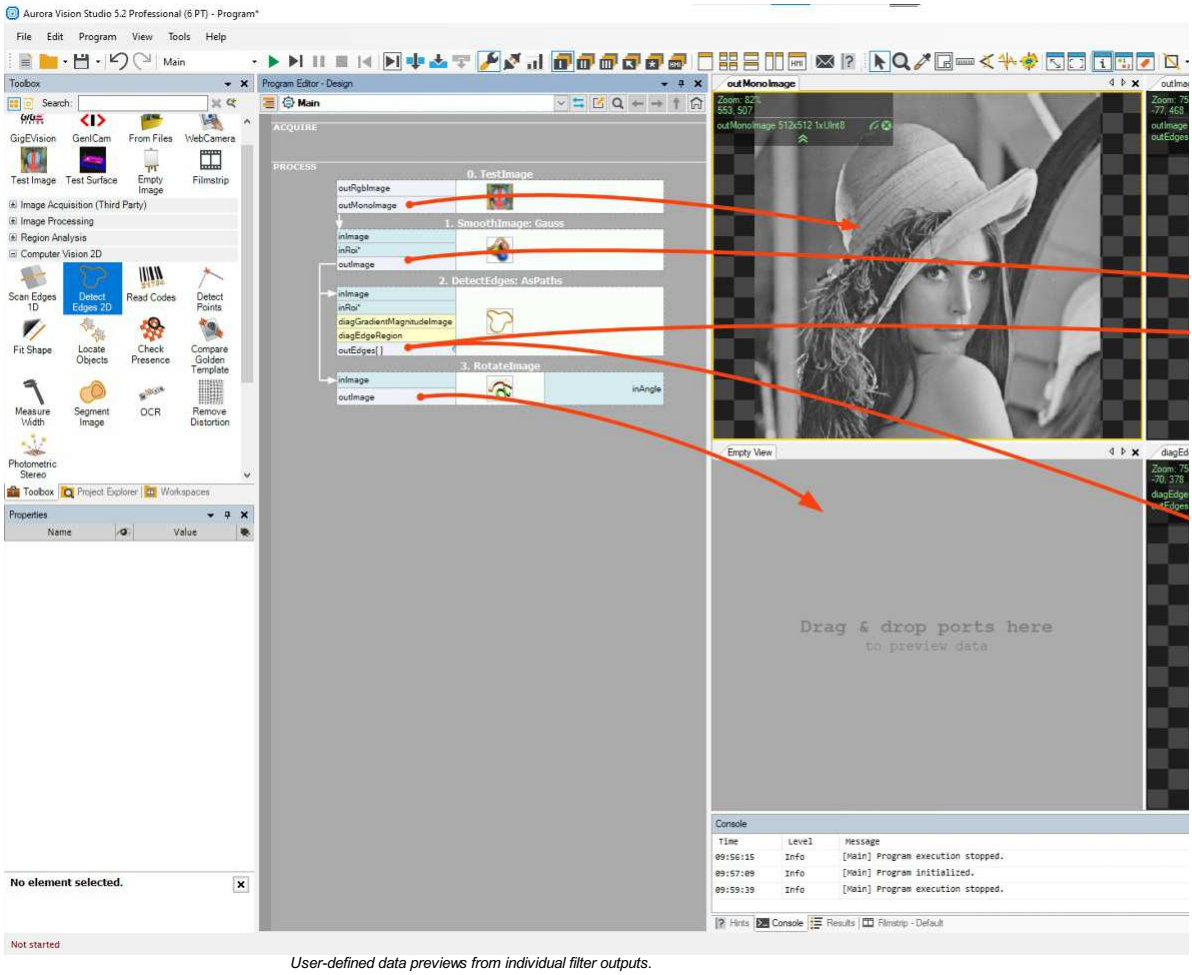
When a project is loaded from a file or when new filters (tools) are added, the filter instances are displayed subdued. They become highlighted after they are invoked (executed). It is also possible to invoke individual filters one by one by clicking *Step Over* or *Step Into*, or by pressing F10 or F11 respectively. The difference between *Step Over* and *Step Into* is related to macrofilters – the former invokes entire macrofilters, whereas the latter invokes individual filters inside.



A program with four filter instances; three of them have been already invoked.

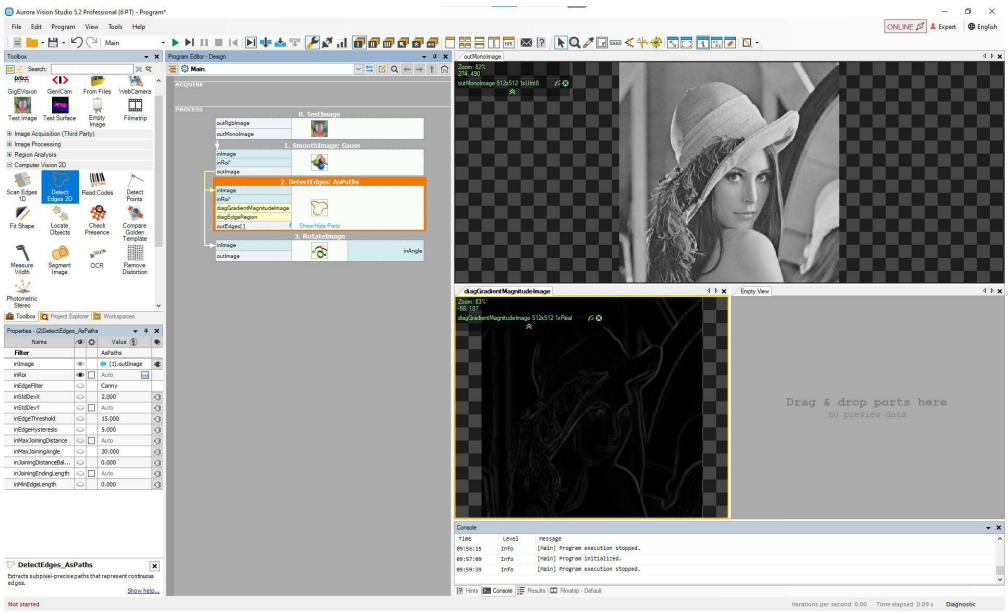
Viewing Results

Once the filters (tools) are executed, their output data can be displayed in the Data Previews panels. To display a value of a particular output, just drag and drop from the port of the filter to a Data Preview panel. Remember to press and hold left mouse button during entire movement from the port to the preview. Multiple data can be often displayed in multiple layers of a single preview. This is useful especially for displaying geometrical primitives, paths or regions over images.



User-defined data previews from individual filter outputs.

In bigger projects you will also find it useful to switch between three different layouts, which can be created to visualize different stages of the algorithm, as well as to the automatic layout mode, which is very useful for interactive analysis of individual filters:



Automatic data previews – the layout adapts to the currently selected filter.

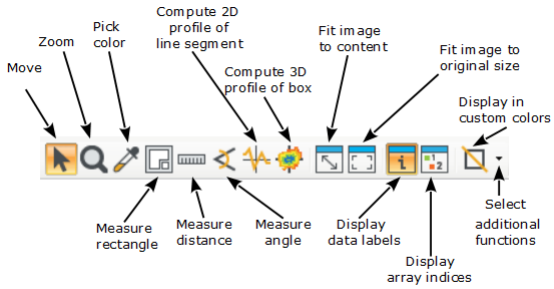
Analysing Data

Data displayed in the Data Preview panels can be analyzed interactively. There are different tools for different types of data available in the main window toolbar. These tools depend on currently selected preview which is marked with a yellow border when window is docked.

Measure rectangle - Selection

Measure distance - Length

For the most common Image type the Data Preview window has the following appearance:

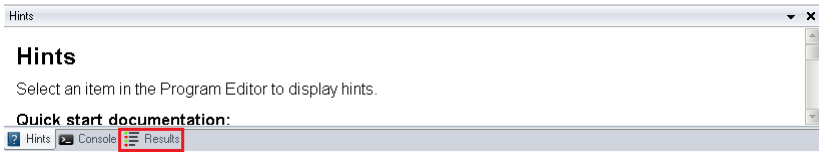


Additionally, there are several usability enhancements:

- **Mouse Wheel** – zooms in or out the image.
- **3rd Mouse Button + Drag** – moves the view.
- **Right Click** – opens a context menu and allows to save the view to an image file.

Results Control

While analyzing your application, it is useful to switch to the Results control available near the bottom of the screen. If you cannot see it, you need to enable it through *View » Results*.



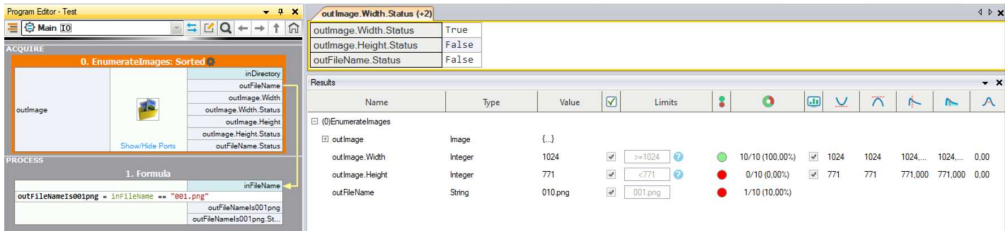
The location of the Results control (when enabled).

Upon clicking on a filter or checking a group of filters, a table with appropriate data will be displayed.

- **Name**
  Displays the name of the selected filter or filters as well as some of their outputs, including diagnostic ones (if the Diagnostic Mode is enabled).
- **Type**
  Displays the type of data returned by the given output.
- **Value**
  Displays the type of data returned by the given output in the last iteration.
- **Checked**
  If the given output can be subject to a Pass/Fail inspection (so any output returning a numerical value but also such types as String or Bool), you may check the box visible in this column in the corresponding row, thus enabling the following few columns. The application must be stopped for it to be allowed.

  Below you can see a modified Fiducial Markers example. Two filters display the application of the Results control.

  - **Limits**
    In this parser you can insert a simple formula describing the criterion for a Pass (hover the cursor over the question mark to see examples). The application must be stopped for it to be allowed.
  - **Pass/Fail status**
    A green or red dot will appear signifying whether the criterion described under Limits has been met or not.
  - **Pass/Fail status statistics**
    Displays the number of iterations in which this particular output has so far met the criterion described under Limits.
  - **Monitored**
    If the given output can be subject to a mathematical analysis, you may check the box visible in this column in the corresponding row in order to enable the following few statistics. The application must be stopped for that to be allowed.
  - **Minimum**
    Displays the lowest value so far returned.
  - **Maximum**
    Displays the highest value so far returned.
  - **Mean**
    Displays the mean value of those so far returned.
  - **Median**
    Displays the median value of those so far returned.
  - **Standard Deviation**
    Displays the standard deviation value of those so far returned.



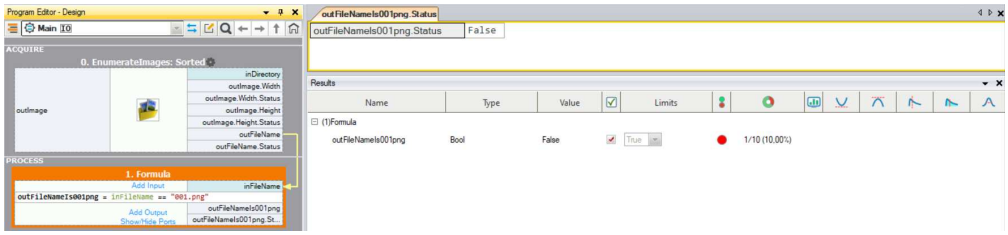
The Results control in action: types Integer and String.

After performing ten iterations on equally-sized images, we can see that:

- The **outImage.Width** Integer output returned True on all ten iterations (because the value was the same and the parser allowed it to be either the same or higher).
- The **outImage.Height** Integer output returned False on all ten iterations (because the value was never lower and the parser only allowed it to be lower).
- The **outFileName** String output returned True on one out of ten iterations (because the number within the name kept increasing so it did not stay the same).

Note how next to the parsers applied to numerical values there is a blue question mark . If you hover your cursor over it, you will see suggestions of formulas that you can use within the parser. These suggestions are not available with non-numerical types of data. Also note how after executing the program, new out...Status outputs became available in the filter (they are shown in the preview window above the Results control).

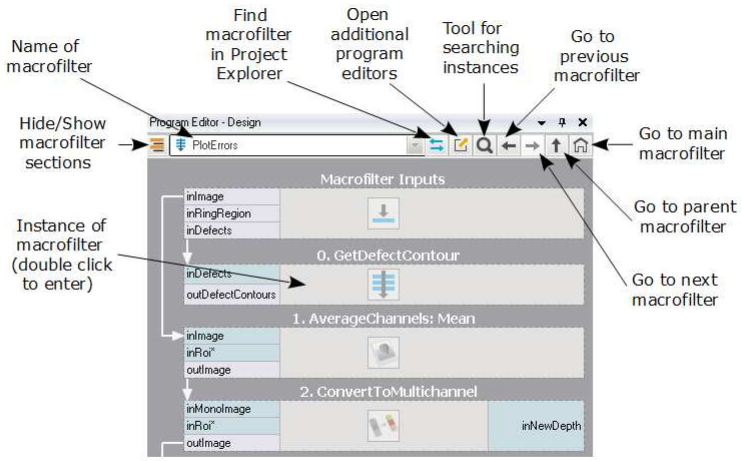
Below you can see the file name being verified again, this time as the Bool type output **outFileNameIs001png**. The Results control shows the identical effect to that of the outFileName String output above—except that this time the Limits parser can only be set either to True or to False.



The Results control in action: type Bool.

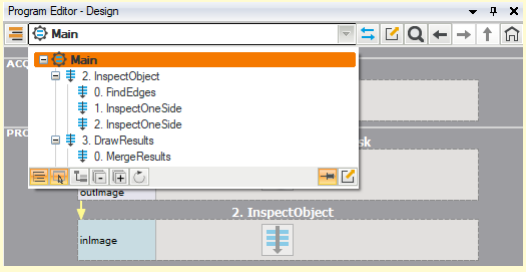
Browsing Macrofilters

Except for the most simple applications, programs in Aurora Vision Studio are composed of many so called macrofilters (subprograms). Instances of macrofilters in the Program Editor can be recognized through the icon which depicts several blue bars. Double clicking on an instance opens the macrofilter in the Program Editor.

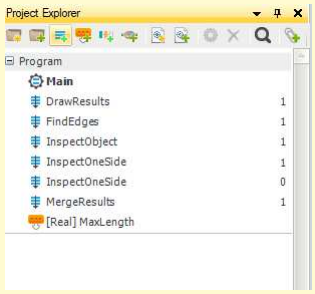


Browsing macrofilters in the Program Editor.

There are two other ways of browsing macrofilters. One is through the Macrofilter Navigator at the top of the Program Editor, which displays *instances* of macrofilters (how they are actually used and nested in the program). Another is through the Project Explorer, which displays *definitions* of macrofilters (a plain list of all macrofilters from which the user can create instances by dragging and dropping them to the Program Editor; double clicking on an item here opens the macrofilter in the Program Editor).



Instances of macrofilters in the Macrofilter Navigator.



Definitions of macrofilters in the Project Explorer.

c++

Definitions of macrofilters correspond to definitions of functions in C++, whereas instances of macrofilters correspond to function calls on a call-stack. Unlike the C++, there is no recurrence in Aurora Vision Studio and each macrofilter definition has a constant and finite number of instances. Thus, we actually have a static call-tree instead of a dynamic call-stack. This makes program understanding and debugging much easier.

Analysing Operation of a Single Macrofilter

The *Iterate Current Macro* command can be very useful when you want to focus on a single macrofilter in a program with many macrofilters. It executes the whole program and pauses each time when it comes to the end of the currently selected macrofilter instance.

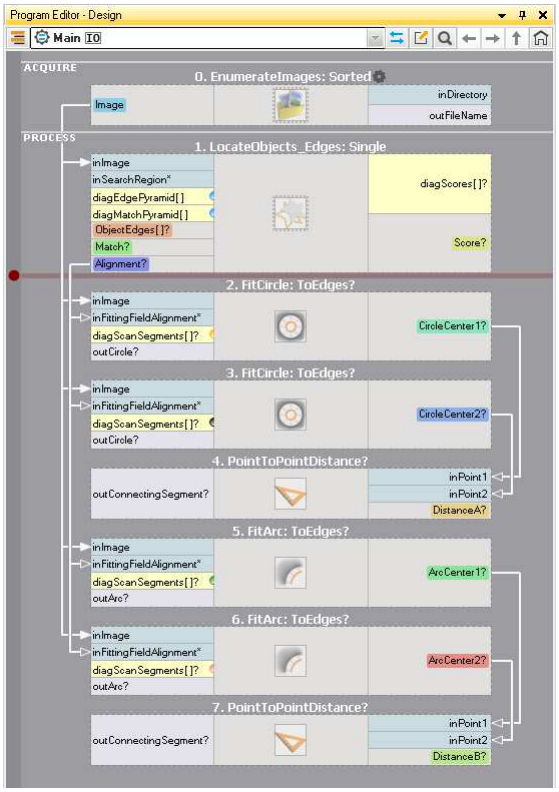
c++

The *Iterate Current Macro* command is very similar to setting a breakpoint at the end of some function in a C++ debugger.

More information about program iteration is available in Testing and Debugging.

Execution Breakpoints

Pausing the program execution in any algorithm location is a fundamental technique while debugging program in any programming language. In Aurora Vision Studio such a pausing can be done with breakpoints in the program editor. Breakpoints are visualized with red circle in the left margin of the program editor and a line next to it:



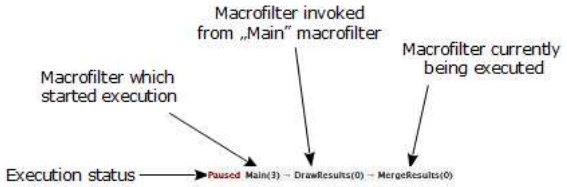
Breakpoints can be activated (toggled) in several ways:

- by clicking appropriate locations within the program editor margin,
- through the context menu of the filter blocks or macrofilter outputs block,
- with F9 shortcut on selected filter block or macrofilter outputs block.

You can read more about Breakpoints in Testing and Debugging.

Knowing Where You Are

At each moment of a program execution you can see on the Application Status Bar which macrofilter instance is currently being executed. This is called a *call-stack*, because not only the name of the macrofilter is displayed, but also all the names of the parent macrofilters.



The call-stack of the Application Status Bar

The current position of the execution process is also marked with a green line or frame in the Program Editor:

Execution marker showing the exact position of the execution process.

Execution marker showing that the execution process is currently inside this macrofilter instance.

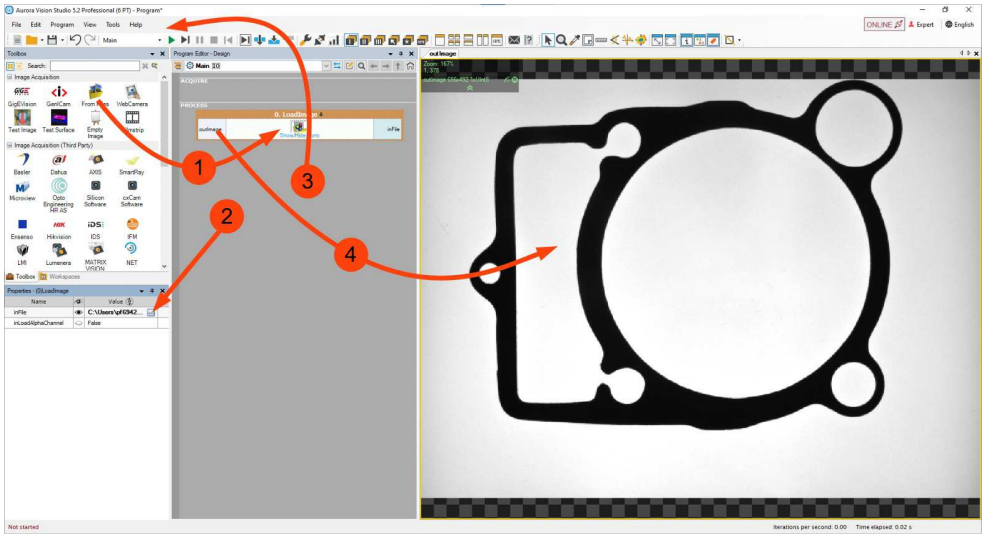
Acquiring Images

Acquiring Images from Files

Aurora Vision Studio is not limited to any fixed number of image sources. Instead, image acquisition is a regular part of the library of filters (tools). In particular, to load an image from a file, the user needs to create a program with an instance of the LoadImage filter.

Four steps are required to get the result ready:

1. Add a LoadImage filter to the Program Editor:
   a. Either by choosing it from the *Image Acquisition* section of the Toolbox (recommended).
   b. Or by dragging and dropping it from the Image :: Image IO category of the Filter Catalog.
2. Select the new filter instance and click on "..." by the **inFile** port in the Properties window. Then select a PNG, JPG, BMP or TIFF file.
3. Run the program.
4. Drag and drop the **outImage** port to the Data Previews panel.



Creating a program that loads an image from a file.

Enumerating Images from a Directory

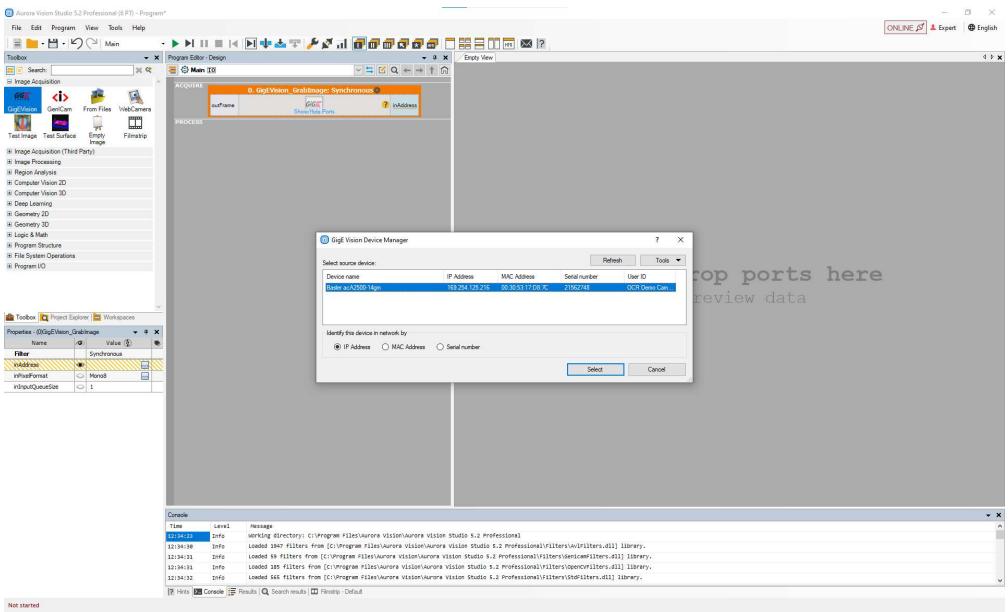
If you have multiple images, e.g. recorded from a camera, and you want to simulate that camera with the files, you can use the EnumerateImages filter (tool). It will make your program work in a loop until all images are read from the directory specified with the **inDirectory** parameter.

Acquiring Images from Cameras

For acquiring images from cameras and frame grabbers there are filters (tools) in several different categories:

- **GigE Vision**
  Provides an interface to all GigE Vision compatible devices. See also Working with GigE Vision Devices.
- **GenICam**
  Provides an interface to all GenICam / GenTL compatible devices. See also Working with GenTL Devices.
- **Camera Support**
  Provides multiple interfaces to drivers provided by individual camera vendors: NET (SynView, ICube), Allied Vision Technologies (Vimba), Basler (Pylon), Kinect, Matrix Vision (mvGenTLAcquire), LMI (Gocator), IFM, PointGrey (FlyCapture, Spinnaker), The Imaging Source (Imaging Control), XIMEA (m3api).
- **Camera Support :: Web Camera**
  Provides an interface to DirectShow based cameras, e.g. to standard web cameras.
- ***User Filters***
  Non-standard cameras can be connected by writing User Filters in C++. A sample implementation for cameras from IDS is available.

It is advisable to use the general GigE Vision or GenICam interfaces in the first place as they provide the most complete feature set and the most comprehensive support in the graphical interface of Aurora Vision Studio. Vendor specific interfaces are required for older and non-standard camera models.



Connecting to a GigE Vision compatible camera.

Preview and Data Presenting

Creating and Closing Data Previews

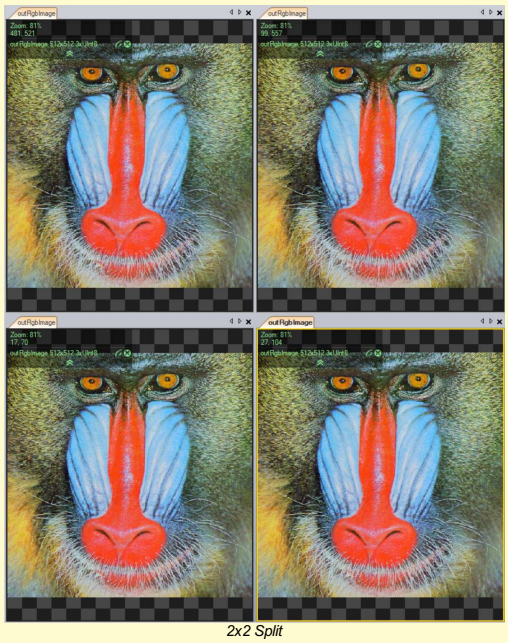
Once you execute filters (tools) in the program it is possible to display their inputs and outputs on the *Data Preview* panel. To do so, you can drag the input or the output of the filter and drop it on the panel to the right of the *Program Editor*. Similar types of data such as **Image** and **Region** or **StringArray** and **IntegerArray** can be displayed on the same preview.

There are several ways of removing elements from the data preview. You can:

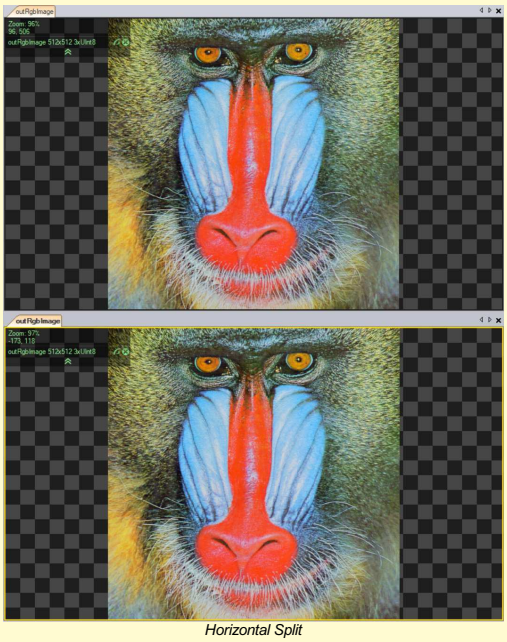
- Click the *X* at the top right corner. In this way, you will remove the last data that has been added to the preview. If the closed one is also the last, the preview window is closed.
- Use the mouse wheel button and press it when mouse cursor is placed on the name of the preview (top left corner).
- Right-click on the name of the preview and select one of the options, which are *Remove Preview*, *Close*, *Close All But This* and *Close All*.

Arranging Data Previews

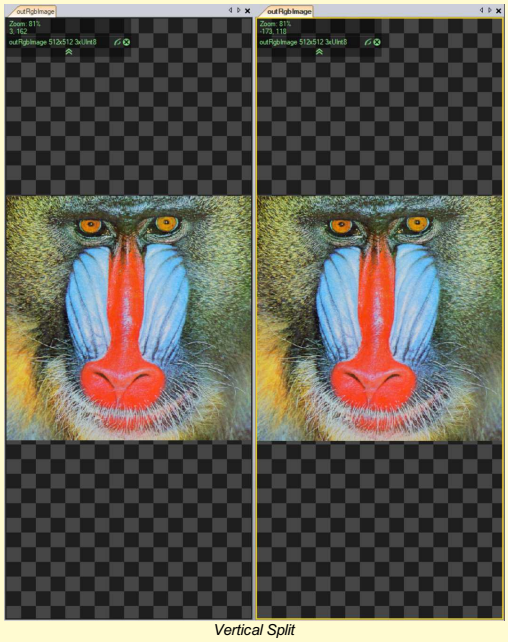
There are five default options that you can use to arrange your preview:  . Despite displaying your preview in a single window, you can use four other options:



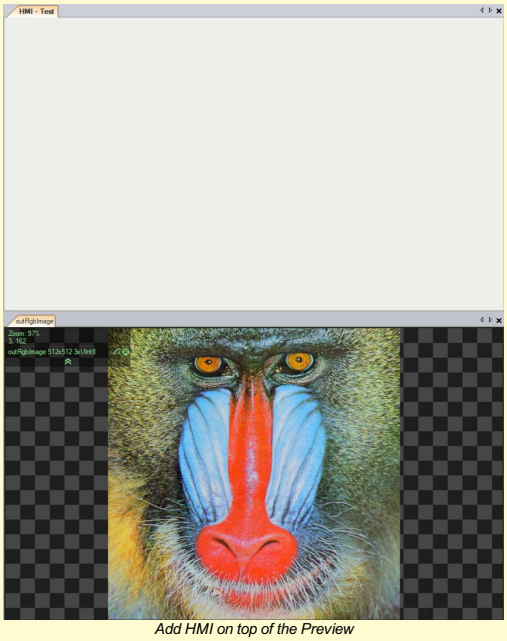
2x2 Split



Horizontal Split



Vertical Split



Add HMI on top of the Preview

Additionally, you can further divide the preview windows by right-clicking on their name and selecting "Split View" option.

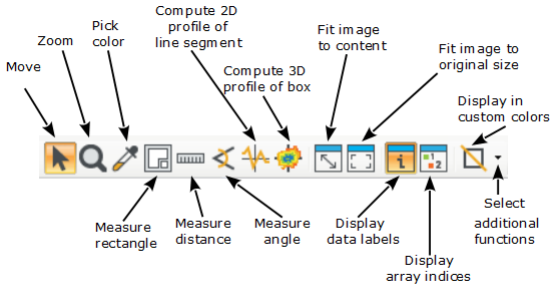
Another option that is at your disposal is docking. You can either double-click on the name of the preview or right-click and select *Undock* option. To restore the window to its previous preview simply right-click on the bar and select *Dock* option. Also, you can click and hold a Tab with the name of data preview and move it to another position. During that process the navigation pane appears and you can specify new appearance of the preview.

Layouts

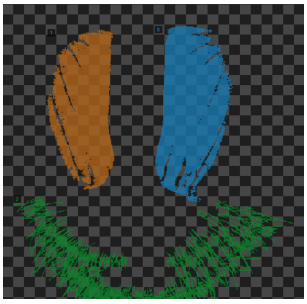
It is possible to have up to three separate preview layouts arranged in different ways and switch between them when working on an algorithm. Additionally, *Auto mode*, which shows the most useful inputs and outputs of the currently active filter (tool), is available. All the above-mentioned features are available on the Toolbar:  . There is also the HMI button that opens the standalone tab where you can design the user interface.

Image Tools

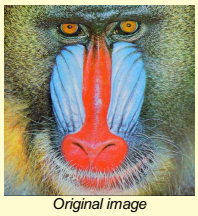
You gain access to the Image Tools after clicking on the image. It contains handy tools that will help you during the analysis of images. Note that they do not modify the image in the program, but rather help you set appropriate parameters and collect data necessary for the inspection, by influencing the preview.

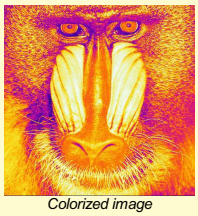


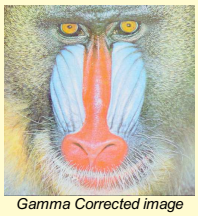
- *Move* and *Zoom* are purely for the navigation on the preview. It is possible to use the same functions with the scroll wheel of your mouse.
- The *Pick Color* tool is handy if you want to check the intensity in the image or RGB/HSV values before performing further operations on the image.
- The next three icons are all about the measurement. *Measure rectangle* allows to select a rectangle and measure its width and height. *Measure distance* returns the distance in pixels between two points on the image. *Measure angle* simply checks the angle defined by three points selected by user.
- Next, there are tools for 1D and 2D image profile analysis. After marking the segment or box on the image and you will receive a graph ready for a quick analysis.
- Following buttons are for fitting the image to the content and returning it to the original size.
- *Display data labels* option allows you to see the labels on the objects and *Display array indices* shows the numbers on the image that correspond to the position of the features in the input array, as seen on the image below:

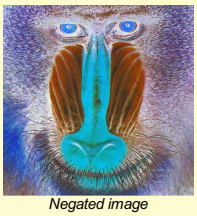


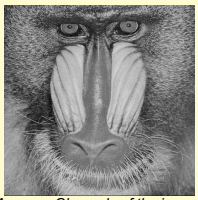
- Possible effects of the *Display in custom colors* tool are displayed below:

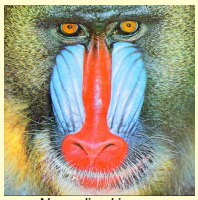

Original image

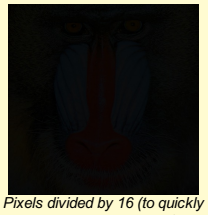

Colorized image


Gamma Corrected image


Negated image


Average Channels of the image


Normalized image


Pixels divided by 16 (to quickly preview 12-bit images)

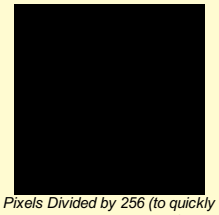
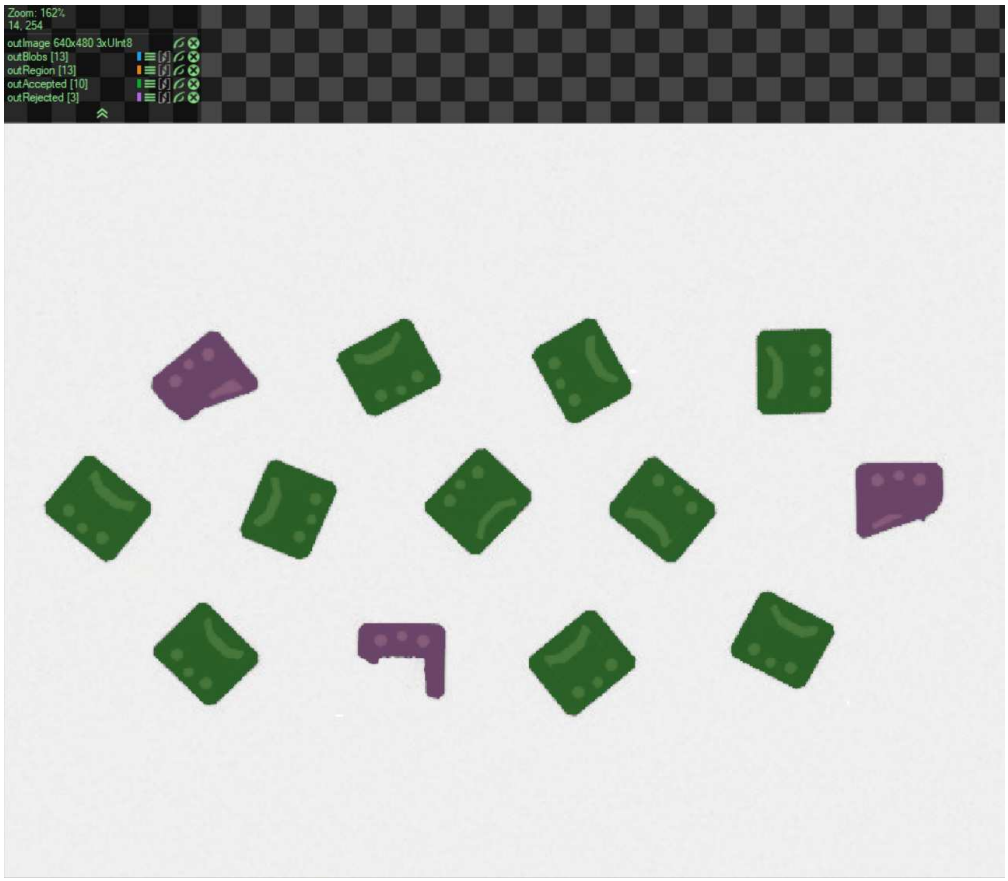

Pixels Divided by 256 (to quickly preview 16-bit images)

Image options on the preview

If you drop an image on the preview you will get access to additional options after right-clicking on it:

- *Save Image As* - saves the image with its original size and other parameters in the specified folder.
- *Save View As* - saves the current view of the image, with other elements (e.g. geometric primitives) located on it, in the specified folder.
- *Copy View As Image* - copies the current view of the image.
- *Zoom to Fit* - scales the image to the size of the preview window so it can fit in it.
- *Zoom to Original size* - returns the image to the original size.
- *Show Information* - turns on/off the information about the current preview.
- *Show Array Indices* - shows sorted numbers on the preview. You can only see the results of that button if it was dropped to the preview as an array.

Preview information

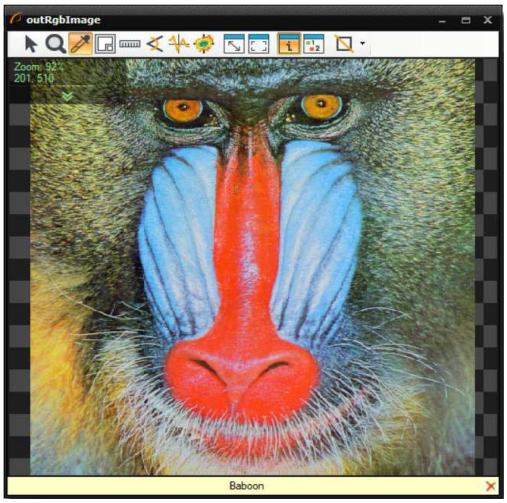


The green text at the top left corner of the preview contains the names of all the outputs and inputs that were dropped on it. They are always followed by additional information. In case of **Image** or **Region** it is usually its size. It is also possible to see a number in the brackets - [4]. It indicates the number of the elements of the array of data. To the right of the names you may notice up to five buttons:

- - it tells you about the color the type of data is being displayed in. If you press it, you will be able to change its shade.
- - by dragging and moving it up and down you can organize the data on the preview. Note that the color changes with the order of data, so it does not stay the same after being moved.
- - it allows you to turn on and off the data from the preview without completely removing it.
- - that button removes the data from the preview.
- - it helps you navigate through elements of the arrays. If you turn it off, you will be able to see all the elements of the array on the preview at once. Turning it on switches the preview to show only one element and allows to iterate them one by one, however, it makes it impossible to see them all.

Adding comments

Adding a comment is a handful option that you can use on all your preview windows. To do so right-click on the name of the preview and choose *Add Comment* option. After that a yellow box at the bottom of the preview will appear. You will be able to write anything of your liking there, as in the image below:



Useful Settings

To change the quality of the previews you can enter *Tools>>Settings>>Previews>>Preview Quality* or *Program>>Preview Quality*. You will be able to choose from three different settings:

- Fast
- Balanced
- High Quality

Note that in case of larger images *Image is too big!* warning pops out. It automatically lowers down the quality of the image even though the best one was selected beforehand.

Users may extract macrofilters in their program. By default that operation removes the previews of the filters that were enclosed. To prevent that you can access *Tools>>Settings>>Editor>>Preserve port previews when extracting macrofilter*.

For more details on the topic, please refer to a video tutorial on our YouTube channel:

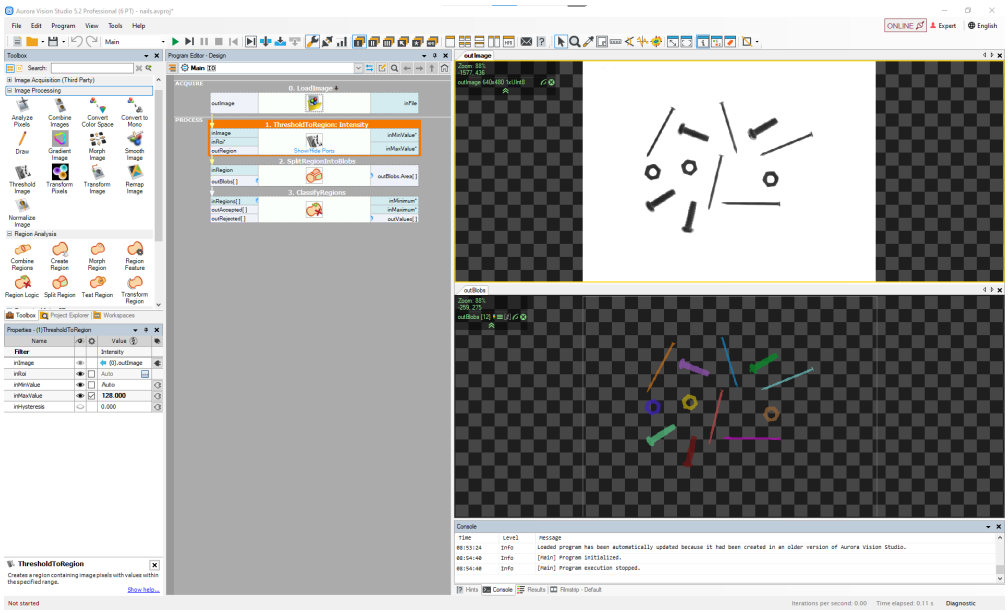
Please note that previews of the 3D data are explained in a separate article: Working With 3D Data.

First Program: Simple Blob Analysis

This article demonstrates the basic workflow in Aurora Vision Studio with an example of simple blob analysis. The task here is to separate nails from other objects which are present in the input image.

Extracting Blobs

To start this simple demonstration we load an image from a file – with the LoadImage filter (tool), which is available in the *Image Acquisition* section of the Toolbox, the *From File* group. The image used in the example has been acquired with a backlight, so it is easy to separate its foreground from the background simply with the ThresholdToRegion filter (*Image Processing → Threshold Image*). The result of this filter is a Region, i.e. a compressed binary image or a set of pixel locations that correspond to the foreground objects. The next step is to transform this single region into an array (a list) of regions depicting individual objects. This is done with the SplitRegionIntoBlobs filter (*Region Analysis → Split Region*):



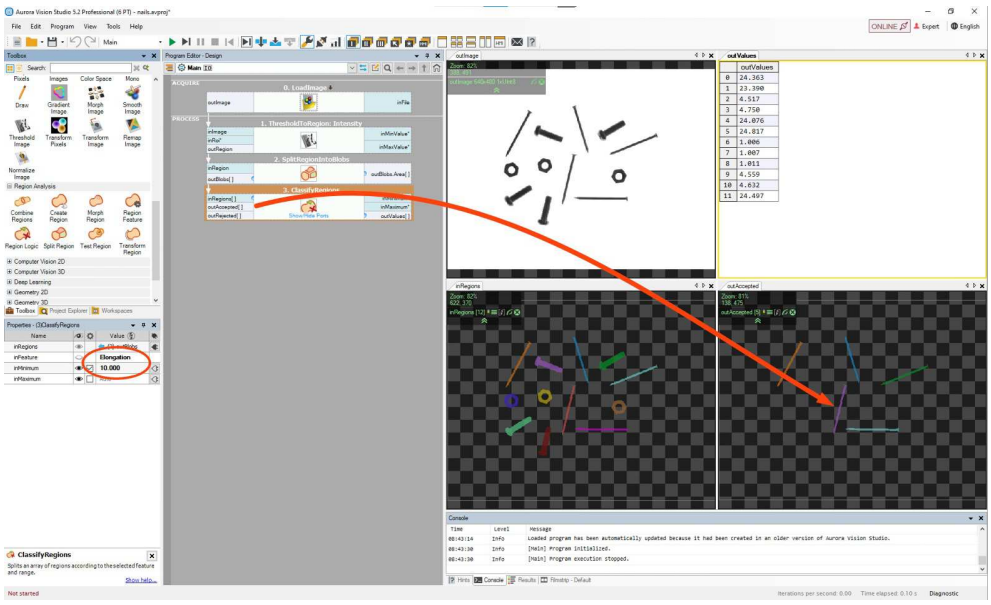
Extracting blobs from an image.

Notes:

- Connections between filters are created by dragging with a mouse from a filter output to an input of another filter.
- The data previews on the right are created by dragging and dropping filter outputs.
- The input file is available here: parts.png.

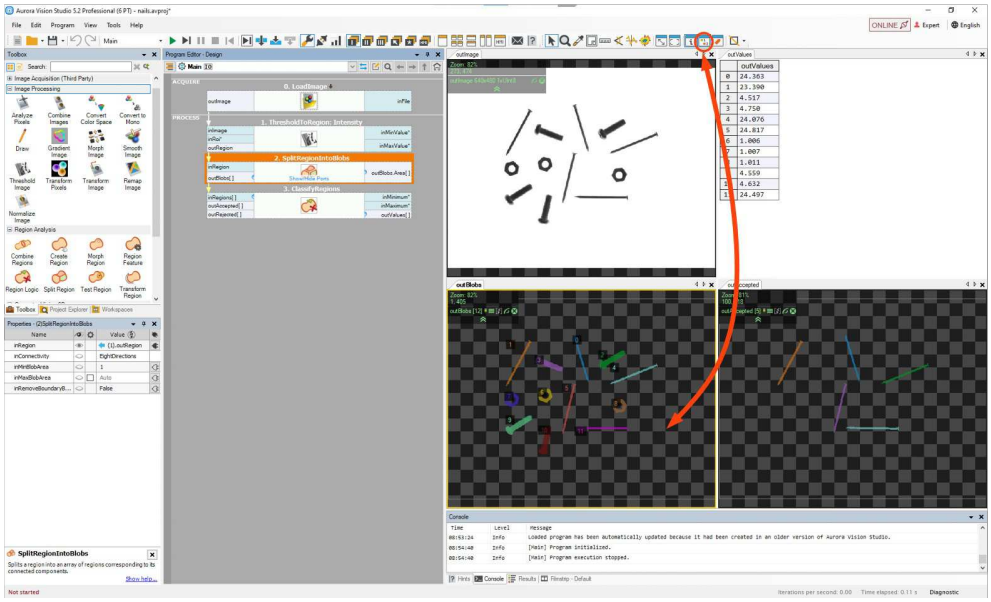
Classifying the Blobs

At this stage what we have is an array of regions. This array has 12 elements, of which 4 are nails. To separate the nails from other objects, we can use the fact that they are longer and thinner. The ClassifyRegions filter (*Region Analysis → Region Logic*) with **inFeature** input set to *Elongation* and **inMinimum** set to *10* will return only the nails on its **outAccepted** output:



Classifying blobs by the "elongation" feature.

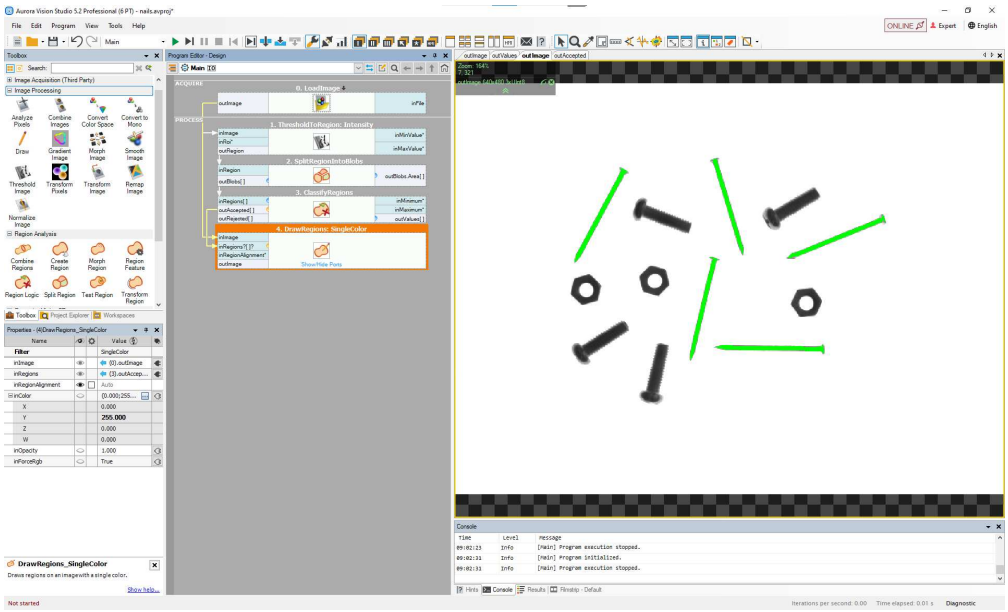
There is also the **outValues** output which contains the feature values of the individual blobs. This can also be displayed in the Data Previews as a table of real numbers. The indexes in this table correspond to the blobs, which can be shown by using the "Show Indexes of Elements" option in the selected data preview toolbar:



Showing indexes of individual blobs.

Drawing the Results

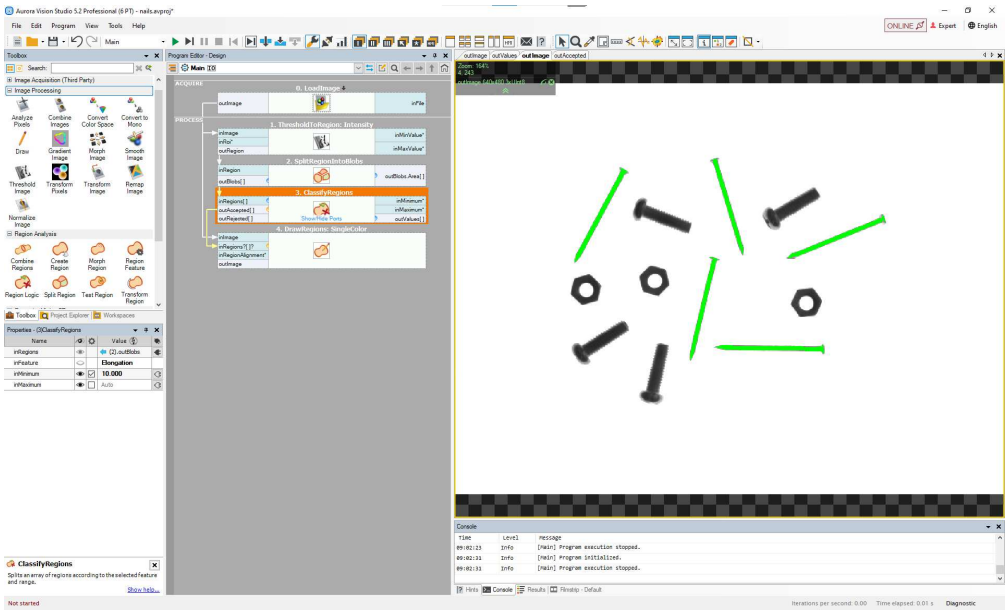
Finally, we can create an output image (e.g. for displaying in the HMI) with the nails marked in green. For this purpose we use the DrawRegions_SingleColor filter, which needs to have its **inImage** and **inRegions** inputs appropriately connected. The **inColor** input defines the required color and can be edited through the Properties window.



Drawing the nails in green.

Getting the Number of Elements

If we want to obtain also the number of elements found, we can right-click on the **outAccepted** output and select *Property Outputs → Count*, which creates an additional output named **outAccepted.Count**. In this case we are getting the number *5*:



Counting the found objects.

Conclusion

As this example demonstrates, creating programs in Aurora Vision Studio consists of selecting filters (tools) from the Toolbox (or from the Filter Catalog), connecting them and configuring. Data previews are created by dragging and dropping filter ports to the data preview area. This simple workflow is common for the most basic programs, as the one just shown, as well as for highly advanced industrial applications which can contain multiple image sources, hundreds of filters and arbitrarily complex data-flow logic.

Note: This program is available as "Nails Screws and Nuts" in the official examples of Aurora Vision Studio.

3. User Interface

Table of content:

Complexity Levels

Introduction

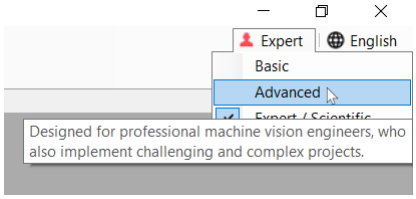
Aurora Vision Studio has three levels of feature complexity. At lower levels we hide advanced features and ones that may be confusing.

Available Levels

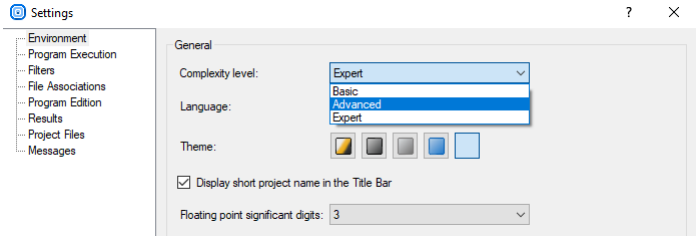
- **Basic** – Designed for production engineers who want to quickly build simple machine vision projects without devoting much time to learning the full capabilities of the software.
- **Advanced** – Designed for professional machine vision engineers who also implement challenging and complex projects.
- **Expert / Scientific** – Gives access to experimental and scientific filters (tools), which are not recommended for typical machine vision applications, but might be useful for research purposes.

Changing Complexity Level

Complexity Level can be changed at any time. It can be done by clicking on the level name in the upper-right corner of Aurora Vision Studio:



Complexity Level can also be changed in the application settings:

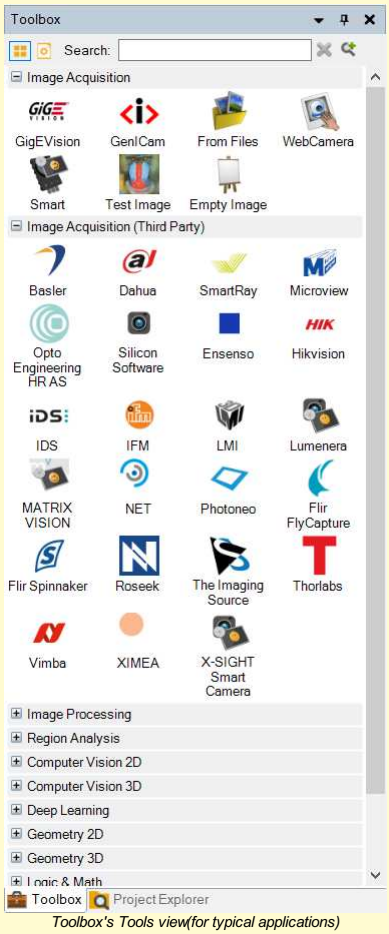


Finding Filters

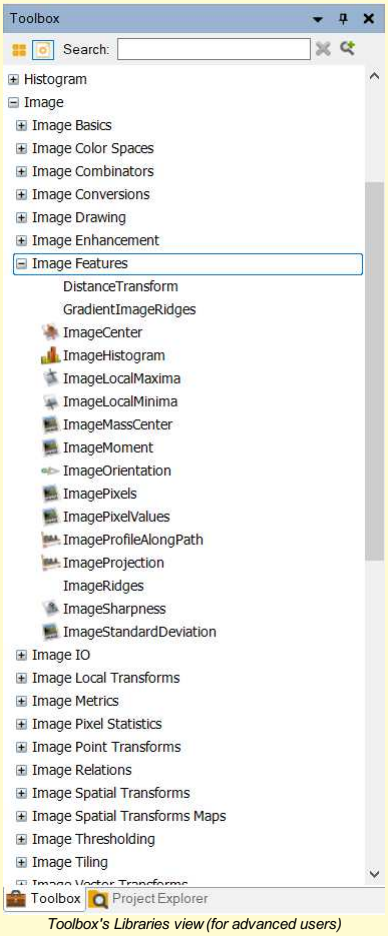
Introduction

There are many hundreds of ready-for-use filters (tools) in Aurora Vision Studio implementing common image processing algorithms, planar geometry, specialized machine vision tools as well as things like basic arithmetics or operating system functions. On the one hand, it means that you get instant access to results of tens of thousands of programmers' work hours, but it also means that there are quite a lot of various libraries and filter categories that you need to know. This article advises you how to cope with that multitude and how to find filters that you need.

The most important thing to know is that there are two different catalogs of filters, designed for different types of users:



Toolbox's Tools view(for typical applications)



Toolbox's Libraries view (for advanced users)

The Toolbox's Tools view is designed for use in typical machine vision applications. It is task-oriented, most filters are grouped into tools and they are supported with intuitive illustrations. This makes it easy to find the filters you need the most. It does not, however, contain all the advanced filters, which might be required in more challenging applications. To access the complete list of filters, you should use the Toolbox's Libraries view. This

catalog is organized in libraries, categories and subcategories. Due to its comprehensiveness, it usually takes more time to find what you need, but there is also an advanced text-based search engine, which is very useful if you can guess a part of the filter name.

Toolbox

Sections

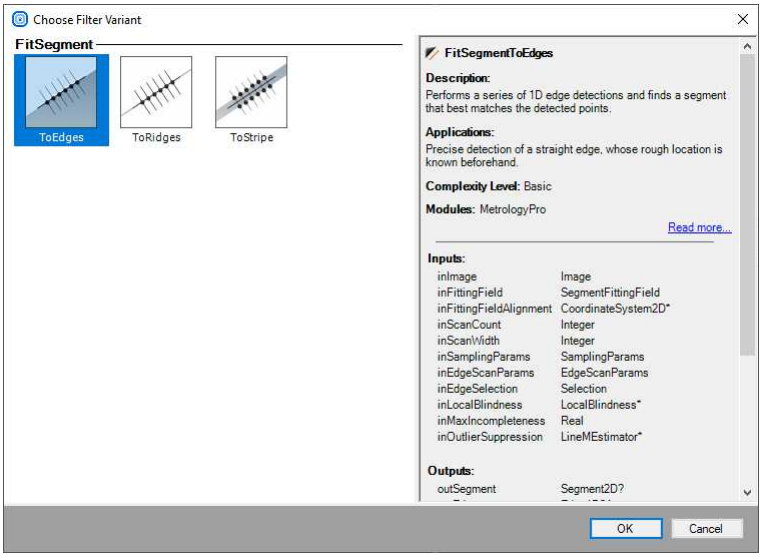
When you use the Toolbox's Tools view, the general idea is that you will most probably start with a filter (tool) from the first section, Image Acquisition, and then follow with filters from consecutive sections:

1. **Image Acquisition**
   – Acquiring images from cameras, frame grabbers or files.
2. **Image Acquisition (Third Party)**
   – Acquiring images from third-party cameras.
3. **Image Processing**
   – Image conversions, enhancements, transformations etc.
4. **Region Analysis**
   – Operations on pixel sets representing foreground objects (blobs).
5. **Computer Vision 2D**
   – Specialized tools for high-level image analysis and measurements.
6. **Computer Vision 3D**
   – Specialized tools for analysis of 3D point clouds.
7. **Deep Learning**
   – Self-learning tools based on deep neural networks.
8. **Geometry 2D**
   – Filters for constructing, transforming and analysing primitives of 2D geometry.
9. **Geometry 3D**
   – Filters for constructing, transforming and analysing primitives of 3D geometry.
10. **Logic & Math**
    – Numerical calculations, arrays and conditional data processing.
11. **Program Structure**
    – Category contains the basic program structure elements.
12. **File System**
    – Filters for working with files on disk.
13. **Program I/O**
    – Filters for communication with external devices.

Choosing Filter from Tools

A tool is a collection of related filters. The process of selecting a filter from the Toolbox thus consists of two steps:

1. First you select a tool in the Toolbox, e.g. "Fit Shape".
2. Then you select a filter from the tool in the "Choose Filter of Group" window.



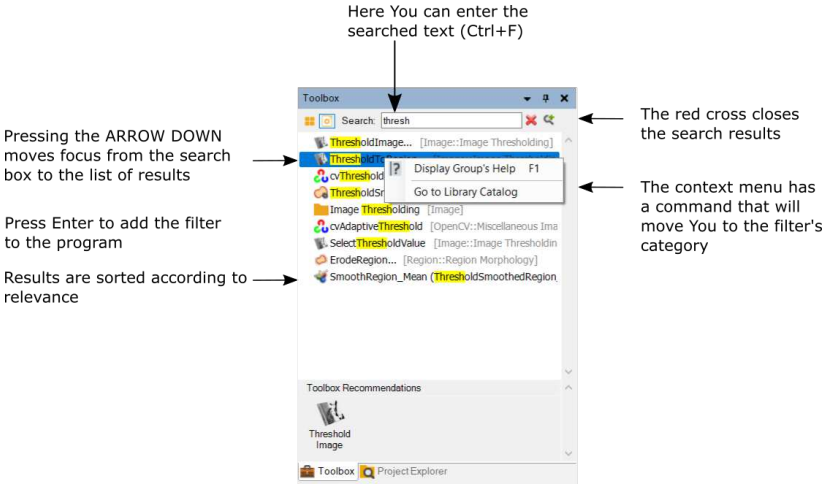
Choosing a filter from a tool (Toolbox).

This dialog can have several sections (e.g. "Fit Segment", "Fit Circle" etc.)

Changing filter variant.

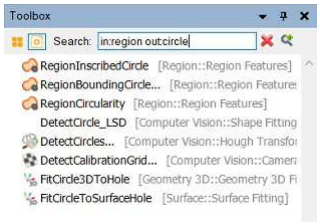
The Search Box

If you know a part of the name of the filter (tool) that you need, or if you can guess it, the Filter Search Box will make your work much more efficient. Just enter the searched text there and you will get a list of filters with most relevant matches at the top:



The Search Box and search results in the Toolbox's Libraries view.

If you can not guess the filter name, but you know what you expect on the inputs and outputs, you can use special queries with "in:" and "out:" operators as the image below depicts:



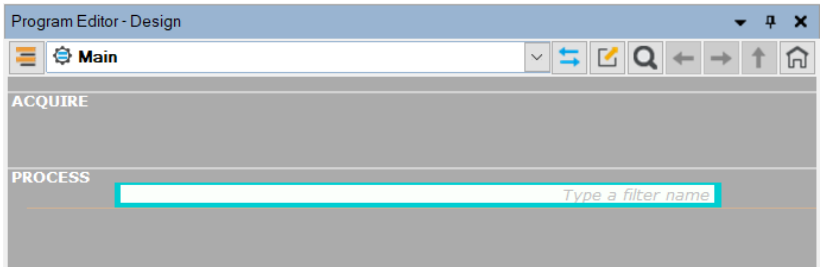
Advanced search with expected inputs and outputs.

As a matter of fact, some advanced users of Aurora Vision Studio stop browsing the categories and just type the filter names in the Search Box to have them quickly added to the program.

Program Editor

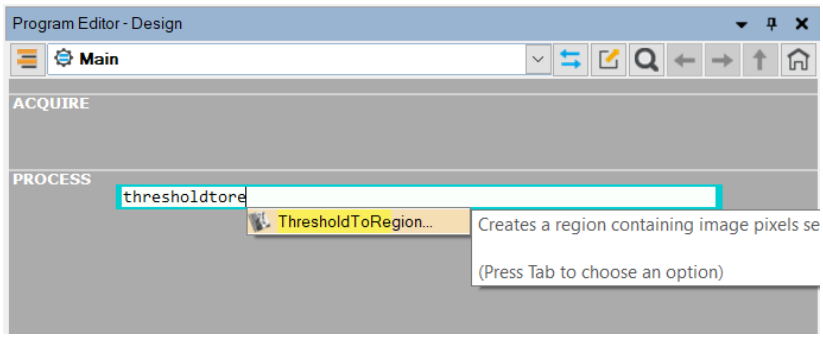
Ctrl+Space / Ctrl+T

When you know the name of a filter, which you would like to add into your program, you can use a keyboard shortcut **Ctrl+Space** or **Ctrl+T** to find it straightaway in the Program Editor, without having to open the Toolbox's Libraries view. After applying this shortcut, you are prompted to type a filter name:



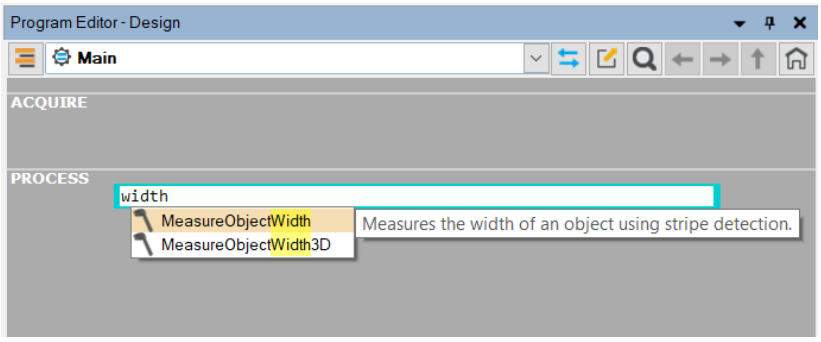
Advanced search using CTRL+SPACE shortcut.

Let us assume that you need to perform simple thresholding on your input image. You already know that there is a ThresholdToRegion filter in our library, which you have utilized many times so far. Instead of time-consuming searching in either Toolbox's Tools view or Libraries view, you can quickly access the desired function by typing its name:



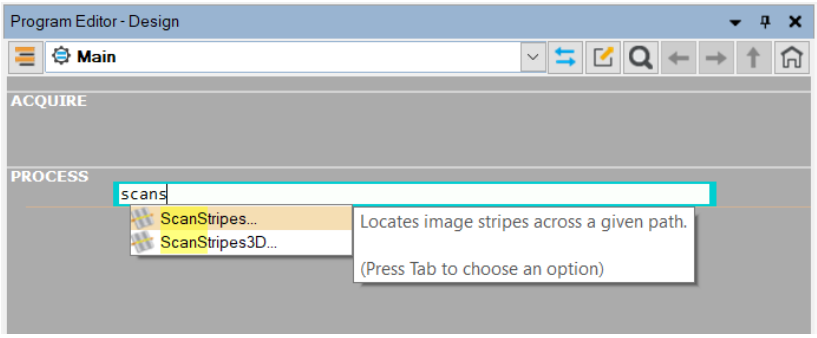
Browsing for a particular filter.

If you do not remember well the name of a filter, you can track it as well by typing only a part of it:



Searching a filter without its full name.

The search engine in the Program Editor also allows you to display the description of a filter in case you want to make sure or remind yourself what the filter is for:



The description of a filter.

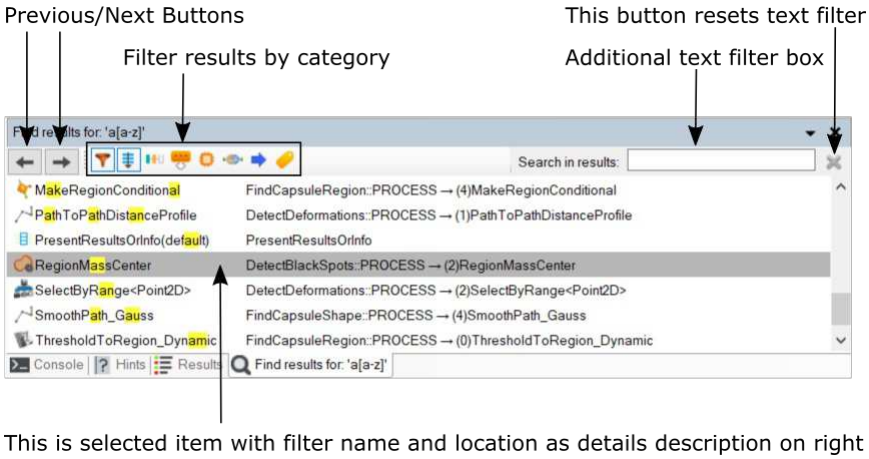
Search Window

Ctrl+F

Creating a large program in Aurora Vision Studio may require finding elements in it's structure. Using Search Window is the best way to find Filters (tools), Macrofilters, Variants and Global parameters which already exist in the project. To open the search window find a Magnifier button in Program Editor or press Ctrl+F. In the search window insert the name of an element you want to find or a part you remember. Press 'Find' button and results of search will appear in the new window. You may also pick some search options like using case sensitivity, matching whole names or regular expression which can narrow your search.

Browsing for elements in project.

In Search Results window you may select element on the list to indicate an object in project. It is possible to filter results with buttons on panel on the top of the window. Additionally if there are too many results they can be filtered with keyword given in the box on top-right.



Managing search results.

Rules

Entering a search phrase also allows to pre-filter data with some keywords. When your project is large it might be useful to use Rules in the search phrase. Here are some examples of using Rules in Search Window.

Name	Syntax	Example	Constraint
Parent	`parent:<MacrofilterName>`	`rectangle parent:Initialize`	Must have an instance in Macrofilter <MacrofilterName>.
Input	`in:<Name>`	`load in:File`	Must contain an input which either name or type contains phrase <Name>.
Output	`out:<Name>`	`load out:Integer`	Must contain an output which either name, type or Data Source Label contains phrase <Name>.

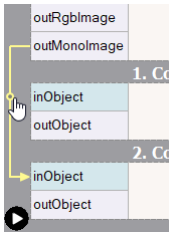
Connecting and Configuring Filters

After a filter is added to the program it has to be configured. This consists in setting its inputs to appropriate constant values in the Properties window or connecting them with outputs of other filters. It is also possible to make an input linked with an external AVDATA file, connected with HMI elements or with Global Parameters.

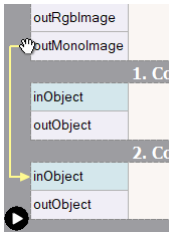
Connecting with Other Filters

Filters receive data from other filters through connections. To create a connection, drag an output of a filter to an input of a filter located below in the Program Editor (upward connections are not allowed). When you drag an output - possible destinations will be highlighted. It is basic way to create program's work-flow.

For convenience, new connections can also be created by forking existing connections (drag from the vertical part of the existing connection):



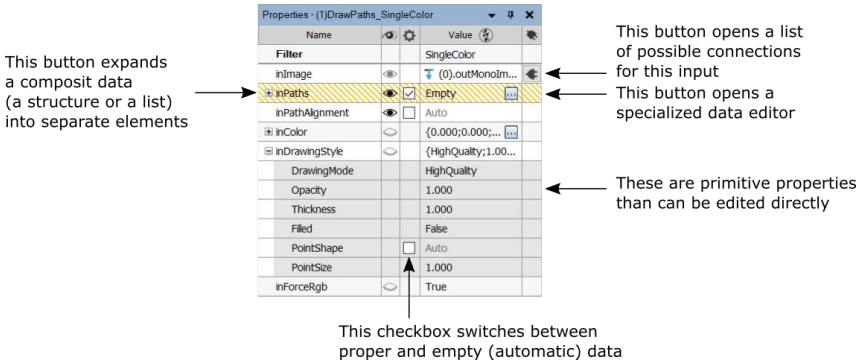
Moreover, it is possible to reconnect an existing connection to another output or input port by dragging the connection's head (near the destination port) or the connection's first horizontal segment (near the source port).



Another way to create connections between filters is by using only the Properties window. When you click the plug ( ) icon in the right-most column, you get a list with possible connections for this input. When the input is already connected the plug icon is solid and you can change the connection to another one.

Setting Basic Properties

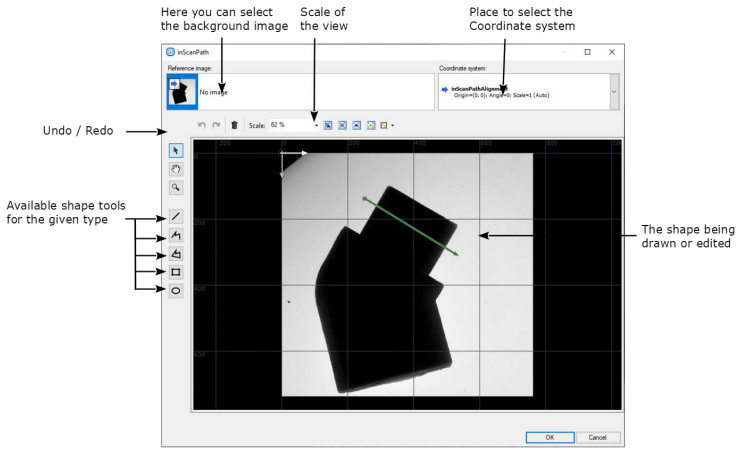
Most of the filters have several parameters which you can set in the Properties window as shown on the picture below. It is very important to go through these parameters is order to get desired results for your specific application. To start, first select a filter in the Program Editor window.



Note: After clicking on the header of the properties table it is possible to choose additional columns.

Editing Geometrical Primitives

To edit geometrical data, such as line segments, circle, paths or regions, click the three dots button ( ) in the Properties window at the input port you want to set or modify. A window similar to the one below will appear. The first thing you will usually need to do, when you open this window for the first time, is to select the background image from the list at the top of the window. This will set a context for your geometric data.



Editing of a path in a context of an image.

Tips:

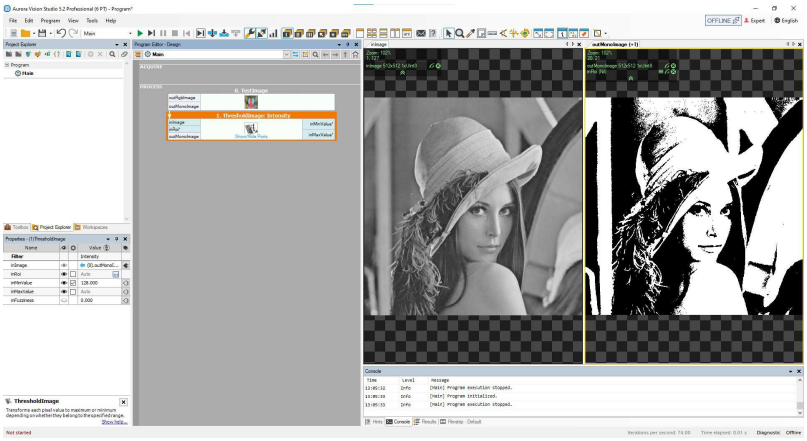
- Select a point and use its context menu to inspect or set the numeric coordinates.
- Use the mouse wheel to zoom the view without changing the tool.
- Hold 3rd mouse button and drag to move the view without changing the tool.
- Hold *Ctrl* to limit the segment angles to the multitudes of 15 degrees.

Testing Parameters in Real Time

One of the greatest features of Aurora Vision Studio is its orientation on rapid development of algorithms. This encompasses the ability to instantly see how different values of parameters affect the results. Due to the dynamic nature of this feature it cannot be presented in a static picture, so please follow the instructions:

1. Create a simple program with a TestImage filter connected to a ThresholdImage filter.
2. Put the output of the ThresholdImage to a data preview.
3. Click *Iterate Current Macro* to execute the program to the last filter, but without ending it as the whole (the execution state will be: *Paused*).
4. Select the ThresholdImage filter in the Program Editor and change its **inMinValue** input in the Properties window.

Now, you will be able to see in real time how changing the value affects the result.



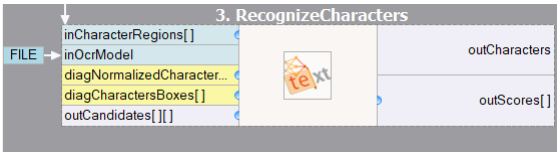
Re-executing a filter after a change of a parameter.

Remarks:

- Please note, that although being extremely useful, this is a "dirty" feature and may sometimes lead to inconsistencies in the program state. In case of problems, stop the program and run it again.
- This feature works only when the filter has already been executed and the program is *Paused*, but NOT *Stopped*.
- By default re-executed is the entire macrofilter. By unchecking the "Global Rerun Mode" setting the re-execution can be limited to a single filter. This can be useful when there are long-lasting computations in the current macrofilter.
- It is not possible to re-execute i/o filters, loop accumulators or loop generators (because this would lead to undesirable side effects). These filters are skipped when the entire macrofilter is getting re-executed.
- When re-executing a nested instance of another macrofilter, the previews of its internal data are NOT updated.
- Re-executing some filters, especially macrofilters, can take much time. Use the Stop command (Shift+F5) to break it, when necessary.
- If you set an improper value and cause a *Domain Error*, the program will stop and it will have to be started again to use the re-execution feature.
- The filter parameters can also be modified during continuous program execution (*F5*).

Linking or Loading Data From a File

Sometimes data values that have to be used on an input of a filter are stored in an .avdata file, that has been created as a result of some Aurora Vision Studio program (for example creating a custom OCR model). It is possible to load such data to the program with the LoadObject filter, but most often it is more convenient and more effective to link the input port directly to the file. To create such a link choose *Link from AVDATA File...* from the context menu of the input (click with the right mouse button on an input port of a filter).

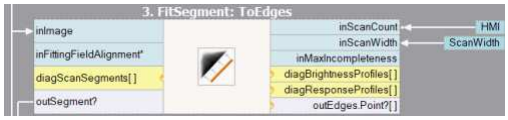


Input data linked from an AVDATA file.

It is also possible to load, not link, data from an .avdata file. This is done with the *Import from AVDATA File...* command in the context menu, which copies the data and makes them part of the current project.

Connecting HMI and Global Parameters

It is also possible to connect filter inputs from outputs of HMI elements and from Global Parameters. Both get displayed as rectangular labels at the sides of the Program Editor as can be seen on the image below:



A filter with connections from HMI and from a Global Parameter.

For further details please refer to the documentation on the specific topics:

- HMI Designer
- Global Parameters

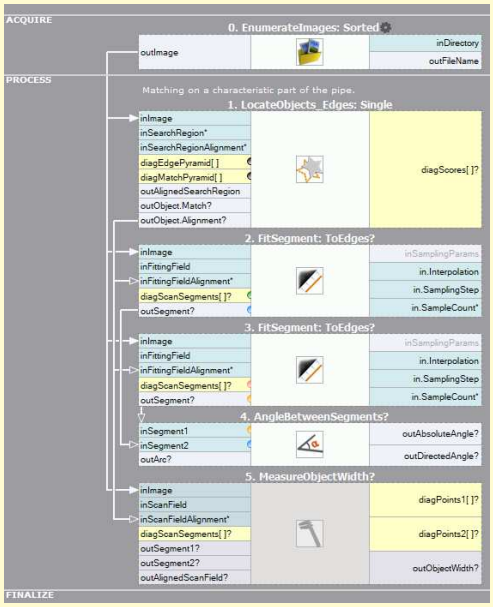
Writable and Readable Global Parameters

It is possible to read and write the value of global parameter by using **ReadParameter** and **WriteParameter** filters. This approach allows users to change global parameter values accordingly to their needs at any point of an algorithm. A few practical applications of this feature are listed below:

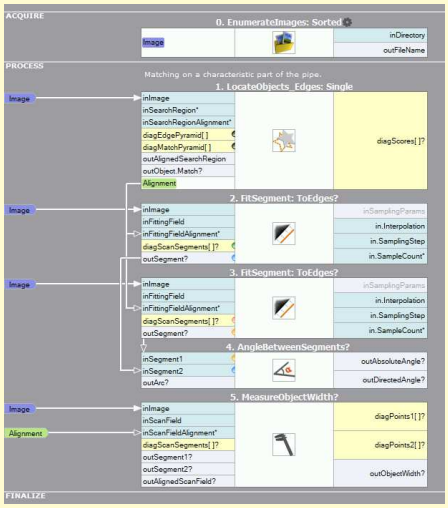
- Managing recipes depending on a signal from PLC,
- Storing data transferred between nested macrofilters,
- Setting global flags.

Labeling Connections

Connections are easier to follow than variables as long as there are not too many of them. When your program becomes complicated, with many intersecting connections, its readability may become reduced. To solve this problem Aurora Vision Studio provides a way to replace some connections with labels. You can right-click on a connection, select "Label All Connections..." - when there is more than one connection or "Label Connection..." when only one connection is present. Then set the name of a label that will be displayed instead. The labels are similar to variables known from textual programming languages – they are named and can be more easily followed if the connected filters are far away from each other. Here is an example:



A program with some long connections becomes not easy to analyze.



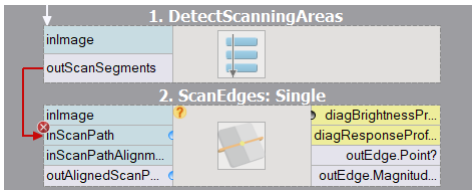
Long connections are replaced with labels for better readability.

Remarks:

- Labeled connections can be brought back (unlabeled) by using the "Un-label This Connection" or "Un-label All Connections" commands available in the context menu of a label.
- Please note, that when your program becomes complicated, the first thing you should consider is reducing its complexity by refactoring the macrofilter hierarchy or rethinking the overall structure. Labeling connections is only a way to visualize the program in a more convenient way and does not make its structure any simpler. It is the user's responsibility to keep it well organized anyway.
- Aurora Vision Studio enforces that all connections between filters are clearly visualized, even if making them implicit would make programming easier in typical machine vision applications. This stems from our design philosophy that assumes that: (1) it is wrong to hide something that the user has to think about anyway, (2) the user should be able to understand all the details of a macrofilter looking at a static screen image. When the amount of connections becomes large despite good program structure you should also consider creating User Structures that may be used for merging multiple connections into one. (Do NOT use global parameters for that purpose).

Invalid Connections

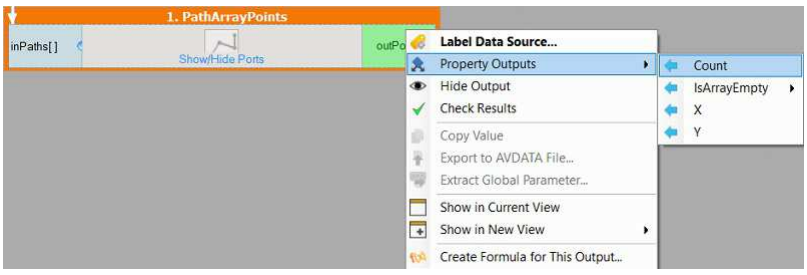
As types of ports in macrofilters and formulas may be changed after connections with other filters have been created, sometimes an existing connection may become invalid. Such an invalid connection will be marked with red line and white cross in the Program Editor:



Invalid connections do not allow to run the program. There are two ways to fix it: replace the invalid connections or revise the types of the connected ports.

Property Outputs

In addition to available filter's outputs, it is possible to get much more information out of a filter. Many data types are represented as structures containing fields, e.g. Point2D consists of "X" and "Y" fields of Real data type. Although these fields are not available as standard outputs, a user can easily add them as additional filter outputs - we call them "Property Outputs". That way, they are directly available for creating connections with inputs of other filters.



Accessing fields of Point2DArray type.

Additional Property Outputs

Some of the properties are calculated based on the current state of the output (for example, checking whether the array is empty or counting its elements), whereas other are derived directly from the internal structure of the type (for example, "X" and "Y" fields composing Point2D structure). The types providing such properties are listed in the table below.

Structure name	Property outputs
Bool	Not
ByteBuffer	Size IsByteBufferEmpty
Histogram	Size
Matrix	IsMatrixEmpty
Path	Points Size Length IsPathEmpty
Profile	Size IsProfileEmpty
Region	Area IsRegionEmpty
Segment2D	Length Direction Center
String	Length IsStringEmpty
Vector2D	Length Direction
Vector3D	Length

There are also special types, which cannot exist independently. They are used for wrapping outputs, which may not be produced under some conditions (Conditional) or optional inputs (Optional), or else for keeping a set of data of specified type (Array). Property outputs of such types are listed in the table below.

Type name	Property outputs
Array	Count IsArrayEmpty
ArrayArray (array of arrays)	Count IsArrayEmpty IsNestedArrayEmpty
Conditional	IsNil
Optional	IsNil

All of the above-mentioned property outputs are specific for the type. However, ports of Array, Optional or Conditional type may have more property outputs, depending on the wrapped type. For example, if the output of a filter is an array of regions, this port will have Count, IsArrayEmpty (both resulting from the array form of the port), IsRegionEmpty and Area (both resulting from the type of the objects held in the array - in this case, regions) as property outputs. However, if the output of a filter is an array of objects without any property outputs (e.g. Integer), only outputs resulting from the array form of the port (Count and IsArrayEmpty) will be available.

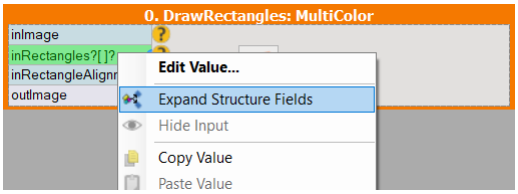
Tip: Avoid using basic filters like Not, ArraySize which will have bigger performance impact than additional property outputs.

What Do *IsNil* and *IsEmpty* Mean:

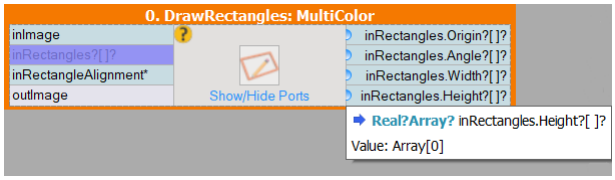
These property outputs return a bool value depending on the content of the data. If the data is *Nil*, what usually means that no object was detected, the property output *IsNil* is set to *True* otherwise it remains *False*. The *IsEmpty* property has a similar use. The only difference is that it reacts if the inputted object is *Empty*. In most common cases one use it for conditional execution of macrofilters or as an input date to formulas. You can read how to deal with *Nil* and *Empty* objects in an article about conditional execution

Expanded Input Structures

Analogously to expanding output properties, it is also possible to expand input structures. This works only for basic structures which allow access to all their fields (e.g. Point2D, but not Image). Please note, that the expanded input structure is replaced by the fields it consists of (unlike by adding property outputs, where the structure itself remains available).



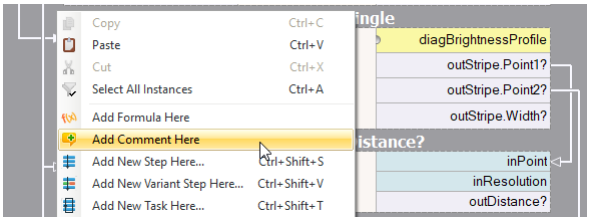
Expanding filter input of Rectangle2D?Array? type.



Result of expanding input of Rectangle2D?Array? type.

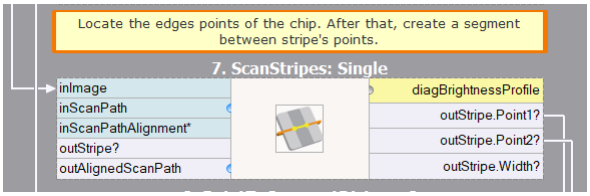
Comment Blocks

Comments are very important part of every computer program. They keep it clear and easy to understand during development and further maintenance. There are two ways to make a comment in Aurora Vision Studio. One way is to add a special comment block. Another option is adding a comment directly in the filter. To add a new comment block to program click the right mouse button on the Program's Editor background and select the "Add Comment Here" option like on the image below.



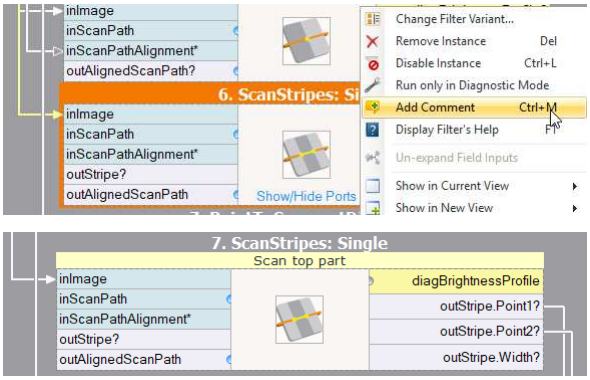
Adding a new comment.

Comment block can be very useful for describing the details of the algorithm.



Comments can be used for describing program steps.

But when you need just a simple tip or short remark, use a "Add Comment" after mouse right click on the filter:



New comment directly in the filter.

Creating Macrofilters

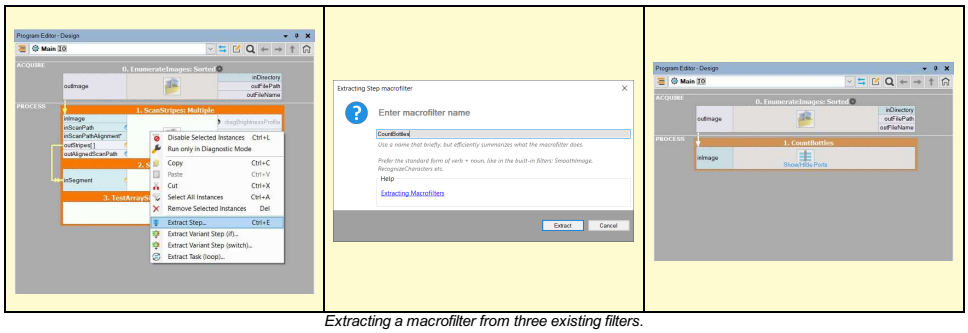
Introduction

Macrofilters play an important role in developing bigger programs. They are subprograms providing a means to isolate sequences of filters and re-use them in other places of the program. After creating a macrofilter you can create its instances. From outside an instance looks like a regular filter, but by double-clicking on it you navigate inside and you can see its internal structure.

When you create a new project, it contains exactly one macrofilter, named "Main". This is the top level macrofilter from which the program execution will start. Further macrofilters can be added by extracting them from existing filters or by creating them in the Project Explorer window.

Extracting Macrofilters (The Quick Way)

The most straightforward way of creating a macrofilter is by extracting it from several filters contained in an existing macrofilter. This is done by selecting several filters in the Program Editor and choosing the *Extract Step...* command from the context menu, as depicted in the picture below:



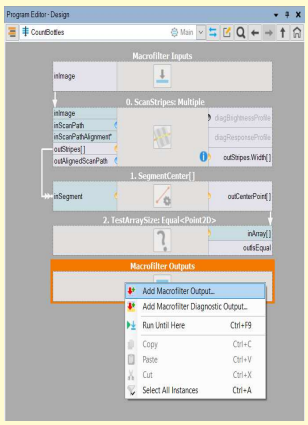
Extracting a macrofilter from three existing filters.

After completing this operation, a new macrofilter definition is created and the previously selected filters are replaced with an instance of this macrofilter. Now, additional inputs and outputs of the new macrofilter can be created by dragging and dropping connections over the new macrofilter instance.

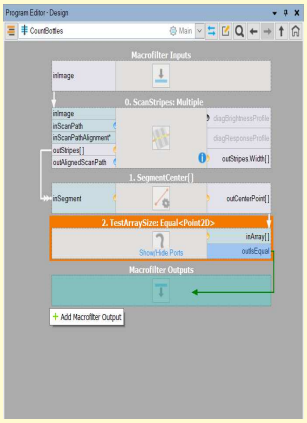
Remark: The "Extract Task (loop)..." command creates a new Task macrofilter which should be used only in special cases. Execution of the Task macrofilter is more complex than execution of the Step macrofilter. Usage of the Task macrofilter for reducing macrofilter complexity may be inappropriate. For more details please read section about Task macrofilters.

Defining the Interface

Being inside of a macrofilter other than "Main" you can see two special blocks: the *Macrofilter Inputs* and the *Macrofilter Outputs*. The context menus of these blocks allow to add new inputs or outputs. Another method of adding them is by dragging connections and dropping them over these blocks.



Adding a new output using the context menu of the Macrofilter Outputs block.



Adding a new output by dragging and dropping a connection.

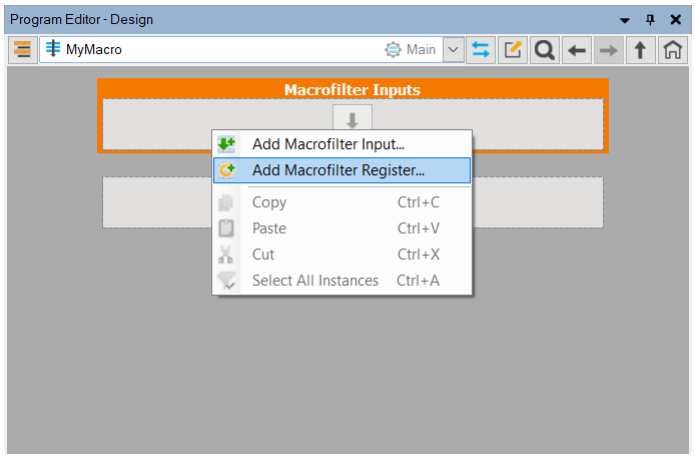
Before the new port is created you need to provide at least its name and type:

Defining a port of a macrofilter.

NOTE: Names of macrofilter inputs and outputs should be clear and meaningful. Names of inputs always start with "in", while names of outputs always start with "out".

Adding Registers

The context menus of macrofilter input and output blocks also contain a command for adding macrofilter registers, *Add Macrofilter Register...*. This is an advanced feature.

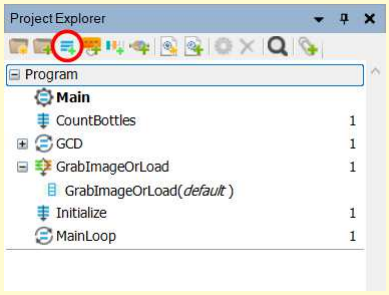


Creating Macrofilters in the Project Explorer

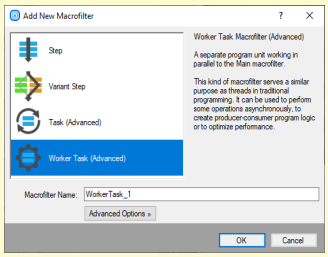
All macrofilter definitions that are contained in the current project can be displayed in the Project Explorer window (by default it is behind the Toolbox on a tab page). Using this control, the user can create or edit macrofilters, but it also acts as a filter catalog from which instances can be created by dragging and dropping the items into the Program Editor.

To create a new macrofilter in the Project Explorer click the *Create New Macrofilter...* button. A new window will appear allowing you to select the name and structure of the new macrofilter.

Please note, that due to their special use *Worker Tasks* can only be created in the Project Explorer. For more information on *Worker Tasks* please refer to Macrofilters article.



Adding Macrofilters in the Project Explorer.



Selecting the name and the structure of a new macrofilter.

Copying Macrofilters

When copying macrofilters in the Program Editor window, you can only create their new instances: modifying one instance will inevitably affect all the others. In order to duplicate the definitions (create brand-new copies) of old macrofilters as independent entities that you will be able to modify separately, you need to do it inside the Project Explorer window.

Copying macrofilters in the Project Explorer - creating new definitions.

Copying macrofilters in the Program Editor - creating new instances.

Please keep in mind, that regardless of whether you copy macrofilters in the Program Editor or the Project Explorer, with other macrofilters nested inside, no new definitions of nested macrofilters will be created.

1. When copying the parent macrofilter (first nest level) in the Program Editor, the number of instances of nested (child) macrofilters will not change.

2. When copying the parent macrofilter (first nest level) in the Project Explorer, new instances of nested (child) macrofilters will be created.

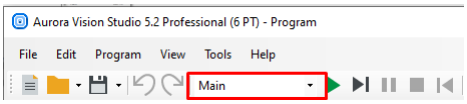
If you want to create a new copy of the whole macrofilter tree (copy of all definitions) you will have to copy each macrofilter separately, starting with the most nested one.

The StartUp Program

It is sometimes useful to have several programs in a single project. The most common scenarios are:

- When some more complex data e.g. custom OCR models or calibrated reference images, need to be prepared in a separate program.
- When it is convenient to have multiple versions of a single program customized for different installations.
- When the highest reliability is important and sets of automated tests need to be performed on recorded images.

In Aurora Vision Studio a program can be any Worker Task macrofilter. The list of all macrofilters fulfilling this criterion can be seen in the StartUp Macrofilter combo box on the Application Toolbar. This control makes it possible to choose the program that will be run. The related macrofilter will be displayed in the Project Explorer with the bold font.



The StartUp Program Combo Box.

Macrofilter Guidelines

Macrofilters organize big projects, but it is responsibility of the user to make this organization clean and effective. When a project grows, especially under the pressure of time, it is easy to forget this and create programs that are difficult to understand and maintain. To avoid this, please follow the following guidelines:

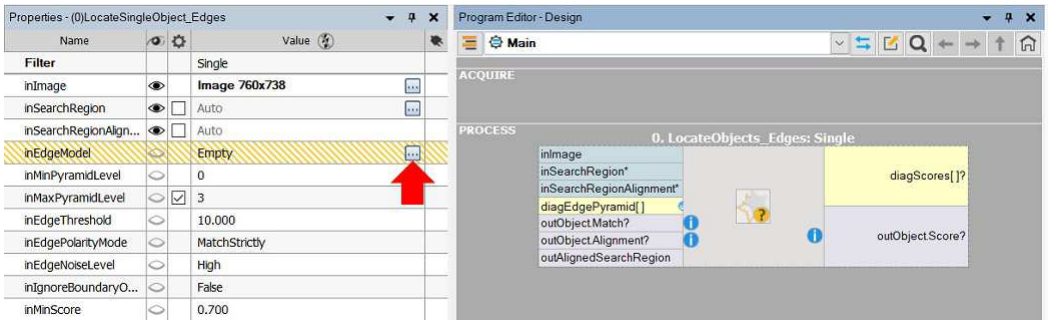
1. Macrofilters should not be too big. For program clarity they should typically consist of 3-15 filters. If you macrofilters have more than 20, it is certainly going to be cause trouble.
2. Each macrofilter should have a clear and single function well reflected in its name. It is recommended to create macrofilters as tools that could possibly be used in many different projects.
3. Macrofilter names do not affect the program execution, but they are very important for effective maintenance. Choose precise and unambiguous English names. Follow the *verb + noun* naming convention whenever possible and avoid abbreviations (this applies also to names of macrofilter ports).
4. Do not mix data analysis with things like communication or visualization in a single macrofilter. These should be separated. If data visualization is needed for HMI, compute results in one macrofilter and then prepare the visualization in another. This will make the data flow more clear.
5. During project development it is very common that the initial program structure becomes inappropriate for what has been added during development, or that it becomes too complicated and unclear. This is when you should consider REFACTORING, i.e. redesigning the program structure, boundaries of macrofilters and revising any complicated sequences of filters. We strongly recommend considering refactoring as a routine part of the job.

Creating Models for Template Matching

Introduction

Template Matching tools are very often used as one of the first steps in industrial inspection applications. The goal is to detect the location of an object. Before this can be done the user has to create a model representing the expected object's shape or structure. To make this step straightforward, Aurora Vision Studio provides an easy user interface. We call it a "GUI for Template Matching".

The GUI for Template Matching is an editor for values of two types: EdgeModel and GrayModel. This means that to open it the user has to select a template matching filter in the program and then click on the button at the **inGrayModel** or **inEdgeModel** input in the Properties window:



Opening GUI for Template Matching.

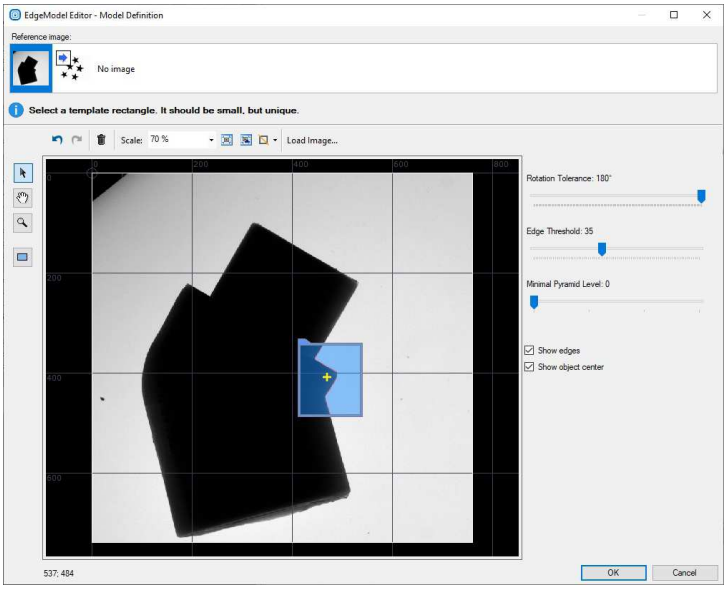
In majority of industrial applications Template Matching algorithms implemented in Aurora Vision Studio easily detect the location of an object using default parameters. However, there are cases where it is needed to tune some of them, mainly in order to achieve higher reliability. It turned out that there is a group of parameters, which is used more often than others. Therefore GUI for Template Matching is available in two variants: Basic and Expert. This is related to Complexity Levels available in Aurora Vision Studio.

Creating a Model

Basic

The Basic window contains the following elements:

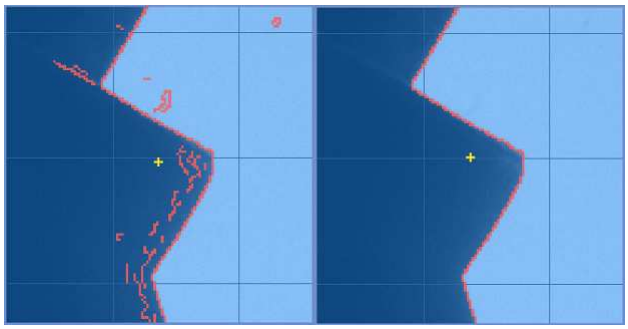
1. At the top: a list of possible template images
2. Below: a simple toolbar that also contains a button for loading a template image from a file
3. On the left: a tool for selecting a rectangular template region
4. On the right: track-bars for setting parameters and some options related to the view
5. In the center: an area for selecting the template region in the context of the selected template image



Basic GUI for Template Matching window

To create a template matching model you need to:

1. Choose a template image from the "Reference image" list.
2. Select a rectangular template region using drawing tool. For maximum performance this rectangle should be as small as possible.
3. Edge-based matching only: Set the **Edge Threshold** parameter, which should be set to value that results in the best quality of the edges. **Edge Threshold** determines the minimum strength (gradient's magnitude) of pixels that can be considered as an edge.



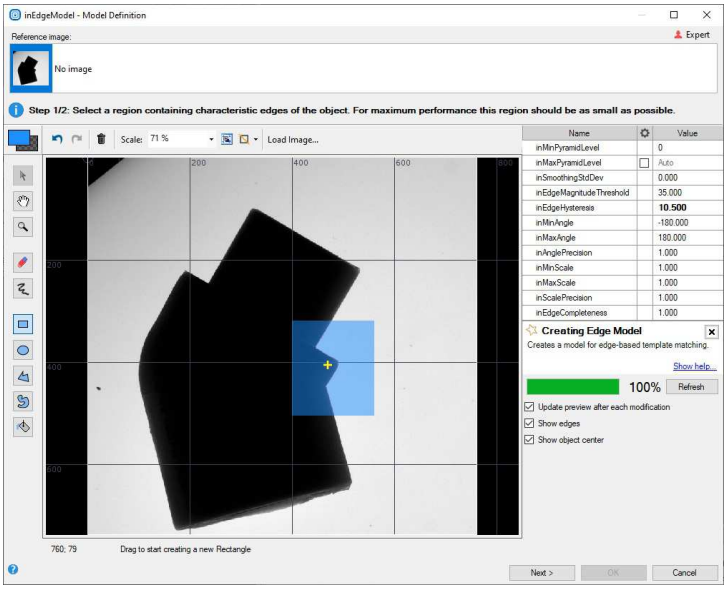
Low quality edges (Edge Threshold = 8) and high quality edges (Edge Threshold = 30)

4. Set the **Rotation Tolerance** in range from 0° to 360°. This parameter determines the maximum expected object rotations. Please note that the smaller the **Rotation Tolerance**, the faster the matching.
5. Set the **Minimal Pyramid Level** parameter which determines the lowest pyramid level used to validate candidates who were found on the higher levels. What is worth mentioning is that setting this parameter to a value greater than 0 may speed up the computation significantly, however, the accuracy of the matching can be reduced. More detailed information about Image Pyramid is provided in Template Matching document in our Machine Vision Guide.

Expert

The Expert window contains the following elements:

1. At the top: a list of possible template images
2. Below: a simple toolbar that also contains a button for loading a template image from a file
3. On the left: a tool for selecting a template region of any shape
4. On the right: parameters of the model, their description and some options related to the view
5. In the center: an area for selecting the template region in the context of the selected template image

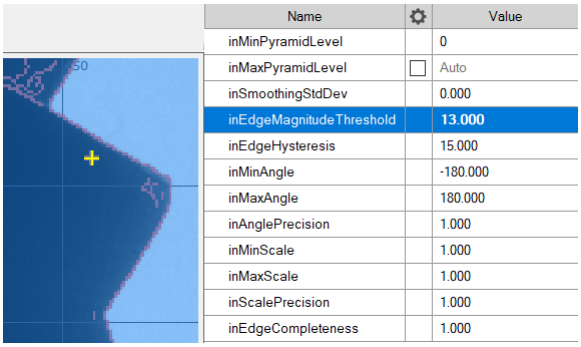


GUI for Template Matching window

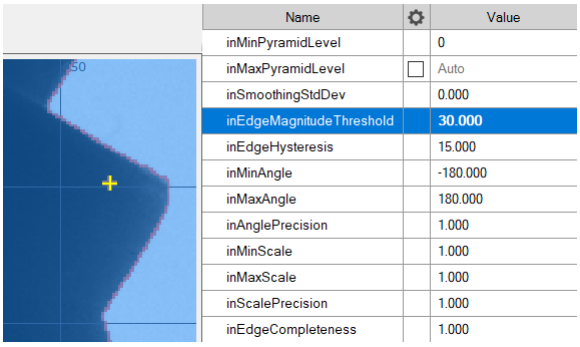
To create a template matching model you need to:

1. Choose a template image from the "Reference image" list.
2. Mark a template region using drawing tools (please note, that the button above the tools switches the current color, i.e. you can both draw or erase shapes). For maximum performance this region should be as small as possible.

3. Edge-base matching only: Set the **inSmoothingStdDev**, **inEdgeMagnitudeThreshold** and **inEdgeHysteresis** parameters to values that result in the best quality of the found edges. It is advisable to first set **inEdgeHysteresis** to zero, then choose a value for **inEdgeMagnitudeThreshold** that assures that all edges have some parts detected and then increase **inEdgeHysteresis**. Small noise might by removed by change the value of parameter **inSmoothingStdDev**. For example:

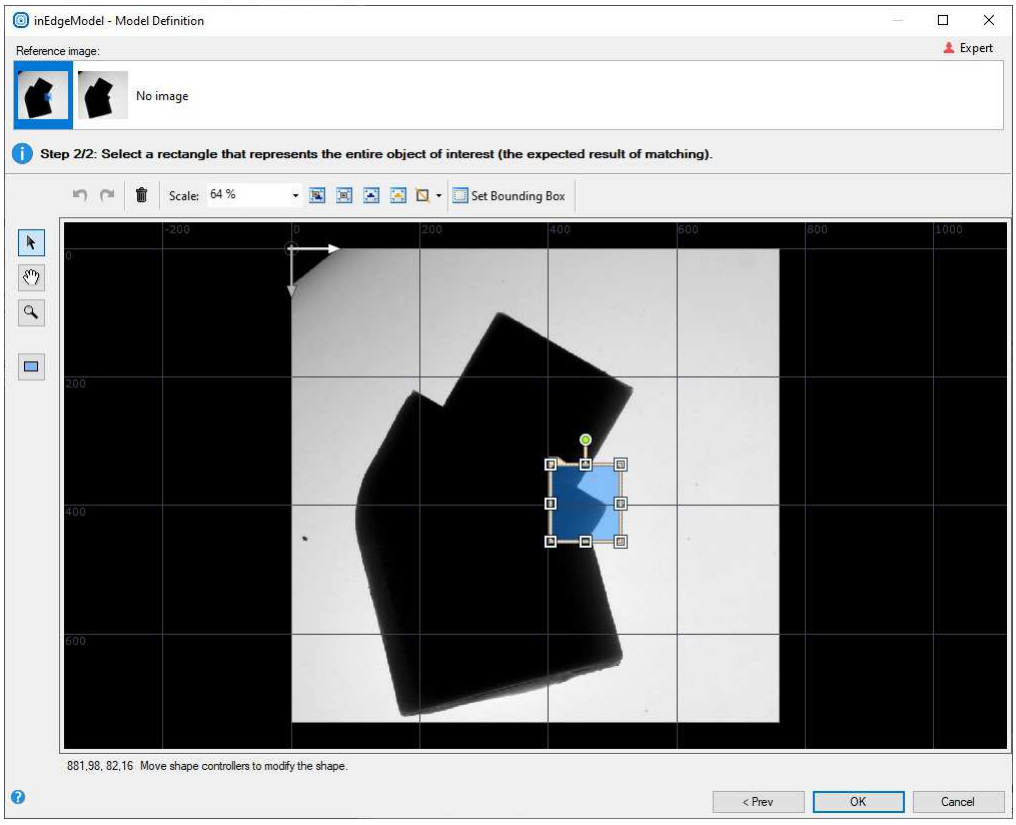


Low quality edges



High quality edges

4. Set the **inMinAngle** and **inMaxAngle** parameters accordingly to the expected range of object rotations (the smaller the range, the faster the matching).
5. Set the **inAnglePrecision** to a value lower that 1.0 if you prefer to lower the angular precision for the benefit of speed.
6. Set the **inMinScale**, **inMaxScale** and **inScalePrecision** to appropriate value if you need to detect objects in different scale. You should be aware of longer detection time when you detect objects in scale.
7. Click the "Refresh" button or check "Update preview after each modification" to review the results.
8. Click the "Next >" button to select template rectangle that represents the entire object of interest (the expected result of matching)

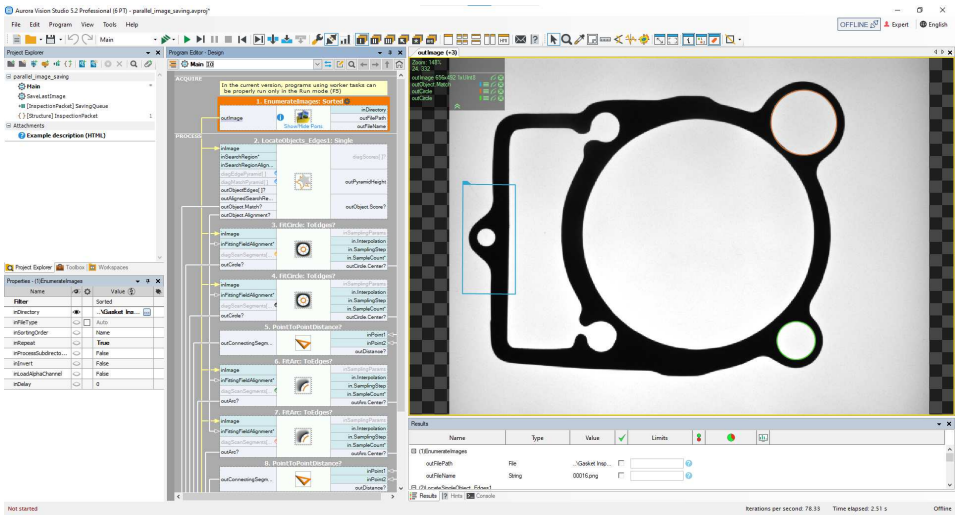


Template rectangle selection

9. Click "OK" to close the window and generate the model.

Performing Template Matching

When the model is ready, performing template matching in an application is straightforward – after connecting the filters and setting the matching parameters, the results are on the outputs of the LocateSingleObject_Edges1 filter (or similar):



Example result of template matching

See also:

Template Matching guide, Template Matching filters

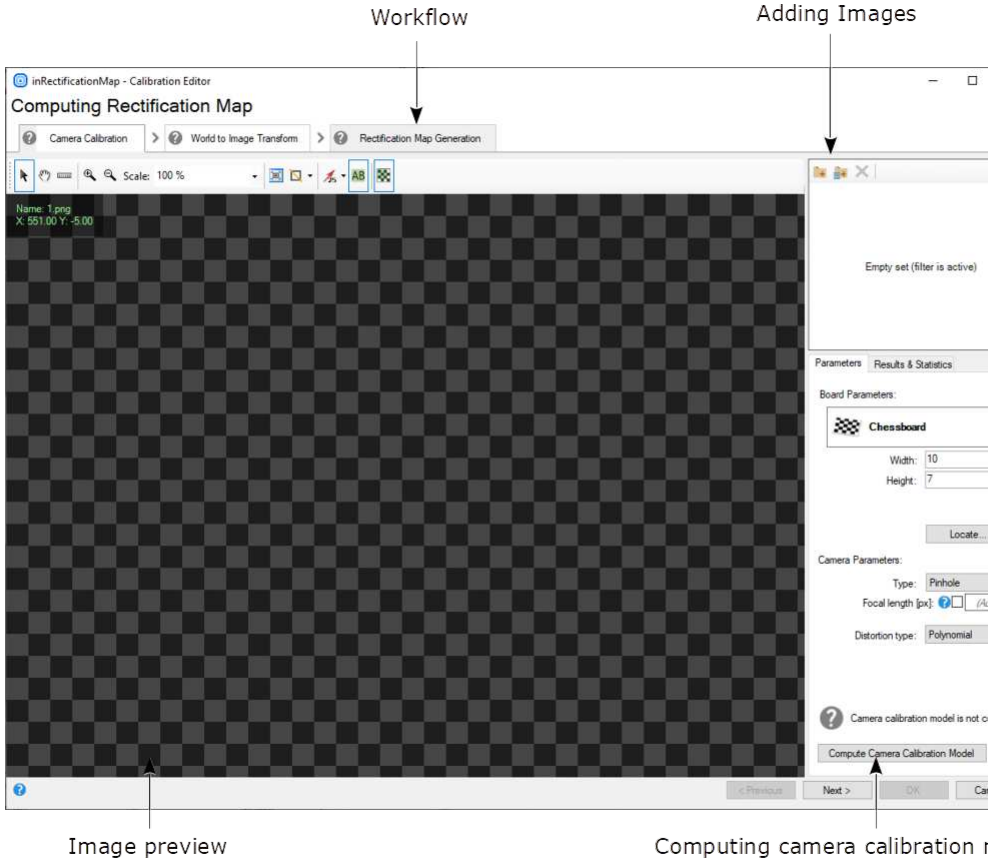
Preparing Rectification Transform Map

Overview

Before you read this article, make sure you are acquainted with Camera Calibration and World Coordinates.

If you want to perform camera calibration in Aurora Vision Studio, there two basic ways to do that. You can either use a calibration editor (plugin) or use filters (manual configuring parameters and connecting outputs with corresponding inputs). Both approaches provide you with the very same results, but the latter way allows you to control intermediate steps and outputs if they are relevant to you for some reason.

This guide will focus mainly on the first approach and show how to perform calibration with the editor step by step. To perform image rectification, the RectifyImage filter should be applied. When you click on the **inRectificationMap** input, the Calibration Editor will be displayed:



The overview of the Calibration Editor.

As you can see the editor consists of three pages:

1. **Camera Calibration** - in which camera lens parameters are computed.
2. **World to Image Transform** - in which perspective and transform converting real-world points to image points are computed.
3. **Rectification Map Generator** - in which fast pixel transform is computed to get a rectified image.

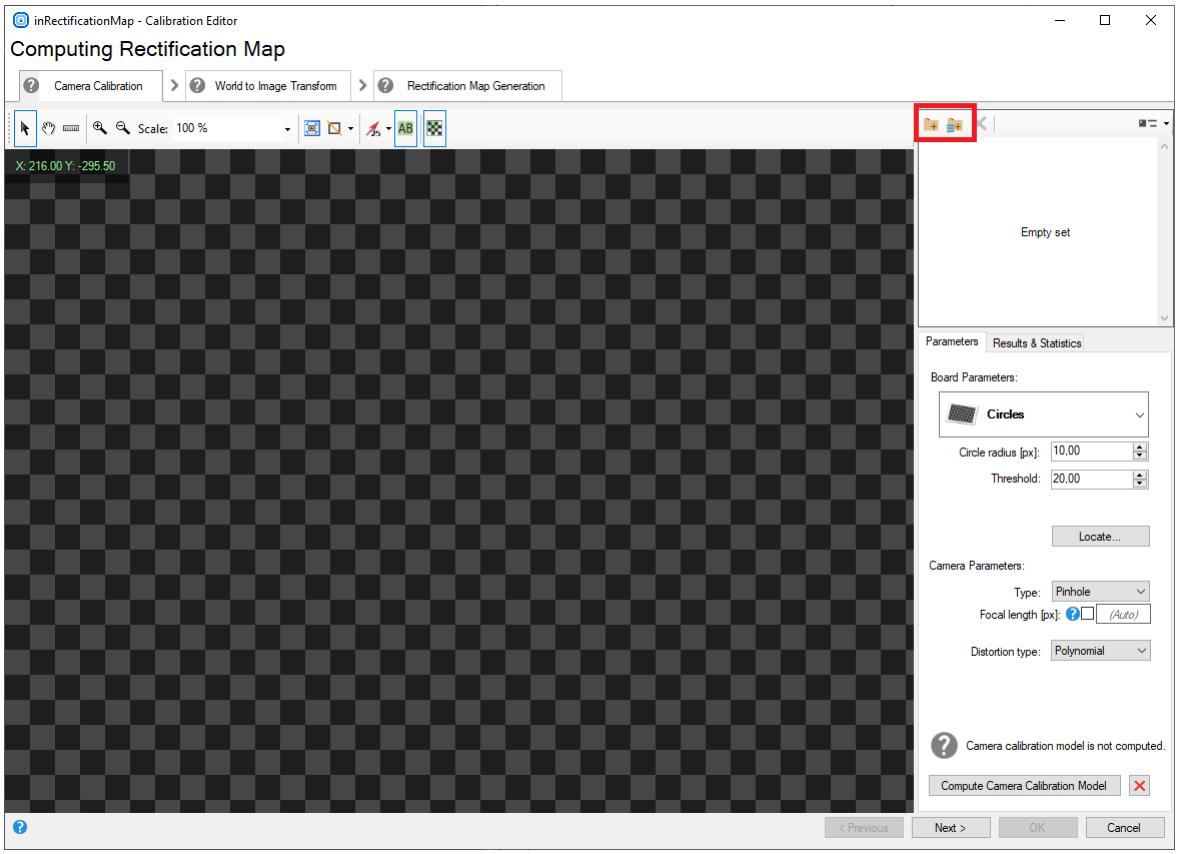
Each of the pages will be individually described, but as it has been already noted, each of the pages in the Calibration Editor invokes corresponding calibration filters. The general overview is in the table below:

Page name	Corresponding filters
Camera Calibration	CalibrateCamera_Pinhole CalibrateCamera_Telecentric
World to Image Transform	CalibrateWorldPlane_Default CalibrateWorldPlane_Labeled CalibrateWorldPlane_Manual CalibrateWorldPlane_Multigrid
Rectification Map Generation	CreateRectificationMap_Advanced CreateRectificationMap_PixelUnits CreateRectificationMap_WorldUnits

Camera Calibration Page

On the first page of the Calibration Editor you have to provide images of calibration boards, using the buttons marked in red.

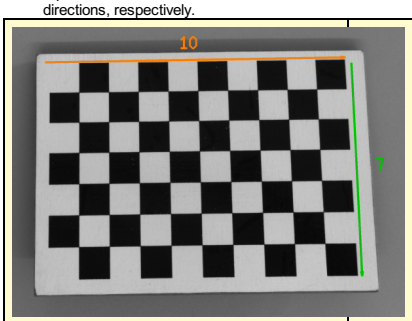
The purpose of this stage is to estimate intrinsic camera parameters. They do not depend on the transformation between the camera and its external world, therefore this step can be done before mounting the camera at the target place.



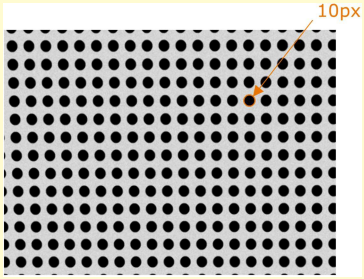
Camera Calibration Page - adding images.

Next, you need to specify board and camera parameters. There are two possible options to choose from, depending on what kind of pattern is used:

- **Chessboard** - where you have to define two parameters of the board: width and height, which correspond to the number of squares in horizontal and vertical directions, respectively.
- **Circles** - where you have to define a single circle's radius and threshold value.



If you are using chessboard pattern, you have to count squares in both dimensions.
In this example, the width is 10 and the height is 7.



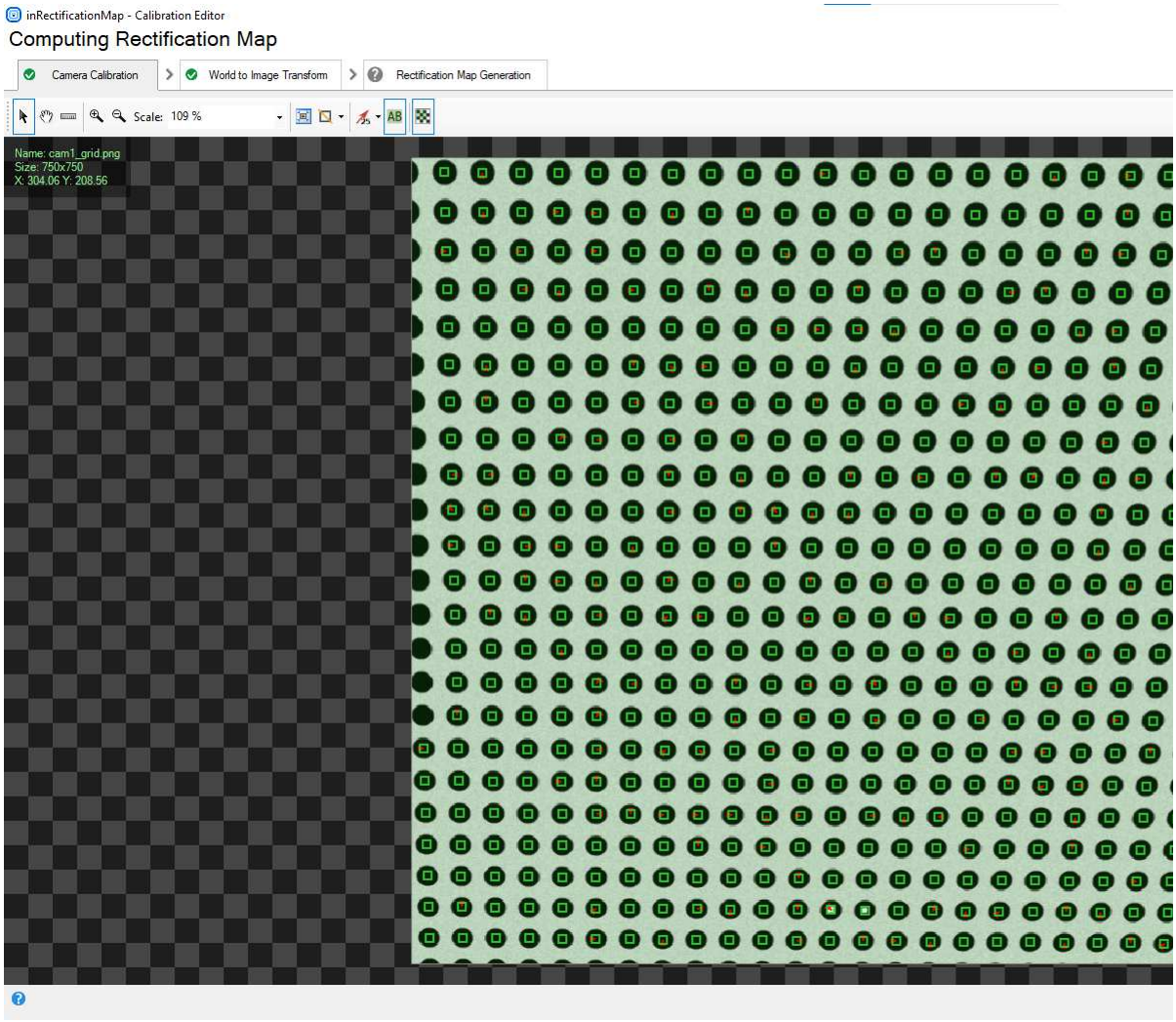
If you are using circle pattern, you have to measure radius of any circle.
In this example, it is about 10px. Note: it is important to use a symmetric board as shown in the image. Asymmetric boards are currently not supported.

Once they are set, you should adjust camera type according to an applied camera, which can be either **pinhole** (which uses a standard perspective projection) or **telecentric** (uses an orthographic projection).

A few distortion model types are supported. The simplest - divisional - supports most use cases and has predictable behavior even when calibration data is sparse. Higher order models can be more accurate, however, they need a much larger dataset of high quality calibration points, and are

usually needed for achieving high levels of positional accuracy across the whole image - order of magnitude below 0.1 px. Of course this is only a rule of thumb, as each lens is different and there are exceptions.

Results & Statistics tab informs you about results of the calibration, especially about reprojection error, which corresponds with reprojection vectors shown in red on the preview in the picture below:

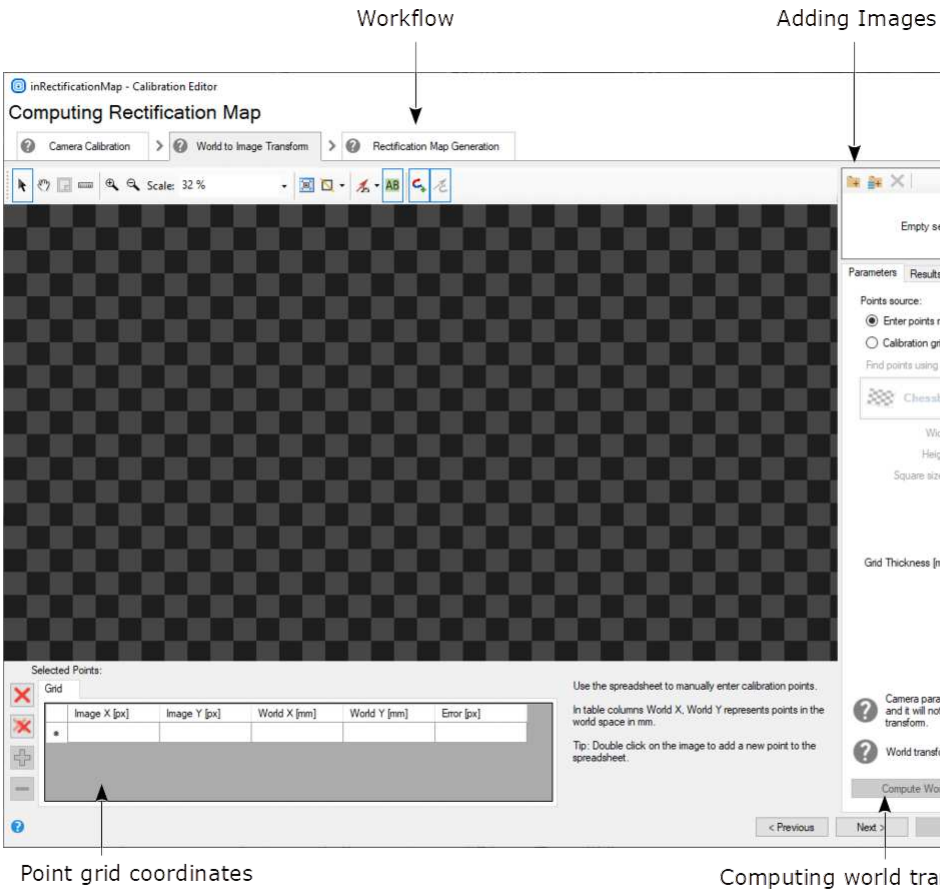


They indicate the direction towards each located point should be moved in order to deliver an immaculate grid. Additionally, there are values of Root Mean Square and Maximal error. The tab also displays value and standard deviation for each computed camera and lens distortion parameter.

For more details, please refer to the corresponding group of filters for this page in the table above.

World to Image Transform Page

Points added to the second page are used to find transformation between real-world and image points.



World to Image Transform - overview.

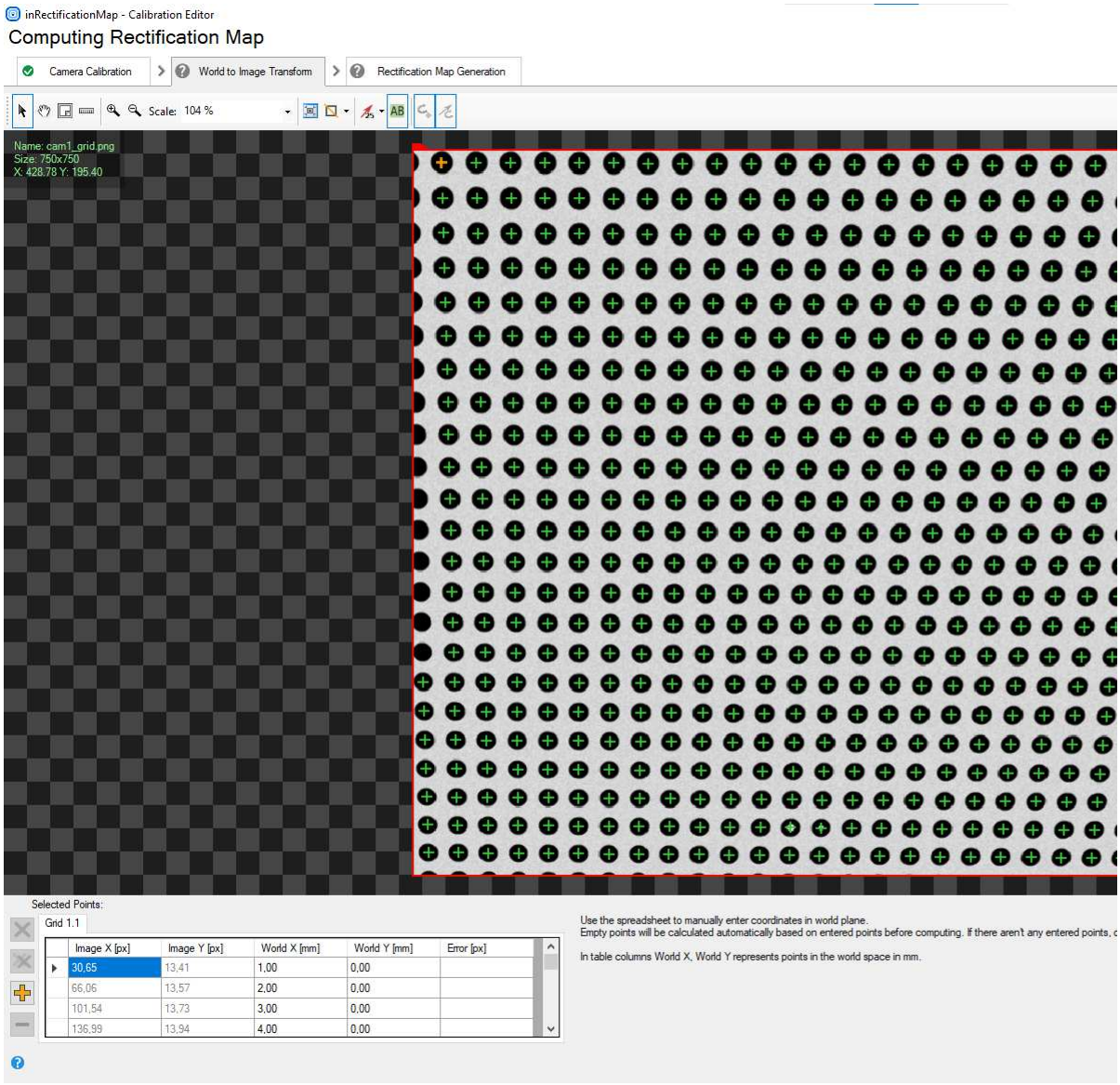
First of all, in this step, you need to add images of the calibration board as well (using the very same buttons as in the previous step). Please note that in this step they have to be taken from a fixed angle in contrast to the previous step, where all of images could be taken from different angles.

In the next step you should indicate points source in either of two ways:

- By **entering points manually**
- By using **calibration grid**

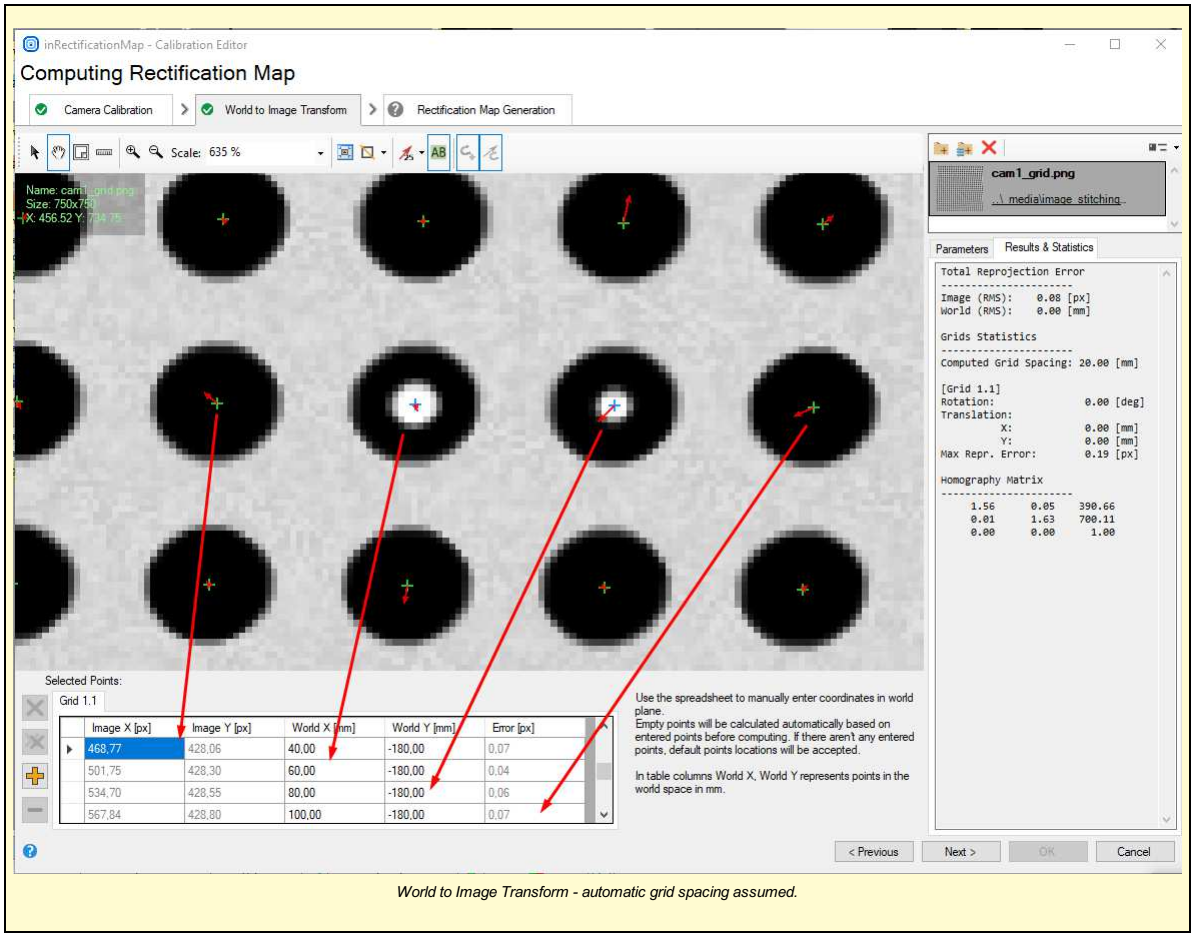
Points can be found automatically using the calibration board as in the previous step.

After successful points location on the calibration board, you should use the spreadsheet to manually enter coordinates in world plane:

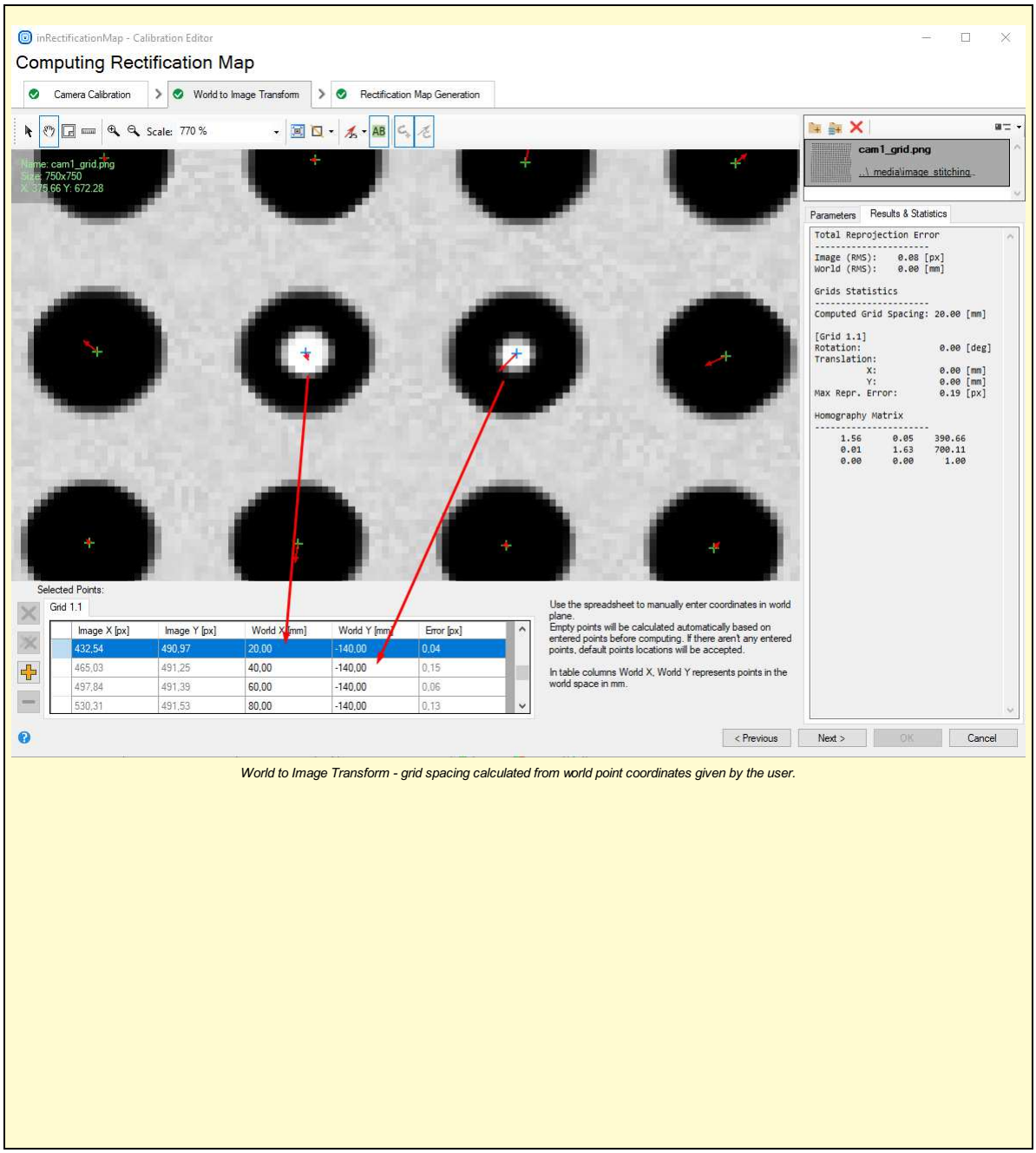


World to Image Transform - modifying points in the spreadsheet.

Points with unspecified world coordinates will have them calculated automatically based on points with specified coordinates. If there are no world coordinates entered, then algorithm will assume some default world point locations, what might produce false results. At least two grid point world coordinates are needed to uniquely determine the world plane position, rotation and spacing.



World to Image Transform - automatic grid spacing assumed.



World to Image Transform - grid spacing calculated from world point coordinates given by the user.

Arrows indicate which points from the calibration grid relate to corresponding rows in the spreadsheet. As you can see each row consists of coordinates in the image plane (given in **pixels**), coordinates in the world plane (given in **mm**), and error (in **pixels**), which means how much a point is deviated from its model location. In this case reprojection vectors, which are marked as small, red arrows, also indicate the deviation from the model.

Colors of points have their own meaning:

- Green Point - the point has been computed automatically.
- Orange Point - the point which has been selected.
- Blue Point - the point has been adjusted manually.

The *Results & Statistics* tab shows information about computed errors for both image and world coordinates.

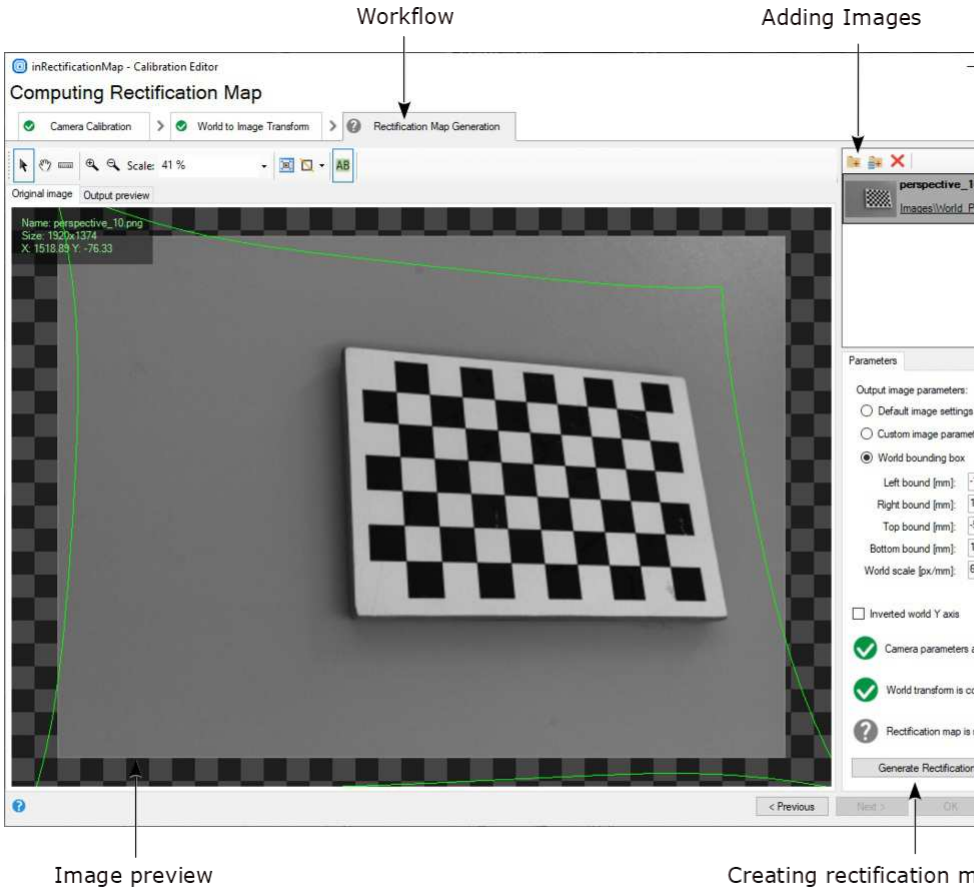
The output reprojection errors are useful tool for assessing the feasibility of computed solution. There are two errors in the plugin: Image (RMS) and World (RMS). The first one denotes how inaccurate the evaluation of perspective is. The latter reflects inaccuracy of labeling of grid 2D world coordinate system. They are independent, however they both influence quality of the solution, so their values should remain low.

For the details, please refer to the corresponding group of filters for this page in the table above.

Rectification Map Generator Page

Last page is used to set parameters of an image after the rectification.

First of all, you need to load images of the calibration board in the same way as in previous steps.



Rectification Map Generation - rectifying the image.

Next, you have to choose one of three basic options of the output image:

- **Default image settings**
- **Custom image parameters**
- **World bounding box**

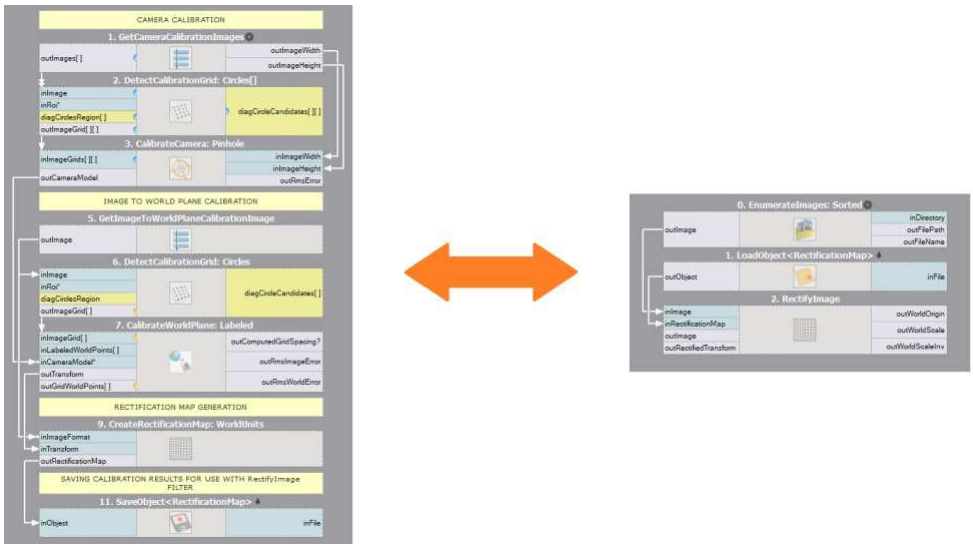
The green frame on the preview informs you about the size and borders of the output image, and depending on what option you have chosen, you can set different parameters. Using *custom parameters* or *world bounding box* will provide you with the same rectification map, but what makes the difference is that you are working in different domains - in the first case you are operating in the image coordinates (given in pixels), whereas in the other one you are operating in the world coordinates (given in millimeters).

When you are done, you can click on the **Generate Rectification Map** button and assess the rectified image displayed on the preview.

For the details, please refer to the corresponding group of filters for this page in the table above.

Relation between the Calibration Editor and filters

The relation between both approaches could be presented in a form of the below graphics:



The left side presents which filters are necessary to generate a rectification map and how to save it using SaveObject. The right side presents how to load the rectification map using LoadObject and passing it to the RectifyImage filter.

This is just an exemplary set of filters which might be applied, but using specific filters depend on the calibration board and other parameters relevant to a case.

Further readings

- Calibration-related list of filters in Aurora Vision Studio

Creating Text Segmentation Models

The graphical editor for text segmentation performs two operations:

1. **Thresholding** an image with one of several different methods to get a single foreground region corresponding to all characters.
2. **Splitting** the foreground region into an array of regions corresponding to individual characters.

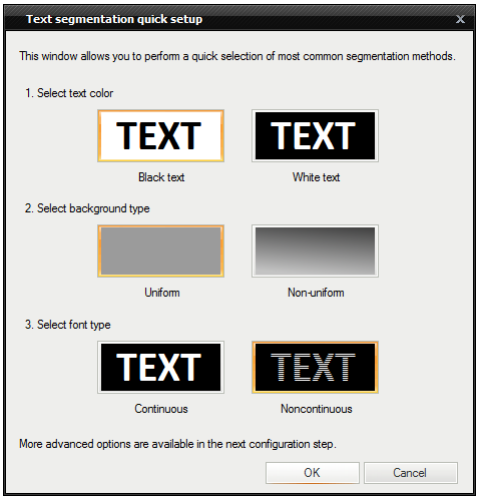
Details about using OCR filters can be found in Machine Vision Guide: Optical Character Recognition.

To configure text extraction please perform the following steps:

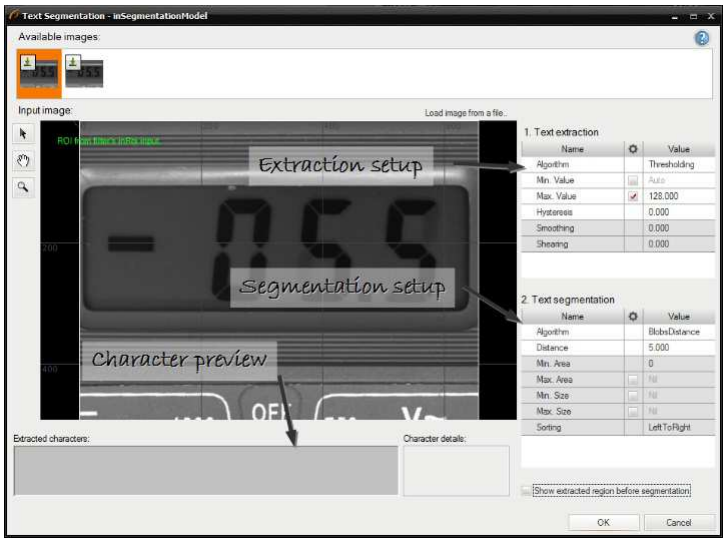
1. Add an ExtractText filter to the program.

2. Set the region of interest on the inRoi input. This step is necessary before performing next steps. The image below shows how the ROI was selected in an example application:

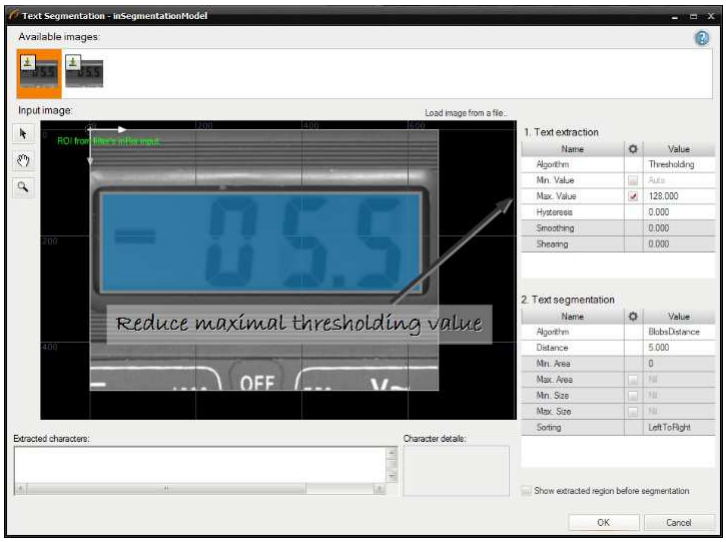
3. Click on the "..." button at the **inSegmentationModel** input to enter the graphical editor.
4. When entering first time, complete the quick setup by selecting most common settings. In this example a black non-continuous text should be extracted from a uniform background. Configuration was set to meet these requirements.



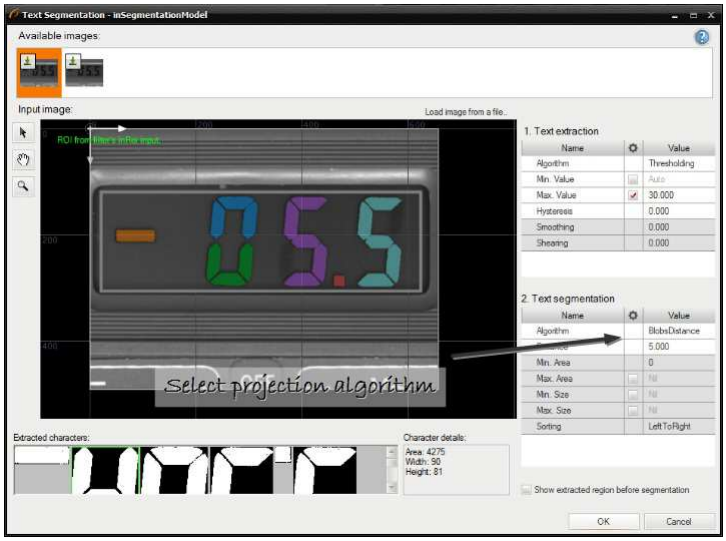
5. After the quick setup the graphical editor starts with some parameter set. Adjust the pre-configured parameters to get best results.



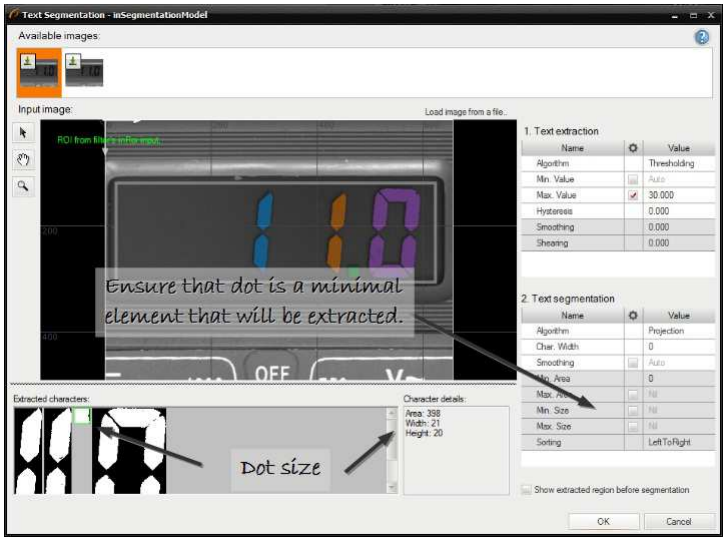
6. Configure a character extraction algorithm. In this case thresholding value is too high and must be reduced.



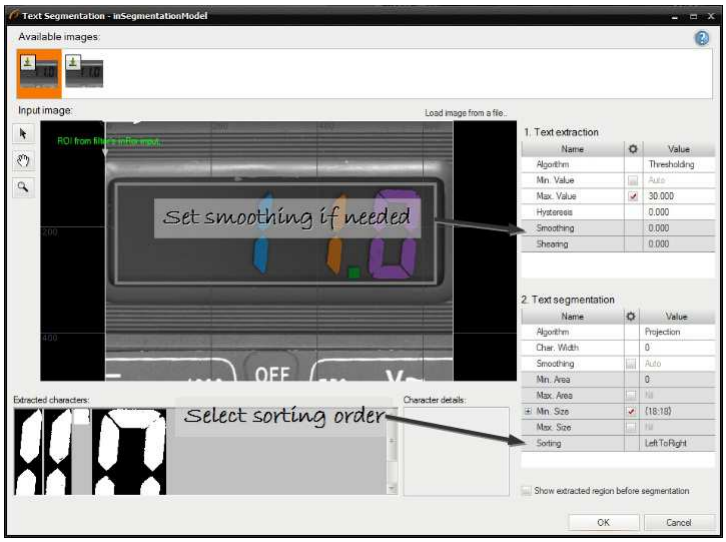
7. Select a character segmentation algorithm.



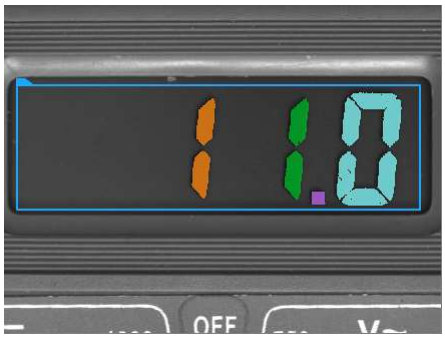
8. Set the minimal and the maximal size of a character. The editor shows the character dimensions when the character is selected in the list below.



9. Select a character sorting order, character deskewing (shearing) and image smoothing. Smoothing is important when images have low quality.

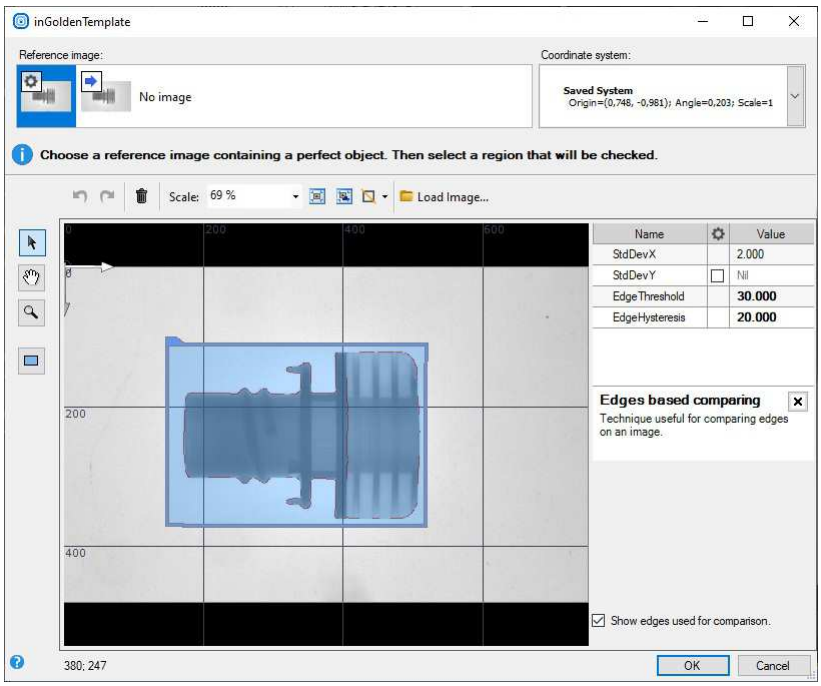


10. Check results using available images.



Creating Golden Template Models

Golden template technique is the most powerful method for finding objects' defects. Editor presented below is available in filters CompareGoldenTemplate_Intensity and in CompareGoldenTemplate_Edges.



To create golden template, select template region and configure its parameters.

Remarks:

- To reduce computation time try to select only necessary part of an object,
- For comparing both edges and surface use two CompareGoldenTemplate filters,
- To create mode programmatically use filter CreateGoldenTemplate.

Creating Models for Golden Template

Introduction

Golden Template is an image comparison technique. It is based on the pixel-to-pixel comparison but uses multiple images and advanced algorithms to create a multi-image model. It is useful for finding general defects of objects that have fixed shape. In order to simplify the process of model creation a "GUI for Golden Template 2" was created. It is possible to access it by clicking on the [...] button at the inModel input in the Properties window:

Opening GUI for Golden Template 2

Image Preparation

To create a golden template model you need at least three same-sized images representing the same object. The object should be placed in the same way in all the images. Otherwise, the final model may not be accurate enough.

The first step is to prepare the images. To be sure that the object is always precisely positioned and have the same size - in both the model and the program - you can use the sequence of filters presented below:

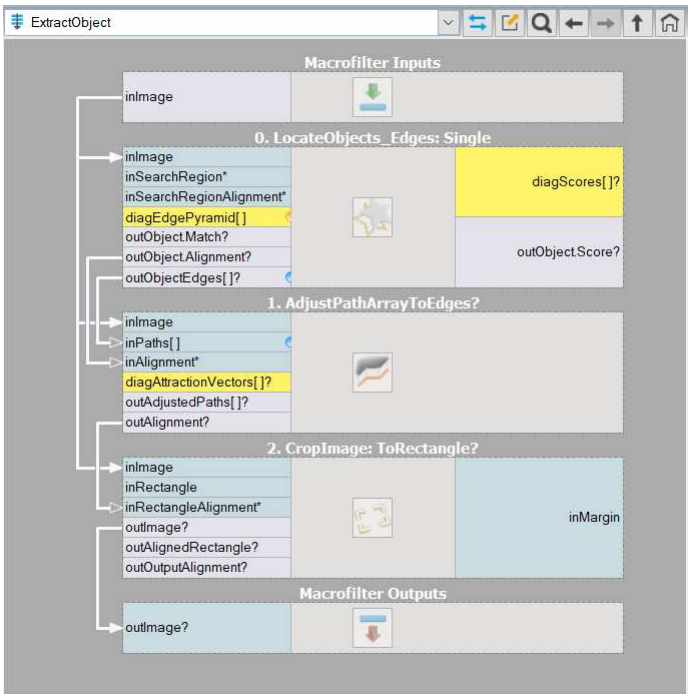


Image preparation process

Following steps were performed there:

- LocateSingleObject_Edges1 filter was used to create a robust model able to locate the logo and remember its alignment
- AdjustPathArrayToEdges allows you to improve the template matching results
- CropImageToRectangle gives evenly-cut images of the object. You should specify the **inRectangle** input manually.

If you have at least three images prepared, you can proceed to the next stage.

Model Creation

Add CreateGoldenTemplate2 filter to the program and click **inModel** in the properties window, as shown in the image at the beginning of the article. After that the following window will appear:

Golden Template Editor

Firstly, add images to the editor. It is possible to either use the drag and drop function or load images from the directory by clicking on the  icon.

The images will be used to create the Golden Template model, so they should present samples without defects.

After loading all the images, draw a mask that represents the object of interest on one image. It applies to all the other images that you loaded. That is why it was necessary to properly position the objects in the images. If no mask is detected the warning will appear. If the object covers most of the field of view, please feel free to mark the whole image.

Now you can check the preview with the "Show average of all images" and adjust the training parameters:

- **Downscale** - resizing the image dividing by the value. It greatly speeds up the computing in exchange for the ability to spot pixel-size defects
- **MaximalDisplacement** - possible error in object positioning, high values may impair detection of small defects, especially near edges
- **LargeDefectSize** - expected diameter of largest, extensive defects
- **NoiseAugmentation** - allows for additional noise presence in the images
- **BrightnessAugmentation** - allows for additional brightness deviation
- **SmoothingAugmentation** - allows for additional smoothing in the images, uses gaussian smoothing with specified standard deviation

Clicking *OK* button will start the training. The editor will close itself afterwards.

The model is created and loaded in the CreateGoldenTemplate2 filter. If you prepare the inspection images in the same way as during model creation (by positioning and cropping them) everything should work properly.

Creating Text Recognition Models

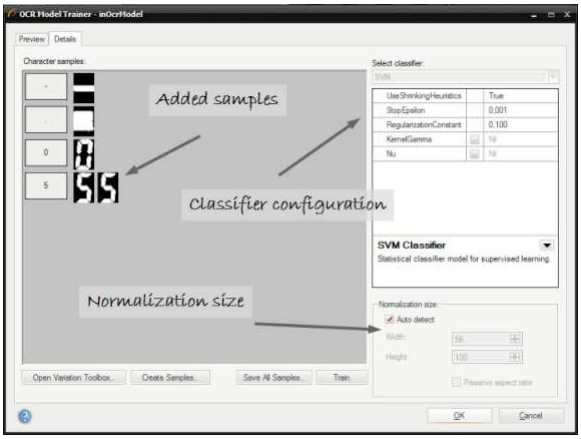
Text recognition editor creates an OCR model for getting text from regions. More details about the OCR technique can be found in Machine Vision Guide: Optical Character Recognition.

To create an OCR model a set of characters should be collected. If recognition score is low after training based on real samples then artificial character variations can created.

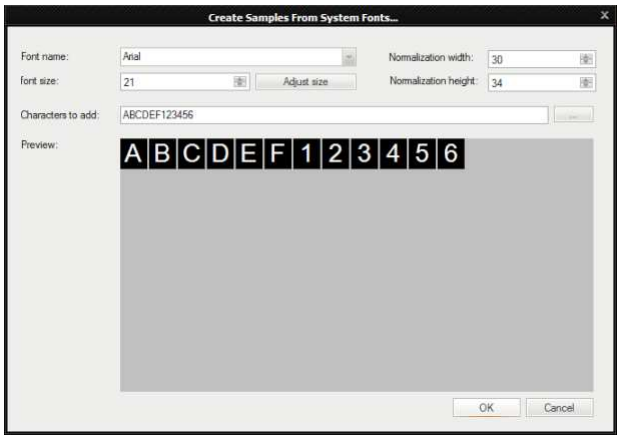
Creation of a model consists of following steps:

1. **Collecting real samples** - after opening the editor characters are visible and can be added to a training set.

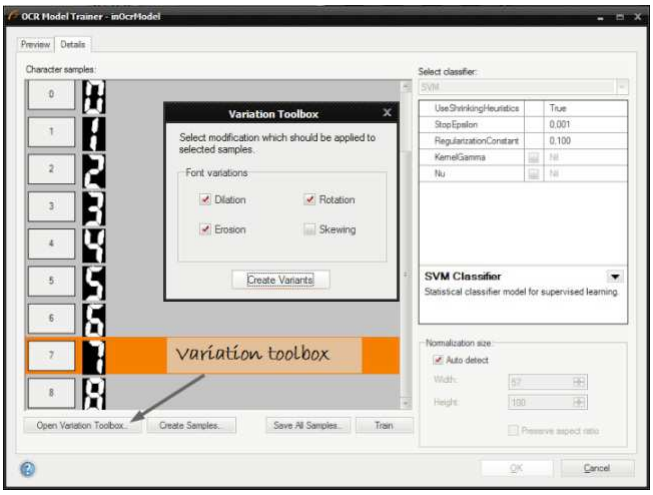
After the training character samples can be viewed in the details tab:



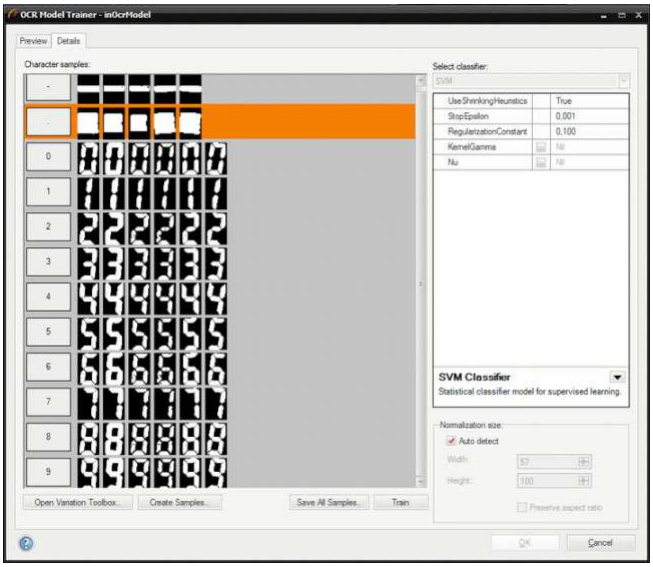
2. **Creating artificial samples** - when no samples are available user can create a training set using systems fonts.



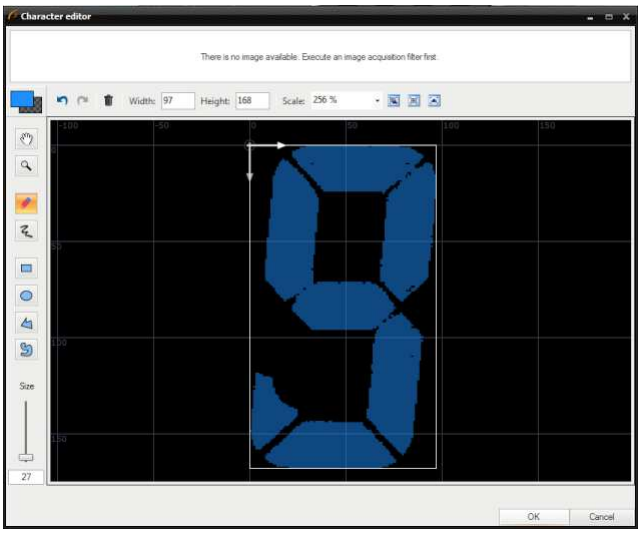
3. **Creating character variations** in case when no more samples are available and the training result is not fine the editor can modify existing samples to create a new set.



The training set after adding new samples variations:



4. **Editing samples** - in case when gathered samples contain noises, or its quality is low, user can edit them manually. The image below show how to edit a character '8' to get character '9'.

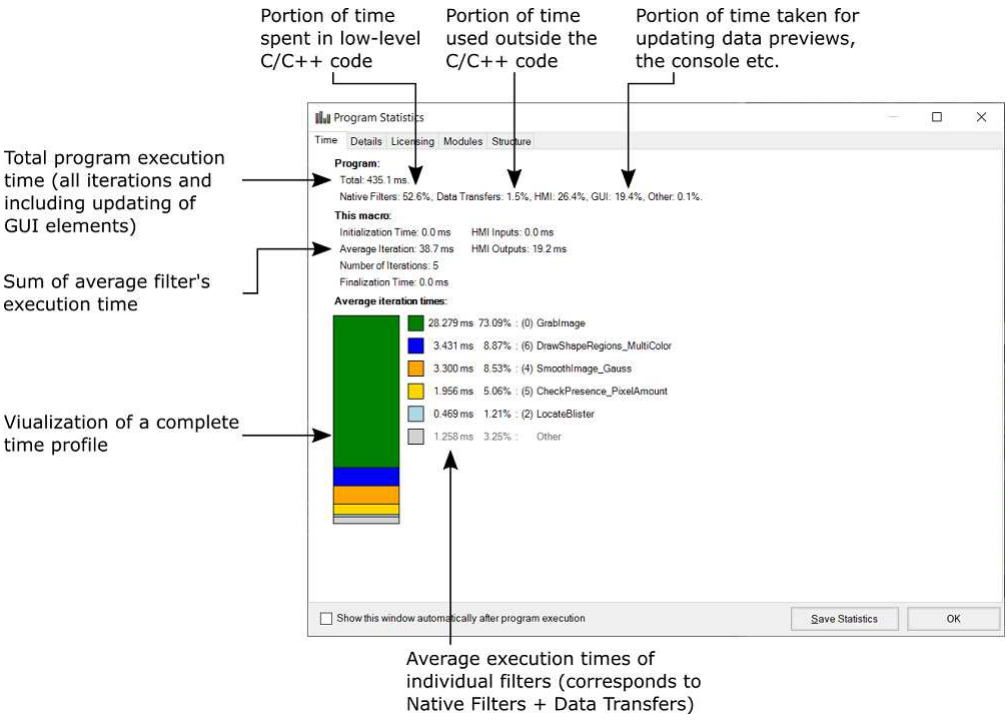


Note:

- Each training character should have this same number of samples.
- In cases when some characters are very similar number of samples can be increased to improve classification.
- Character samples can be stored in an external directory to perform experiments on them.

Analysing Filter Performance

The Program Statistics window contains information about the time profile of the selected macrofilter. This is very important as real vision algorithms need to be very fast. However, before starting program optimization we must know what needs to be optimized, so that later we optimize the right thing. Here is how the required information is presented:



As can be seen on the above illustration, program execution time is affected by several different factors. The most important is the "Native Filters" time, which corresponds to the core data processing tools. Additional time is consumed by "Data Transfers", which is related to everything that happens on connections between filters – this encompasses automatic conversions as well as packing and unpacking arrays on array and singleton connections. Another statistic called "Other" is related to the virtual machine that executes the program. If it value is significant, then C++ code generation might be worth considering. The last element, "GUI", corresponds to visualization of data and program execution progress. You can expect that this part is related only to the development environment and can possibly be reduced down to zero in the runtime environment.

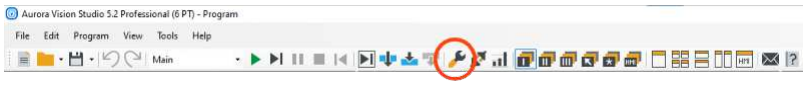
Remarks:

- In practice, performance statistics may vary significantly in consecutive program executions. It is advisable to run the program several times and check if the statistics are coherent. It might also be useful to add the EnumerateIntegers filter to your program to force a loop and collect performance statistics not from one, but from many program iterations.
- Turn off the diagnostic mode when testing performance.
- Data Preview Panels, animations in the Program Editor and even the Console Window can affect performance in Aurora Vision Studio. Choose *Program » Previews Update Mode » Disable Visualization* to test performance with minimal influence of the graphical environment. (Do not be surprised however that nothing is visible then).
- Even with all windows closed there are some background threads that affect performance. Performance may still be higher when you run the program with the Executor (runtime) application.
- Please note that the first program iteration might be slower. This is due to the fact that in the first iteration memory buffers are allocated, filters are initialized, communication with external devices is established etc.

See also: Optimizing Image Analysis for Speed.

Seeing More in the Diagnostic Mode

Programs can be executed in two different modes: *Diagnostic* and *Non-Diagnostic*. The difference between them is in the computation of values on the diagnostic outputs. Values of this kind of outputs are computed only in the *Diagnostic* mode. They can be helpful in debugging programs but are not necessary in its final version. In the *Non-Diagnostic* mode, execution is faster because no diagnostic values are computed.



The Diagnostic Mode switch.

The execution mode can be easily changed in Aurora Vision Studio using a button on the Application Toolbar. Outside of Aurora Vision Studio, programs are always executed in the *Non-Diagnostic* mode to provide the highest performance.

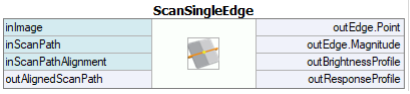
Diagnostic Filter Instances

Filters that have inputs connected to diagnostic outputs of some filters above them, are said to be diagnostic too. They are executed only when the program runs in the *Diagnostic* mode.

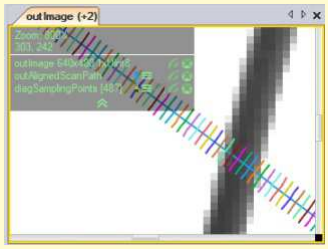
Example

The ScanSingleEdge filter has three diagnostic outputs:

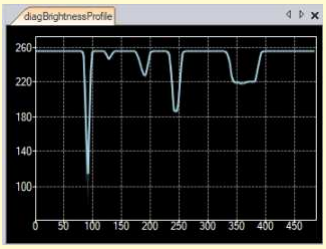
- **diagBrightnessProfile** is the profile of image brightness sampled along the scan path.
- **diagResponseProfile** is the profile derivative after preprocessing.
- **diagSamplingPoints** visualizes the points on the input image from where the brightness samples were taken.



The ScanSingleEdge filter.



Input image with the scan path and the diagnostic sampling points.



The brightness profile.

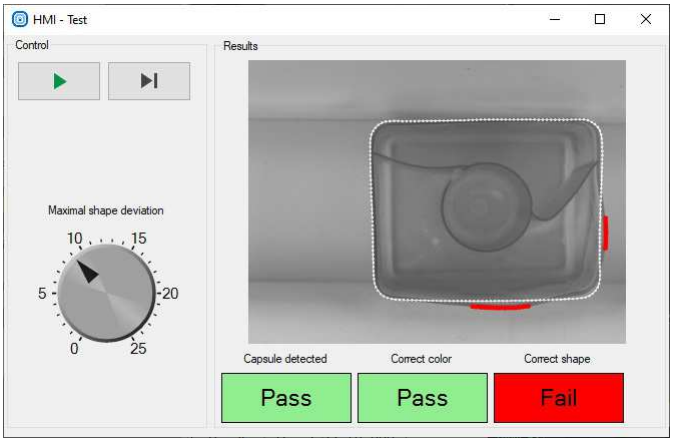
The response profile.

Deploying Programs with the Runtime Application

Introduction

Aurora Vision Executor is a lightweight application that can run programs created with Aurora Vision Studio. The GUI controls that appear in this application are the ones that have been created with the HMI Designer. The end user can manipulate the controls to adjust parameters and can see the results, but he is not able change the project.

Aurora Vision Executor application is installed with the Aurora Vision Studio Runtime package. It can be used on computers without the full development license. Only a runtime license is required. What is more, programs executed in Aurora Vision Executor usually run significantly faster, because there is no overhead of the advanced program control and visualization features of the graphical environment of Aurora Vision Studio.



The screen of Aurora Vision Executor.

Usage

Open a project from a file and use standard buttons to control the program execution. A file can also be started using the Windows Explorer context menu command *Run*, which is default for computers with Aurora Vision Studio Runtime and no Aurora Vision Studio installed.

Please note, that Aurora Vision Executor can only run projects created in exactly the same version of Aurora Vision Studio. This limitation is introduced on purpose – little changes between versions of Studio may affect program compatibility. After any upgrade your application should first be loaded and re-saved with Aurora Vision Studio as it then runs some backward compatibility checks and adjustments that are not available in Executor.

Console mode

It is possible to run Aurora Vision Executor in the console mode. To do so, the `--console` argument is needed to be passed. Note, that this mode makes the `--program` argument required so the application will know which program to run at startup.

Aurora Vision Executor is able to open a named pipe where it's log will be write into. This is possible with `--log-pipe` argument which accepts a pipe name to be opened. One may then connect to the pipe and process Aurora Vision Executor log live. This can be easily done e.g. in C#:

```
var logPipe = new NamedPipeClientStream(".", "myProjectPipe", PipeDirection.In);
logPipe.Connect();

byte[] buffer = new byte[1024];
int count = 0;
while (logPipe.IsConnected && (count = logPipe.Read(buffer, 0, 1024)) > 0)
{
 Console.WriteLine(Encoding.UTF8.GetString(buffer, 0, count));
}
```

Available Aurora Vision Executor arguments are as follows:

--program
: Path to the program to be loaded

--log-level
: Sets the logged information level

--console
: Runs the application in the console mode

--auto-close
: Automatically closes the application when program is finished. Meaningful only in console mode.

--language
: Specifies the language code to use as the user interface language.

--attach
: Attaches application process to the calling process console.

--log-pipe
: Creates a named pipe which will be populated with log entries during application lifetime. Meaningful only in console mode.

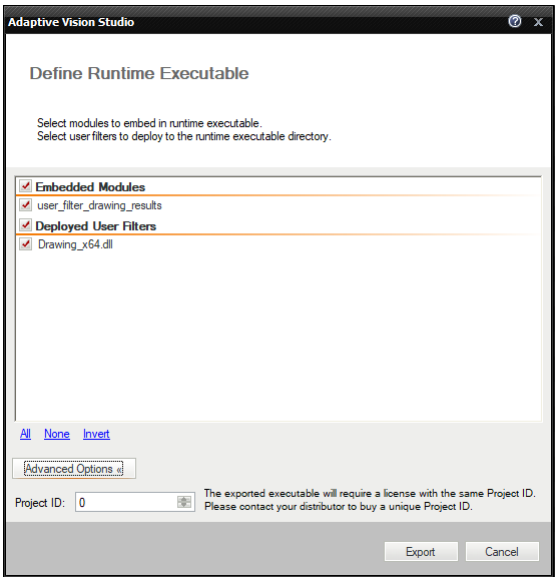
--help
: Displays help

Runtime Executables

Aurora Vision Executor can open .avproj files, the same as Aurora Vision Studio, however it is better to use .avexe files here. Firstly one can have a single binary executable file for the runtime environment. Secondly this file is encrypted so that nobody is able to look at project details. To create one, open a project in Aurora Vision Studio and use *File » Export to Runtime Executable...*. This will produce an .avexe file that can be executed directly from the Windows Explorer.

If Aurora Vision project contains any User Filter libraries, it is crucial to put their *.dll files into the appropriate directory when running in Aurora Vision Executor. This is when exporting to .avexe file might also be a handy option. While defining the .avexe contents, it is possible to select all the User Filters libraries that the exported project depends on. Selected libraries are deployed then to the same directory as generated .avexe file and the

.avexe itself is set to use all User Filter libraries from its directory.



Defining the Runtime Executable.

In case there are any other dependencies, e.g. exposed by used User Filter libraries, one can add them into the Aurora Vision project as an attachment in Project Explorer and also deploy with .avexe file during export.

Project ID available in *Advanced Options* is an additional parameter that makes only a specific license able to run the application. At the customer request, Aurora Vision Team can generate the key (Project ID) and add it to the Runtime license. The same Project ID must be set for Runtime Executable export to connect the application with the specific license. If you leave this field default, any license will be able to run the .avexe file.

Trick: Configuration File as a Module Not Exported to AVEXE

It is often convenient to have an configuration file separated from the executable so that various parameters can be adjusted for a particular installation (but not made available to the end user). This can be easily implemented with a simple programming idiom:

1. Use global parameters in your project for values that might require adjusting.
2. Place the global parameters in a separate module (through the Project Explorer window).
3. Exclude the module when exporting the .avexe file.
4. In the runtime environment copy the .avexe file and the module (as a separate file) with global parameters.
5. Open the config module in the Notepad to edit it when needed.

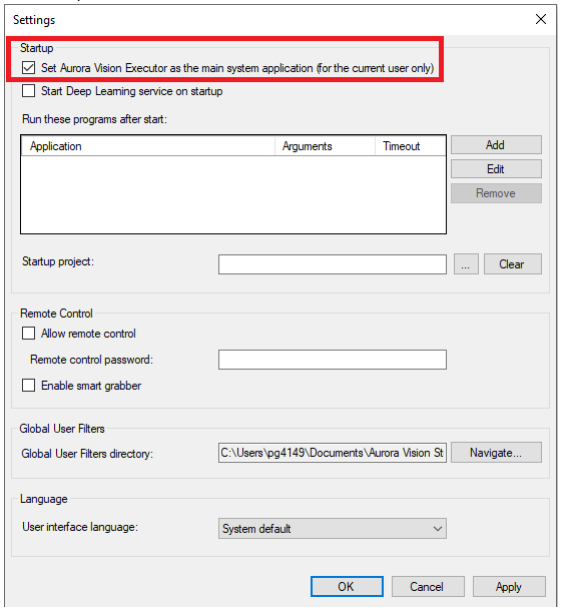
Other Runtime Options

Please note that Aurora Vision Executor is only one of several options for creating end-user's applications. Other available options are:

- .NET Macrofilter Interface Generator – generates a native .NET assembly (a DLL file) and makes it possible to invoke macrofilters created in Aurora Vision Studio as simple class methods from a .NET project. Internally the execution engine of Aurora Vision Studio is used, so modifying the related macrofilters does not require to re-compile the .NET solution. The HMI can be implemented with WinForms, WPF or similar technologies.
- C++ Code Generator – generates native C++ code (a CPP file) that is based on Aurora Vision Library C++. This code can be integrated with bigger C++ projects and the HMI can be implemented with Qt, MFC or similar libraries. Each time you modify the program in Studio, the C++ code has to be re-generated and re-compiled.

Set Aurora Vision Executor as the "system shell"

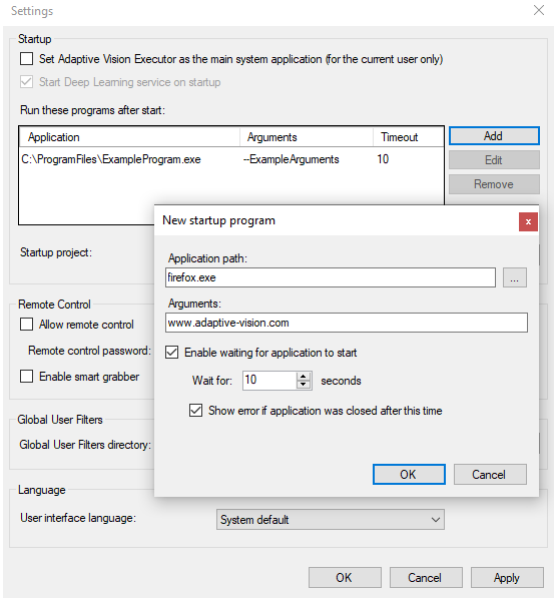
On Windows systems it is possible to set Aurora Vision Executor as the "system shell", thus removing Desktop, Menu Start etc. completely. Go to Settings in Aurora Vision Executor and the Startup section. Mark the *Set Aurora Vision Executor as the main system application (for the current user only)*. Please be informed that this option requires administrator privileges.



Set Aurora Vision Executor as the "system shell"

Startup Applications

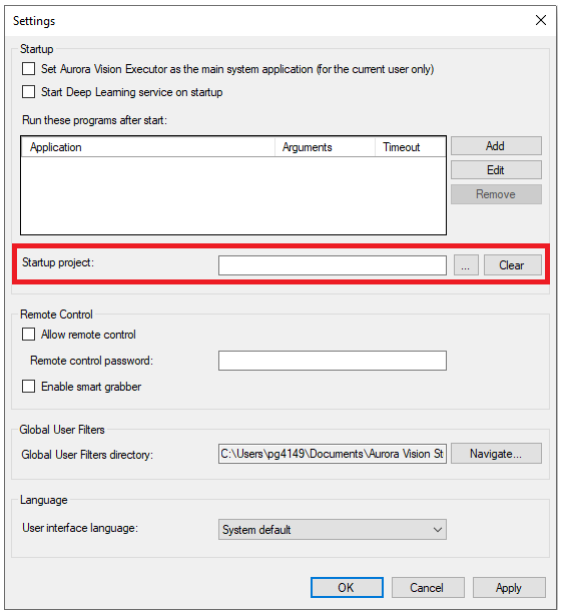
It is possible to run any process before starting a program in Aurora Vision Executor. Go to Settings in Aurora Vision Executor and the Startup section. To define a new startup program select the *Add* button on the right. In a *New startup program* dialog box you need to specify the application path (obligatory) and arguments (optional). It is similar to typing the application name and command-line arguments in the *Run* dialog box of the Windows Start menu. The added program will appear in the list. All added programs will start each time you run Aurora Vision Executor.



Defining Startup Applications

Startup Project

It is possible to choose the project the Aurora Vision Executor should run after the startup . Go to Settings in Aurora Vision Executor and the Startup section. To define the startup project select the ... button on the right. In an *Open* dialog box you need to specify the startup project path (obligatory). The added project's path will appear in the box. It will start each time you run Aurora Vision Executor.



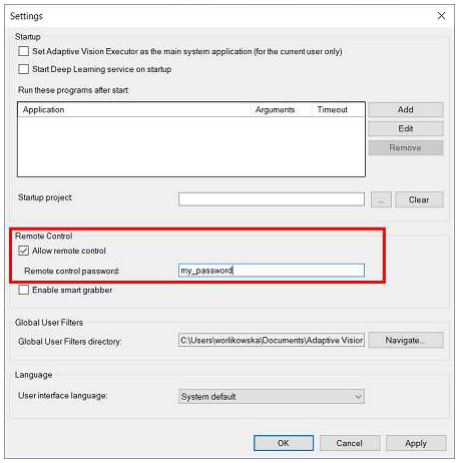
Defining Startup Project

If Deep Learning Service is installed you can choose to run it on start by selecting 'Start Deep Learning service on startup'.

Remote Access to the Runtime Application

Introduction

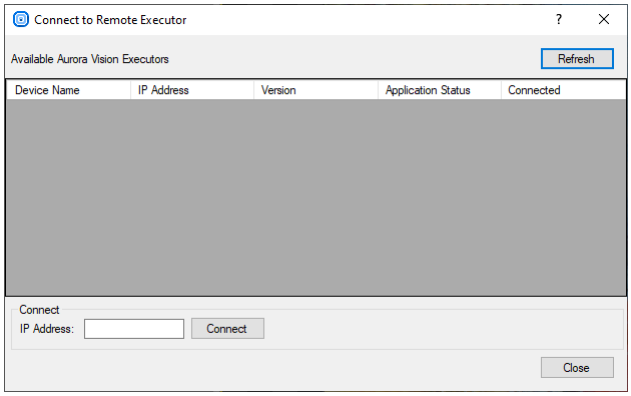
It is possible to use Aurora Vision Studio to control an Aurora Vision Executor running on a remote computer, if only the two computers are in the same Ethernet subnet. To enable remote access in Aurora Vision Executor, the option "Allow remote control" should be enabled. For security reasons, it is also possible to protect connection with a password:



Enable remote executor.

Usage

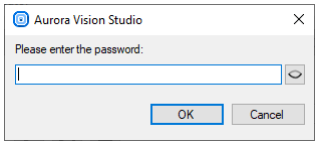
In Aurora Vision Studio the list of available remote systems is accessible in the Connect to Remote Executor window which opens up after choosing the File › Connect to Remote Executor menu command:



Browse remote systems.

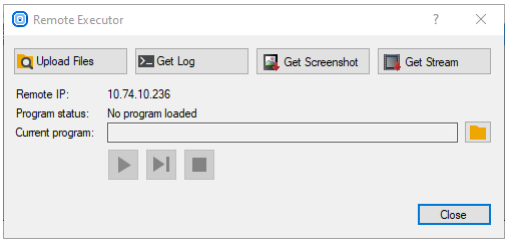
If an expected remote system is not visible on the list, please verify that: (1) it is configured for the same local area network, and (2) it can access Ethernet through its firewall (in the first place verify that in the Windows' Control Panel, Windows Firewall section, the option "Notify me when Firewall blocks a new program" is enabled). In some local networks information about communication type and ports may be useful to assign application access through firewall. In this case please use UDP ports 6342 and 6343 or TCP port 6342.

When connecting to a selected device, the program will ask for the password (if it has been set):



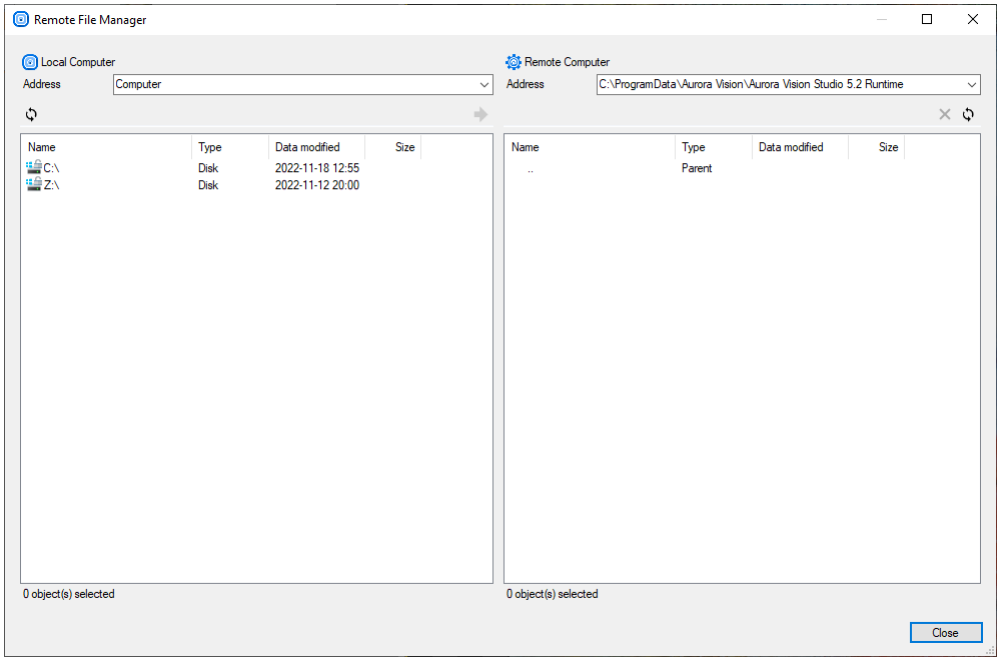
Password protection window.

After successful connection, management of the selected executor becomes possible. Now you are able to do several actions: Upload Files, Control Program and Get Diagnostics.



Remote executor window.

Upload Files:

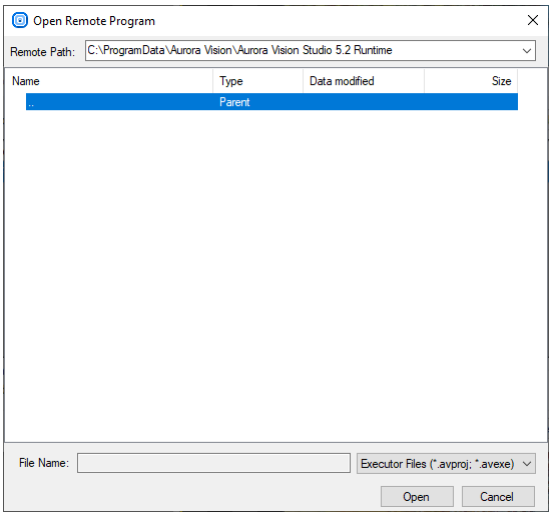


Upload files window.

To Manage Files on remote executor, click Upload Files button. In this window it is possible to send program files to the remote executor. By default all files are stored in the directory <CommonApplicationData>\Aurora Vision\Aurora Vision Executor.

Program Control:

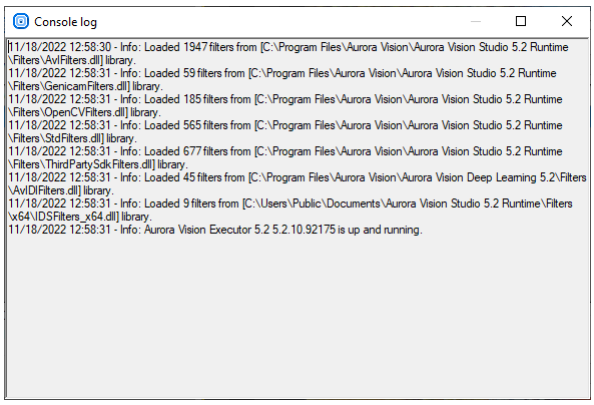
In the Remote executor window you are able to control currently executed program. In the middle of the window we can see the current program status and the path to the currently loaded program. Next there are four buttons controlling program execution. Finally there is an option to open another program.



Remote file dialog window.

Diagnostics:

In the Remote Executor window there is available a few diagnostic tools. Here, it is possible to preview the execution log and a screen preview from the currently connected executor. All diagnostic options are located at the top of the Remote executor window.



Log preview window.

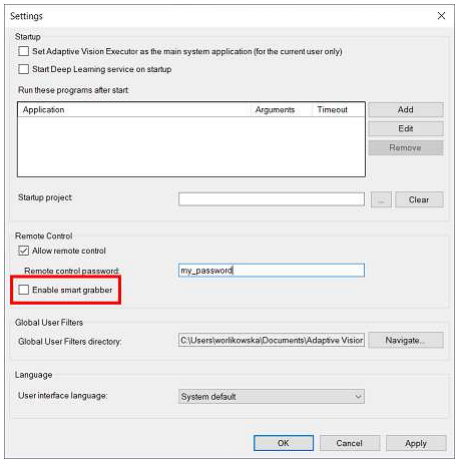
Remote Image Acquisition

Introduction

It is also possible to get images from a connected device using the remote executor. This method offers an easy way to acquire images using a different image protocols. Remote image acquisition filters enable to run this same code on the smart cameras (client side) as good as on developer computers without any additional modifications.

Usage

To enable remote image acquisition it is necessary to stop the executor program and enable the "Enable smart grabber" option:



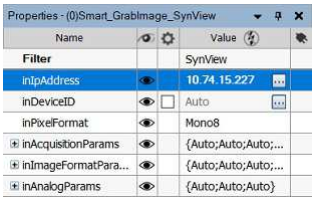
Enable Remote Image Acquisition.

In Aurora Vision Studio it is possible to use the filters from the category Camera Support\Smart to get images from a device connected to the remote Executor system.

Currently the available image acquisition protocols are:

- **AvSMART** – access to AvSMART,
- **GenICam** – access to GenICam complaint devices,
- **Roseek** – access to Roseek cameras,
- **SynView** – access to NET GmbH cameras,
- **WebCamera** – access to DirectX complaint image sources like web cameras or frame grabbers.

To grab images using one of these protocols it is necessary to set **inIpAddress** in the filter input. The IP Address can be checked in the Connect to Remote Executor window when "Allow remote control" option is enabled in the Executor.

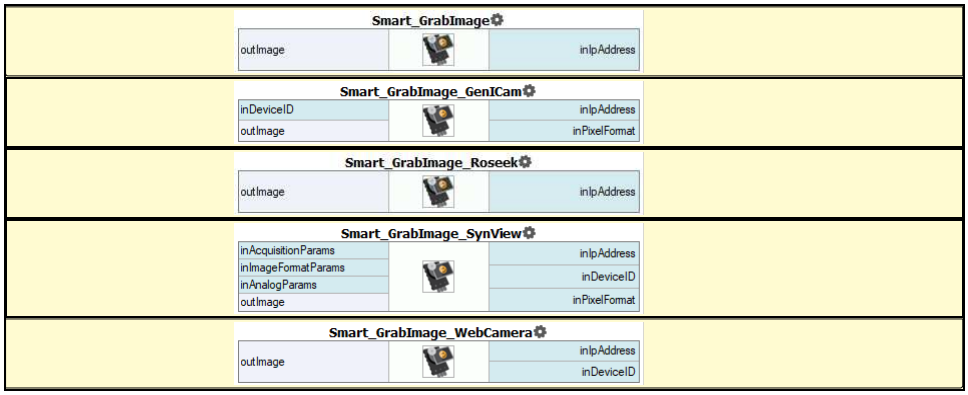


Smart_GrabImage_SynView filter properties.

If system detect that the provided IP address describes local machine the Smart_GrabImage filters will perform all operation on local computer. So it is no need to perform any changes during transferring project from developers machine to the client side device.

If the program is deployed to a device with another IP address then the input **inIpAddress** should be changed to the IP of the Executor or left empty. If the input **inIpAddress** is empty then the image is always grabbed using an appropriate local image acquisition protocol.

Available filters for the remote image acquisition:



If **inDeviceID** is set to*NIL*the first found device will be used.

Performing General Calculations

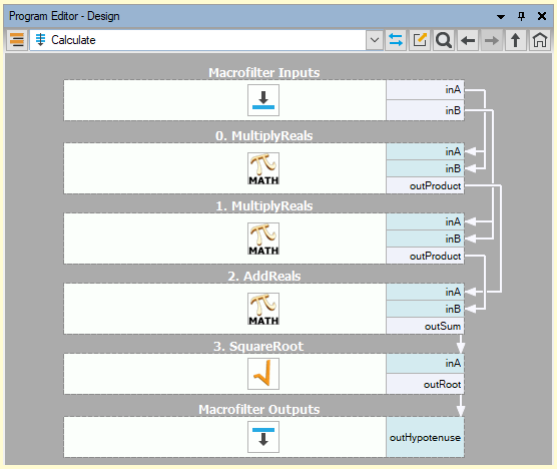
Introduction

Apart from using image processing or computer vision tools, most often it is also necessary to perform some general calculations on numeric values or geometrical coordinates. There are two ways to do this in Aurora Vision Studio:

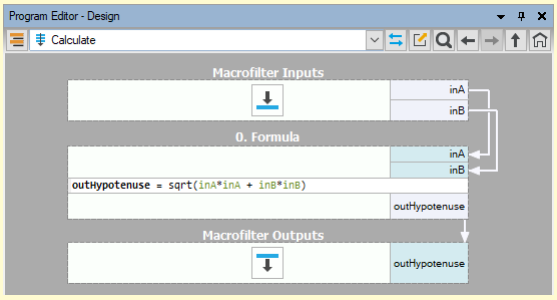
1. Using filters from the Standard Library.
2. Using special formula blocks.

Example

Let us assume that we need to compute the hypotenuse $\sqrt{a^2+b^2}$. Here are the two possible solutions:



Calculations using standard filters.



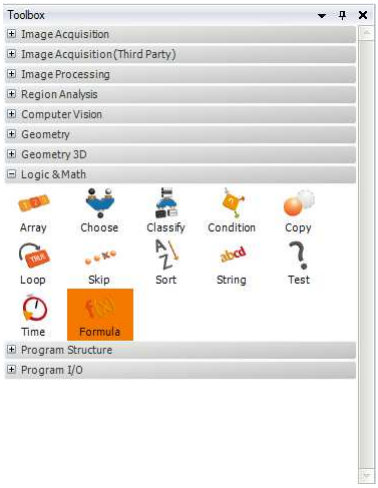
Calculations using formula blocks.

The second approach, using formula blocks, is the most recommended. Data flow programming is just not well suited for numerical calculations and standard formulas, which can be defined directly in formula blocks, are much easier to read and understand. You can also think of this feature as an ability to embed little spreadsheets into your machine vision algorithms.

Creating Formula Blocks

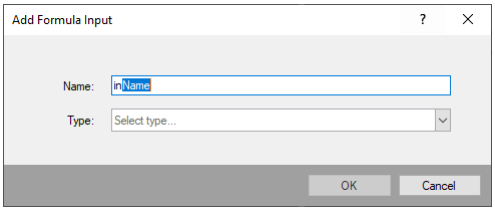
To create a formula block:

1. Drag and drop the Formula tool from the Toolbox (*Logic & Math* section) to the Program Editor:



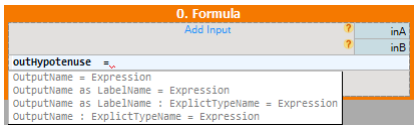
2. Add inputs and outputs using the context menu or by clicking *Add Input* and *Add Output* links:

3. For each input define the name and the type:



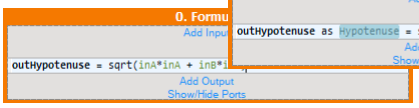
4. In order to create an output, first press *Add Output* button. It will create a new output with template name: *outName*:

Replace the highlighted *Name* with meaningful name. While typing a Tooltip with available options should appear:



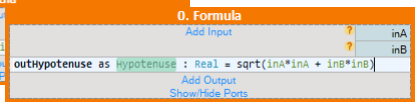
Example usage of each option:

- After the name type equal sign =o and the expression. There is no need to define a type of output. The type will be assigned automatically.



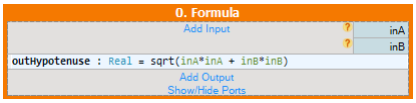
- After the name add *as* followed by *LabelName*. This label will later appear as a formula output.

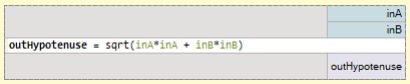
- After the name add *label* as described in the previous point. Then add colon *:* and directly specify the output type:



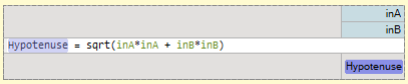
Final formula should look like this:

- In the last option output type is defined directly. After the name add colon *:* and directly specify the output type:



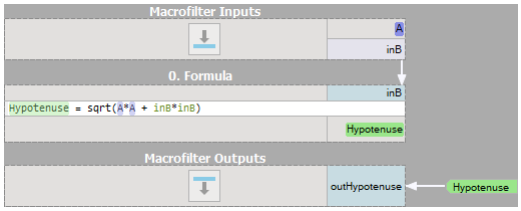


Without output label.



With output label.

Not only the output data of the formulas can be labeled but also inputs and outputs of other filters outside of the formula. This way you can perform a direct call to another labeled data inside a formula without explicitly defining new formula inputs. In the image below the input *A* of the macrofilter is labeled (highlighted in violet color) and a direct call of this label (A) is performed inside the formula block. Later the output *Hypotenuse* is connected via a label with the macrofilter output, instead of linking it directly using an outgoing arrow connection.



Remarks

- Existing formulas can also be edited directly in a formula block in the Program Editor.
- It is also possible to create new inputs and outputs by dragging and dropping connections onto a formula block.
- Formula blocks containing incorrect formulas are marked with red background. Programs containing such filters cannot be run.
- When defining a formula for an output it is possible to use other outputs, provided that they are defined earlier. The order can be changed through the outputs context menu.
- For efficiency reasons it is advisable not to use "heavy" objects in formulas, such as images or regions.

Syntax and Semantics

For complete information about the syntax and semantics please refer to the Formulas article in the Programming Reference.

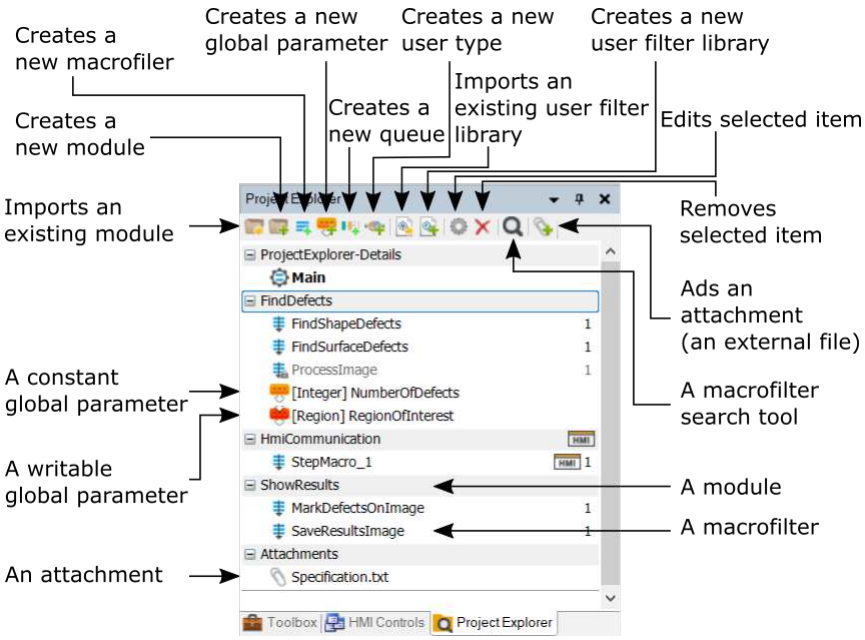
Managing Projects with Project Explorer

Introduction

Project Explorer is a window displaying elements which are contained in the currently opened project:

- Modules
  - Macrofilters (definitions)
  - Global Parameters
  - User Types
- User Filter libraries
- Attachments

Its main goal is to provide a single place to browse, add, remove, rename or open the items, which are grouped into categories in the same way as the filters in the Filter Catalog. A category may correspond to a standard module or to a module of a User Filter library. There is also one special category, Attachments, which appears when the user adds an external file to the project (it might be for example a text document with a piece of documentation).



Modules in the Project Explorer.

Opening Macrofilters

As described in Running and Analysing Programs, there are two ways of navigating through the existing macrofilters. One of them is with the Project Explorer window, which displays *definitions* of macrofilters, not the instances. After double-clicking on a macrofilter in the Project Explorer, however, a macrofilter instance is opened in the Program Editor. As one macrofilter definition can have zero, one or many instances. Some special rules apply to which of the instances it is:

- If possible, the most recently executed instance is opened.
- If no instance has been executed yet, the most recently created one is opened.
- If there are no instances at all the "ghost instance" is presented, which allows editing the macrofilter, but will never have any data on the output ports.

Macrofilter Counter

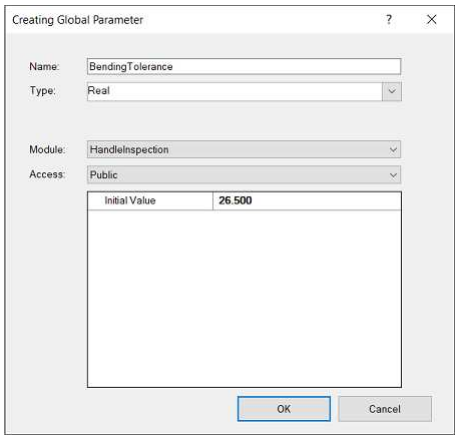
Macrofilter counter shows how many times a given macrofilter is used in the program.

ADVANCED NOTE: If a macrofilter X is used in a macrofilter Y and there are multiple instances of the macrofilter Y, we still consider macrofilter X being used once. Number of uses is not the same as number of instances.

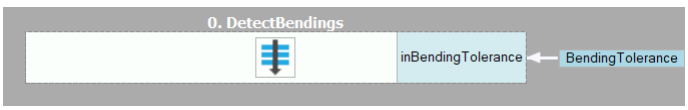
Global Parameters

If some value is used many times in several different places of a program then it should be turned into a global parameter. Otherwise, consecutive changes to the value will require the error-prone manual process of finding and changing all the occurrences. It is also advisable to use global parameters to clearly distinguish the most important values from the project specification – for example the expected dimensions and tolerances. This will make a program much easier to maintain in future.

In Aurora Vision Studio global parameters belong to specific modules and are managed in the Project Explorer. To create one, click the *Create New Global Parameter...* button and then a dialog box will appear where you will provide the name, the type and the value of the new item. After a global parameter is created it can be dragged-and-dropped on filter inputs and appropriate connections will be created with a visual label displaying the name of the parameter.



Creating a global parameter.

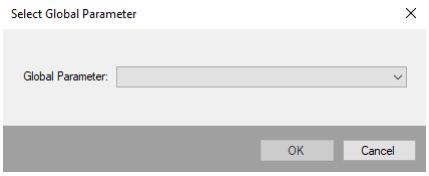


Global parameter used in a program.

Global parameters contained in a project can also be edited in the Properties window after being selected in the Project Explorer or in the Program Editor.

Since version 4.11 it is possible to create and manage global parameters with dedicated filters: WriteParameter and ReadParameter. They are available in the Program Structure category of the Toolbox.

While adding a new input, "Select Global Parameter" window is displayed:



You can select an already created global parameter or create a new one.

Global parameter "inAddress" created within the filter WriteParameter can be read with the ReadParameter filter.

Thanks to these filters you can easily read or write the values of global parameters anywhere in your algorithm. In order to facilitate development the icon of the global parameter has different appearance depending on whether it is overwritten somewhere in the program. The color of the icon will be red then so that you will know that this value may change during the execution of your application.

To see how Global Parameters work in practice, check out our official example: HMI Handling Events.

Remarks:

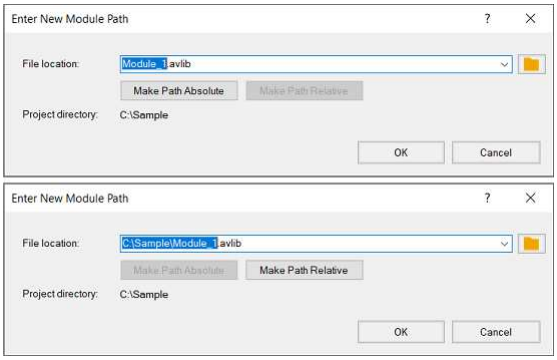
- Connected filters are not re-executed after the global parameter is changed. This is due to the fact, that many filters in different parts of the program can be connected to one global parameters. Re-executing all of them could cause unexpected non-local program state changes and thus is forbidden.
- Do NOT use writable global parameters unless you really must. In most cases data should be passed between filters with explicit connections, even if there are a lot of them. Writable global parameters should be used only for some very specific tasks, most notably for configuration parameters that may be dynamically loaded during program execution and for high level program statistics that may be manipulated through the HMI (like the last defect time).

Modules

When a project grows above 10-20 macrofilters it might be appropriate to divide it into several separate modules, each of which would correspond to some logical part. It is advisable to create separate modules for things like i/o communication, configuration management or for automated unit testing. Macrofilters and global variables will be then grouped in a logical way and it will be easier to browse them.

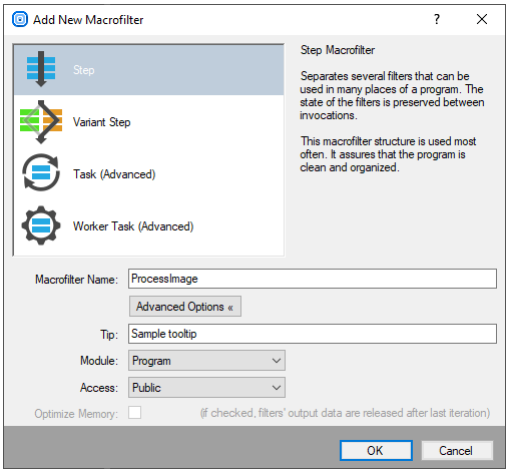
Modules are also sometimes called "libraries of macrofilters". This is because they provide a means to develop sets of common user's tools that can be used in many different projects. This might be very handy for users who specialize in specific market areas and who find some standard tasks appearing again and again.

To create a module, click the *Create New module...* button and then a dialog box will appear. In there you specify the location and name of the module. The path may be absolute or relative. Modules are saved with extension **.avlib**. Saving and updating the module files happens when the containing program is saved.

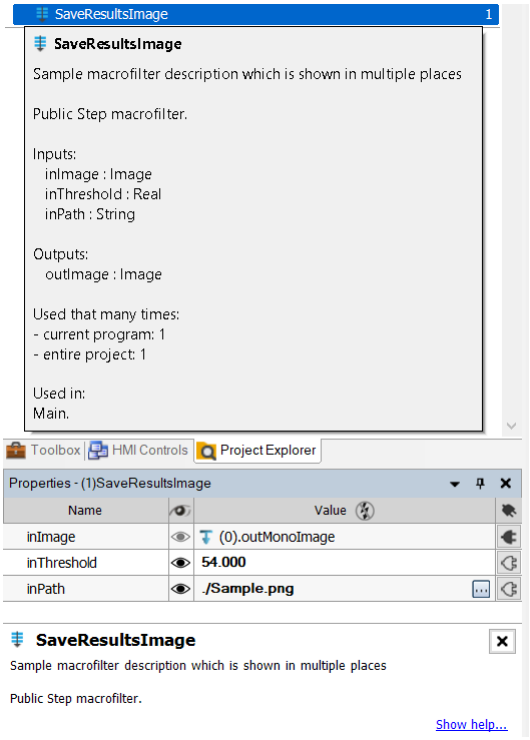


Module creation windows for both absolute and relative paths.

After creating a module you may move already existing macrofilters into the module by dragging them onto the module in Project Explorer. Clicking the *Create New Macrofilter...* button allows you to create a new macrofilter with the given name. During that you can access *Advanced options* by clicking the appropriate button. There you can specify the parent module and access to the macrofilter. Access can be either *private* or *public*. Private macrofilters cannot be directly used outside its module. You can also provide a tooltip (description) for the macrofilter. This will show up in the properties section of an instance and when hovered over in the Project Explorer. Tooltip should describe the macrofilter's purpose and be concise. They are especially important when creating an

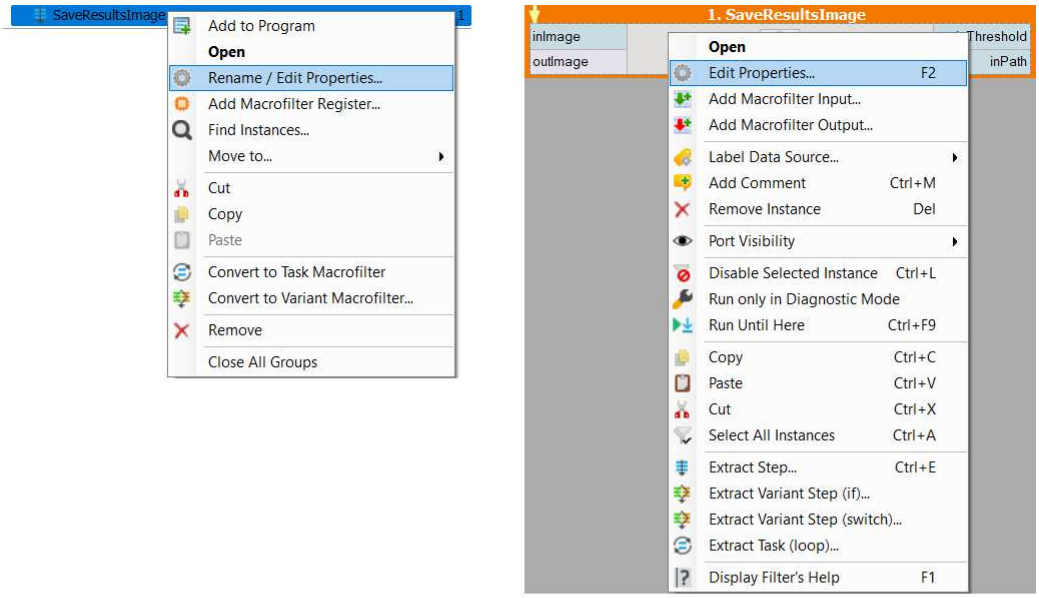


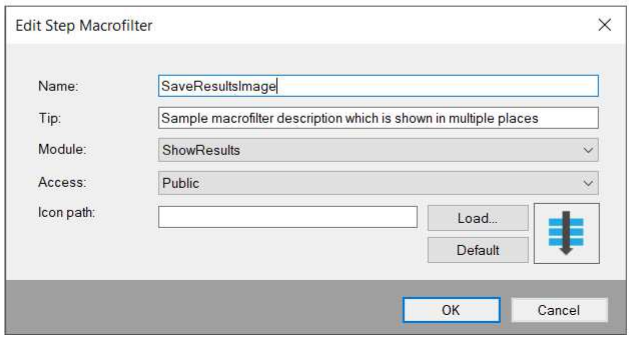
Macrofilter creation with advanced options.



View showing a sample tooltip.

Right clicking a macrofilter and selecting *Edit properties...* allows to modify their properties, including access, after their creation.

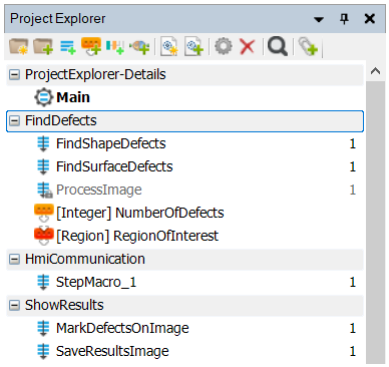




Ways to access the editing window. The option to change the icon is highlighted

The image below shows a structure of an example program. The macrofilters have been grouped into two Modules. Module *FindDefects* has macrofilters related to defect detection and a global parameter used by the macrofilters. Notice how the *ProcessImage* macrofilter is grayed out. It indicates it is a *private* macrofilter (here it is used by *FindShapeDefects*). *ProcessImage* cannot be used outside its module.

The other module has macrofilters related to showing and storing the results of the inspection.



Example program structure with macrofilters grouped into modules.

Here are some guidelines on how to use modules in such situations:

- Create a separate module for each set of related, standard macrofilters.
- Give each module a unique and clear name.
- Use the English language and follow the same naming conventions as in the native filters.
- Create the common macrofilters in such a way, that they do not have to be modified between different projects and only the values of their parameters have to be adapted.
- If some of the macrofilters are intended as implementation only (not to be used from other modules), mark them as *private*.

It is important to note that modules containing filters interfacing with the HMI should not be shared between programs. Every filter port connected to the HMI has a unique identifier. Those identifiers vary between programs - ports of the same filter in different programs will have different identifiers.

Generally it is a good practice to create a separate module for all things related to the HMI. That way every other module can be shared between programs without any problems.

Importing Modules

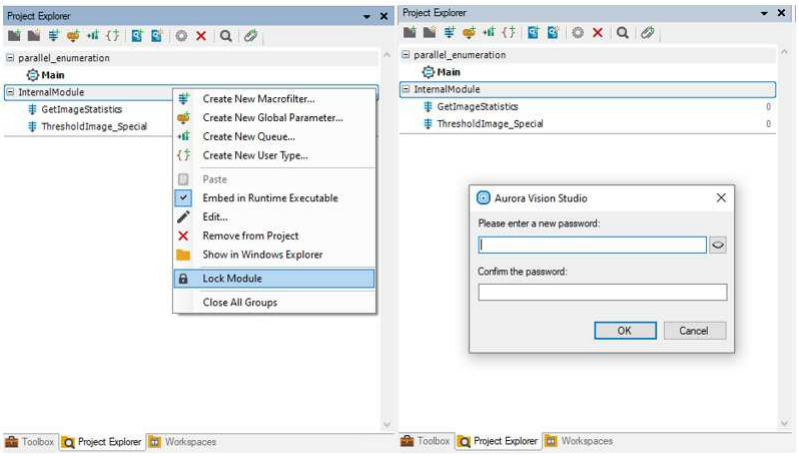
If you want to use a module which had been created before click the *Import Existing Module...* . This will open a window in which you can select modules to add. Now the path to the module will be linked to the project. Similarly to creating a new module you can choose whether the path to it will be relative or absolute.

Remember that modules are separate files and as such can be modified externally. This is especially important with modules which are shared between multiple projects at the same time.

See also: Trick: INI File as a Module Not Exported to AVEXE.

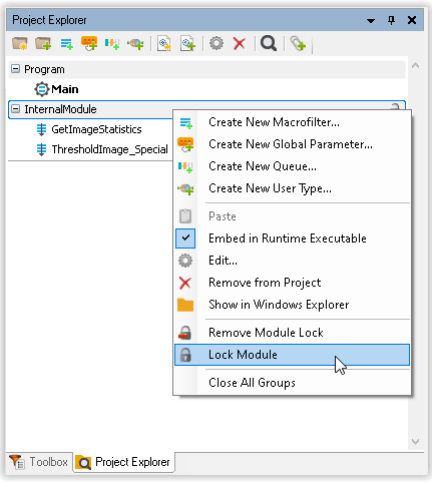
Locking Modules

Sometimes users would like to hide the contents of some of their macrofilter to protect them against unauthorized access. They can do this by placing them inside a module and locking it. To do this it is necessary to right click on the module in the Project Explorer and select Lock Module option. User will be then prompted to provide a password for this specific module.

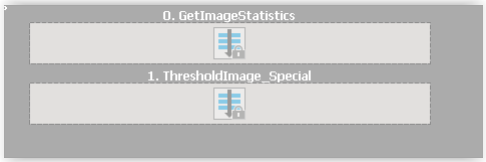


Adding lock to the module.

After creating password user should see an open lock icon next to the module name in the Project Explorer. You can work with this module exactly the same way as with any other module. What distinguishes it from other modules is that it can be locked, which will make checking implementation of macrofilters included inside not possible. Such modules will be similar to filters in this matter. This operation will also encrypt the avlib file. It is important to lock the module every time you finish working on it.



Locking module in project explorer.



Macrofilters from locked modules in the program.

In order to unlock a module you need to select the Unlock Module option in the ProjectExplorer and enter the correct password. To completely remove module locking feature from a selected module, use the Remove Module Lock option.

Keyboard Shortcuts

Introduction

Many of Aurora Vision Studio actions can be invoked with keyboard shortcuts. Most of them have default shortcuts - such as copying with `Ctrl+C`, pasting with `Ctrl+V`, finding elements with `Ctrl+F`, saving with `Ctrl+S`, navigating with arrow keys etc.
In some controls - e.g. formula editor - there are present commonly used keyboard shortcuts for text editors: navigating through whole words with `Ctrl+Left/Right Arrow`, selecting consecutive letters/lines (`Shift+Arrow keys`), selecting whole words (`Ctrl+Shift+Left/Right Arrow`), increasing indents with `Tab`, decreasing them with `Shift+Tab` etc.

In this article you can find listed all of the less-known shortcuts.

Table of Contents

Shortcuts Table

Command	Shortcut
Program Editor Back to Top	
Run program	F5
Run program with Aurora Vision Executor	Ctrl+F5
Run program until selected point	Ctrl+F9
Stop program	Shift+F5
Pause program at current executing step	Ctrl+Alt+Pause
Iterate program	F6
Iterate current macrofilter	Ctrl+F10
Step over	F10
Step into	F11
Step out	Shift+F11
Insert new filter instance	Ctrl+T Ctrl+Space
Load recent project	Alt+Number
Rename currently open macrofilter	F2
View program statistics	F8
Toggles breakpoint for currently selected filter and macrofilter outputs block	F9
Copy element	Ctrl+Insert Ctrl+C
Paste element	Shift+Insert Ctrl+V
Change currently selected macrofilter	Ctrl+Number
Navigate to home macrofilter	Alt+Home
Navigate to parent macrofilter	Shift+Enter
Open next macrofilter in history	Alt+Right Arrow
Open previous macrofilter in history	Alt+Left Arrow
Extract step from selected filter/filters	Ctrl+E
Add new Formula instance to current program	Ctrl+Shift+E
Add new Comment instance to current program	Ctrl+Shift+K
Create step macrofilter and add instance	Ctrl+Shift+S

Create task macrofilter and add instance	Ctrl+Shift+T
Create variant macrofilter and add instance	Ctrl+Shift+V
Remove currently opened macrofilter with all its instances	Ctrl+Shift+R
Undock currently opened macrofilter	Ctrl+U
Enable/Disable selected filter instance/instances	Ctrl+L
Add new/Edit existing filter comment	Ctrl+M
Copy currently moving instance	Hold Ctrl while moving instance
Navigate and select elements instances	Shift+Up/Down/Home/End
Move selected filters	Alt+Up Alt+Down
HMI Designer Back to Top	
Move currently selected control	Arrows
Move currently selected control to the edge of its container	Ctrl+Arrows
Resize currently selected control	Shift+Arrows
Resize currently selected control to the edge of its container	Ctrl+Shift+Arrows
Properties Control Back to Top	
Hide/Show selected property	Ctrl+H
Restore default value of selected property	Ctrl+D
Enable/Disable port in current macrofilter variant	Ctrl+E
Set/Unset optional value	Ctrl+P
Enable/Disable port in current macrofilter variant	Ctrl+E
Change scale on property value slider from 1 to 10	Hold Shift while modifying value with slider
Change scale on property value slider from 1 to 0.1	Hold Ctrl while modifying value with slider
Change scale on property value slider from 1 to 0.01	Hold Ctrl+Shift while modifying value with slider
Insert a new line into the text	Shift+Enter
3D View Back to Top	
Show bounding box	B
Show grid	G
Increase point size	]
Decrease point size	[
Move view up	Q Up Arrow
Move view down	Z Down Arrow
Move view left	A Left Arrow
Move view right	D Up Arrow
Zoom in	W Page Up
Zoom out	S Page Down
Decrease rotation angle	Hold CTRL while rotating view
Deep Learning Editors Back to Top	
Change current class	0 - 9
Move one image up	Page Up
Move one image down	Page Down
Save current state of the model	Ctrl+S
Automatic training	Alt+F2

Generate the report	Alt+F3
Deep Learning Editors - Anomalies Detection only Back to Top	
Change the current image's annotation	Space
Label image as good and go to the next one	G
Label image as bad and go to the next one	B
Deep Learning Editors - Object Classification only Back to Top	
Remove selected ROI	Delete
Deep Learning Editors - Instance Segmentation only Back to Top	
Remove single instance	Delete
Add new instance	Space
Split the instances into separate blobs	S
Deep Learning Editors - Point Location only Back to Top	
Remove selected point	Delete

Working with 3D data

Introduction

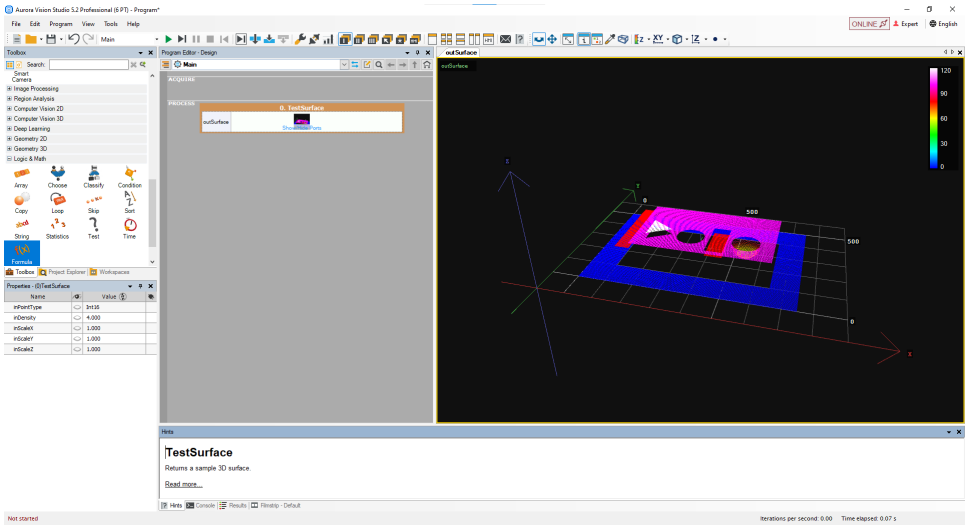
This article summarizes how to work with 3D data previews. Aurora Vision Studio allows you to work not only with images or conventional data types, but also with point clouds of following types:

- Surface
- Point3DGrid
- Point3DArray

Please note that the term *surface* may be used in this article to denote a point cloud of Surface type.

Toolbar

The toolbar appears the moment a point cloud is previewed:



The location of 3D tools in Aurora Vision Studio.

3D tools available in the toolbar.

Button (Tool)	Description of the tool
Rotate	This button permits to rotate a point cloud around the rotation center point.
Pan	This button permits to move the rotation center point.
Resetting view	This button permits to revert changes made by rotating or panning a point cloud.
Probe Point Coordinates	This button permits to obtain information about coordinates of a point.
Bounding Boxes	This button permits to display the bounding box of a point cloud.
Coloring Mode	This button permits to switch between different coloring modes which are divided into 4 categories: • Solid (whole point cloud is of a single color) • Along X-axis (greater values along the X-axis are colored according to the colors' scale bar in the top right corner of the preview) • Along Y-axis (greater values along the Y-axis are colored according to the colors' scale bar in the top right corner of the preview) • Along Z-axis (greater values along the Z-axis are colored according to the colors' scale bar in the top right corner of the preview)
Grid Mode	This button permits to display a grid plane to your liking. There are also 4 modes available: • None (no grid) • XY (the grid is displayed in XY plane) • XZ (the grid is displayed in XZ plane) • YZ (the grid is displayed in YZ plane)
View Projection Mode	This button permits to display a point cloud in 4 different modes: • Perspective (default mode) • Up (overhead view – showing a point cloud from above – looking down at XY plane) • Back (view from the back – looking at XZ plane) • Left (side view – looking at YZ plane)
World Orientation	This button permits to display a coordinate system in 4 different modes: • Right Handed Z Up • Right Handed Y Up • Left Handed Z Up • Left Handed Y Up
Point size	This button permits to control the size of a single point within a point cloud. There are 5 possible sizes: • Very small • Small • Medium • Large • Very large

Using a computer mouse

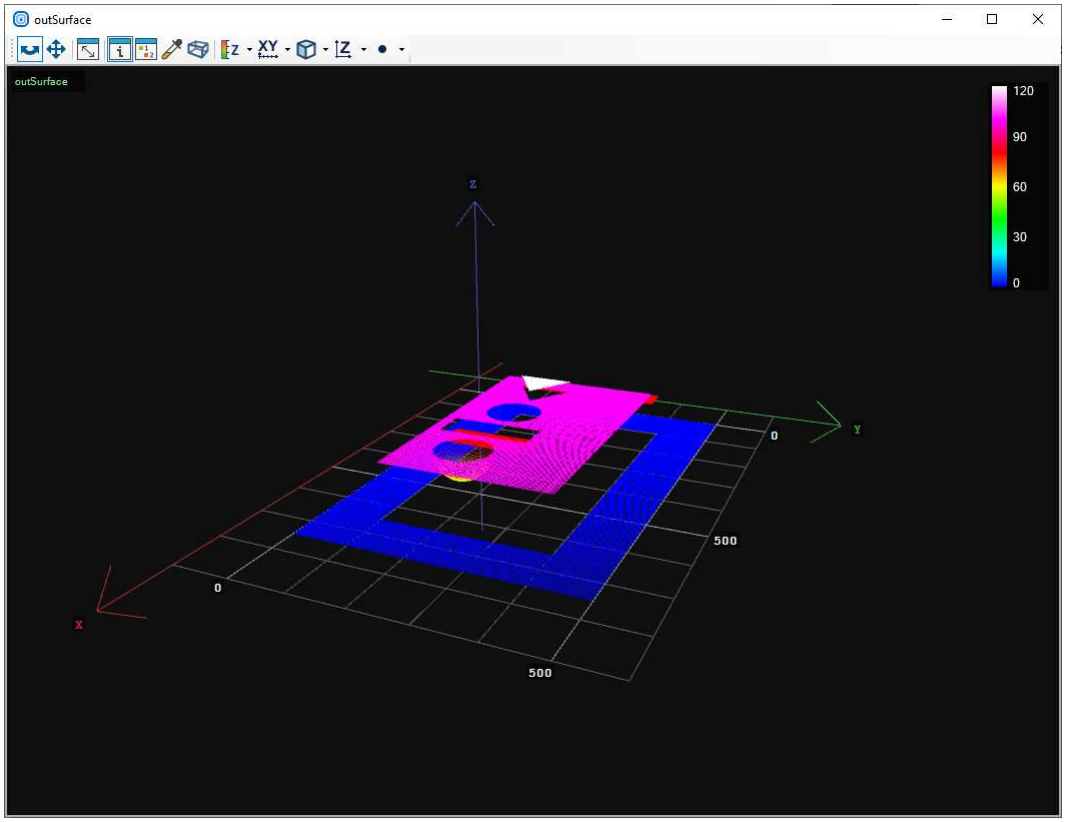
Aurora Vision Studio also allows you to control a 3D preview with your computer mouse. You can perform some of the above features to analyze the point cloud without a necessity to find and switch between buttons in the toolbar.

	If you click **Left Mouse Button** on a preview, it works as if you used the Rotate button, so you can rotate a point cloud around a fixed center point until the button is released.
	If you click **Right Mouse Button** on a point cloud, a dropdown list with all features described in the previous section appears.
	If you use **Mouse Wheel** on a point cloud, you can zoom it in and out.
	If you press **Mouse Wheel** on a point cloud, you can move the rotation center point of the coordinate system (works like the "Pan" button).

Preview

When you drag and drop the output, which is of 3D data type, and you have already previewed some images, string or real values, please note that it can be only previewed in a separate view. It is not possible to display an image and a point cloud in the very same view.

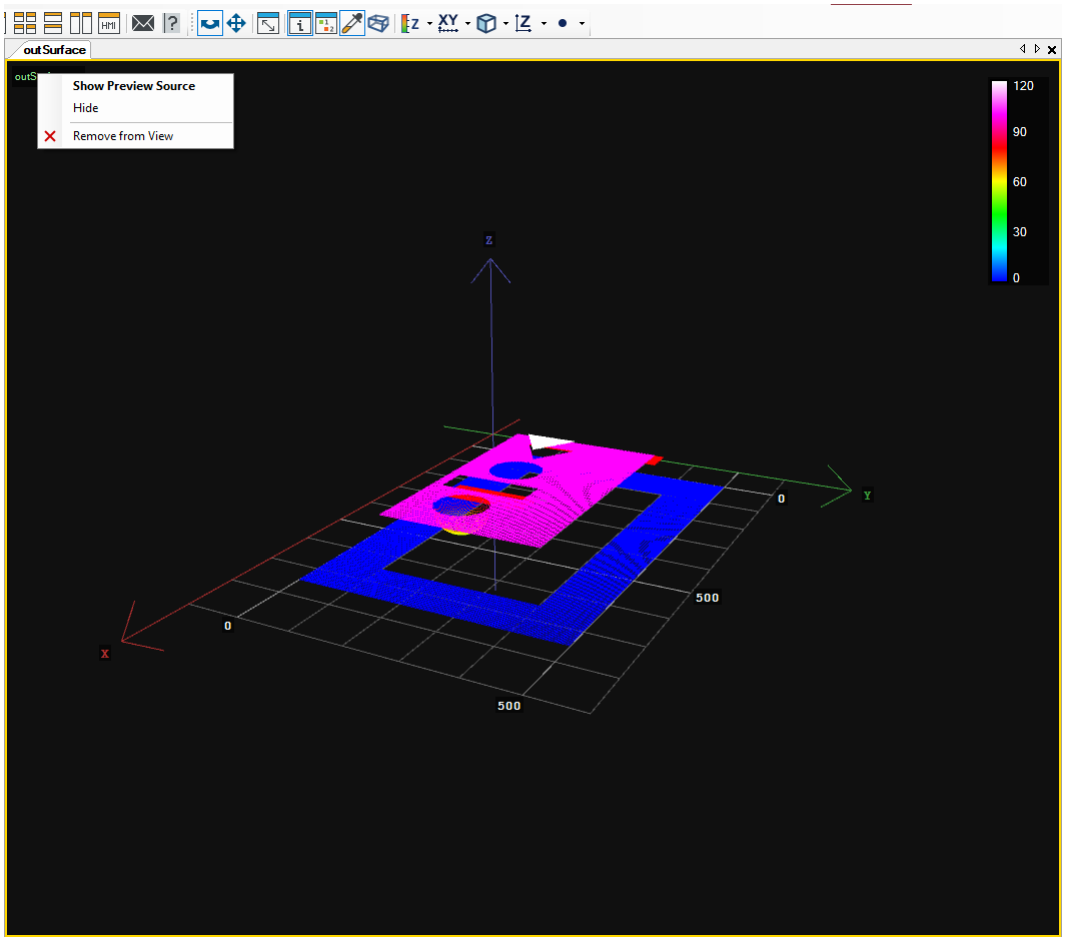
While analyzing 3D data, please pay attention to the colors' scale bar in the top right corner of the preview:



Colors' scale bar.

It might be helpful if you want to estimate the coordinate value based on color.

You can also hide or remove surfaces from a view. To do it, you have to right-click on the information about a preview in the top left corner of the view:



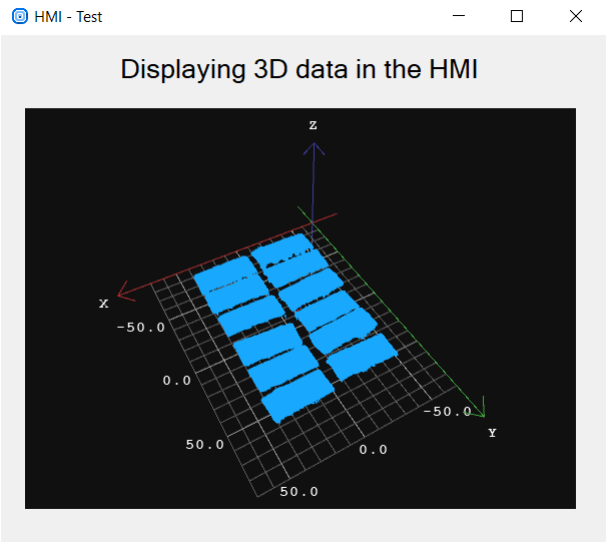
Dropdown list with possible actions on previews.

As it is shown in the above image, you can:

- Show Preview Source - it will indicate the filter in the Program Editor, of which the output has been previewed
- Hide - it will hide only indicated surface without removing the preview
- Remove from View - it will delete the preview, leaving an empty view

HMI

3D data can be displayed in the HMI as well - using the View3DBox control:



View3DBox HMI control - a point cloud is displayed in the same manner as in the preview.

It is worth mentioning that in this HMI control you can also rotate and change the rotation center of the point cloud. For more details about designing and using HMI, please refer to HMI Controls.

Creating Deep Learning Model

Contents:

Introduction

Deep Learning editors are dedicated graphical user interfaces for DeepModel objects (which represent training results). Each time a user opens such an editor, he is able to add or remove images, adjust parameters and perform new training.

Since version 4.10 it is also an option to open a Deep Learning Editor as a stand-alone application, which is especially useful for re-training models with new images in production environment.

Requirements:

- A DeepLearning license is required to use Deep Learning editors and filters.
- Deep Learning Service must be up and running to perform model training.

Currently available deep learning tools are:

1. **Anomaly Detection** – for detecting unexpected object variations; trained with sample images marked simply as good or bad.
2. **Feature Detection** – for detecting regions of defects (such as surface scratches) or features (such as vessels on medical images); trained with sample images accompanied with precisely marked ground-truth regions.
3. **Object Classification** – for identifying the name or the class of the most prominent object on an input image; trained with sample images accompanied with the expected class labels.
4. **Instance Segmentation** – for simultaneous location, segmentation and classification of multiple objects in the scene; trained with sample images accompanied with precisely marked regions of each individual object.
5. **Point Location** – for location and classification of multiple key points; trained with sample images accompanied with marked points of expected classes.
6. **Read Characters** – for location and classification of multiple characters; this tool uses a pretrained model and cannot be trained, so it is not described in this article
7. **Object Location** – for location and classification of multiple objects; trained with sample images accompanied with marked bounding rectangles of expected classes.

Technical details about these tools are available at Machine Vision Guide: Deep Learning while this article focuses on the training graphical user interface.

Workflow

You can open a **Deep Learning Editor** via:

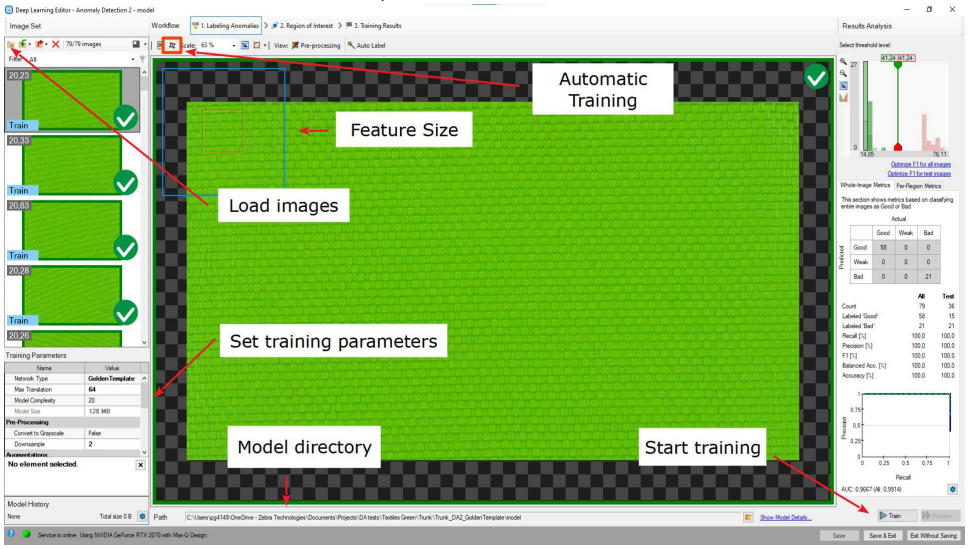
- a filter in Aurora Vision Studio:
  1. Place the relevant DL filter (e.g. DL_DetectFeatures or DL_DetectFeatures_Deploy) in the Program Editor.
  2. Go to its Properties.
  3. Click on the button next to the **inModelDirectory** or **inModelId.ModelDirectory** parameter.
- a standalone **Deep Learning Editor** application:
  1. Open a standalone **Deep Learning Editor** application (which can be found in the **Vision Studio installation folder** as "DeepLearningEditor.exe", in Aurora Vision folder in **Start menu** or in Aurora Vision Studio application in **Tools menu**).
  2. Choose whether you want to create a new model or use an existing one:
     - **Creating a new model**: Select the relevant tool for your model and press OK, then select or create a new folder where files for your model will be contained and

The Deep Learning model preparation process is usually split into the following steps:

1. **Loading images** – load training images from disk
2. **Labeling images** – mark features or attach labels on each training image
3. **Setting the region of interest (optional)** – select the area of the image to be analyzed
4. **Adjusting training parameters** – select training parameters, preprocessing steps and augmentations specific for the application at hand
5. **Training the model and analyzing results**

press OK.

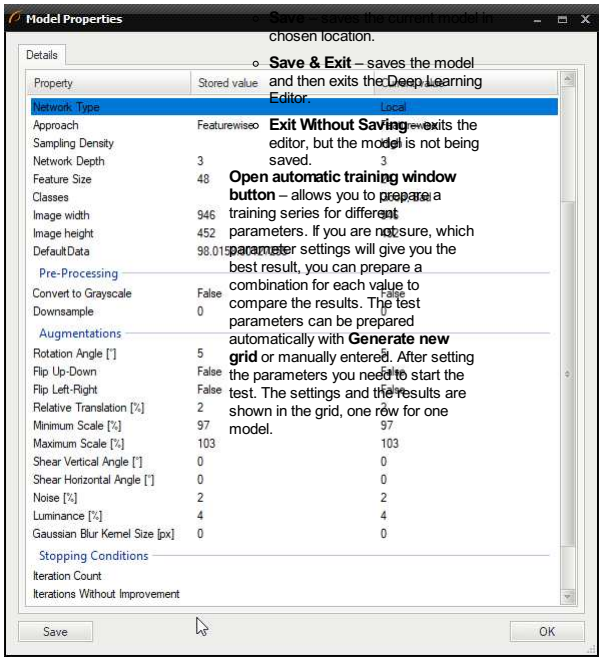
- **Choosing existing model**: Navigate to the folder containing your model files – either write a path to it, click on the button next to field to browse to it or select one of the recent paths if there are any; then press OK.



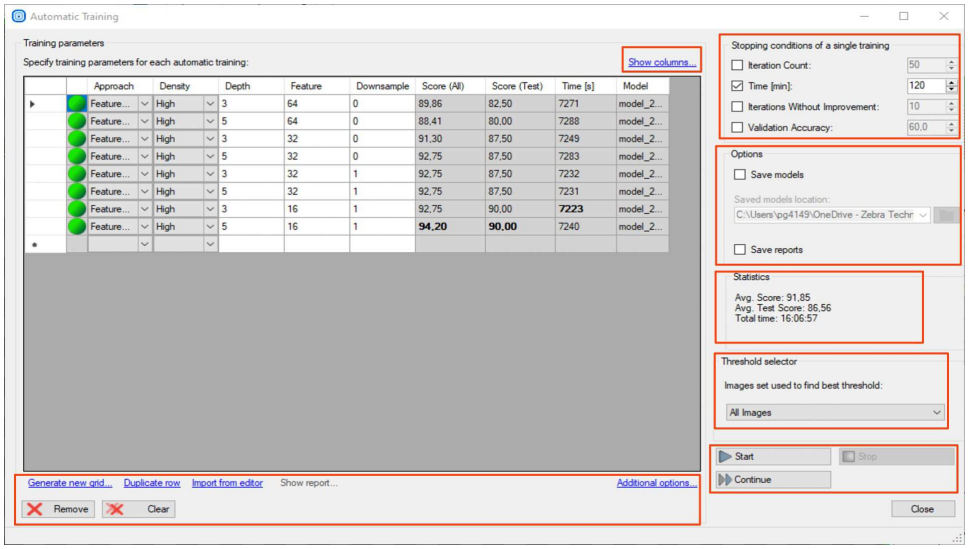
Overview of a Deep Learning Editor.

Important features:

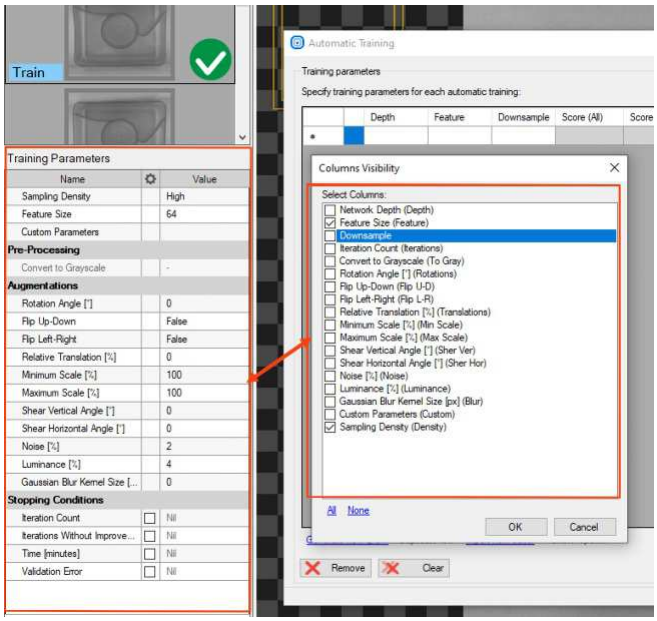
- **Pre-processing button** – located in the top toolbar; allows you to see the changes applied to a training image e.g. grayscale or downsampling.
- **Current Model directory** – located in the bottom toolbar; allows you to switch a model being in another directory or to simply see which model you are actually working on.
- **Show Model Details button** – located next to the previous control; allows you to display information on the current model and save them to a file.
- **Train & Resume buttons** – allow you to start training or resume it in case you have changed some of training parameters.
- Saving buttons:
  - **Save** – saves the current model in chosen location.
  - **Save & Exit** – saves the model and then exits the Deep Learning Editor.
  - **Exit Without Saving** – exits the editor, but the model is not being saved.



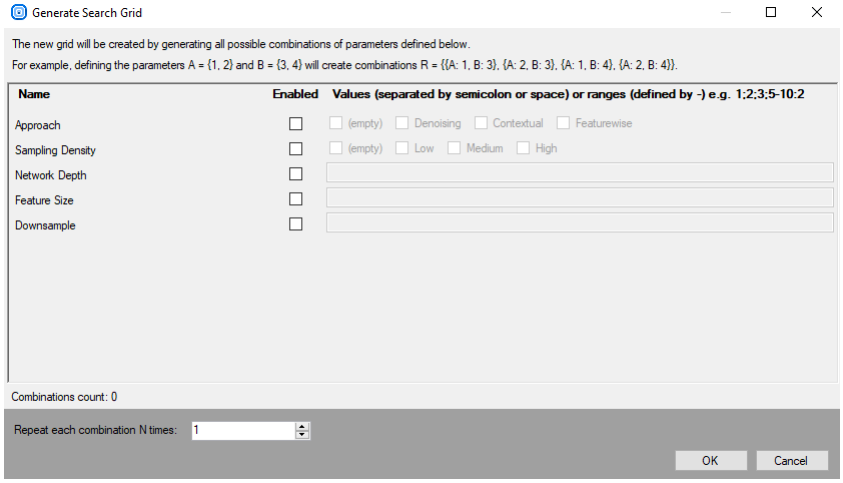
- **Open automatic training window button** – allows you to prepare a training series for different parameters. If you are not sure, which parameter settings will give you the best result, you can prepare a combination for each value to compare the results. The test parameters can be prepared automatically with **Generate new grid** or manually entered. After setting the parameters you need to start the test. The settings and the results are shown in the grid, one row for one model.



Show columns – hides / shows model parameters you will use in your test. The view is common for all deep learning tools. To create an appropriate grid search, choose these parameters which are correct for the used tool (the ones you can see in Training Parameters). For DL_DetectAnomalies2 choose the network type first to see the appropriate parameters.



Generate new grid – prepares the search grid for the given parameters. Only parameters chosen in **Show columns** are available. The values should be separated with ; sign.



Duplicate rows – duplicates a training parameters configuration. If the parameters inside the row won't be modified, this model will be trained twice.

Import from editor – copies the training parameters from **Editor Window** to the last search grid row.

Show report – shows the report for the chosen model (the chosen row). This option is available only if you choose **Save Reports** before starting the training session.

Additional options

- **Export grid to CSV file** – exports the grid of training parameters to a CSV file.
- **Import grid from CSV file** – imports the grid of training parameters from a CSV file.

Remove – removes a chosen training configuration.

Clear – clears the whole search grid.

Stopping conditions of a single training – determines when a single training stops.

- Iteration Count
- Time
- Iterations without improvement
- Validation Accuracy

Options

- **Save models** – saves each trained model in the defined folder.
- **Save reports** – saves a report for each trained model in the folder defined for saving models.

Statistics – shows statistics for all trainings.

- **Avg. Score** – shows average score of all trained models for all images.
- **Avg. Test Score** – shows average score of all trained models for test images.
- **Total time** – sums up time of each training.

Threshold selector – choses for which image group the best threshold is searched.

- **All Images**
- **Test Images**

Start – starts the training series with the first defined configuration.

Stop – stops the training series.

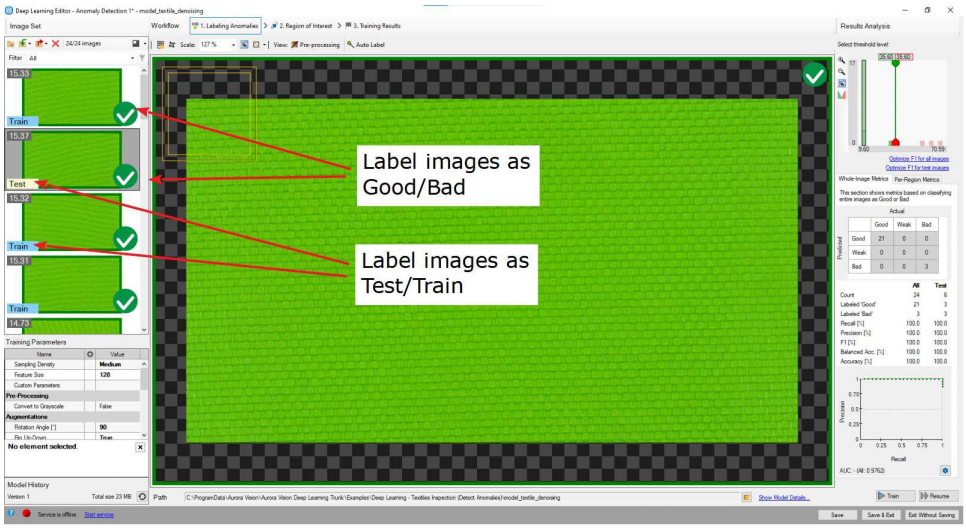
Continue – continues the stopped training series with the next configuration of the parameters.

Detecting anomalies 1

In this tool the user only needs to mark which images contain correct cases (good) or incorrect ones (bad).

1. Marking Good and Bad samples and dividing them into Test and Training data.

Click on a question mark sign to label each image in the training set as **Good** or **Bad**. Green and red icons on the right side of the training images indicate to which set the image belongs to. Alternatively, you can use **Add images and mark...**. Then divide the images into **Train** or **Test** by clicking on the left label on an image. Remember that all bad samples should be marked as **Test**.



Labeled images in Deep Learning Editor.

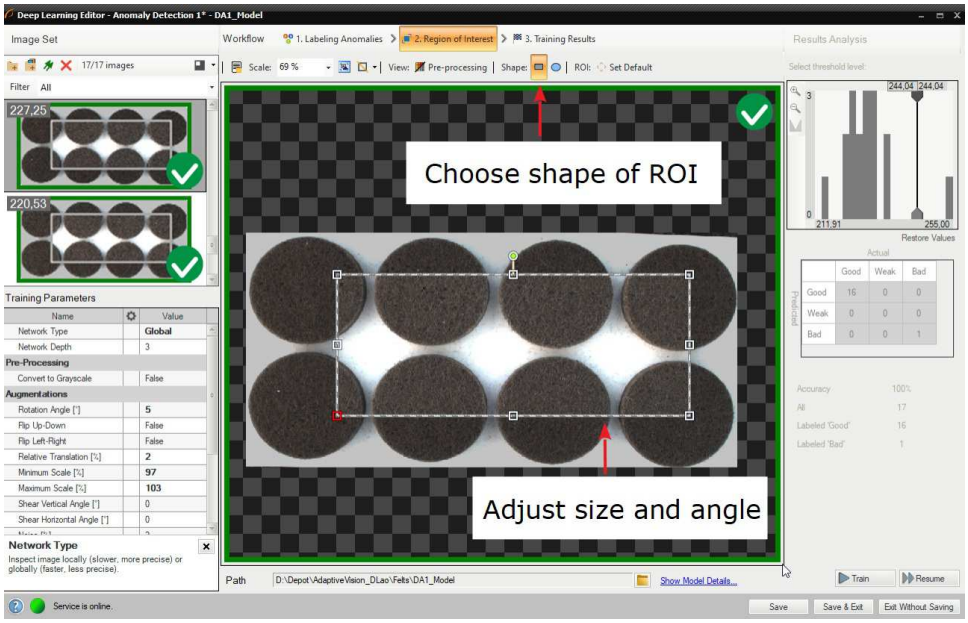
2. Configuring augmentations

It is usually recommended to add some additional sample augmentations, especially when the training set is small. For example, the user can add additional variations in pixels intensity to prepare the model for varying lighting conditions on the production line. Refer to "Augmentation" section for detailed description of parameters: Deep Learning – Augmentation.

3. Reducing region of interest

Reduce region of interest to focus only on the important part of the image. Reducing region of interest will speed up both training and inference.

Please note that the Region of Interest in this tool is the same for each image in a training set and it cannot be adjusted individually. As a result, this Region of Interest is automatically applied during execution of a model, so a user has no impact on its size or shape in the Program Editor.



By default region of interest contains the whole image.

4. Setting training parameters

- **Sampling density** (for the Featurewise approach in DL_DetectAnomalies1 and in DL_DetectAnomalies2 only) – selecting from: Low, Medium and High.
- **Feature size** (for the Local type only) – the width of the inspection window. In the top left corner of the editor a small rectangle visualizes the selected feature size.
- **Stopping conditions** – define when the training process should stop.

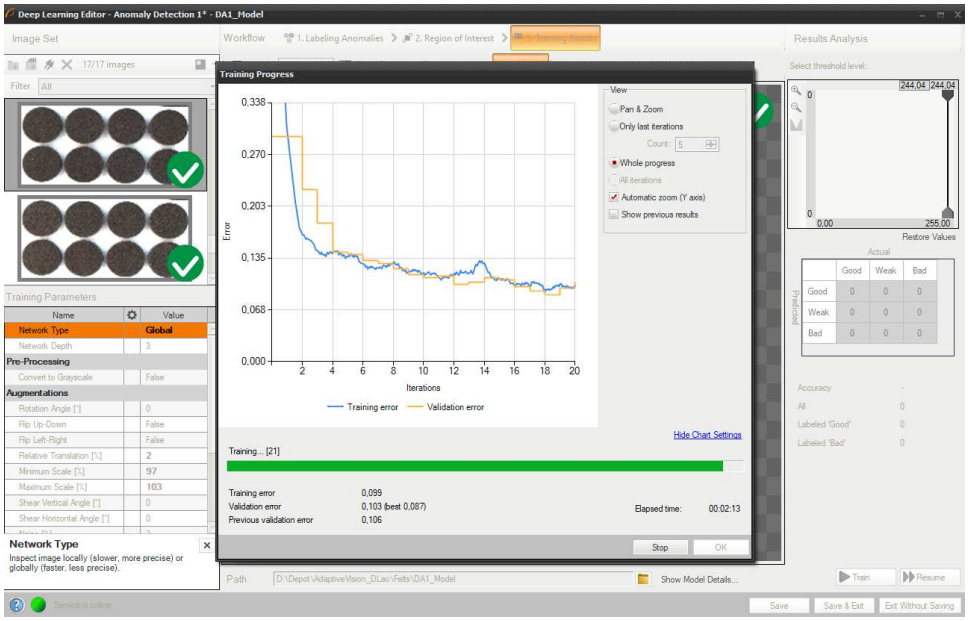
For more details read Deep Learning – Setting parameters.

5. Performing training

During training, two figures are successively displayed: training error and validation error. Both charts should have a similar pattern.

More detailed information is displayed on the chart below:

- current training statistics (training and validation),
- number of processed samples (depends on the number of images and feature size),
- elapsed time.



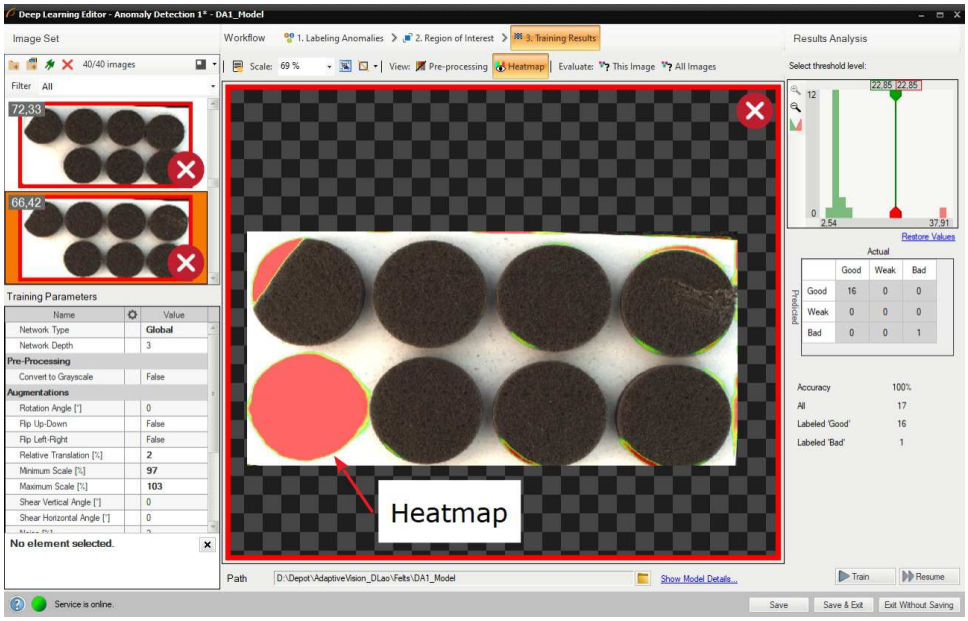
Training process consists of computing training and validation errors.

Training process can take its time depending on the selected stopping criteria and available hardware. During this time training can be finished manually anytime by clicking Stop button. If no model is present (the first training attempt) model with the best validation accuracy will be saved. Consecutive training attempts will prompt user about old model replacement.

6. Analyzing results

The window shows a histogram of sample scores and a heatmap of found defects. The left column contains a histogram of scores computed for each image in the training set. Additional statistics are displayed below the histogram.

To evaluate trained model, **Evaluate: This Image** or **Evaluate: All Images** buttons can be used. It can be useful after adding new images to the data set or after changing the area of interest.



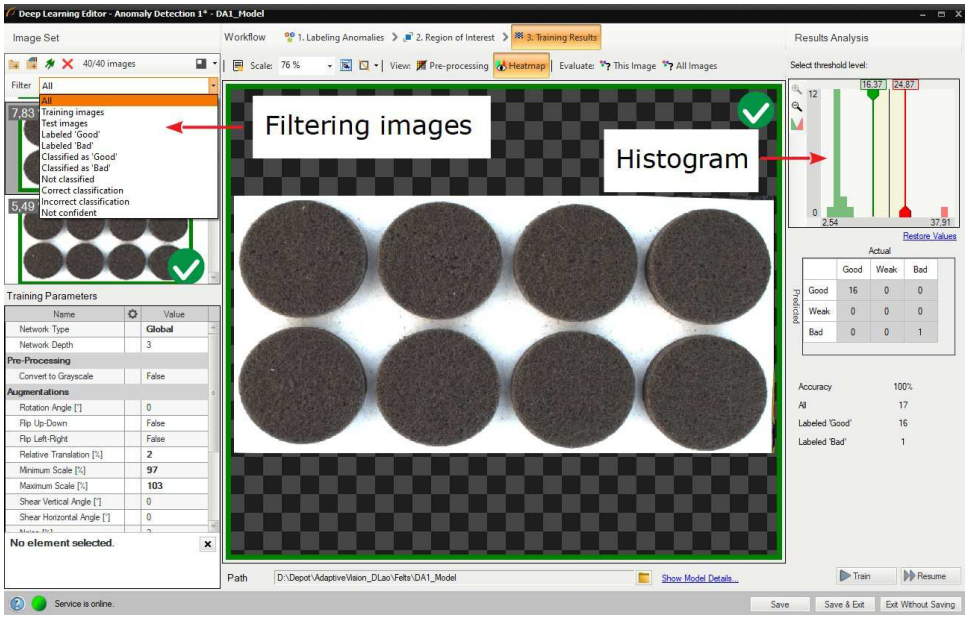
Heatmap presents the area of possible defects.

After training two border values are computed:

1. Maximum good sample score (T1) – all values from 0 to T1 are marked as Good.
2. Minimum bad sample score (T2) – all values greater than T2 are marked as Bad.

All scores between T1 and T2 are marked as "Low quality". Results in this range are uncertain and may be not correct. Filters contain an additional output **outIsConfident** which determines the values which are not in the T1-T2 range.

After evaluation additional filtering options may be used in the the list of training images.



Filtering the images in the training set.

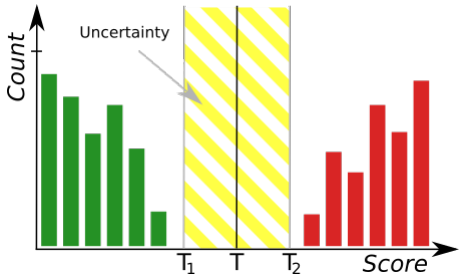
Interactive histogram tool

DetectAnomalies filters measure deviation of samples from the normal image appearances learned during the training phase. If the deviation exceeds a given threshold, the image is marked as anomalous. The suggested threshold is automatically calculated after the training phase but it can also be adjusted by the user in the Deep Learning Editor using an interactive histogram tool described below.

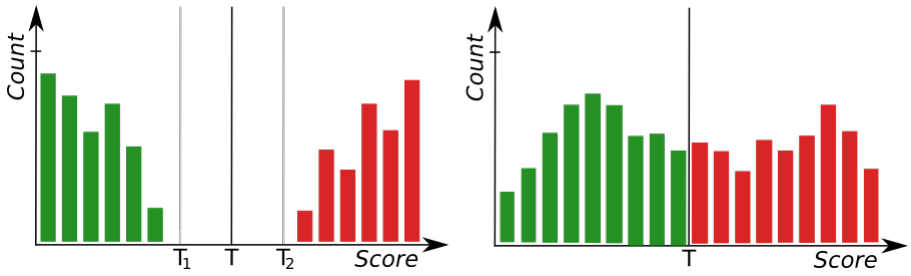
After the training phase, scores are calculated for every training sample and are presented in the form of a histogram; good samples are marked with green, bad samples with red bars. In the perfect case, the scores for good samples should be all lower than for bad samples and the threshold should be automatically calculated to give the optimal accuracy of the model. However, the groups may sometimes overlap because of:

1. incorrectly labeled samples,
2. bad Feature Size,
3. ambiguous definition of the expected defects,
4. high variability of the samples appearance or environmental conditions.

In order to achieve more robust threshold, it is recommended to perform training with a large number of samples from both groups. If the number of samples is limited, our software makes it possible to manually set the uncertainty area with additional thresholds (the information about the confidence of the model can be then obtained from the hidden outIsConfident filter output).



The histogram tool where green bars represent correct samples and red bars represent anomalous samples. T marks the main threshold and T1, T2 define the area of uncertainty.

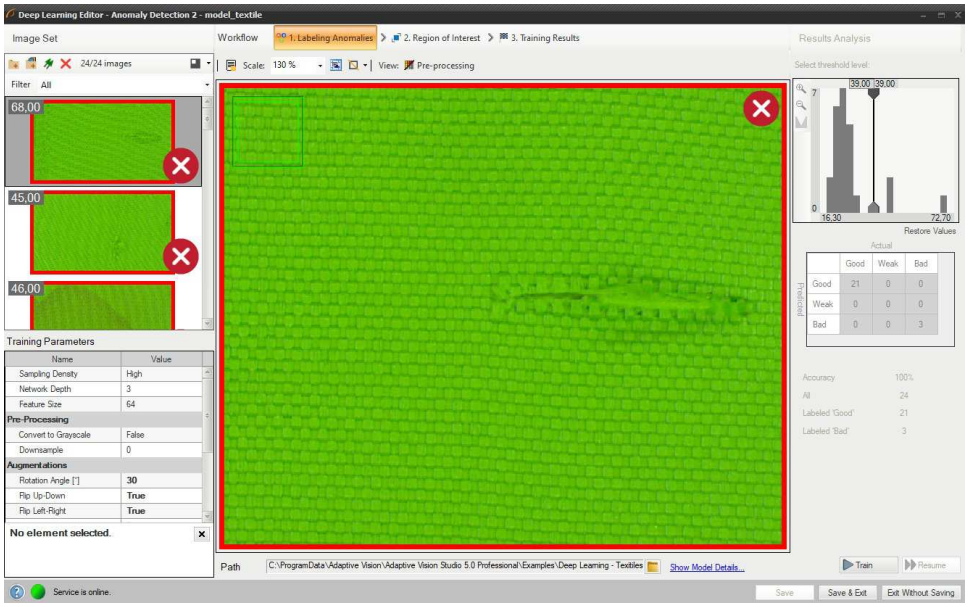


Left: a histogram presenting well-separated groups indicating a good accuracy of the model. Right: a poor accuracy of the model.

Detecting anomalies 2 (classificational approach)

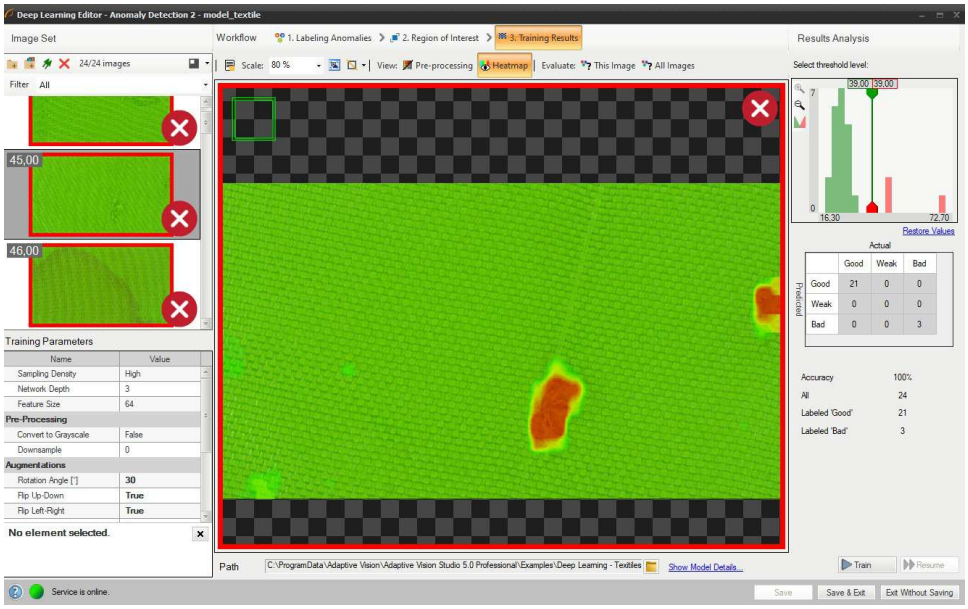
DL_DetectAnomalies2 is another filter which can be used for detecting anomalies. It is designed to solve the same problem, but in a different way. Instead of using image reconstruction techniques Anomaly Detection 2 performs one-class classification of each part of the input image.

As both tools are very similar the steps of creating a model are the same. There are only a few differences between those filters in the model parameters section. In case of DL_DetectAnomalies2 the user does not need to change iteration count and network type. Instead, it is possible to set the Sampling density that defines the step of analysis using the inspection window. The higher Sampling density, the more precise heatmaps, but longer training and inference times.



Results are marked as rectangles of either of two colors: green (classified as good) or red (classified as bad).

The resulting heatmap is usually not as accurate spatially as in case of using reconstructive anomaly detection, but the score accuracy and group separation on the histogram may be much better.



Heatmap indicates the most possible locations of defects.

Detecting features (segmentation)

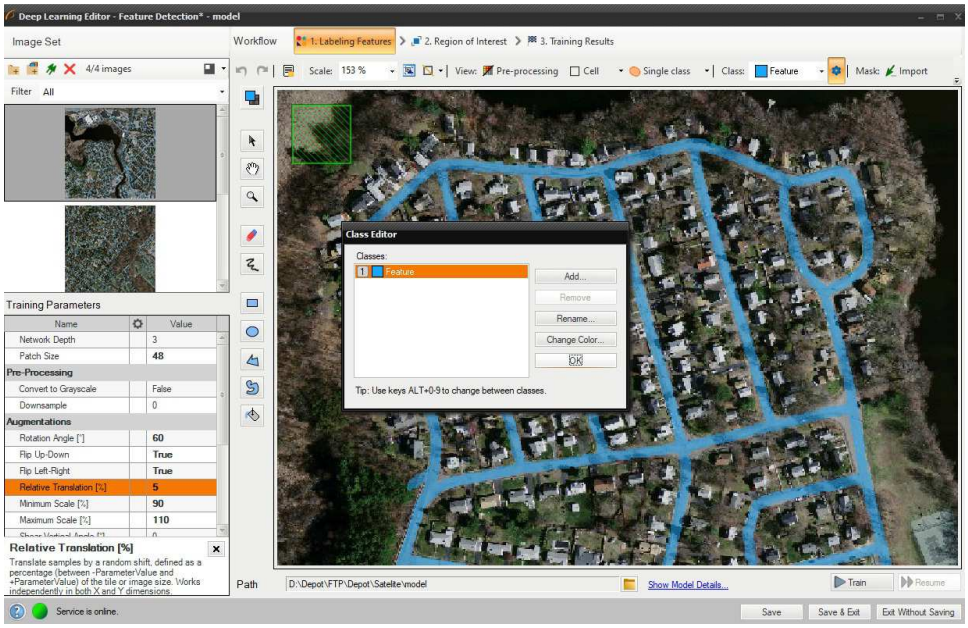
In this tool the user has to define each feature class and then mark features on each image in the training set. This technique is used to find object defects like scratches or color changes, and for detecting image parts trained on a selected pattern.

1. Defining feature classes (Marking class)

First, the user has to define classes of defects. Generally, they should be features which user would like to detect on images. Multiple different classes can be defined but it is not recommended to use more than a few.

Class editor is available under sprocket wheel icon in the top bar.

To manage classes, **Add**, **Remove** or **Rename** buttons can be used. To customize appearance, color of each class can be changed using **Change Color** button.



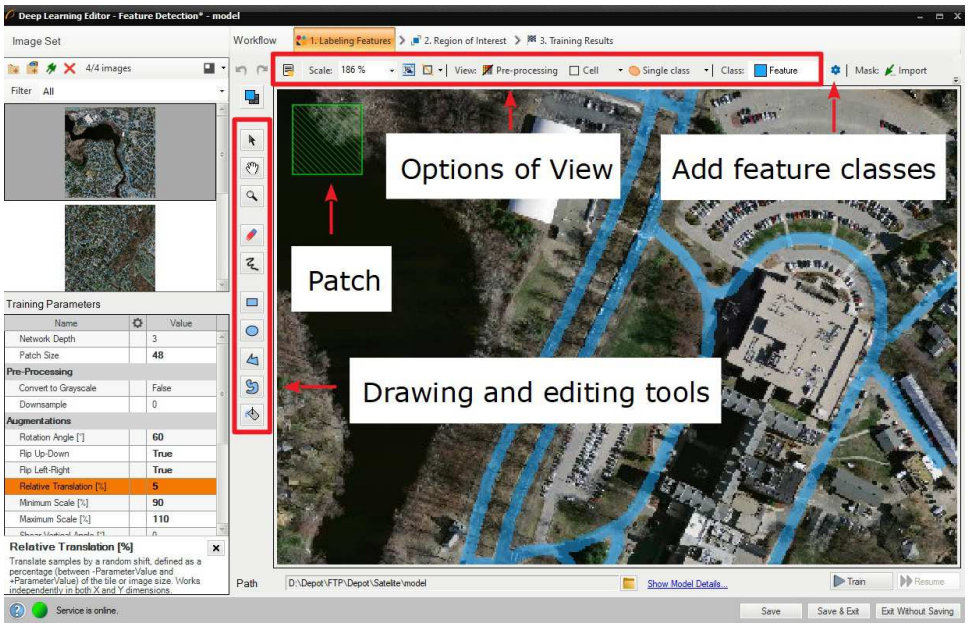
In this tool it is possible to define more classes of defects.

The current class for editing is displayed on the left, the user can select different class after click.

Use drawing tool to mark features on the input images. Tools such as **Brush** or **Rectangle** can be used for selecting features.

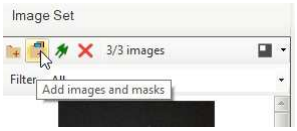
In addition, class masks can be imported from external files. There are buttons for **Import** and **Export** created classes so that the user can create an image of a mask automatically prior to a Deep Learning model.

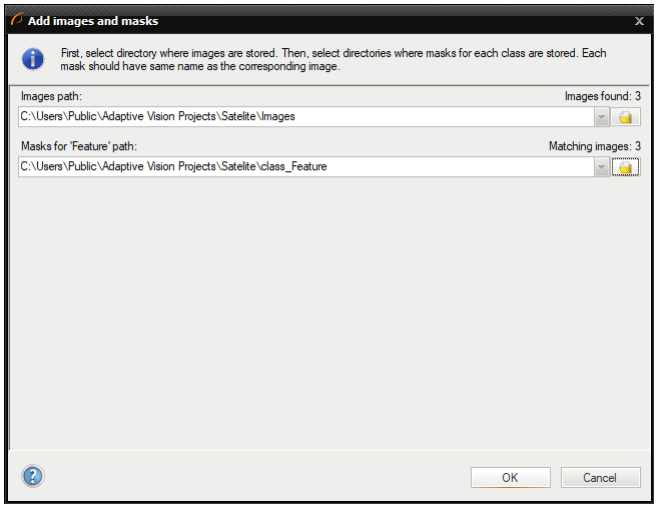
The image mask should have the same size as the selected image in the input set. When importing an image mask, all non-black pixels will be included in the current mask.



The most important features of the tool.

The user can also load multiple images and masks at the same time, using **Add images and masks** button.





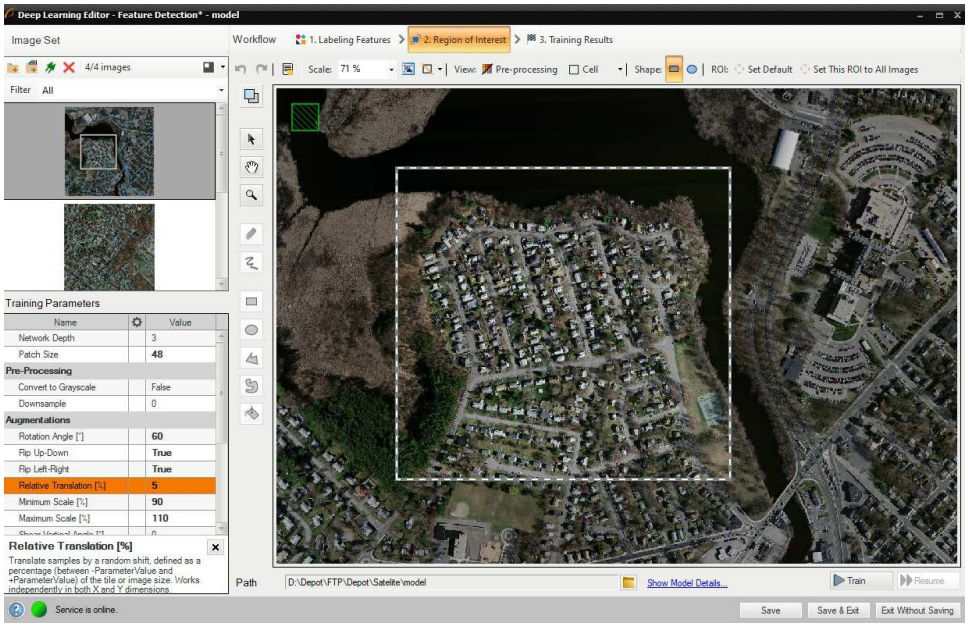
Selecting path to images and masks.

Directory containing input images should be selected first. Then, directories for each feature class can be selected below. Images and masks are matched automatically using their file names. For example, let us assume that "images" directory contains images 001.png, 002.png, 003.png; "mask_class1" directory contains 001.png, 002.png, 003.png; and "mask_class2" directory contains 001.png, 002.png, 003.png. Then "images\001.png" image will be loaded together with "mask_class1\001.png" and "mask_class2\001.png" masks.

2. Reducing region of interest

The user can reduce the input image size to speed up the training process. In many cases, number of features on an image is very large and most of them are the same. In such case the region of interest also can be reduced.

On the top bar there are tools for applying the current ROI to all images as well as for resetting the ROI.



Setting ROI.

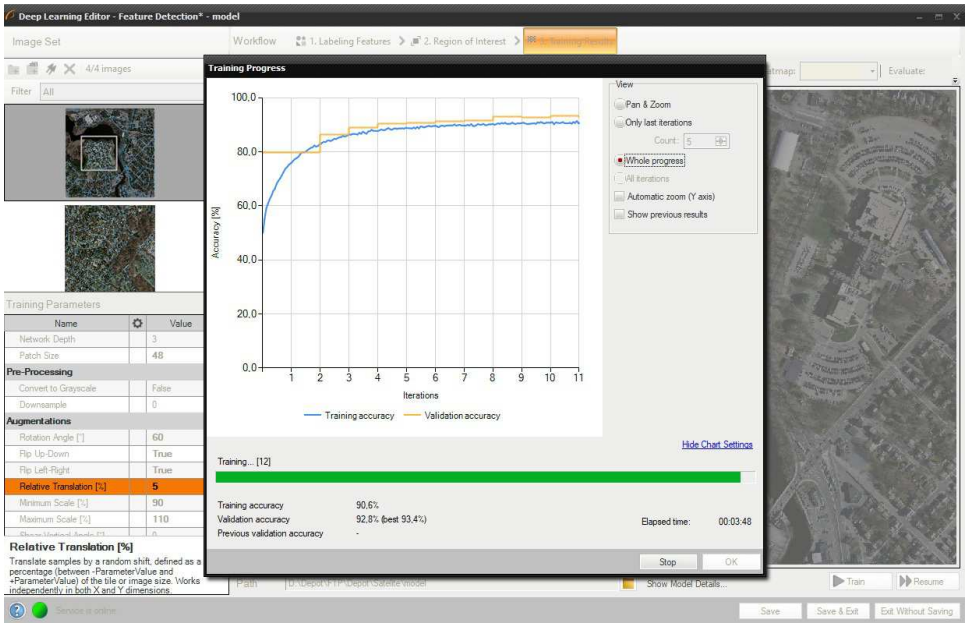
3. Setting training parameters

- **Network depth** – chooses one of several predefined network architectures varying in their complexity. For bigger and more complex image patterns a higher depth might be necessary.
- **Patch size** – the size of an image part that will be analyzed with one pass through the neural network. It should be significantly bigger than any feature of interest, but not too big – as the bigger the patch size, the more difficult and time consuming is the training process.
- **Stopping conditions** – define when the training process should stop.

For more details please refer to Deep Learning – Setting parameters and Deep Learning – Augmentation.

4. Model training

The chart contains two series: training and validation score. Higher score value leads to better results.



In this case training process consists of computing training and validation accuracies.

5. Result analysis

Image scores (heatmaps) are presented in blue-yellow-red colors palette after using the model to evaluation of image. The color represents the probability of the element belonging to the currently selected feature class.

Evaluate: This Image and **Evaluate: All Images** buttons can be used to classify images. It can be useful after adding new images to the data set or after changing the area of interest.

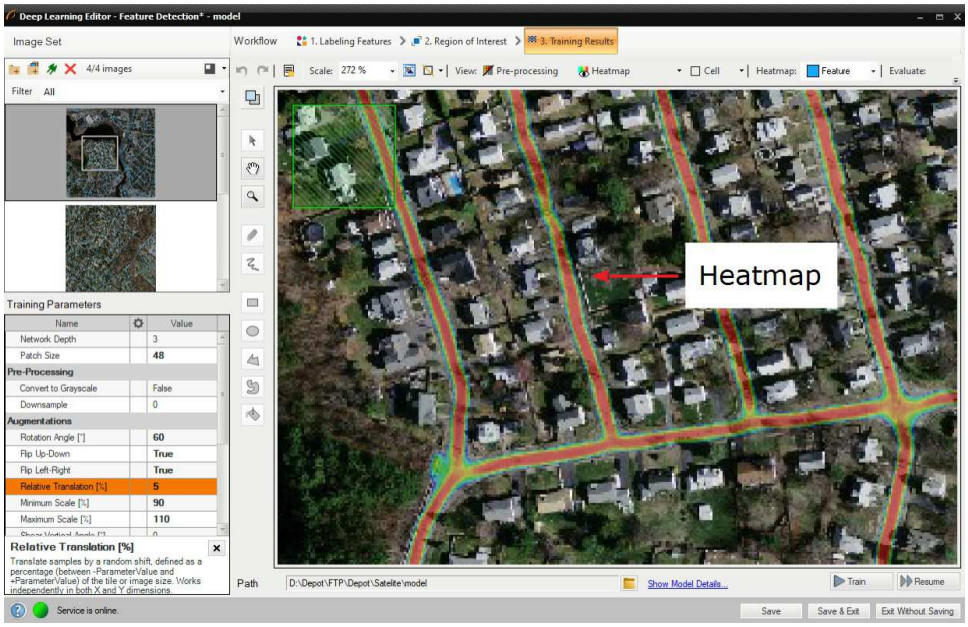


Image after classification.

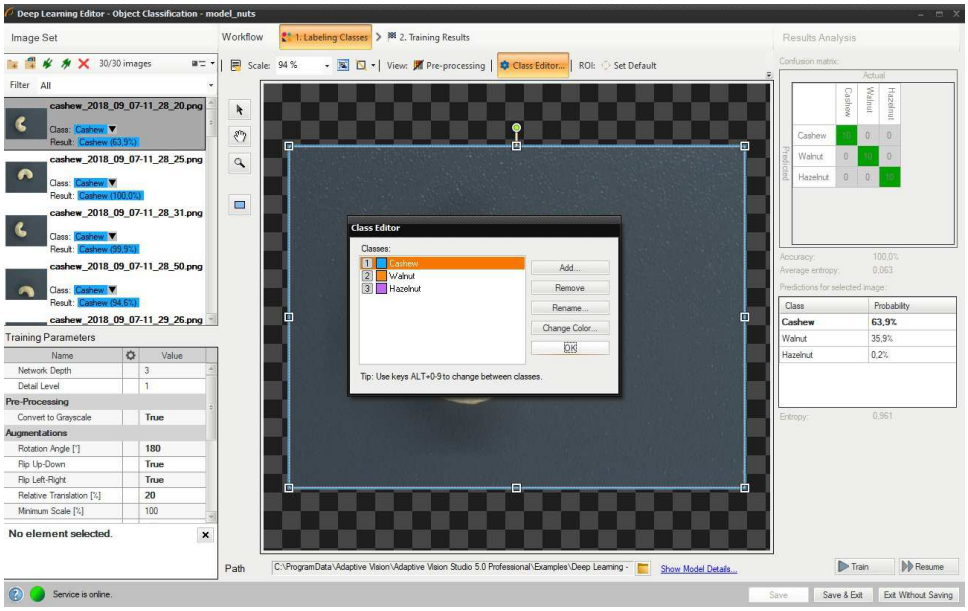
In the top left corner of the editor, a green rectangle visualizes the selected patch size.

Classifying objects

In this too, the user only has to label images with respect to a desired number of classes. Theoretically, the number of classes that a user can create is infinite, but please note that you are limited by the amount of data your GPU can process. Labeled images will allow to train model and determine features which will be used to evaluate new samples and assign them to proper classes.

1. Editing number of classes

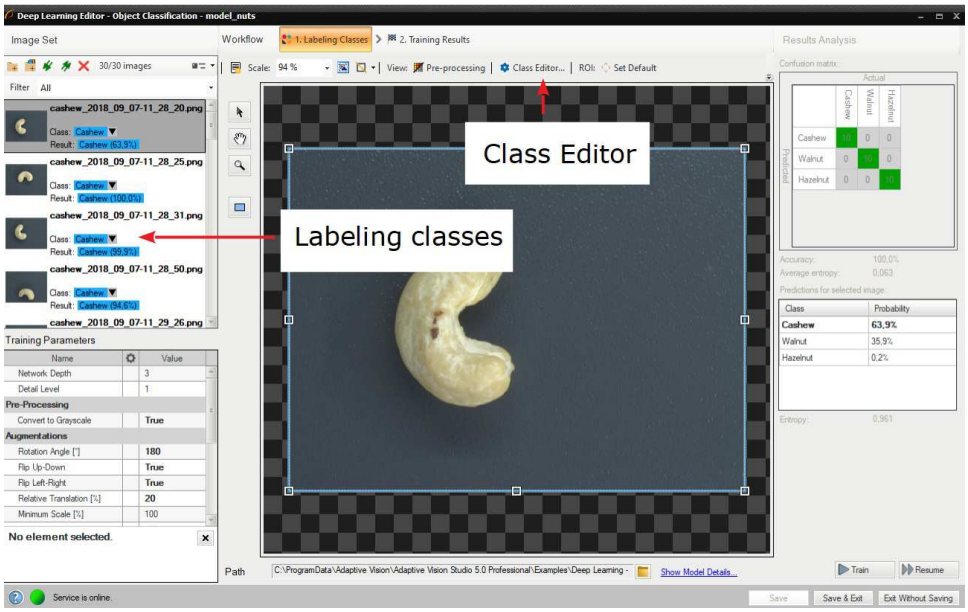
By default two classes are defined. If the problem is more complex than that, the user can edit classes and define more if needed. Once the user is ready with definition of classes, images can be labeled.



Using Class Editor.

2. Labeling samples

Labeling of samples is possible after adding training images. Each image has a corresponding drop-down list which allows for assigning a specific class. It is possible to assign a single class to multiple images by selecting desired images in Deep Learning Editor.

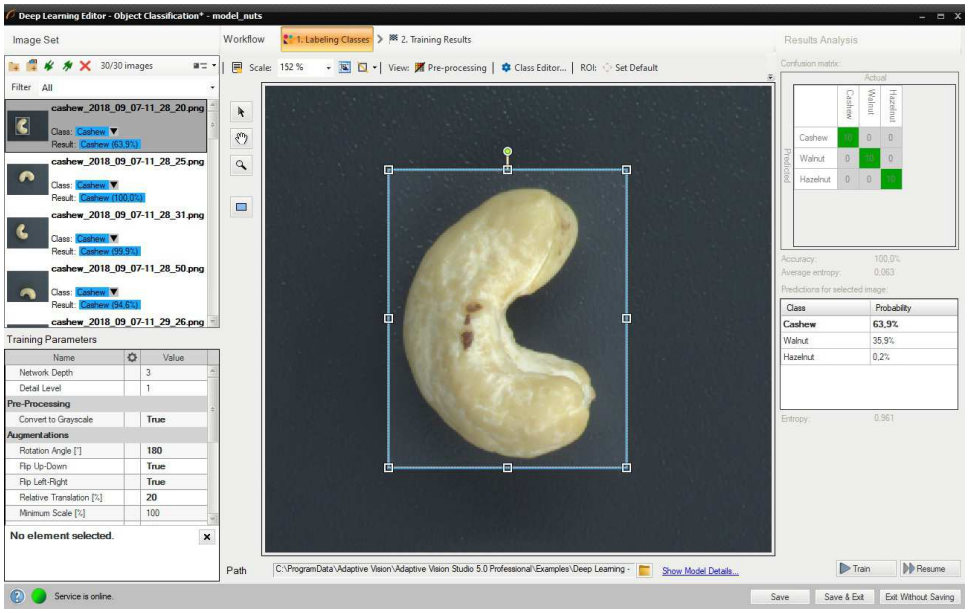


Labeling images with classes.

3. Reducing region of interest

Reduce the region of interest to focus only on the important part of the image. Reducing region of interest will speed up both training and classification. By default the region of interest contains the whole image.

To get the best classification results, use the same region of interest for training and classification.



Changed region of interest.

4. Setting training parameters

- **Network depth** – predefined network architecture parameter. For more complex problems higher depth might be necessary.
- **Detail level** – level of detail needed for a particular classification task. For majority of cases the default value of 1 is appropriate, but if images of different classes are distinguishable only by small features, increasing value of this parameter may improve classification results.
- **Stopping conditions** – define when the training process should stop.

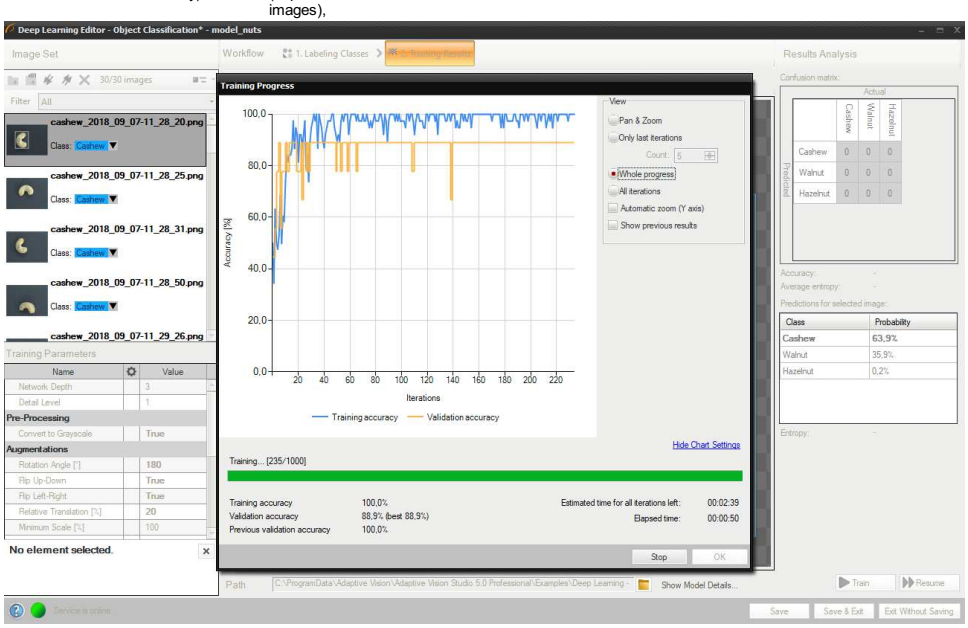
For more details please refer to Deep Learning – Setting parameters and Deep Learning – Augmentation.

5. Performing training

During training two series are visible: training accuracy and validation accuracy. Both charts should have a similar pattern.

More detailed information is displayed below the chart:

- current training statistics (training and validation accuracy),
- number of processed samples (depends on the number of images),
- elapsed time.



Training object classification model.

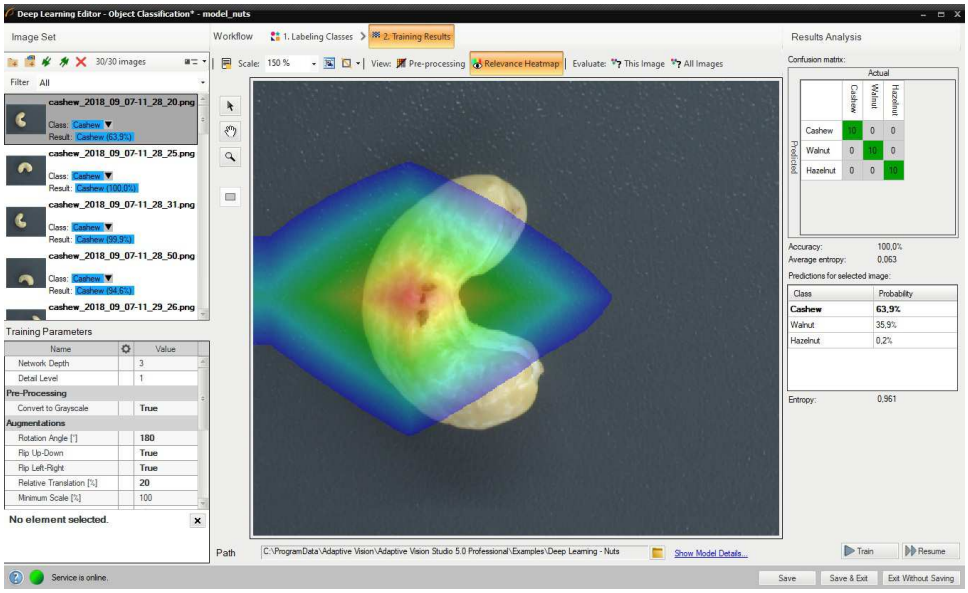
Training process can take a couple of minutes or even longer. It can be manually finished if needed. The final result of one training is one of the partial models that achieved the highest validation accuracy (not necessarily the last one). Consecutive training attempts will prompt the user whether to save a new model or keep the old one.

6. Analyzing results

The window shows a confusion matrix which indicates how well the training samples have been classified.

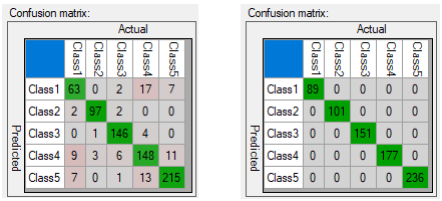
The image view contains a heatmap which indicates which part of the image contributed the most to the classification result.

Evaluate: This Image and **Evaluate: All Images** buttons can be used to classify training images. It can be useful after adding new images to the data set or after changing the area of interest.



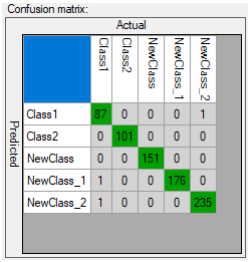
Confusion matrix and class assignment after the training.

Sometimes it is hard to guess the right parameters in the first attempt. The picture below shows confusion matrix that indicates inaccurate classification during the training (left).



Confusion matrices for model that needs more training (left) and for model well trained (right).

It is possible that confusion matrix indicates that trained model is not 100% accurate with respect to training samples (numbers assigned exclusively on main diagonal represent 100% accuracy). User needs to properly analyze this data, and use to his advantage.



Confusion matrix indicating good generalization.

Too many erroneous classifications indicate poor training. Few of them may indicate that model is properly focused on generalization rather than exact matching to training samples (possible overfitting). Good generalization can be achieved if images used for training are varied (even among single class). If provided data is not varied within classes (user expects exact matching), and still some images are classified outside main diagonal after the training, user can:

- increase the network depth,
- prolong training by increasing number of iterations,
- increase amount of data used for training,
- use augmentation,
- increase the detail level parameter.

Segmenting instances

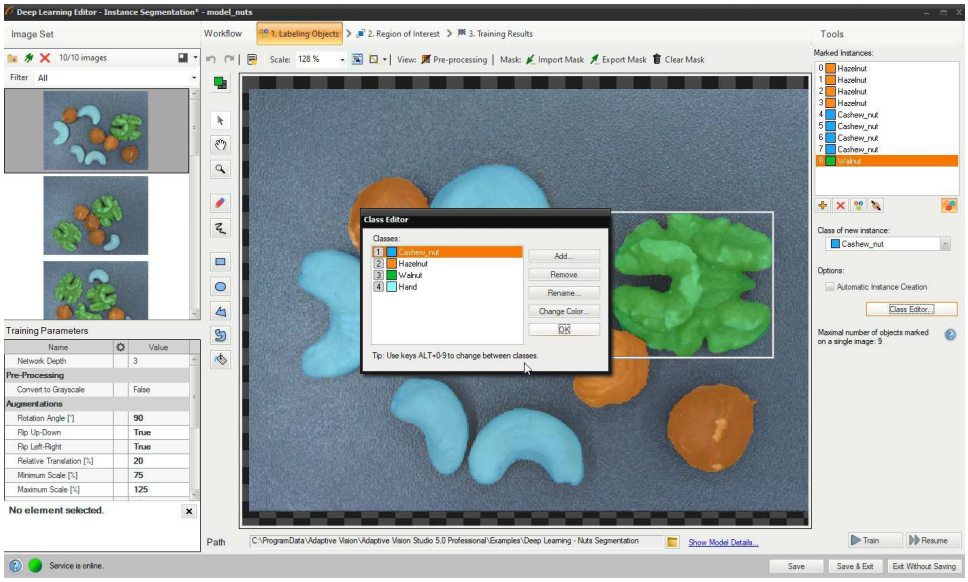
In this tool a user needs to draw regions (masks) corresponding to the objects in the scene and specify their classes. These images and masks are used to train a model which then in turn is used to locate, segment and classify objects in the input images.

1. Defining object classes

First, a user needs to define classes of objects that the model will be trained on and that later it will be used to detect. Instance segmentation model can deal with single as well as multiple classes of objects.

Class editor is available under the Class Editor button.

To manage classes, Add, Remove or Rename buttons can be used. To customize appearance, color of each class can be changed using the Change Color button.



Using Class Editor.

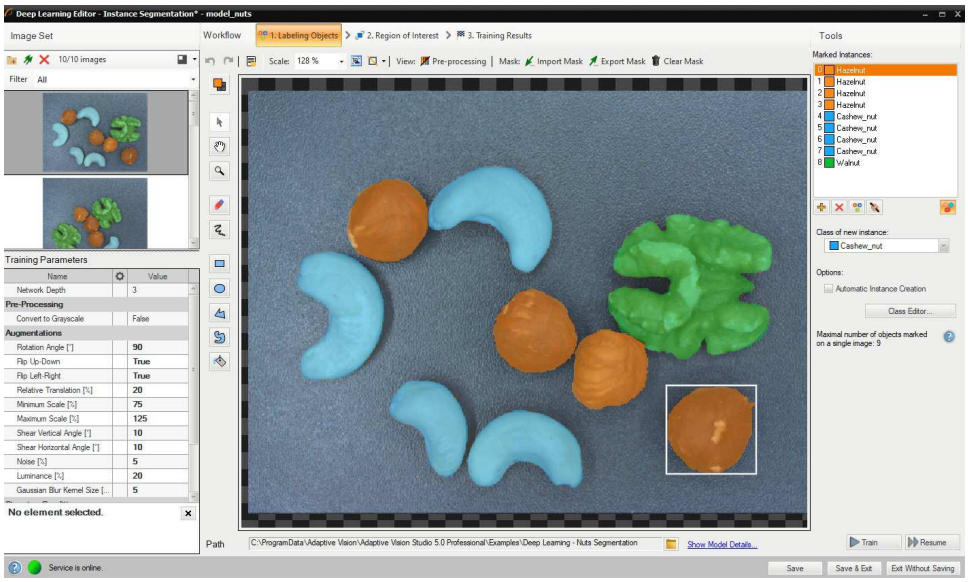
2. Labeling objects

After adding training images and defining classes a user needs to draw regions (masks) to mark objects in images.

To mark an object the user needs to select a proper class in the Current Class drop-down menu and click the Add Instance button (green plus). Alternatively, for convenience of labeling, it is possible to apply **Automatic Instance Creation** which allows a user to draw quickly masks on multiple objects in the image without having to add a new instance every time.

Use drawing tool to mark objects on the input images. Multiple tools such as brush and shapes can be used to draw object masks. Masks are the same color as previously defined for the selected classes.

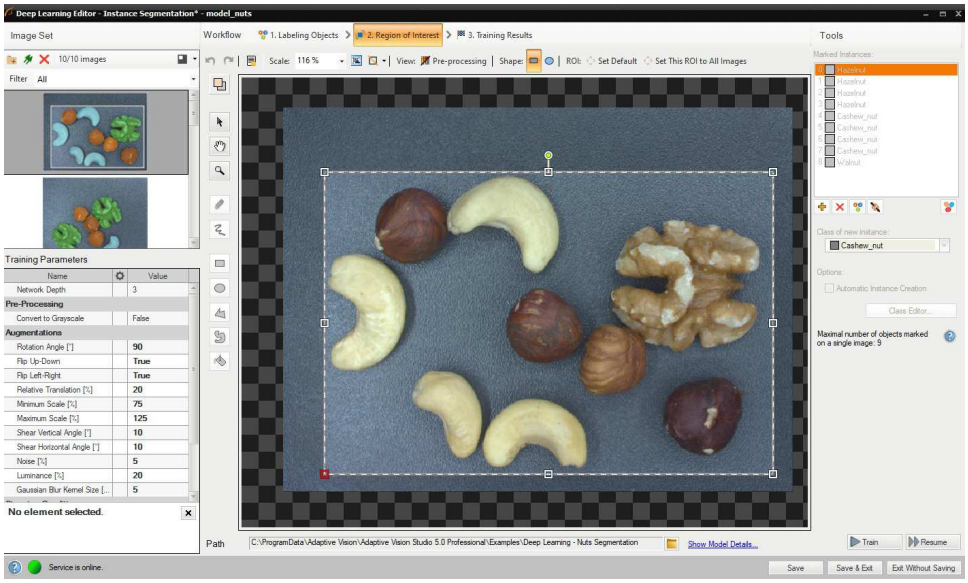
The Marked Instances list in the top left corner displays a list of defined objects for the current image. If an object does not have a corresponding mask created in the image, it is marked as "(empty)". When an object is selected, a bounding box is displayed around its mask in the drawing area. Selected object can be modified in terms of a class (Change Class button) as well as a mask (by simply drawing new parts or erasing existing ones). Remove Instance button (red cross) allows to completely remove a selected object.



Labeling objects.

3. Reducing region of interest

Reduce region of interest to focus only on the important part of the image. By default region of interest contains the whole image.



Changing region of interest.

4. Setting training parameters

- **Network depth** – predefined network architecture parameter. For more complex problems higher depth might be necessary,
- **Stopping conditions** – define when the training process should stop.

For more details read Deep Learning – Setting parameters.

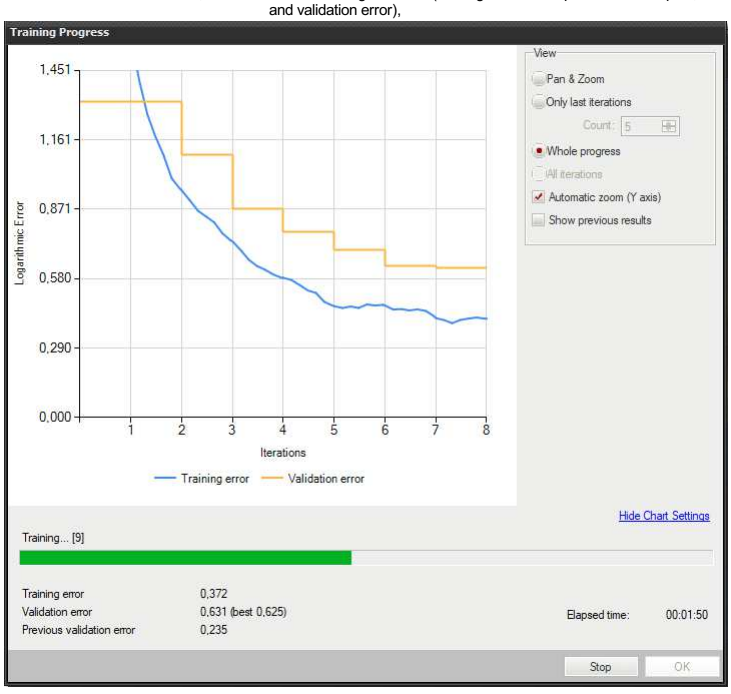
Details regarding augmentation parameters. Deep Learning – Augmentation

5. Performing training

During training two main series are visible: training error and validation error. Both charts should have a similar pattern. If a training was run before the third series with previous validation error is also displayed.

More detailed information is displayed below the chart:

- current iteration number,
- current training statistics (training and validation error),
- number of processed samples,
- elapsed time.



Training instance segmentation model.

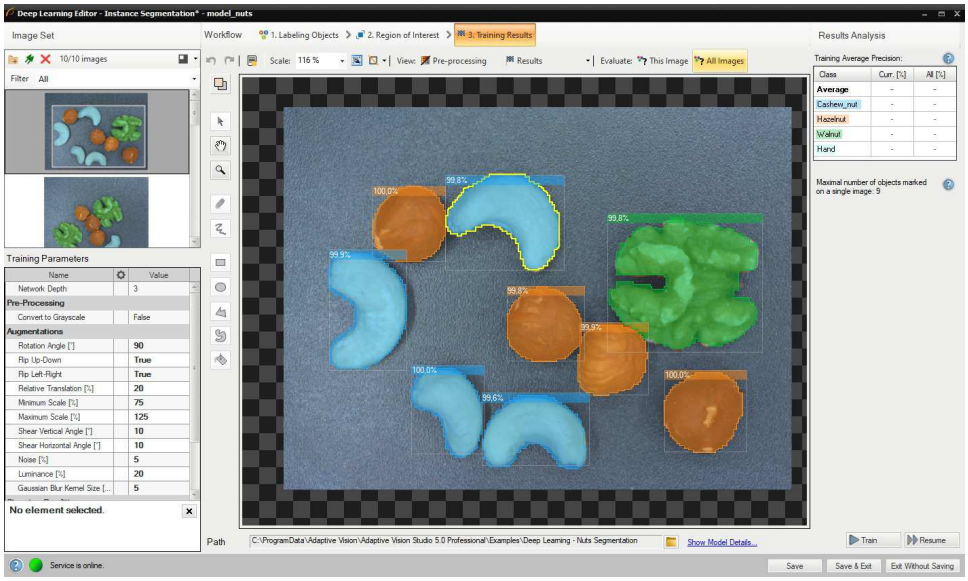
Training may be a long process. During this time, training can be stopped. If no model is present (first training attempt) model with best validation accuracy will be saved. Consecutive training attempts will prompt a user whether to replace the old model.

6. Analyzing results

The window shows the results of instance segmentation. Detected objects are displayed on top of the images. Each detection consists of following data:

- class (identified by a color),
- bounding box,
- model-generated instance mask,
- confidence score.

Evaluate: This Image and **Evaluate: All Images** buttons can be used to perform instance segmentation on the provided images. It can be useful after adding new images to the data set or after changing the area of interest.



Instance segmentation results visualized after the training.

Instance segmentation is a complex task therefore it is highly recommended to use data augmentations to improve network's ability to generalize learned information. If results are still not satisfactory the following standard methods can be used to improve model performance:

- providing more training data,
- increasing number of training iterations,
- increasing the network depth.

Locating points

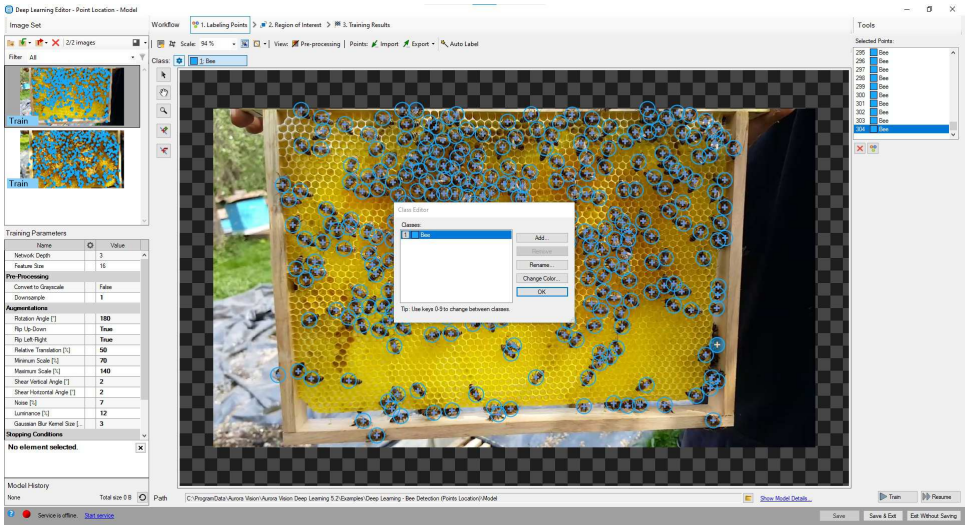
In this tool the user defines classes and marks key points in the image. This data is used to train a model which then is used to locate and classify key points in images.

1. Defining classes

First, a user needs to define classes of key points that the model will be trained on and later used to detect. Point location model can deal with single as well as multiple classes of key points.

Class editor is available under the Class Editor button.

To manage classes, Add, Remove or Rename buttons can be used. Color of each class can be changed using Change Color button.



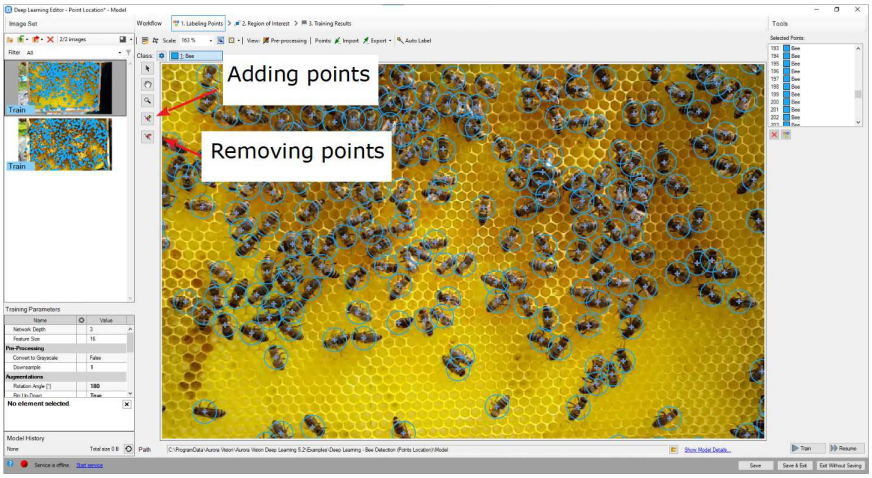
Using Class Editor.

2. Marking key points

After adding training images and defining classes a user needs to mark points in images.

To mark an object a user needs to select a proper class in the Current Class drop-down menu and click the Add Point button. Points have the same color as previously defined for the selected class.

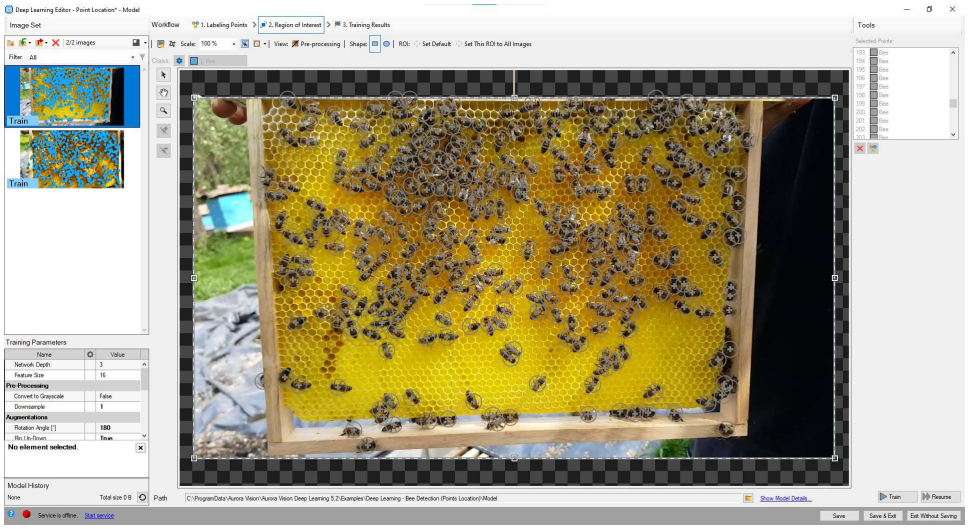
The Selected Points list in the top left corner displays a list of defined points for the current image. A point can be selected either from the list of directly on the image area. A selected point can be moved, removed (Remove Point button) or has its class changed (Change Class button).



Marking points.

3. Reducing region of interest

Reduce region of interest to focus only on the important part of the image and to speed up the training process. By default region of interest contains the whole image.



Changing region of interest.

4. Setting training parameters

- **Network depth** – predefined network architecture parameter. For more complex problems higher depth might be necessary.
- **Feature size** – the size of an small object or of a characteristic part. If images contain objects of different scales, it is recommended to use feature size slightly larger than the average object size, although it may require experimenting with different values to achieve optimal results.
- **Stopping conditions** – define when the training process should stop.

For more details read Deep Learning – Setting parameters.

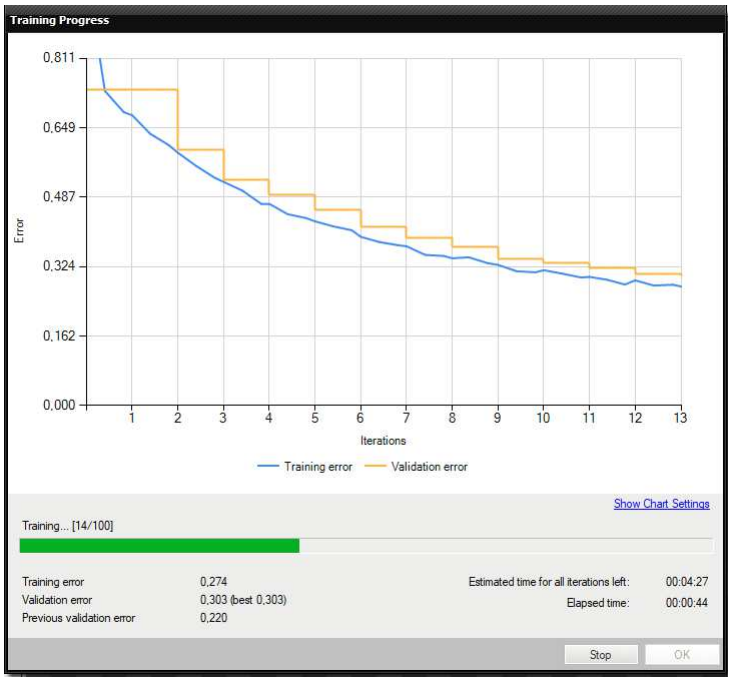
Details regarding augmentation parameters. Deep Learning – Augmentation

5. Performing training

During training, two main series are visible: training error and validation error. Both charts should have a similar pattern. If a training was run before the third series with previous validation error is also displayed.

More detailed information is displayed below the chart:

- current iteration number,
- current training statistics (training and validation error),
- number of processed samples,
- elapsed time.



Training point location model.

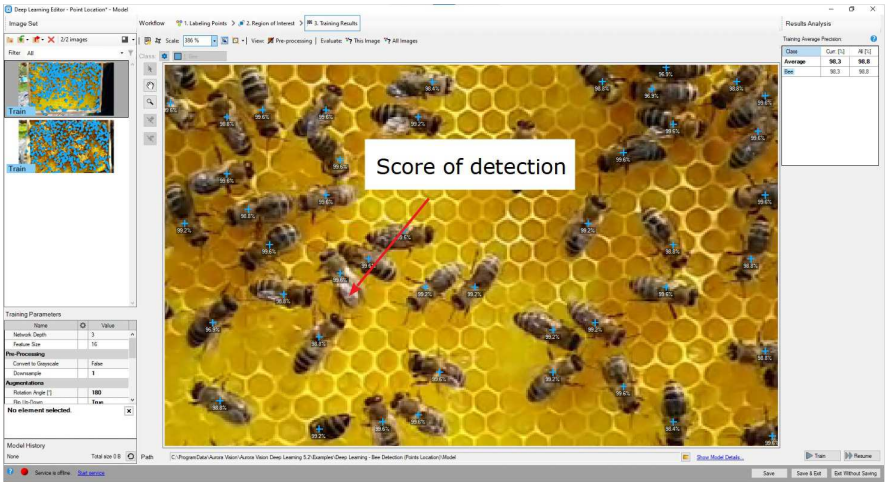
Training may be a long process. During this time, training can be stopped. If no model is present (first training attempt) model with best validation accuracy will be saved. Consecutive training attempts will prompt user whether to replace the old model.

6. Analyzing results

The window shows results of point location. Detected points are displayed on top of the images. Each detection consists of following data:

- visualized point coordinates,
- class (identified by a color),
- confidence score.

Evaluate: This Image and **Evaluate: All Images** buttons can be used to perform point location on the provided images. It may be useful after adding new training or test images to the data set or after changing the area of interest.



Point location results visualized after the training.

It is highly recommended to use data augmentations (appropriate to the task) to improve network's ability to generalize learned information. If results are still not satisfactory the following standard methods can be used to improve model performance:

- changing the feature size,
- providing more training data,
- increasing number of training iterations,
- increasing the network depth.

Locating objects

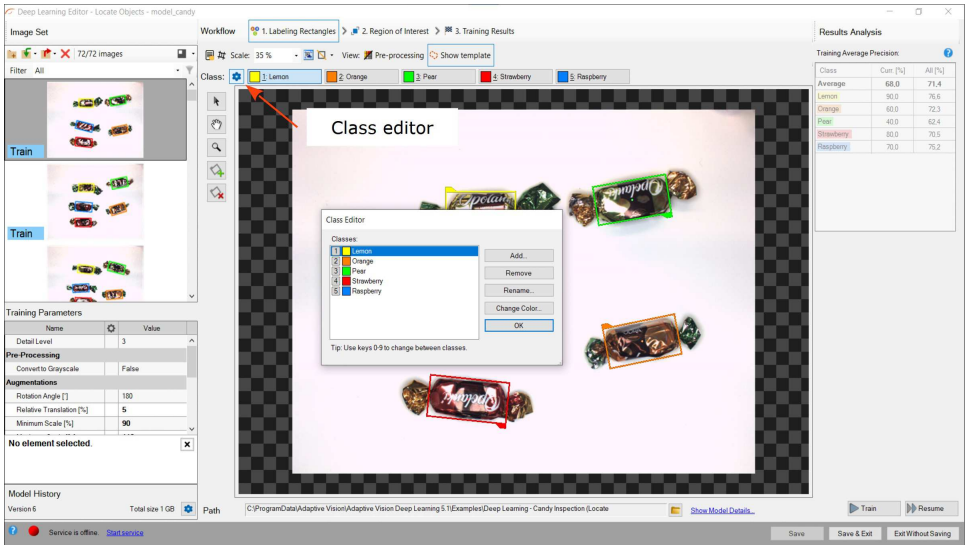
In this tool a user needs to draw rectangles bounding the objects in the scene and specify their classes. These images and rectangles are used to train a model to locate and classify objects in the input images. This tool doesn't require from a user to mark the objects as precisely as it is required for segmenting instances.

1. Defining classes

First, a user needs to define classes of objects that the model will be trained on and later used to detect. Object location model can deal with single as well as multiple classes of objects.

Class editor is available under the Class Editor button.

To manage classes, Add, Remove or Rename buttons can be used. Color of each class can be changed using Change Color button.



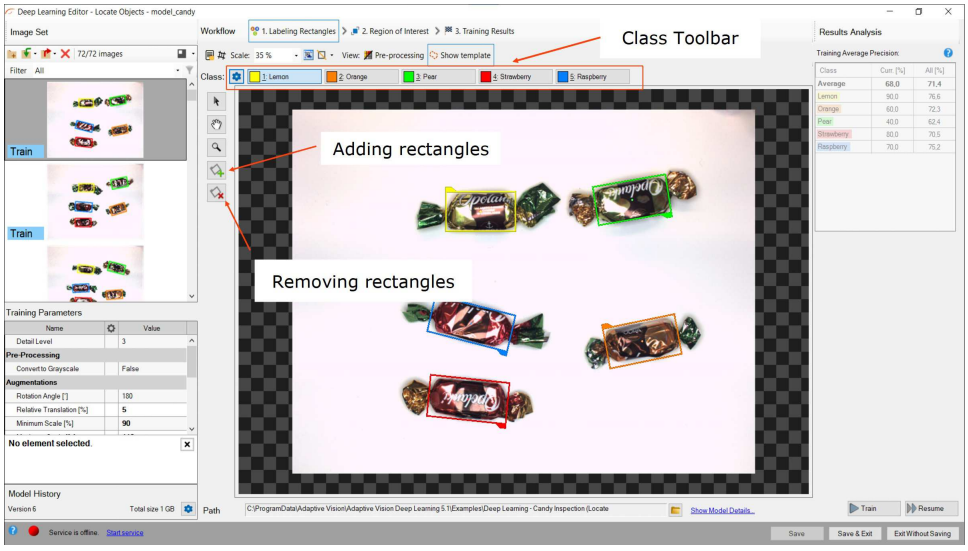
Using Class Editor.

2. Marking bounding rectangles

After adding training images and defining classes a user needs to mark rectangles in images.

To mark an object a user needs to click on a proper class from the Class Toolbar and click the Creating Rectangle button. Rectangles have the same color as previously defined for the selected class.

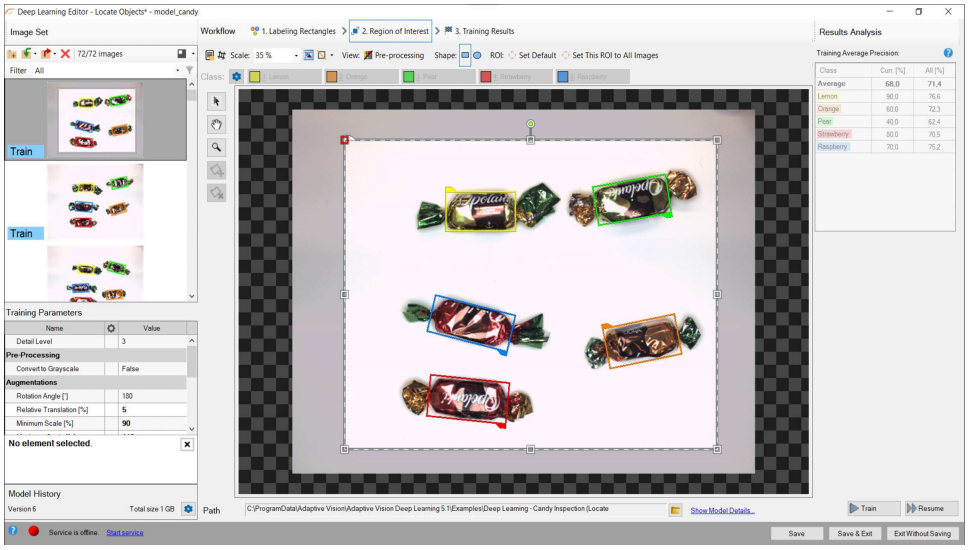
A rectangle can be selected directly on the image area. A selected rectangle can be moved, rotated and resized to fit it to the object, removed (Remove Region button) or has its class changed (*Right click on the rectangle » Change Class button*).



Marking rectangles.

3. Reducing region of interest

Reduce region of interest to focus only on the important part of the image and to speed up the training process. By default region of interest contains the whole image.



Changing region of interest.

4. Setting training parameters

- **Detail level** – level of detail
- **Stopping conditions** – define

For more details read Deep Learning – Setting parameters.

needed for a particular classification task. For majority of cases the default value of 3 is appropriate, but if images of different classes are distinguishable only by small features, increasing value of this parameter may improve classification results.

when the training process should stop.

Details regarding augmentation parameters. Deep Learning – Augmentation

5. Performing training

During training, two main series are visible: training error and validation error. Both charts should have a similar pattern. If a training was run before the third series with previous validation error is also displayed.

More detailed information is displayed below the chart:

- current iteration number,
- current training statistics (training and validation error),
- number of processed samples,
- elapsed time.

Training object location model.

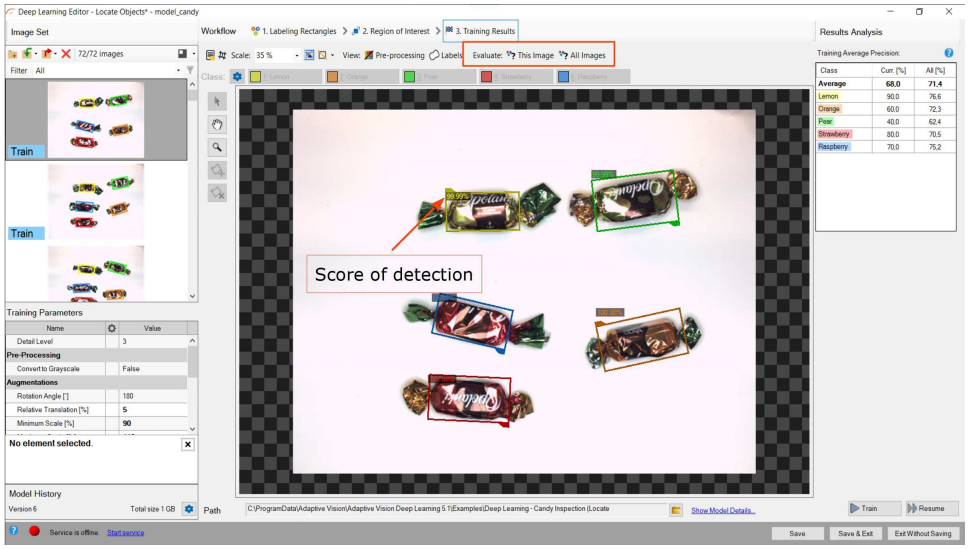
Training may be a long process. During this time, training can be stopped. If no model is present (first training attempt) model with best validation accuracy will be saved. Consecutive training attempts will prompt user whether to replace the old model.

6. Analyzing results

The window shows results of point location. Detected points are displayed on top of the images. Each detection consists of following data:

- visualized rectangle (object) coordinates,
- class (identified by a color),
- confidence score.

Evaluate: This Image and **Evaluate: All Images** buttons can be used to perform object location on the provided images. It may be useful after adding new training or test images to the data set or after changing the area of interest.



Object location results visualized after the training.

It is highly recommended to use data augmentations (appropriate to the task) to improve network's ability to generalize learned information. If results are still not satisfactory the following standard methods can be used to improve model performance:

- changing the detail level,
- providing more training data,
- increasing number of training iterations or extending the duration of the training.

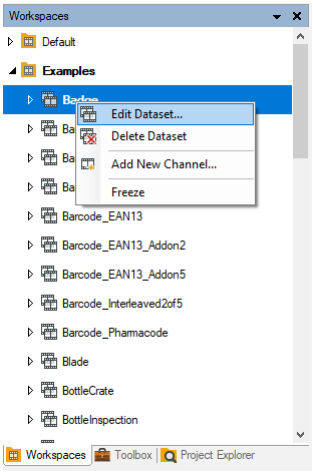
See also:

- Machine Vision Guide: Deep Learning – Deep Learning technique overview,
- Deep Learning Installation – installation and configuration of Aurora Vision Deep Learning.

Managing Workspaces

Workspace window enables user a convenient way to store datasets grouped by this same category or purpose. For example single workspace may be created for a single project like "Part Inspection" and each included dataset may represent

inspection from different day or images of the different part types.



Example Workspace window with examples dataset.

Using Filmstrip Control

1. Worker Tasks
2. HMI Events
3. New, powerful formulas
4. Program Editor Sections and Minimal View
5. Results control
6. Module encryption
7. Elements of the User Interface
8. Program Display
9. The Basic Workflow
10. Data
11. Filters (Tools)
12. Connections
13. Macrofilters
14. Sections
15. Executing Programs
16. Results Control
17. Browsing Macrofilters
18. Execution Breakpoints
19. Knowing Where You Are
20. Toolbox
21. Setting Basic Properties
22. Editing Geometrical Primitives
23. Testing Parameters in Real Time
24. Linking or Loading Data From a File
25. Labeling Connections
26. Invalid Connections
27. Property Outputs
    1. Additional Property Outputs
28. Expanded Input Structures
29. Comment Blocks
30. Extracting Macrofilters (The Quick Way)
31. Creating Macrofilters in the Project Explorer
32. Trick: Configuration File as a Module Not Exported to AVEXE
33. Macrofilter Counter
34. Global Parameters
35. Modules
36. Importing Modules
37. Locking Modules
38. Table of Contents

- Introduction
- Workflow
- Detecting anomalies 1

1. 4. Setting training parameters

- Detecting anomalies 2 (classificational approach)
- Detecting features (segmentation)
- Classifying objects
- Segmenting instances
- Locating points
- Locating objects

1. Overview
2. Common Tasks
3. See Also

- Introduction

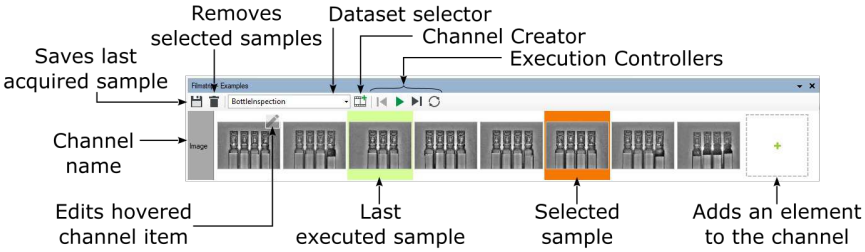
1. Prerequisites
2. User Filter Libraries Location
3. Adding New Global User Filter Libraries
4. Adding New Local User Filter Libraries

- 5. Instance Segmentation
- 6. Point Location
- 7. Locating objects
- 8. Reading Characters
- 9. Troubleshooting

1. 1. Installation guide
2. 2. Aurora Vision Deep Learning Library and Filters
3. 3. Aurora Vision Deep Learning Service
4. 4. Aurora Vision Deep Learning Examples
5. 5. Aurora Vision Deep Learning Standalone Editor
6. 6. Logging
7. 7. Troubleshooting
8. References

Overview

Filmstrip control is a powerful tool for controlling the execution of the program in the Offline Mode where the Filmstrip data is accessed with the ReadFilmstrip filters and through the bound outputs of the Online-Only filters.

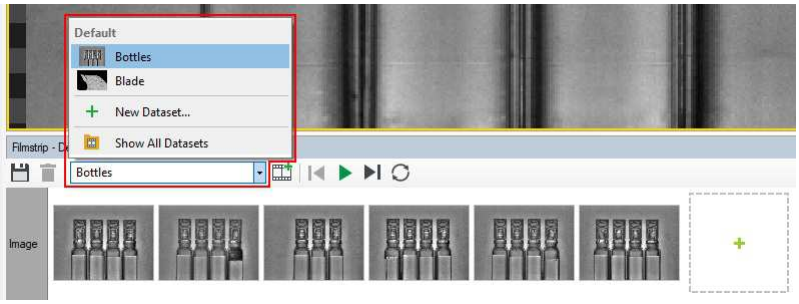


Filmstrip window with a one-channel dataset.

The data presented in the Filmstrip control is arranged in the grid layout, where the rows represents the Channels and the columns represents the Samples.

Additionally, the control enables most common operations over the current workspace, i.e., adding Datasets and Channels.

Changing the current dataset is as easy as selecting one from the datasets combo box:



Dataset selection.

To keep the program consistent, the to be selected dataset must contain channels with the same names as channels bound in the current Worker Task.

Common Tasks

- Dragging a channel from the Filmstrip control onto the Program Editor empty area inserts the ReadFilmstrip filter assigned to that channel,
- Dragging a channel from the Filmstrip control onto the filter instance output binds the output with that channel, as long as:
  - The filter is the Online-Only filter,
  - The filter is in the ACQUIRE section of the Worker Task,
  - The output data type is the same as the channel data type.
- Dragging files onto the Filmstrip control empty area creates a new Channel with the dragged files included.
- Double-click on the Filmstrip sample executes one program iteration with the clicked sample. Requirements:
  - The Offline mode is active
  - There is at least one channel assigned in the Single-Threaded application's Worker Task or in the Multi-Threaded application's Primary Worker Task
- Dragging files onto the existing channel within the Filmstrip control appends the dragged data to that channel, if only the dragged data type match the channel type.

See Also

1. Managing Workspaces - extensive description how to manage dataset workspaces in Aurora Vision Studio.
2. Offline Mode - the mode that enables access to the Channel data items.

4. Extensibility

Table of content:

Creating User Filters

Note 1: Creating user filters requires C/C++ programming skills.

Note 2: With your own C/C++ code you can easily crash the entire application. We are not able to protect against that.

Table of Contents

Introduction

User filters are written in C/C++ and allow the advanced users to extend capabilities of Aurora Vision Studio with virtually no constraints. They can be used to support a new camera model, to communicate with external devices, to add application-specific image processing operations and more.

Prerequisites

To create a user filter you will need:

- an installed Microsoft Visual Studio 2015/2017/2019 for C++, Express Edition (free) or any higher edition,
- the environment variable `AVS_PROFESSIONAL_SDK5_3` in your system (depending on the edition; a proper value of the variable is set during the installation of Aurora Vision Studio),
- C/C++ programming skills.

User filters are grouped in user filter libraries. Every user filter library is a single .dll file built using Microsoft Visual Studio. It can contain one or more filters that can be used in programs developed with Aurora Vision Studio.

User Filter Libraries Location

There are two types of user filter libraries:

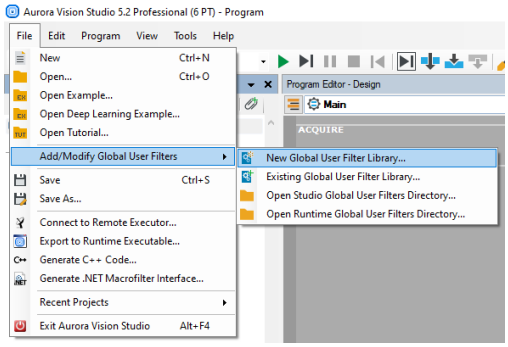
- **Global** – once created or imported to Aurora Vision Studio they can be used in all projects. The filters from such libraries are visible in the Libraries tab of the Toolbox.
- **Local** – belong to specific projects of Aurora Vision Studio. The filters from such libraries are visible only in the project that the library has been added to.

A solution (.sln file) of a global user filter library can be located in any location on your hard disk, but the default and recommended location is `Documents\Aurora Vision Studio 5.3 Professional\Sources\UserFilters` (the exact path can vary depending on the version of Aurora Vision Studio). The output .dll file built using Microsoft Visual Studio and containing global user filters has to be located in `Documents\Aurora Vision Studio 5.3 Professional\Filters\x64` (this time the exact path depends on the version and the edition) and this path is set in the initial settings of the generated Microsoft Visual Studio project. For global user filter libraries, this path must not be changed because Aurora Vision Studio monitors this directory for changes of the .dll files. The Global User Filter .dll file for Aurora Vision Executor has to be located in `Documents\Aurora Vision Studio 5.3 Runtime\Filters\x64` (again, the exact path depends on the version and edition). For 32 bit edition the last subdirectory should be changed from *x64* to *Win32*. The Local User Filter .dll file for Aurora Vision Executor has to be located in path configured in the User Filter properties. You can modify this path by editing user filters library properties in Project Explorer.

A local user filter library is a part of the project developed with Aurora Vision Studio and both source and output .dll files can be located anywhere on the hard drive. Use the Project Explorer task panel to check or modify paths to the output .dll and Microsoft Visual Studio solution files of the user filter library. The changes of the output .dll file are monitored by Aurora Vision Studio irrespectively of the file location. It's a good practice to keep the local user filter library sources (and the output .dll) relatively to the location of the developed project, for example in a subdirectory of the project.

Adding New Global User Filter Libraries

To add a new user filter library, start with *File » Add/Modify Global User Filters » New Global User Filter Library...*.



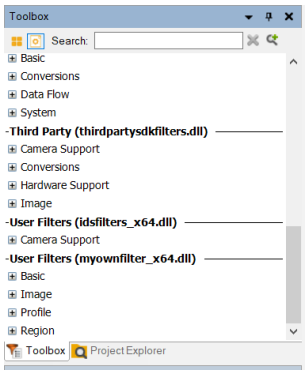
The other option is to use *Create New Local User Filter Library* button from Project Explorer panel.

A dialog box will appear where you should choose:

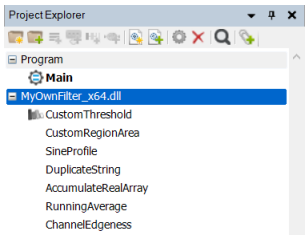
- name for the new library,
- type of the library: local (available in current project only) or global (available in all projects),
- location of the solution directory,
- version of Microsoft Visual Studio (2015, 2017 or 2019),
- whether Microsoft Visual Studio should be opened automatically,
- whether the code of example user filters should be added to the solution - good idea for users with less experience with user filters programming.

If you choose Microsoft Visual Studio to be opened, you can build this solution instantly. A new library of filters will be created and after a few seconds loaded to Aurora Vision Studio. Switch back to Aurora Vision Studio to see the new filters in:

- Appropriate categories of Libraries tab (global user filters, category in Libraries tab is based on the category set in filter code)



- Project Explorer (local user filters)

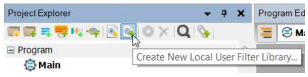


You can work simultaneously in both Microsoft Visual Studio and Aurora Vision Studio. Any time the C++ code is built, the filters will get reloaded. Just re-run your program in Aurora Vision Studio to see what has changed.

If you do not see your filters in the above-mentioned locations, make sure that they have been compiled correctly in an architecture compatible with your Aurora Vision Studio architecture: x86 (Win32) or x64.

Adding New Local User Filter Libraries

To add a new local user filter use the "Create New Local User Filter Library.." button in the Project Explorer panel as on the image below:



Developing User Filters

User Filter Project Configuration

User filter project files (.sln and .vcproj) are generated by Aurora Vision Studio during adding new user filter library.

The settings of user filter project are gathered in *.props* file available in *props* subdirectory of the Aurora Vision Studio SDK (environment variable `AVS_PROFESSIONAL_SDK5_3`), typically `C:\Program Files\Aurora Vision\Aurora Vision Studio 5.3 Professional\SDK\props`.

If you want to configure the existing project to be a valid user filter library, please use the proper *.props* file (file with *v140* suffix is dedicated for Microsoft Visual Studio 2015, *v141* for 2017 and *v142* for 2019).

Basic User Filter Example

Example below shows whole source code for basic user filter. In this example filter is making a simple threshold operation on an 8-bit image.

```
#include "UserFilter.h"
#include "AVL_Lite.h"

#include "UserFilterLibrary.hxx"

namespace avs
{
 // Example image processing filter
 class CustomThreshold : public UserFilter
 {
 private:
  // Non-trivial outputs must be defined as a filed to retain data after filter execution.
  avl::Image outImage;

 public:
  // Defines the inputs, the outputs and the filter metadata
  void Define() override
  {
   SetName  (L"CustomThreshold");
   SetCategory (L"Image::Image Thresholding");
   SetImage (L"CustomThreshold_16.png");
   SetImageBig (L"CustomThreshold_48.png");
   SetTip  (L"Binarizes 8-bit images");

   //      Name      Type    Default  Tool-tip
   AddInput   (L"inImage",    L"Image",   L"",  L"Input image"    );
   AddInput   (L"inThreshold",   L"Integer<0, 255>", L"128",  L"Threshold value");
   AddOutput   (L"outImage",    L"Image",      L"Output image"   );
  }

  // Computes output from input data
  int Invoke() override
  {
   // Get data from the inputs
   avl::Image inImage;
   int inThreshold;

   ReadInput(L"inImage", inImage);
   ReadInput(L"inThreshold", inThreshold);

   if (inImage.Type() != avl::PlainType::UInt8)
    throw atl::DomainError("Only uint8 pixel type are supported.");

   // Get image properties
   int height = inImage.Height();

   // Prepare output image in this same format as input
   outImage.Reset(inImage, atl::NIL);

   // Enumerate each row
   for (int y = 0; y < height; ++y)
   {
    // Get row pointers
    const atl::uint8* p = inImage.RowBegin<atl::uint8>(y);
    const atl::uint8* e = inImage.RowEnd<atl::uint8>(y);
    atl::uint8*       q = outImage.RowBegin<atl::uint8>(y);

    // Loop over the pixel components
    while (p < e)
    {
     (*q++) = (*p++) < inThreshold ? 0 : 255;
    }
   }

   // Set output data
   WriteOutput(L"outImage", outImage);

   // Continue program
   return INVOKE_NORMAL;
  }
 };

 // Builds the filter factory
 class RegisterUserObjects
 {
 public:
  RegisterUserObjects()
  {
   // Remember to register every filter exported by the user filter library
   RegisterFilter(CreateInstance<CustomThreshold>);
  }
 };

 static RegisterUserObjects registerUserObjects;
}
```

Structure of User Filter Class

A user filter is a class derived from the *UserFilter* class defined in *UserFilter.h* header file. When creating a filter without state you have to override two methods:

- **Define** – defines the interface of the filter, including its name, category, inputs and outputs.
- **Invoke** – defines the routine that transforms inputs into outputs.

When creating *a filter with state* (storing information from previous invocations) the class is going to have some data fields and two additional methods have to be overridden:

- **Init** – initializes the state variables, invoked at the beginning of a Task parenting the instance of the filter, may be invoked multiple times during filter instance lifetime. Always remember to invoke base type **Invoke()** method.
- **Stop** – deinitializes the state variables, including releasing of external and I/O resources (like file handles or network connections), may not affect data on filter outputs, invoked at the end of a Task parenting the filter instance (to pair every **Init** call).
- **Release** – releases output variables memory, invoked at the end of a Task macrofilters marked to release memory.

When a user filter class is created it has to be registered. This is done in the *RegisterUserObjects* function which is defined at the bottom of the sample user filters' code. You do not need to call it manually, it's called by Aurora Vision Studio while loading filters from the .dll file.

Structure of *Define* Method

Use the *Define* method to set the name, category, image (used as the icon of the filter) and tooltip for your filter. All of this can be set by using proper *Set...* methods.

The *Define* method should also contain a definition of the filter's external interface, which means: inputs, outputs and diagnostic outputs. The external interface should be defined using *AddInput*, *AddOutput* and *AddDiagnosticOutput* methods. These methods allow to define name, type and tooltip for every input/output of filter. For inputs a definition of the default value is also possible.

Aurora Vision Studio uses a set of additional attributes for ports. To apply attribute on a port use *AddAttribute* method. Example:

```
AddAttribute(L"ArraySync", L"inA inB");
```

List of attributes:

Attribute Name	Description	Example	Comment
ArraySync	Defines a set of synchronized ports.	`L"inA inB"`	Informs that arrays in inA and inB require the same number of elements.
UserLevel	Defines user level access to the filter.	`L"Advanced"`	Only users with Advanced level will find this filter in Libraries tab.
UsageTips	Defines additional documentation text.	`L"Use this filter for creating a line."`	This is instruction where this filter is needed.
AllowedSingleton	Filter can accept singleton connections on input	`L"inA"`	User can connect a single value to inA which has Array type.
FilterGroup	Defines element of filter group.	`L"FilterName default ## Description for group"`	Creates a FilterName with default element VariantName. More detailed description in "Defining Filters Groups"
Tags	Defines alternative names for this filter.	`L"DrawText DrawString PutText"`	When user types "DrawText" this filter will be in result list.
CustomHelpUrl	Defines alternative URL for this filter.	`L"http:\\adaptive-vision.com"`	When user press F1 in the Program Editor alternative help page will be opened.

Defining Filters Groups

Several filters can be grouped into a single group, which can be very helpful for user to change variant of very similar operations.

To create filter group define attribute L"FilterGroup" for default filter with parameters. **L"FilterName<VariantName> default ## Description for group"**. Notice "default" word. Text after "##" defines the tooltip for whole group.

If default filter is defined you can add another filter using L"FilterGroup" with parameter **L"FilterName<NextVariant>"**

Example usage:

```
// Default filter FindCircle -> Find: Circle
AddAttribute(L"FilterGroup", L"Find default ## Finds an object on the image");
...

// Second variant FindRectangle -> Find: Rectangle
AddAttribute(L"FilterGroup", L"Find");
...

// Third variant FindPolygon -> Find: Polygon
AddAttribute(L"FilterGroup", L"Find");
...
```

As result a filter group Find will be created with three variants: Circle, Rectangle, Polygon.

Using custom user filter icons

Using methods `SetImage` and `SetImageBig` user can assign a custom icon for user filter. Filter icon must be located in this same directory as output output user filter DLL file.

There are four types of icons:

- **Small Icon** - icon with size 16x16 pixels used in Libraries tab, set by `SetImage`, name should end with "_16"
- **Medium Icon** - icon of size 24x24 pixel, created automatically from Big Icon,
- **Big Icon** - icon of size 48x48 pixel, set by `SetImageBig`, name should end with "_48",
- **Description Icon** - icon of size 72x72 used in filter selection from group, name is created by replacing "_48" from `SetImageBig` by "_D". For given `SetImageBig` as "custom_48.png" a name "custom_D.png" will be generated.

Structure of *Invoke* Method

An *Invoke* method has to contain 3 elements:

1. **Reading data from inputs**

   To read the value passed to the filter input, use the *ReadInput* method. This is a template method supporting all Aurora Vision Studio data types. *ReadInput* method returns the value (by reference) using its second parameter.

2. **Computing output data from input data**

   It is the core part. Any computations can be done here

3. **Writing data to outputs**

   Similarly to reading, there is a method *WriteOutput* that should be used to set values returned from filter on filter outputs.

   Data types that don't contain blobs (i.e. *int*, *avl::Point2D*, *avl::Rectangle2D*) can be simply returned by passing the local variable to the *WriteOutput* method. Output variables with blobs (i.e. *avl::Image*, *avl::Region*) should be declared at least in a class scope.

```
class MyOwnFilter : public UserFilter
{
 int Invoke()
 {
  int length;
  // ... computing the length value...
  WriteOutput("outLength", length);
 }

 // ...
}
```

   All non-trivial data types like Image, Region or ByteBuffer should be defined as a filter class filed.

   This solution has two benefits:

   1. Reduces performance overhead for creating new objects in each filter execution,
   2. Assures that types which contains blobs are not released after the filter execution.

   For the sake of clarity it is good habit to define all filter variables as class members.

```
class MyOwnFilter : public UserFilter
{
 private:
          // Non-trivial type data
  avl::Image image;

 int Invoke()
 {
  // ... computing image ...
  WriteOutput("outImage", image);
 }

 // ...
}
```

Invoke has to return one of the four possible values:

- **INVOKE_ERROR** - when something went wrong and program cannot be continued.
- **INVOKE_NORMAL** - when everything is OK and the filter can be invoked again.
- **INVOKE_LOOP** - when everything is OK and the filter requests more iterations.
- **INVOKE_END** - when everything is OK and the filter requests to stop the current loop.

For example the filter ReadVideo returns **INVOKE_LOOP** whenever a new frame is successfully read and **INVOKE_END** when there is the end. **INVOKE_NORMAL** is returned by filters that do not have any influence on the current loop continuation or exiting (for example ThresholdImage).

All filter outputs should be assigned by WriteOutput before filters return status. Missing assigned may result random data access in complex program structure. On INVOKE_END result filter should set up output values as last iterations of filter.

In case of error also exceptions can be thrown. User *atl::DomainError* for signaling problems connected with input data. All hardware problems should be signaled using *atl::IoError*. For more information please read Error Handling

Using Arrays

User filters can process not only single data objects, but also arrays of them. In Aurora Vision Studio, arrays are represented by data types with suffix *Array* (i.e. IntegerArray, ImageArray, RegionArrayArray). Multiple *Array* suffixes are used for multidimensional arrays. In C++ code of user filters, *atl::Array<T>* container is used for storing objects in arrays:

```
atl::Array< int > integers;
atl::Array< avl::Image > images;
atl::Array< atl::Array< avl::Region > > regions2Dim;
```

For more information about types from *atl* and *avl* namespaces, please refer the documentation of *Aurora Vision Library*.

Diagnostic Mode Execution and Diagnostic Outputs

User filters can have diagnostic outputs. Diagnostic outputs can be helpful during developing programs in Aurora Vision Studio. The main purpose of this feature is to allow the user to view diagnostic data on the Data Previews, but they can also participate in the data flow and can be connected to an input of any filter. This type of connection is called a diagnostic connection and makes the destination filter to be executed in the *Diagnostic* mode (filter will be invoked only in the *Diagnostic* mode of program execution).

When a program is executed in the *Non-Diagnostic* mode, values of the diagnostic outputs shouldn't be (for performance purposes) computed by any filter. In user filters, you should use the **IsDiagnosticMode()** method for conditional computation of the data generated by your filter for diagnostic outputs. If the method returns *True*, execution is in the *Diagnostic* mode and values of the diagnostic outputs should be computed, otherwise, the execution is in the *Non-Diagnostic* mode and your filter shouldn't compute such values.

Filter Work Cancellation

Aurora Vision Studio allows to stop execution of each filter during the time consuming computations. To use this option function **IsWorkCancelled()** can be used. If function returns value *True* the long computation should be finished because user pressed the "Stop" button.

Using Dependent DLL

User filter libraries are often created as wrappers of third party libraries, e.g. of APIs for some specific hardware. These libraries often come in the form of DLL files. For a user filter to work properly, the other DLL files must be located in an accessible disk location at runtime, or the user gets the error code 126, *The specified module could not be found*. MSDN documentation specifies possible options in the article Dynamic-Link Library Search Order. From the point of view of user filters in Aurora Vision Studio, the most typical option is the one related to changing the PATH environment variable – almost all camera manufacturers follow this way. For local user filters it is also allowed to add dependent dll in the same directory as the user filter dll directory.

Alternatively, it is possible to create a new directory for a global user filter library: `Documents\Aurora Vision Studio 5.3 Runtime\Filters\Deps_x64`, after which the user filter's dependent DLL files can be stored inside of it.

Advanced Topics

Using the Full Version of AVL

By default, user filters are based on *Aurora Vision Library Lite* library, which is a free edition of Aurora Vision Library Professional. It contains data types and basic functions from the 'full' edition of Aurora Vision Library. Please refer to the documentation of Aurora Vision Library Lite and Aurora Vision Library Professional to learn more about their features and capabilities.

If you have bought a license for the 'full' Aurora Vision Library, you can use it in user filters instead of the Lite edition. The following steps are required:

- In compiler settings of the project, add additional include directory `$(AVL_PATH5_3)\include` (*Configuration Properties* | *C/C++* | *General* | *Additional Include Directories*).
- In linker settings of the project, add new additional library directory `$(AVL_PATH5_3)\lib\$(Platform)` (*Configuration Properties* | *Linker* | *General* | *Additional Library Directories*).
- In linker settings of the project, replace `AVL_Lite.lib` additional dependency with `AVL.lib` (*Configuration Properties* | *Linker* | *Input* | *Additional Dependencies*).
- In source code file, change including `AVL_Lite.h` to `AVL.h`.

Accessing Console from User Filter

It is possible to add messages to the console of Aurora Vision Studio from within the *Invoke* method. Logging messages can be used for problems visualization, but also for debugging. To add the message, use one of the following functions:

```
bool LogInfo   (const atl::String& message);
bool LogWarning(const atl::String& message);
bool LogError  (const atl::String& message);
bool LogFatal  (const atl::String& message);
```

Generic User Filters

Generic filters are filters that do not have a strictly defined type of the data they process. Generic filters have to be concretized with a data type before they can be used. There are many generic filters provided with Aurora Vision Studio (i.e. ArraySize) and user filters can be generic as well.

To create a generic user filter, you need to define one or more ports of the user filters as generic. In the call of *AddInput* method the second parameter (data type) has to contain < *T* >. Example usage:

```
AddInput ("inArray", "Array", "", "Input array");
AddInput ("inObject", "", "", "Object of any type");
```

In the *Invoke* method of a user filter, the *GetTypeParam* function can be used to resolve the data type that the filter has been concretized with. Once the data type is known, the data can be properly processed using the *if-else* statement. Please see the example below.

```
atl::String type = GetTypeParam(); // Getting type of generic instantiation as string.
int arrayByteSize = -1;

if (type == "Integer")
{
    atl::Array< int > ints = GetInputArray< int >("inArray");
    arrayByteSize = ints.Size() * sizeof(int);
}
else if (type == "Image")
{
    atl::Array< avs::Image > images = GetInputArray< avs::Image >("inArray");
    arrayByteSize = 0;
    for (int i = 0; i <  images.Size(); ++i)
        arrayByteSize += images[i].pitch * images[i].height;
}
```

Creating User Types in User Filters

When creating a User Filter add to the project an AVTYPE file with a user types description. The file should contain type descriptions in a format the same like the one used for creating User Types in a program. See Creating User Types. Sample user type description file:

```
enum PartType
{
 Nut
 Bolt
 Screw
 Hook
 Fastener
}

struct Part
{
 String  Name
 Real Width
 Real Height
 Real Tolerance
}
```

In your C++ code declare structures/enums with the same field types, names and order. If you create an enum then you can start using this type in your project instantly. For structures you must provide *ReadData* and *WriteData* functions overrides for serialization and deserialization.

In these functions you should serialize/deserialize all fields of your structure in the same order you declared them in the type definition file.

To support structure *Part* from the previous example in your source code you should add:

Structure declaration:

```
struct Part
{
 atl::String  Name;
 float   Width;
 float   Height;
 float   Tolerance;
};
```

Structure deserialization function:

```
void ReadData(atl::BinaryReader& reader, Part& outPart)
{
 ReadData(reader, outPart.Name);
 ReadData(reader, outPart.Width);
 ReadData(reader, outPart.Height);
 ReadData(reader, outPart.Tolerance);
}
```

Structure serialization function:

```
void WriteData(atl::BinaryWriter& writer, const Part& inValue)
{
 WriteData(writer, inValue.Name);
 WriteData(writer, inValue.Width);
 WriteData(writer, inValue.Height);
 WriteData(writer, inValue.Tolerance);
}
```

Enum declaration:

```
enum PartType
{
 Nut,
 Bolt,
 Screw,
 Hook,
 Fastener
};
```

It is not required for custom serialization / deserialization of enum types.

The file with user type definitions has to be registered. This is done in the *RegisterUserObjects* class constructor which is defined at the bottom of the user filter code. You need to add there a registration of your file as *RegisterTypeDefinitionFile("fileName.avtype")*. The file name is a path to your type definitions file. The path should be absolute or relative to the User Filter dll file.

You can use types defined in a User Filter library in this User Filters library as well as in all other modules of the project. If you want to use the same type in multiple User Filters libraries then you should declare these types in each User Filters library.

The following Example Program: "User Filter With User Defined Types" demonstrates usage of User Types in User Filters.

Troubleshooting and Examples

Upgrading User Filters to Newer Versions of Aurora Vision Studio

When upgrading project with User Filters to more recent version of Aurora Vision you should manually edit the User Filter vcxproj file in your favorite text editor e.g. notepad. Make sure to close the solution file in Microsoft Visual Studio before performing the modifications. In this file you should change all occurrences of *AVS_PROFESSIONAL_SDKxx* (where is xx is your current version of Aurora Vision) to `AVS_PROFESSIONAL_SDK5_3`, save your changes and rebuild the project.

After successful build you can use your User Filter library in the new version of Aurora Vision.

During compilation you can receive some errors if you use in your code function which has changed its interface. In such case, please refer the documentation and release notes to find out how the function was changed in the current version.

Remarks

- If you get problems with PDB files being locked, kill the *mspdbsrv.exe* process using Windows Task Manager. It is a known issue in Microsoft Visual Studio. You can also switch to use the *Release* configuration instead.
- User filters can be debugged. See Debugging User Filters.
- A user filter library (in .dll file) that has been built using SDK from one version of Aurora Vision Studio is not always compatible with other versions. If you want to use the user filter library with a different version, it may be required to rebuild the library.
- If you use Aurora Vision Library ('full' edition) in user filters, Aurora Vision Library and Aurora Vision Studio should be in the same version.
- A solution of a user filter library can be generated with example filters. If you're a beginner in writing your own filters, it's probably a good idea to study these examples.
- Only compiling your library in the release configuration lets you use it on other computer units. You cannot do so if you use a debug configuration.

Example: Image Acquisition from IDS Cameras

One of the most common uses of user filters is for communication with hardware, which does not (fully) support the standard GenICam industrial interface. Aurora Vision Studio comes with a ready example of such a user filter – for image acquisition from cameras manufactured by the IDS company. You can use this example as a reference when implementing support for your specific hardware.

The source code is located in the directory:

```
%PUBLIC%\Documents\Aurora Vision Studio 5.3 Professional\Sources\UserFilters\IDS
```

Here is a list of the most important classes in the code:

- **CameraManager** – a singleton managing all connections with the IDS device drivers.
- **IDSCamera** – a manager of a single image acquisition stream. It will be shared by multiple filters connected to the same device.
- **IDS_BaseClass** – a common base class for all user filter classes.
- **IDS_GrabImage**, **IDS_GrabImage_WithTimeout**, **IDS_StartAcquisition** – the classes of individual user filters.

The CameraManager constructor checks if an appropriate camera vendor's dll file is present in the system. The user filter project loads the library with the option of Delay-Loaded DLL turned on to correctly handle the case when the file is missing.

Requirement: To use the user filters for IDS cameras you need to install IDS Software Suite, which can be downloaded from IDS web page.

After the project is built in the appropriate Win32/x64 configuration, you will get the (global) user filters loaded to Aurora Vision Studio automatically. They will appear in the Libraries tab of the Toolbox, "User Filters" section.

Example: Using PCL library in Aurora Vision Studio

This example shows how to use PCL in an user filter.

The source code is located in the directory:

```
%PUBLIC%\Documents\Aurora Vision Studio 5.3 Professional\Sources\UserFilters\PCL
```

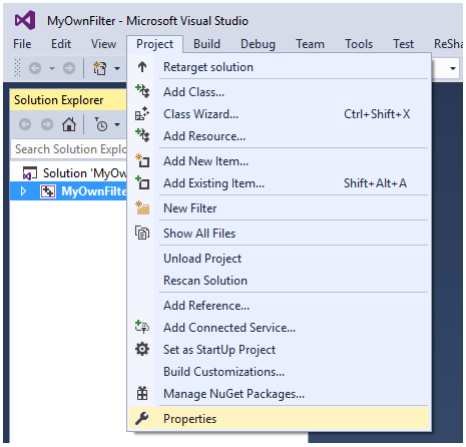
To run this example PCL Library must be installed and system PCL_ROOT must be defined.

Debugging User Filters

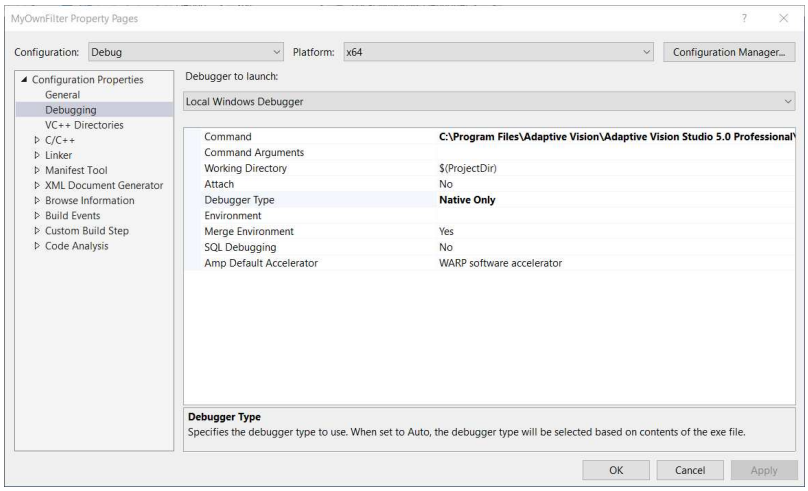
Debugging User Filters with Aurora Vision Studio Running

To debug your user filters, follow the instructions below:

1. If the Microsoft Visual Studio solution of your user filter library is not opened, open it manually. For global user filters it is typically located in *My Documents\Aurora Vision Studio Professional\Sources\LibraryName*, but can be located in any other location that you have chosen while creating the library. For local user filters, you can check the location of the solution file in the Project Explorer task pane.
2. Make sure that Aurora Vision Studio is not running.
3. Select *Debug* configuration.
4. Go to the project properties:



5. Go to *Debugging* section:

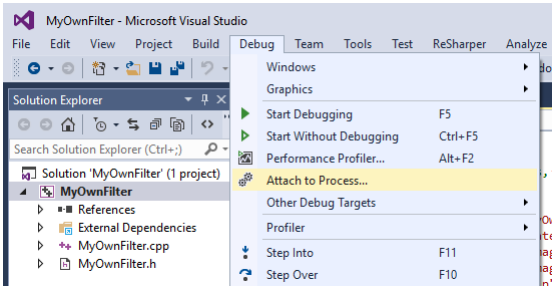


6. Set *Command* to the executable of Aurora Vision Studio.
7. Set *Debugger Type* to *Native Only*.
8. Set a breakpoint in your code.
9. Launch debugging by clicking *F5*.
10. Have your filter executed in Aurora Vision Studio. At this point it should get you into the debugging session.

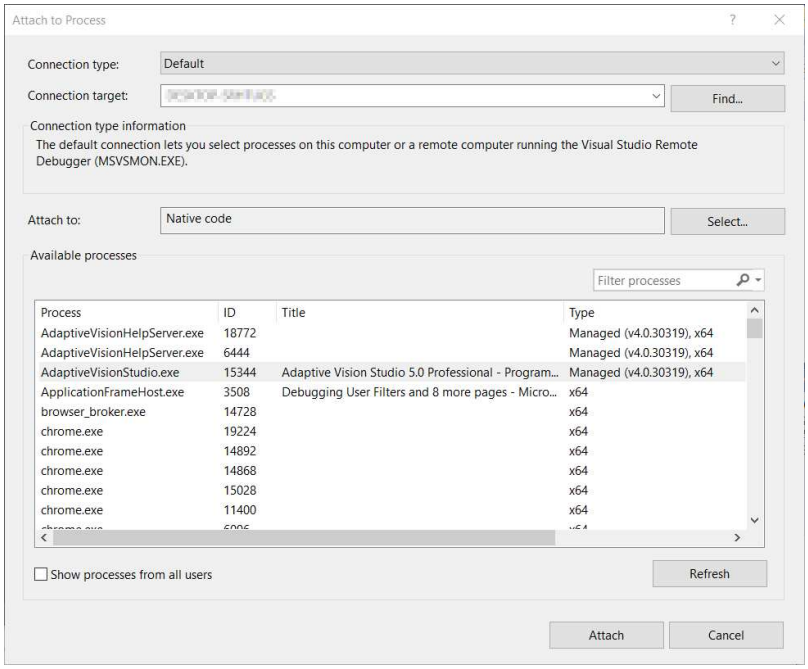
Debugging User Filters by attaching to Aurora Vision Studio process

You can attach the Microsoft Visual Studio debugger to a running process. Follow the instructions below:

1. Run Aurora Vision Studio and load your project.
2. Load solution of the User Filter in Microsoft Visual Studio.
3. From the *Debug* menu, choose *Attach to Process*.



4. In the *Attach to Process* dialog box, find AuroraVisionStudio.exe process from the *Available Processes* list.
5. In the *Attach to* box, make sure *Native code* option is selected.



6. Press *Attach* button.
7. Set a breakpoint in your code.
8. Have your filter executed in Aurora Vision Studio. At this point it should get you into the debugging session.

Debugging Tips

- User filters have access to the Console window of Aurora Vision Studio. It can be helpful during debugging user filters. To write on the Console, please use one of the functions below:
  - `bool LogInfo (const atl::String& message);`
  - `bool LogWarning (const atl::String& message);`
  - `bool LogError (const atl::String& message);`

  Functions are declared (indirectly) in the *UserFilter.h* header file that should be used in every file with user filters source code.
- To write messages on the Output window of the Microsoft Visual Studio, please use standard *OutputDebugString* function (declared in *Windows.h*).

Creating User Types

In Aurora Vision Studio it is possible for the user to create custom types of data. This can be especially useful when it is needed to pass multiple parameters conveniently throughout your application or when creating User Filters.

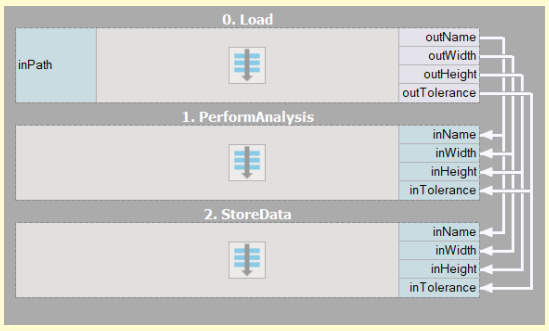
The user can define a structure with named fields of specified type as well as his own enumeration types (depicting several fixed options).

For example, the user can define a structure which contains such parameters as: *width, height, value* and *position* in a single place. Also, the user can define named program states by defining an enumeration type with options: *Start, Stop, Error, Pause,* etc.

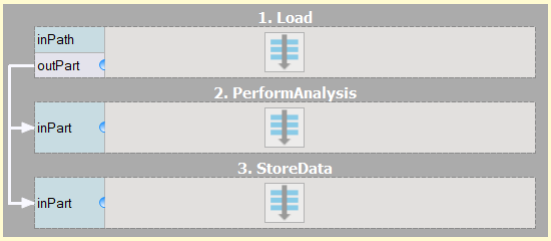
Usage

In an example project information such as: part name, part width, part height and its tolerance is needed for checking product quality. All this data elements must be accessed during image analysis.

This problem can be solved without user defined types, but creating a lot of connections can make the program structure too complex. The pictures below show a comparison between working with a user's structure and passing multiple values as separate parameters.



A solution without user types – more connections, less readable.

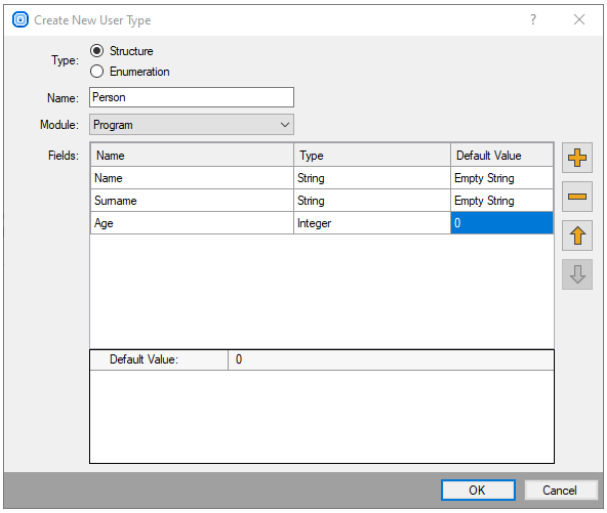


A solution with user types – fewer connections, more readable.

Creating User Types in a Program

User types are created with a graphical editor available through the Project Explorer window.

Use this icon to open the graphical editor.

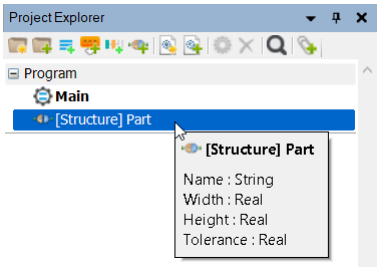


Graphical user type editor.

Alternatively, you can save your project, open the main AVCODE file (e.g. with Notepad++) and at the beginning of the file enter a type declaration:

```
struct Part
{
 String  Name
 Real Width
 Real Height
 Real Tolerance
}
```

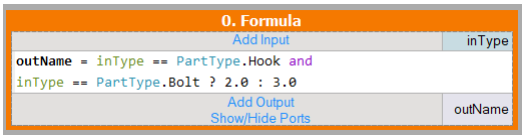
Save your file and reload the project. Now the newly created type can be used as any other type in Aurora Vision Studio.



After reloading the project the custom made type is available in Aurora Vision Studio.

Also custom enumeration types can be added this way. To create a custom enumeration type add the code below to the top of your AVCODE file.

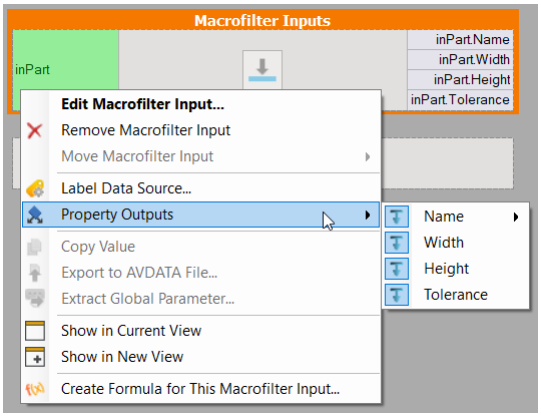
```
enum PartType
{
 Nut
 Bolt
 Screw
 Hook
 Fastener
}
```



Custom enumeration types can be used like other types.

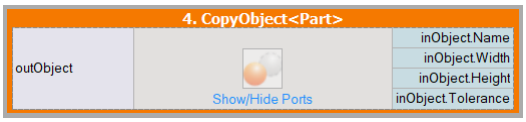
Accessing Structure Fields

To access information contained in a user structure its fields must be expanded. The picture below shows how to expand a type on an input of a macrofilter.



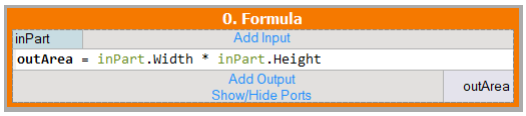
User type fields expanded on a macrofilter's inputs.

User type objects can be created with the CopyObject filter.



User type fields expanded on the CopyObject input.

User defined types can also be accessed with formulas.



Computation using the user defined type.

Saving User Types

User defined types work in Aurora Vision Studio, so filters such SaveObject, WriteToString, WriteToXmlNode or TcpIp_WriteObject can be used to store and transfer user data.

Related Program Examples

User defined types can be studied in the following Program Examples: Brick Destroy, User Defined Types, User Filter With User Defined Types.

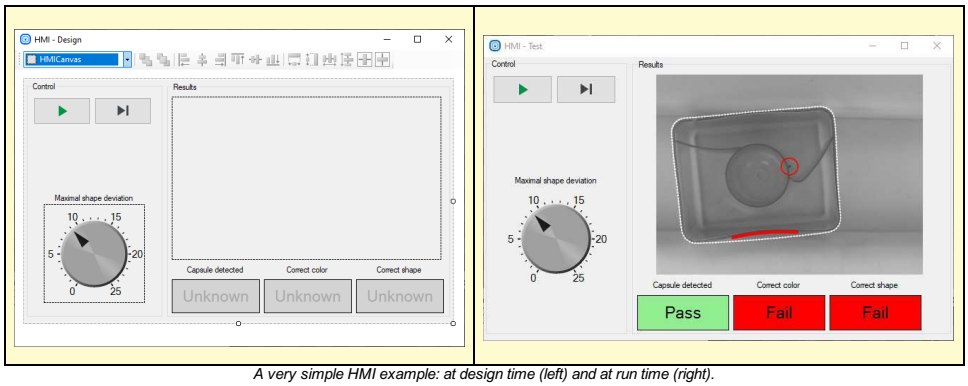
5. Human Machine Interface (HMI)

Table of content:

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.

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

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 Interface and then creating entire HMI in the C# programming language.
- Using C++ Code Generator and then creating entire HMI in the C++ programming language.

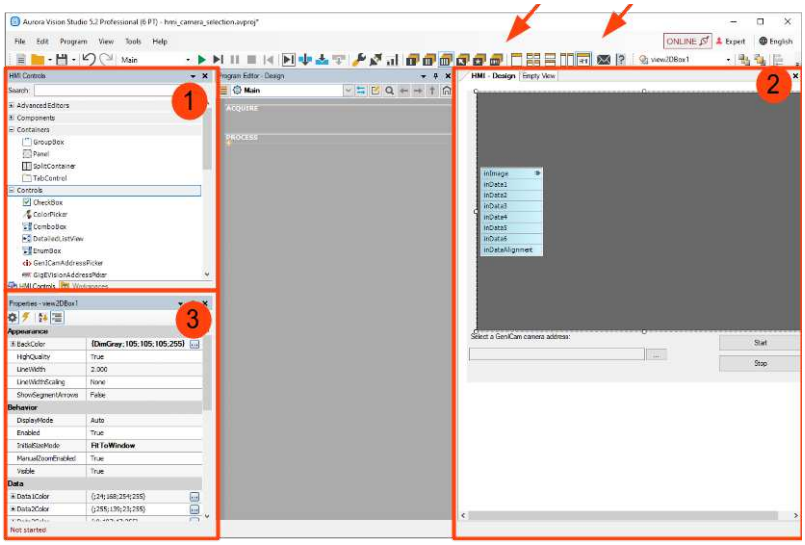
  and creating the user interface with Qt library.

  Note: Aurora Vision Executor with HMI support is only available on Microsoft Windows operating system. For Linux we recommend generating C++ code

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.

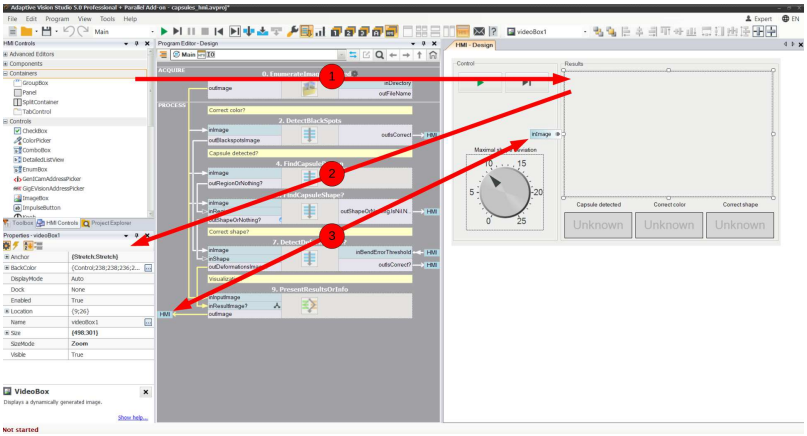
Removing HMI

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:

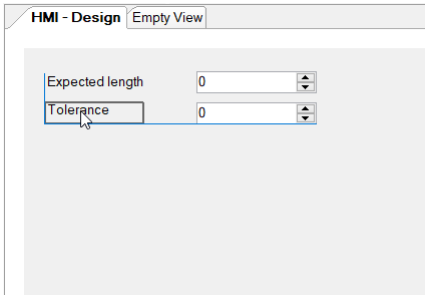
1. Drag & drop a control from HMI Controls to HMI Editor. Set its location and size.
2. Set properties of the selected control.
3. Drag & drop a connection between the control's input or output with an appropriate filter port in Program Editor.



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.



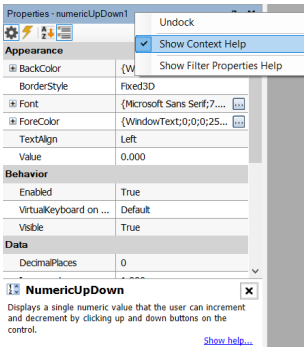
Snaplines assist in setting layout of the controls.

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

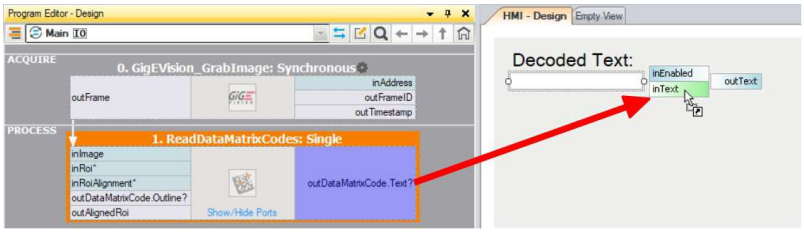
- **Anchor** set to *Right* or *Bottom* keeps the control in a fixed distance from the right or bottom edge of its parent window.
- **Anchor** set to *Stretch* makes the control resize with the parent window, keeping fixed distances from both edges.
- **Dock** property binds the control to one of the 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:

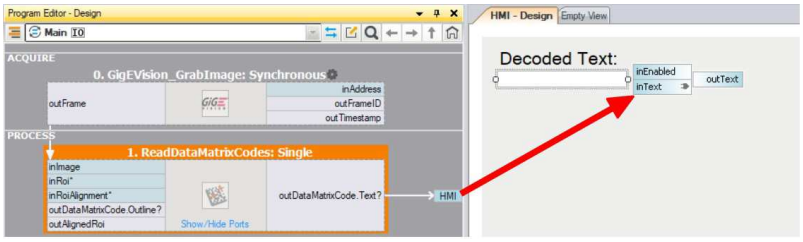


3. Connecting HMI controls is very similar to connecting filter inputs and outputs. First, the control which we want to connect has to be selected. Labels of inputs and outputs of the control will appear next to it. Then one should drag an output of a control onto a compatible filter input, or an output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



Connecting a filter with a control.

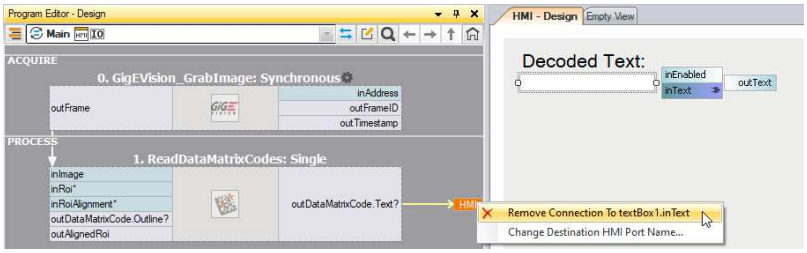
Aurora Vision Studio will assist this process, by disallowing connections between incompatible ports. After the connection is made, the input or output label will include a little plug icon, and the connected filter will have an "HMI" label next to its input or output.



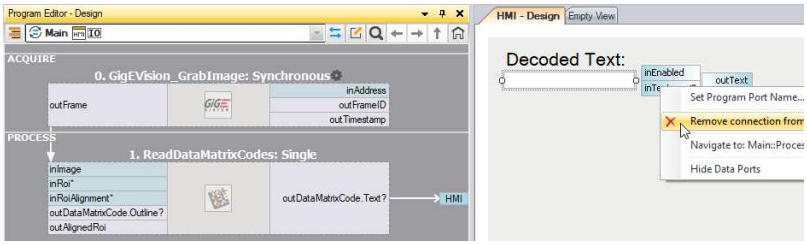
Connection between a filter and a control was established.

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.

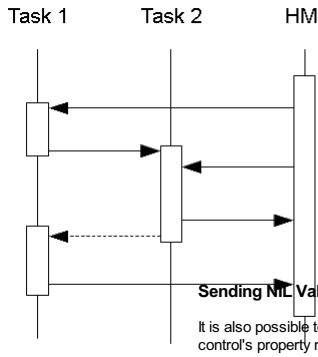
This has some important implications:

- 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 can NOT send a value to a control and expect that this value will be immediately available when a 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.

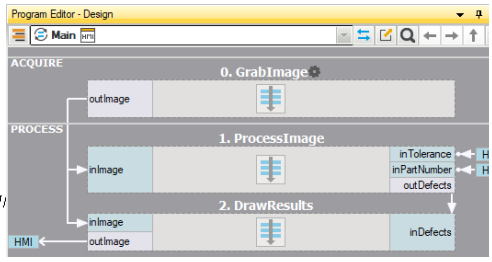
Recommendations

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. Fox 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 on the picture below:



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.



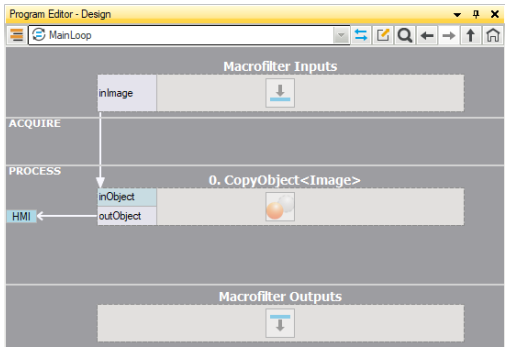
A typical simple structure of a program with HMI connections.

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). Is 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:

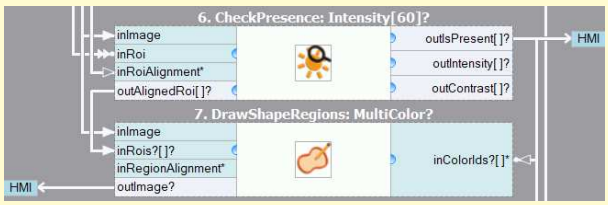
- Image Drawing – for preparing images overlaid with geometrical primitives, text labels etc.
- CropImage, ResizeImage, MirrorImage etc. – for transforming an image in an arbitrary way before display.

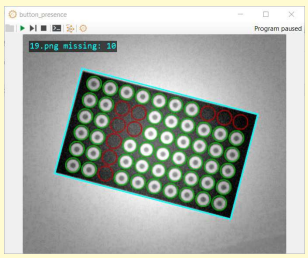
  Note: ZoomingVideoBox control has a built-it support for panning and zooming the displayed images, so no additional filters are required.
- FormatRealToString, FormatIntegerToString, FormatPoint2DToString etc. – for formatting numerical values into strings with properties such as the number of decimal 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:





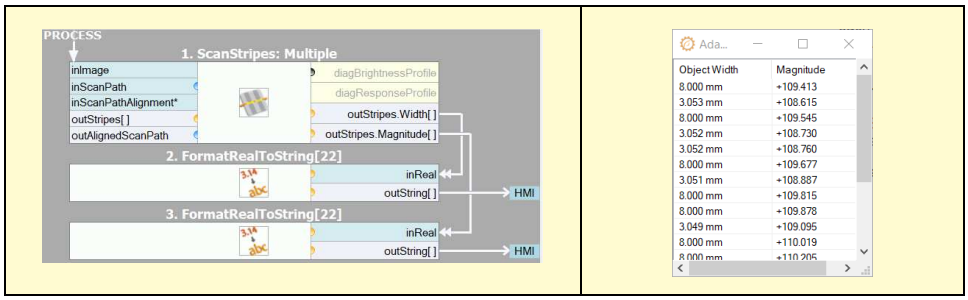
If more overlays are required on a single image, then we use multiple Image Drawing filters connected sequentially.

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

Introduction

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

- Components – with non-visual elements which extend the HMI window functionality.
- Containers – for organizing the layout with panels, groups, splitters and tab controls.
- Controls – with standard controls for setting parameters and controlling the application state.
- File System – for choosing files and directories.
- Indicators – for displaying inspection results or status.
- Logic and Automation – for binding some HMI properties with each other, also with basic AND/OR conditions.
- Multiple Pages – for creating multi-screen applications.
- Password Protection – for limiting access to some parts of an HMI to authorized personnel.
- Shape Editors – with controls that allow the end user to define object models or measurement primitives.
- Shape Array Editors – with controls that allow the end user 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 - with several variants of the VideoBox control 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 couple 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 the same for different standard controls. Here is a summary of the most important ones:

- **AutoSize** – specifies whether a control will automatically size itself to fit its contents.
- **BackColor** – the background color of the component.
- **BackgroundImage** – the background image used for the control.
- **BorderStyle** – controls how the border of the control looks like.

Three sample labels with different background colors.

Three sample labels with different border styles.

- **Enabled** – can be used to make the control not editable by the end user.
- **Font** – defines the size and style of the font used to display text in the control.
- **ForeColor** – foreground color of the control, which is used to display text.
- **Text** – a string being displayed in the control.
- **Visible** – can be used to make the control invisible (hide it).

Sample buttons, first enabled, second disabled.

Three sample labels with different fonts.

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

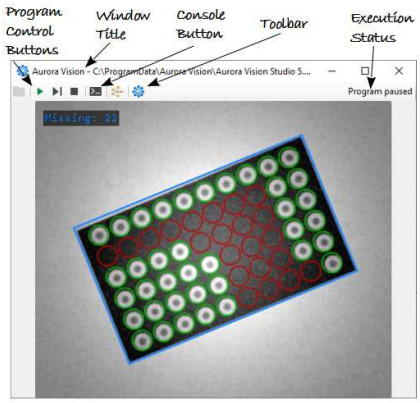
Sample custom style for a button and a label.

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

HMICanvas control represents the entire window of the created application. Other controls will be 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** – 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.

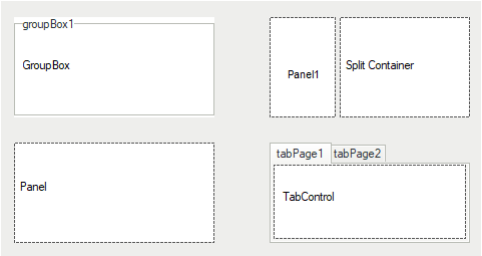
Remarks:

- 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 movable boundary 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 onto it all the controls you want to group together. Other method in order to do that is clicking on a desired container in "Containers" section and then selecting the area on 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).

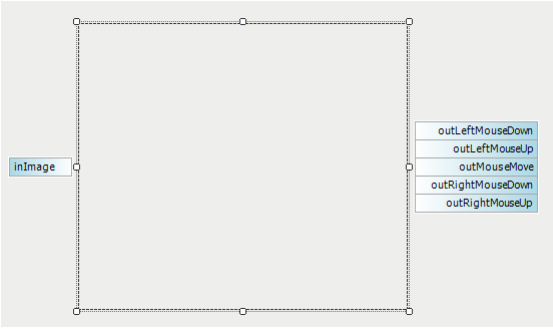
However, when you would like to create more sophisticated HMI Panel with multiple screens, go to The Multi Panel Control

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:

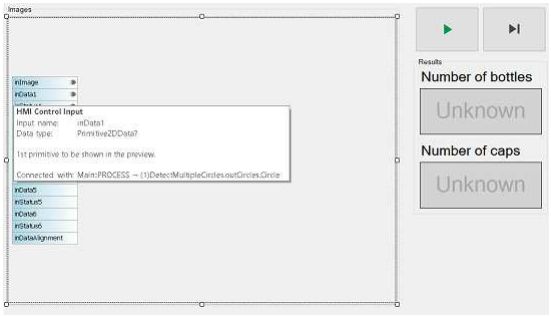
- VideoBox – the basic variant of the image display control.
- SelectingVideoBox – also allows for selecting a box region by the end user.
- ZoomingVideoBox – also allows for zooming and panning the image by the end user.
- FloatingVideoWindow – a component that allows for opening an additional window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



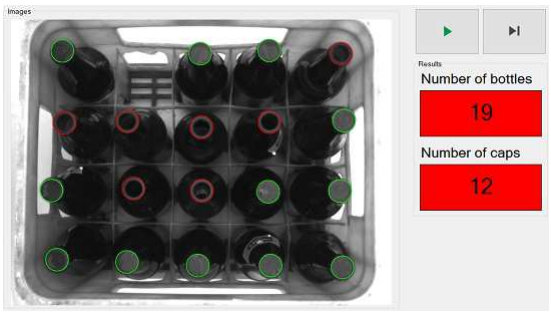
A VideoBox.

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 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.

Notes:

- Do NOT use ImageBox control for displaying inspection results. Due to performance reasons, it should only be used for static images.
- 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.
- 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.

Common properties:

- **DisplayMode** – switches between GDI and DirectX implementations. It depends on the underlying hardware, which one is faster.
- **SizeMode** – specifies how the image should be fitted to the available window space.

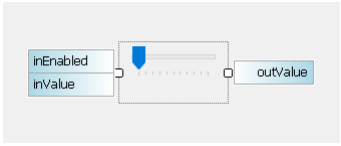
Controls for Setting Parameters

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

- TrackBar, Knob, NumericUpDown – 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.



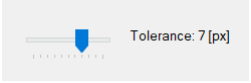
A track-bar.

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.

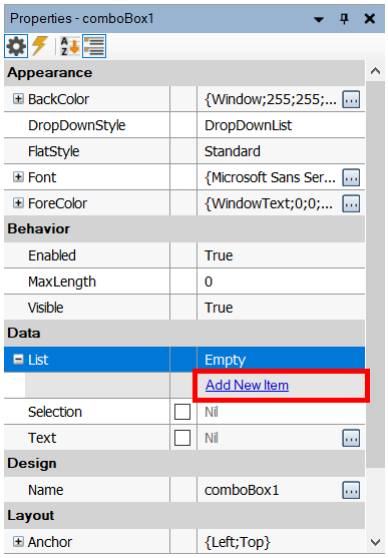
It is also possible to specify the format of the displayed text. The picture below shows a connected Label with the **AutoValueFormat** property set to "Tolerance: % [px]".



TrackBar with a connected Label using an additional text formatting.

Example: Using 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.

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.

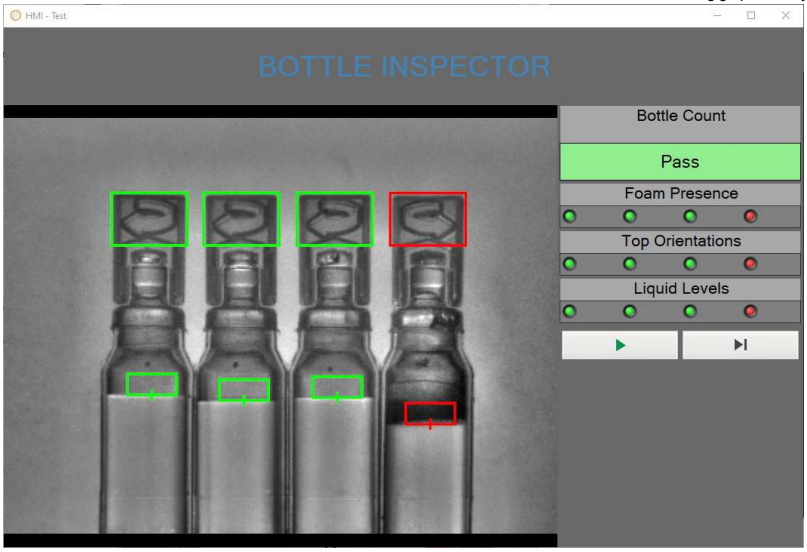
Please note that most of the event triggering outputs e.g. outValueChanged, outSelectionChanged etc. are hidden by default and can be shown with the "Edit Port Visibility..." command in the context menu of the control.

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.



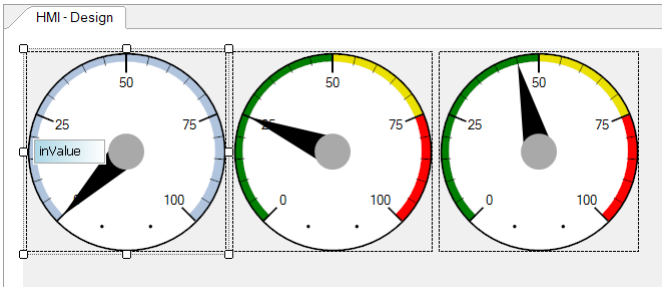
Example use of one PassFailIndicator and several BoolIndicatorBoards.

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.

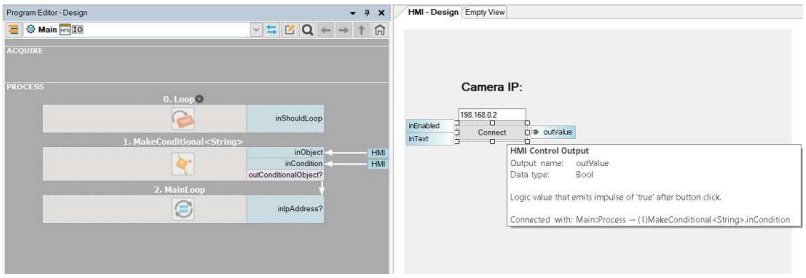


Analog indicators with and without scales

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.

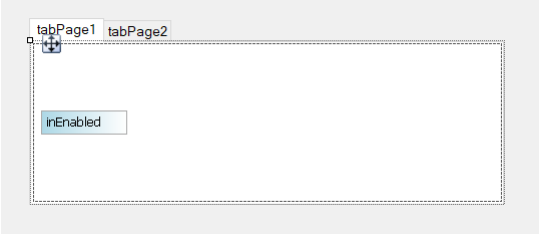


A simple use case of the ImpulseButton.

For more information about events see: Handling HMI Events.

The TabControl Control

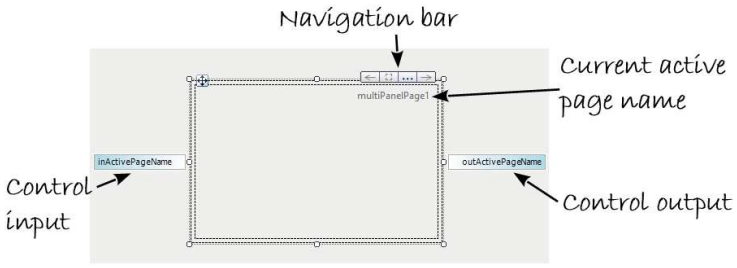
TabControl is the one of the methods to organize HMI panels within a limited space of a form. It is a container with several "tabs", where each page contains a different set of controls.



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.

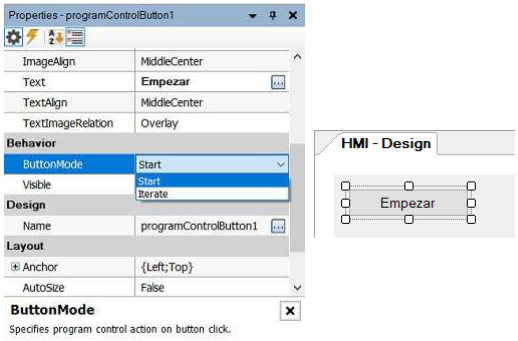
A MultiPanelControl control is usually accompanied by some **MultiPanelSwitchButton** controls that navigate between the pages. Usage of a button of this kind is very simple. Just place it on your window and set the **TargetPage** property (also available through double-clicking on the button). Page 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.

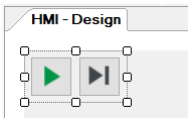
Note: There is a tutorial example HMI Multipanel Control.

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.

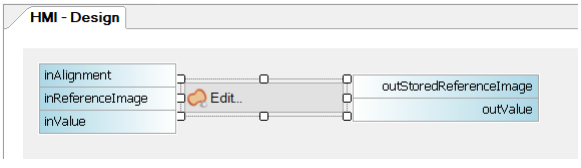
Shape Editors

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.

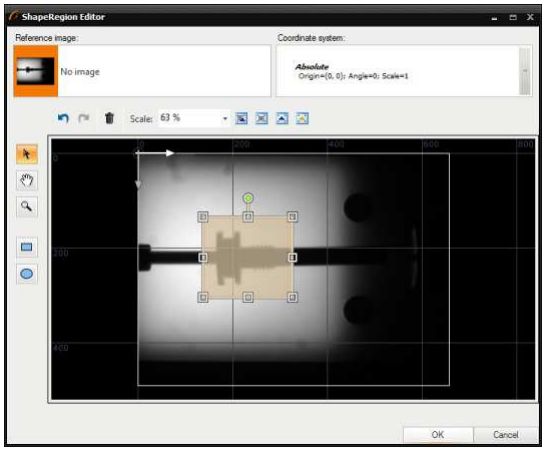
Each such editor has a form of a button with four ports visible by default:

- **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.

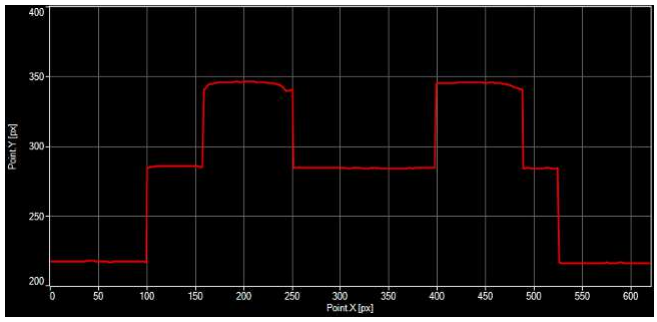
Shape editors can be used for creating input values for filter or modify existing values. Every shape editor has a proper data type assigned to **inValue** and **outValue**. The following list shows data types assigned to individual HMI controls and example filters that can be connected with them.

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

ProfileBox

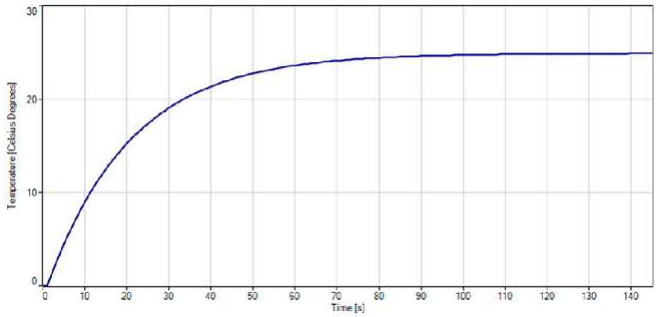
ProfileBox is the HMI control that displays a profile of evenly sampled measurements. ProfileBox is available in the "Indicators" section of the HMI Controls window.

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.



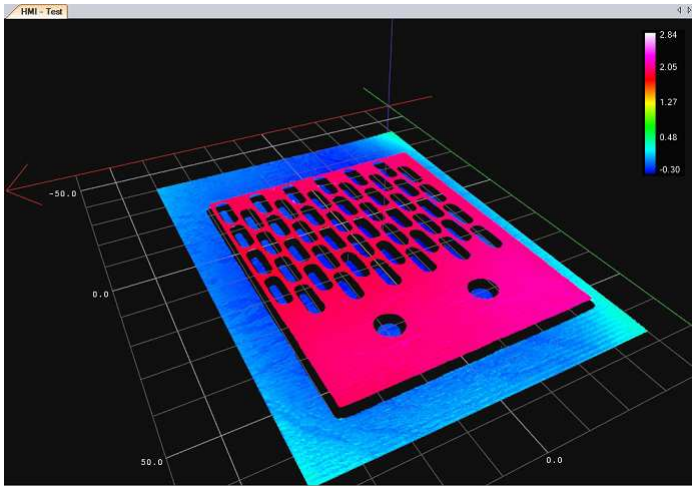
Temperature chart.

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.

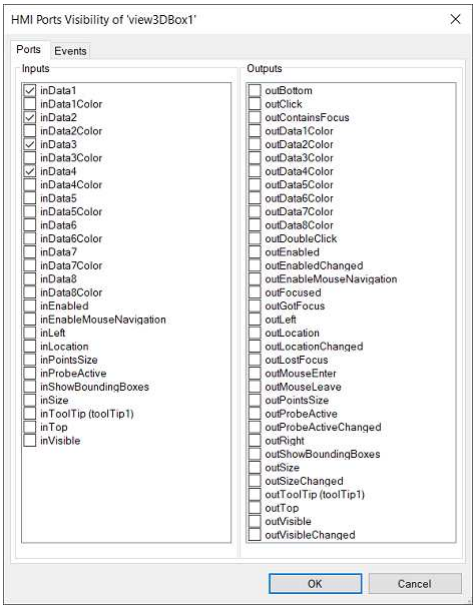
View3DBox

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:

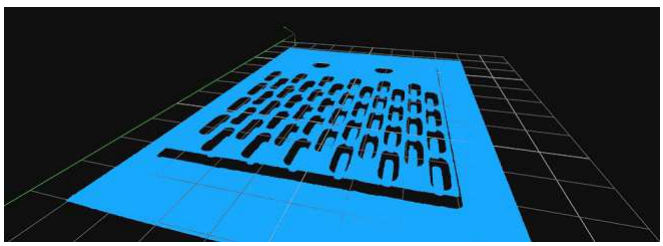
- **EnableAntiAliasing** – when set to true increases the render quality at the expense of performance.
- **GridMode** – specifies visibility and orientation of the measurement grid displayed in the scene.
- **PointSize** – specifies the display size of the cloud points relative to the size of the scene.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

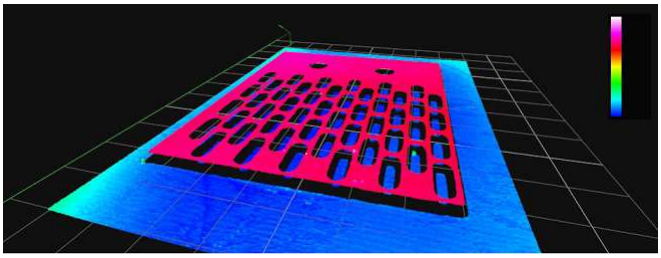
- **Enabled** – indicates whether the control is enabled.
- **EnableMouseNavigation** – when set to true enables the user to change observer position using mouse.
- **InitialViewState** – specifies the initial position of observer in the scene.
- **Visible** – determines whether the control is visible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

ActivityIndicator

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 OCR tools available in Aurora Vision Studio software are described in the following articles:

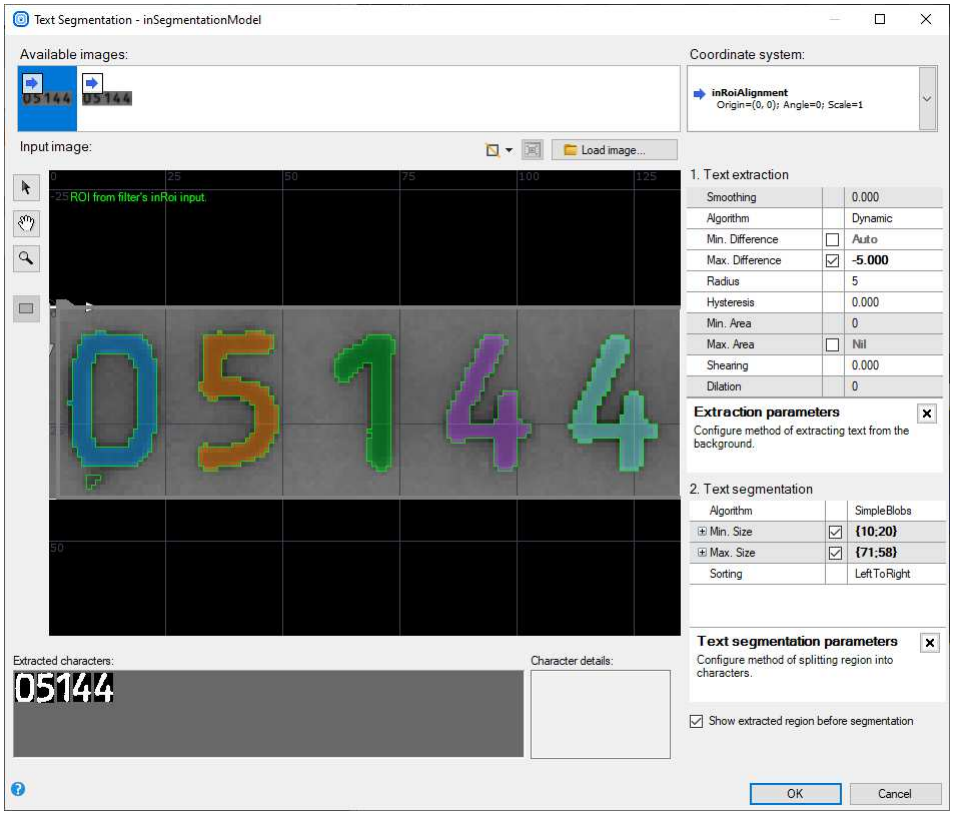
- Machine Vision Guide: Optical Character Recognition,
- Creating Text Segmentation Models,
- Creating Text Recognition Models.

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
- **inReferenceImage** – an image
- **outStoredReferenceImage** – the
- **outValue** – is a result OcrModel.

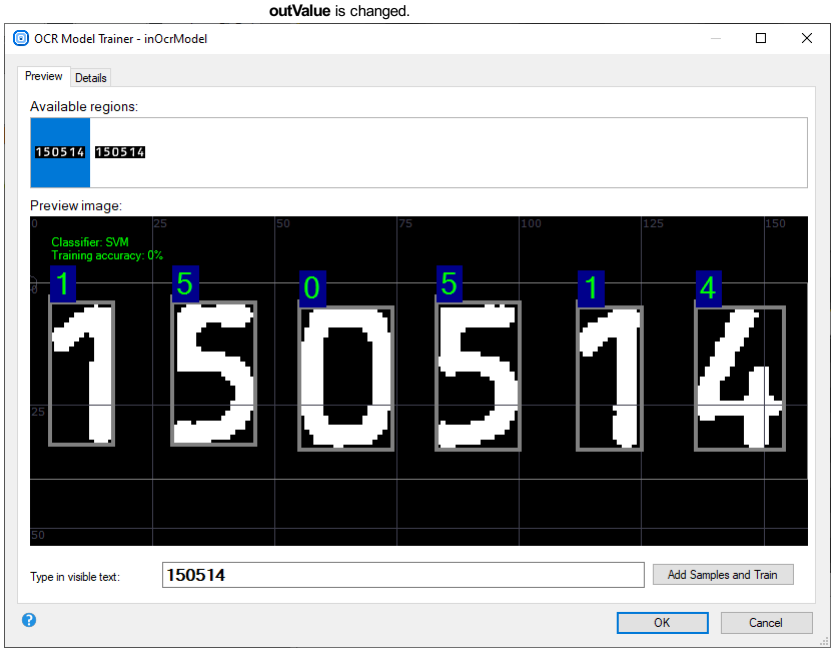
value of RegionArray data type. Characters from ExtractText filter can be connected to that input. The characters will be used during a model training.

set as the editor background.

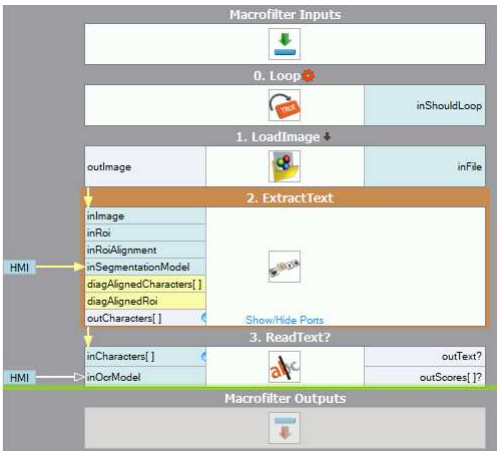
- **outValueChanged** – an output of the bool data type that is set to *True* for one iteration in which the **outValue** is changed.

image set as the editor background that can be sent to the algorithm.
After clicking on OCRModelEditor button, the editor will be opened. However, to be able to open it, the program has to be executed.

This value should be connected to **inOcrModel** input in ReadText filter.



Editor is the same as inOcrModel editor known from ReadText filter.



A simple program which is 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 to check which key is pressed, but there is 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 is most cases.
- **outKeysUP** – emits array of keys that have been recently released.

IntegerArray data type.

To use those values in other place in program the best solution is copying them with **CopyObject** filter with

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.

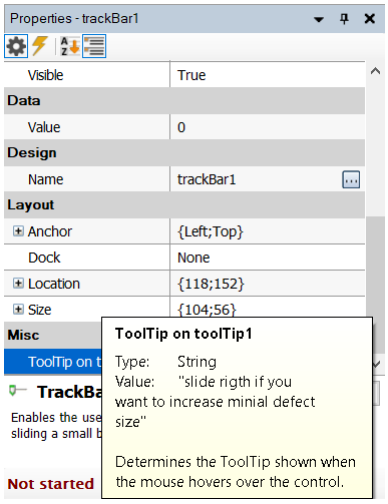
ToolTip

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.

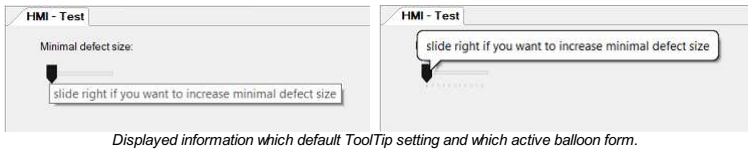
Once ToolTip control is added, each HMI Control has additional parameter available under the "Misc" category in the Properties window. It is possible to use multiple ToolTips with different parameters.



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

ToolTip properties can be adjusted to 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, animation are used when the ToolTip is shown or hidden.
- **UseFading** – when set to true, a fade effect is used when ToolTip are shown or hidden.



Displayed information which default ToolTip setting and which active balloon form.

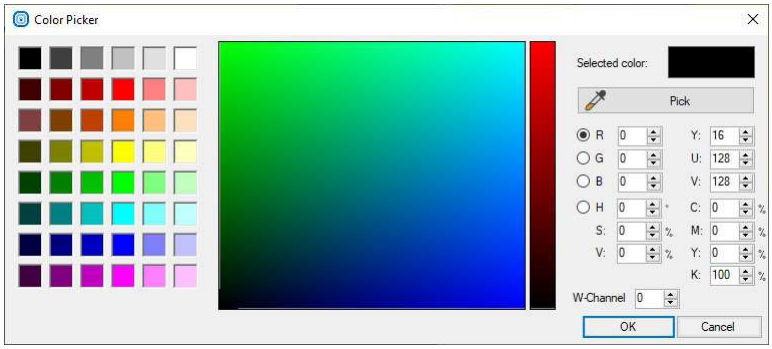
ColorPicker

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

This control can be used i.e. when it is required 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.

Logic and Automation tools

BoolAggregator

BoolAggregator allows to combine multiple boolean signal sources using logic functions for purpose of controls such as EnabledManager.

This function is useful when 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. This controls have no graphical representation in the HMI application and only provide a data processing function.

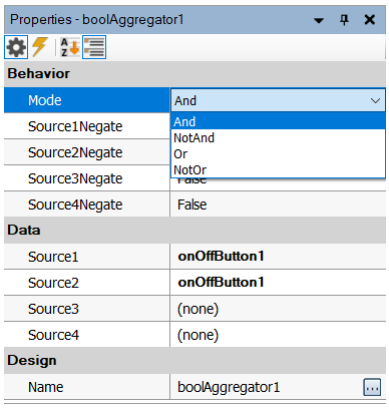
While working with that control, user needs to select boolean value sources to be aggregated. It can be done in the Properties window. Also, additional two values from the program can be aggregated by connecting them to the **inValue1** and **inValue2** inputs.

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; {True, True, True, True} → True)
- **NotAnd** - `not (in1 and in2 and ... and inN)` (e.g.: {True, True, True, True} → False; {True, True, False, True} → 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.

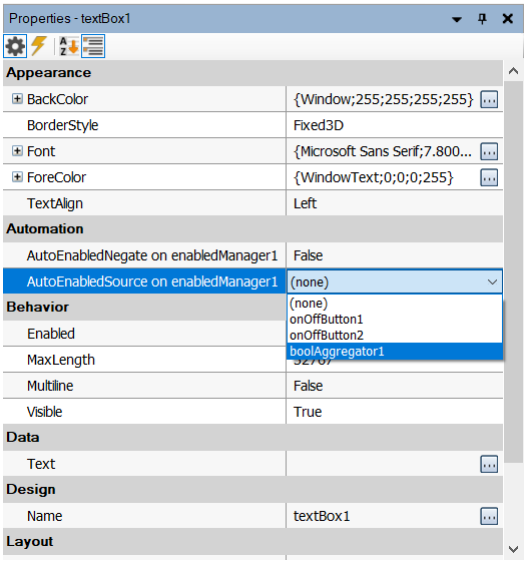
It can be used with BoolAggregator in order to manage controls' enable states in more complex cases.

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 to send an enable signal to every HMI Control. 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.*

EdgeModelEditor

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

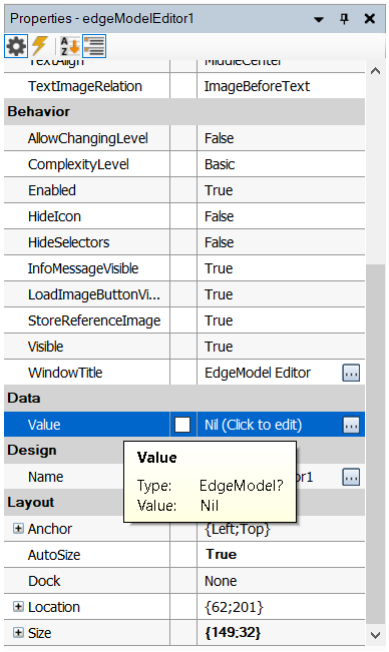
The process of creating a model representing the expected object's shape is described in Creating Models for Template Matching article.

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.

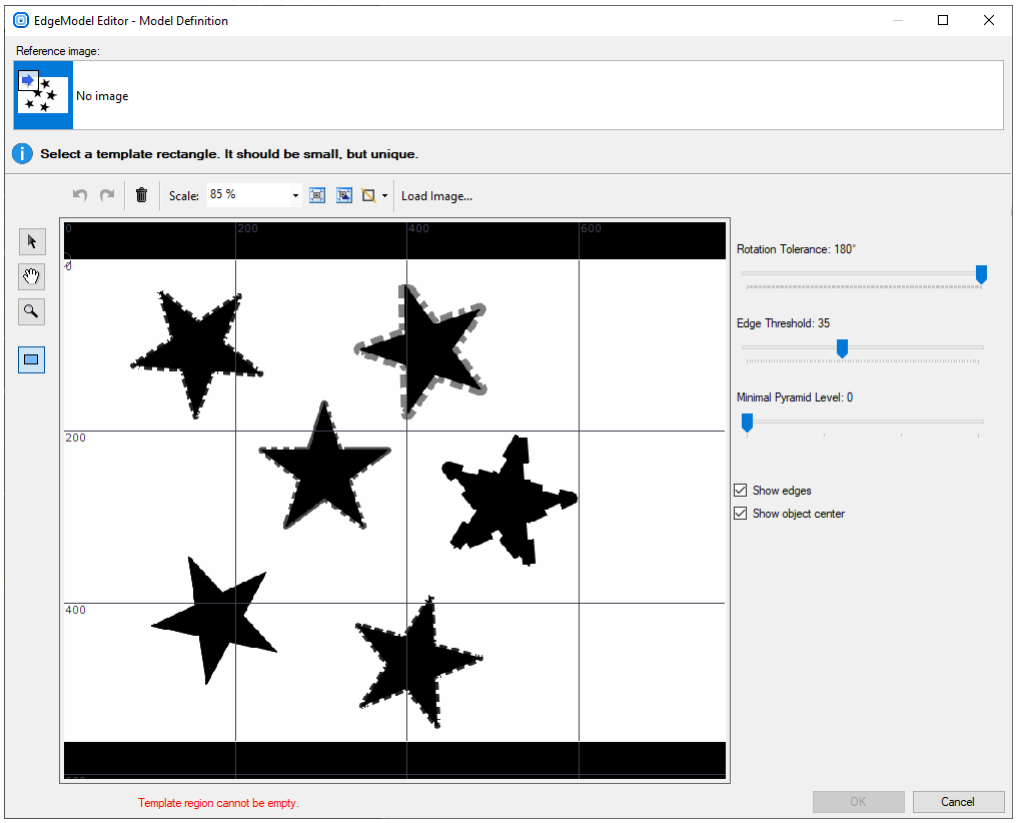
outValue output has EdgeModel data type. This output can be connected to **inEdgeModel** input in **LocateObjects_Edges1** filter.

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 to change 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 runtime environment by using an easy user interface called "GUI for Template Matching".

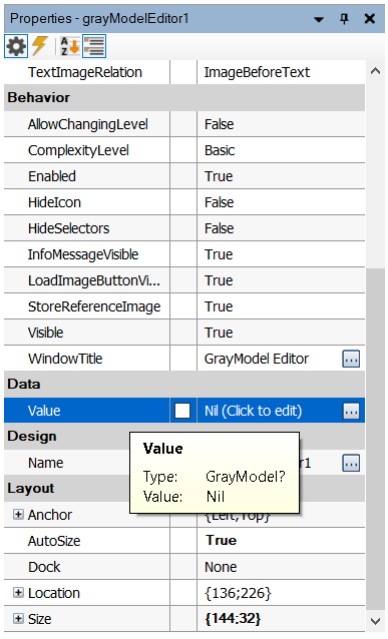
The process of creating a model representing the expected object's shape is described in Creating Models for Template Matching article.

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.

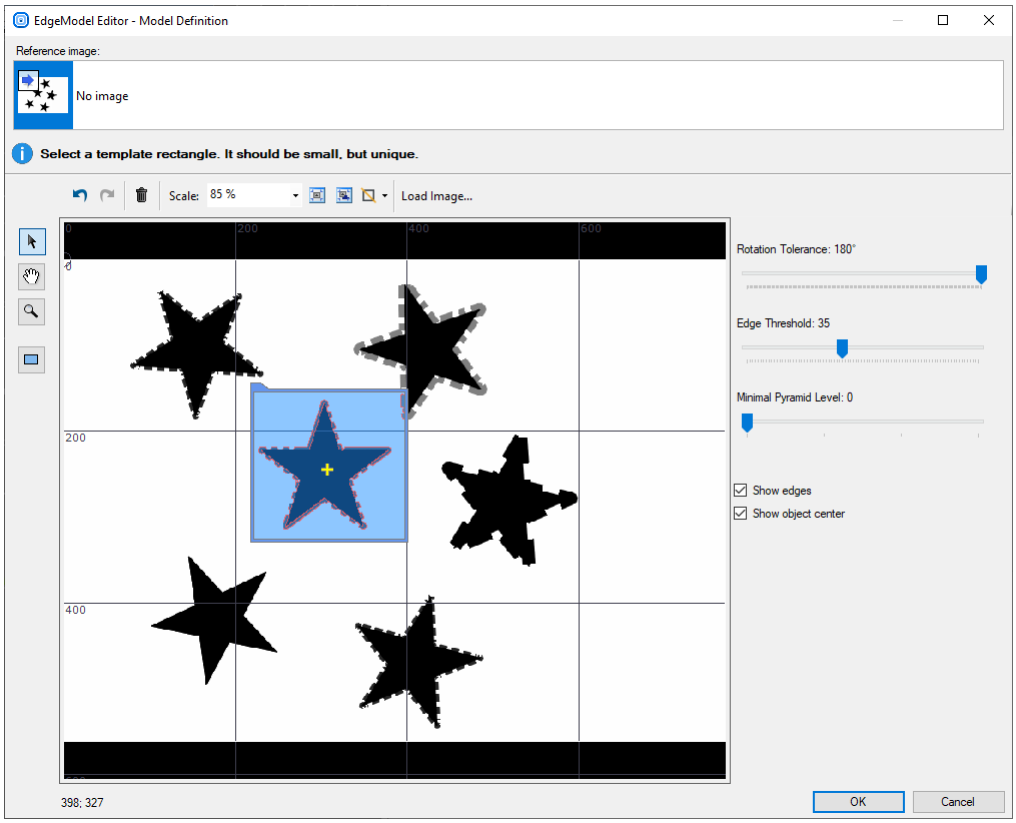
Besides standard parameters available in the Appearance, Design and Layout sections of the Properties window, the GrayModelEditor can be customized using the following parameters:

- AllowChangingLevel – allows to change 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

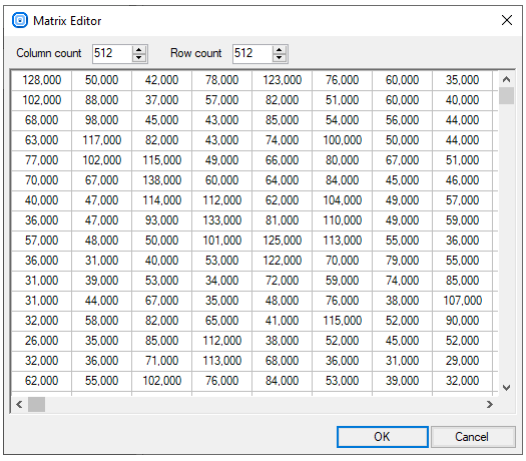
GigEVisionAddressPicker is used to choose GigE Vision device address in runtime environment. This control can be used with GigE Vision filters.

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.

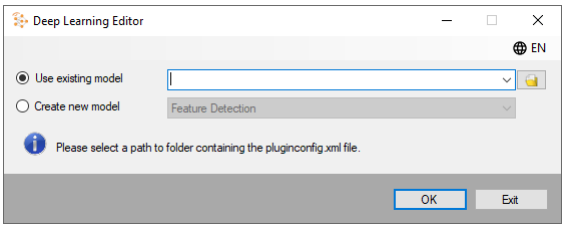
outValue returns matrix which was modified or created in MatrixEditor.



Matrix Editor window.

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.

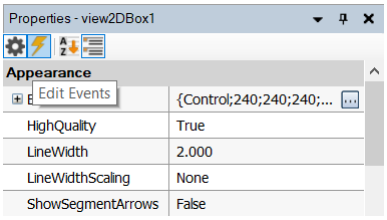
Event Handlers

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.

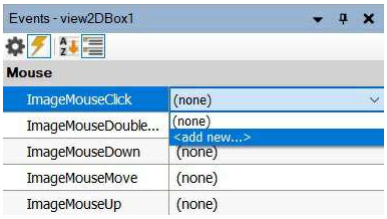
Creating Event Handlers

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:

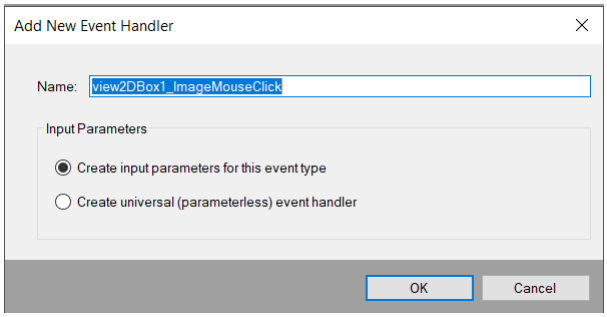


3. This will open the Events window with most commonly used events that can be handled. Click the drop-down list next to the event name and select <add new...> as shown below:

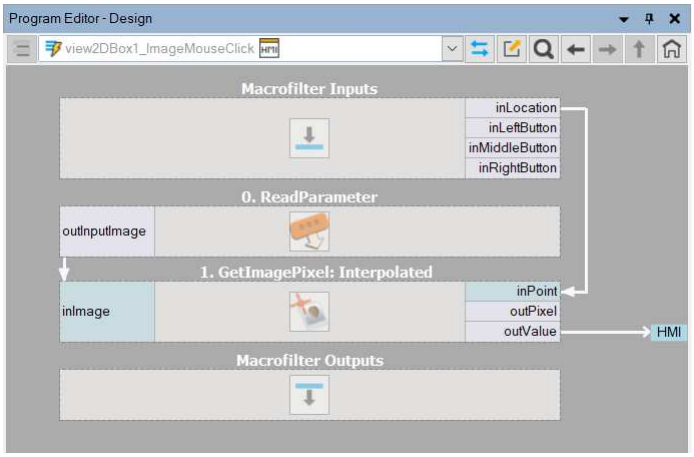


NOTE: There are many more events available. If you want to see the complete list, right-click on the HMI Control, select Edit Ports and Events Visibility... and go to the Events tab.

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

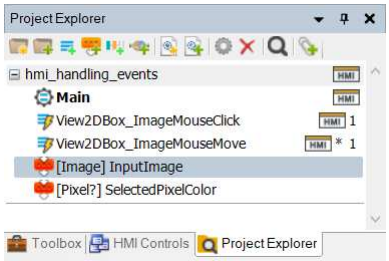


5. Some types of events, e.g. mouse move, can send additional information about the performed action. When creating an event handler you need to specify whether input parameters (containing additional information) should be created. If you select this option, the created event handler will have some 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.

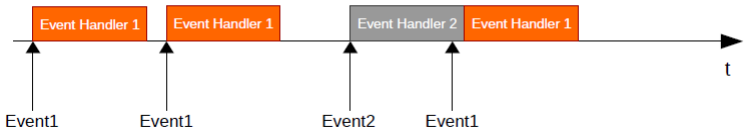
6. Once the event handler is created, it is available in the Project Explorer:



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.



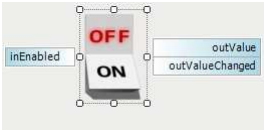
Sample events sequence and a timeline showing the order in which they are handled

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.

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

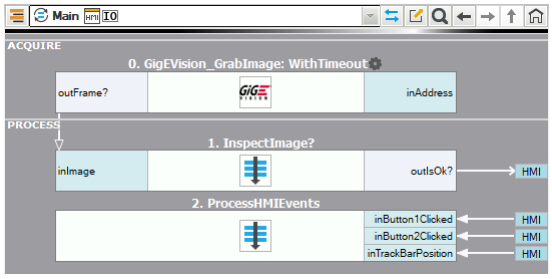
Handling events that switch the application to another mode of operation is described in the Programming Finite State Machines article.

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 combo box is filled with names of all files with required extension, which are found in given folder (therefore, it is 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 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.
- **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.
- **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.
- **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 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.

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 from

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.

Additional properties of the StateControlBox control:

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:

StateAutoLoader control

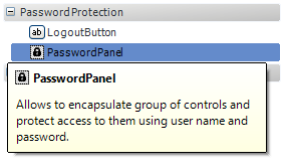
This control can be used to automatically load chosen HMI configuration on application start or to automatically save current HMI state to a configuration file on application close. StateAutoLoader should be placed inside a container control which defines its range. This control is visible only in the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

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.

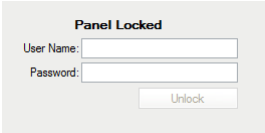
- **CreateSubdirectory** - setting it to *True* will cause that the folder which should contain the file described by the FilePath property will be fully created (when it does not exist yet) before saving a new configuration to it. In case of setting it to *False* (a default value), an attempt to save 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.



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.

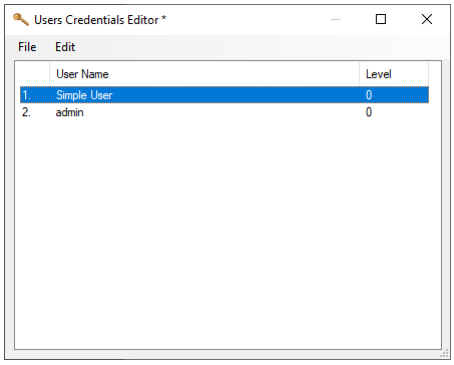
You will be also automatically logged out after the period of time defined in the **AutoLockTimeout** property (in seconds) of the PasswordPanel control if the system input has been idle for that period of time (no keystrokes or mouse clicks/moves).

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.

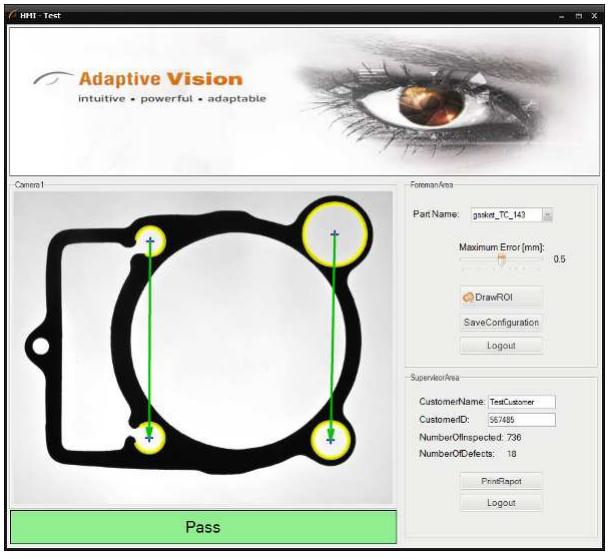
The unlocked area for the unauthorized users is only used for displaying the result of inspection.

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

Time	Level	Message
17:01:24	Info	HMI PasswordPanel "ForemanPanel": User login successful (user name: "Foreman", access level: 0).
17:01:40	Info	HMI PasswordPanel "ForemanPanel": User logged out (user name: "Foreman").
17:01:40	Info	HMI PasswordPanel "ForemanPanel" settings state: MaximumError: "6"
17:01:52	Info	HMI PasswordPanel "ForemanPanel": User login successful (user name: "Supervisor", access level: 1).
17:02:04	Info	HMI PasswordPanel "ForemanPanel": User logged out (user name: "Supervisor").
17:02:04	Info	HMI PasswordPanel "ForemanPanel" settings state: MaximumError: "5"
17:02:17	Info	HMI PasswordPanel "SupervisorPanel": User login failed (user name: "foreman").
17:02:37	Info	HMI PasswordPanel "SupervisorPanel": User login successful (user name: "Supervisor", access level: 1).
17:03:15	Info	HMI PasswordPanel "SupervisorPanel": User logged out (user name: "Supervisor").
17:03:36	Info	HMI PasswordPanel "ForemanPanel": User login successful (user name: "Foreman", access level: 0).

Hints | Console

Console with events logged for audit trial purposes.

Creating User Controls

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 `ShouldSerializeXXX`, `ResetXXX` 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 a XML format and every value must be converted to a text, which is feasible to be saved in XML node. To ensure it is feasible, types of data used in public properties must support conversion from and to a text independent from a localization. Therefore, they need to have assigned a type converter which is capable of conversion in both direction with `System.String` type. Most of simple built-in types and types provided by .NET Framework (as `System.Integer` or `System.Drawing.Size`) already supports such conversion and does 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 edition, the property must be public and its data type must allow

conversion to AMR data. 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 edition (e.g. by setting permitted range). More detailed description regarding conversion to AMR can be find further on 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.

DLL file prepared in such way should be placed in HMIControls folder in installation folder of adequate Aurora Vision product or in HMIControls folder in sub-folder of Aurora Vision product folder in user documents. Loading of the module requires restarting of 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
- In case of using reference 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.

(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 to display a virtual on-screen keyboard, useful on systems with touch panels. The keyboard is, to a limited extend, shown and hidden automatically by the HMI environment. If required, a HMI control can explicitly control showing and hiding of the virtual keyboard. To do so, control must obtain `HMI.IVirtualKeyboardService` interface. This interface provides methods for displaying keyboards of a specified types, avoiding the obstruction of the visibility of specified control or specified part of the screen, as well as methods for hiding 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
- 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
  - TextBox
  - ToggleButton
  - TrackBar
- Shape Editors
  - Arc2DEditor
  - ArcFittingFieldEditor
  - BoxEditor
- Advanced Editors
  - Edge Model Editor
  - Gray Model Editor
  - Matrix Editor
  - OCR Model Editor
  - Text Segmentation Editor
- Deep Learning
  - StartButton
- Indicators
  - ActivityIndicator
  - AnalogIndicator
  - AnalogIndicatorWithScale
  - BoolIndicatorBoard
  - PassFailIndicator
  - ProfileBox
  - View2DBox
  - View2DBox_PassFail
  - View3DBox
- State Management
  - StateAutoLoader
  - StateControlBox
  - StateControlButton
- Components
  - Keyboard Listener
  - ToolTip
  - Virtual Keyboard
- Containers
  - Group Box
  - Panel
  - Split Container
  - Tab Control
- File System
  - DirectoryPicker
  - FilePicker
- Logic and Automation
  - BoolAggregator
  - EnabledManager
- 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
- Video Box
  - FloatingVideoWindow
  - SelectingVideoBox
  - VideoBox

- Circle2DEditor
- CircleFittingFieldEditor
- Line2DEditor
- LocationEditor
- PathEditor
- PathFittingFieldEditor
- Point2DEditor
- Rectangle2DEditor
- RegionEditor
- Segment2DEditor
- SegmentFittingFieldEditor
- SegmentScanFieldEditor
- ShapeRegionEditor
- ShapeRegionDeprecatedEditor
- ZoomingVideoBox

6. Programming Reference

Table of content:

Data Types

Read first: Introduction to Data Flow Programming.

Introduction

Integer number, image or two-dimensional straight line are examples of what is called a *type of data*. Types define the structure and the meaning of information that is being processed. In Aurora Vision Studio types also play an important role in guiding program construction – the environment assures that only inputs and outputs of compatible types can be connected.

Primitive Types

The most fundamental data types in Aurora Vision Studio are:

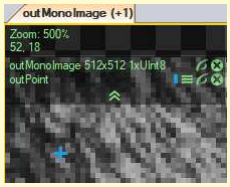
- **Integer** – Integer number in the range of approx. ±2 billions (32 bits).
- **Real** – Floating-point approximation of real numbers (32 bits)
- **Bool** – Logical value – *False* or *True*.
- **String** – Sequence of characters (text) encoded as UTF-16 unicode.

Type Constructs

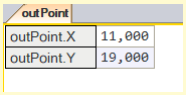
Complex types of data used in Aurora Vision Studio are built with one of the following constructs:

Structures

Composite data consisting of a fixed number of named fields. For example, the Point2D type is a structure composed of two real-valued fields: *X* and *Y*.



A point (blue) in a graphical view.



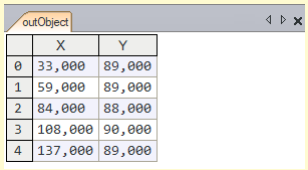
A point in a textual view.

More information about using structures can be found: here.

Arrays

Composite data composed of zero, one or many elements of the same type. In Aurora Vision Studio, for any type *X* there is a corresponding *XArray* type, including *XArrayArray*. For example, there are types: Point2D, Point2DArray, Point2DArrayArray and more.

A point array in a graphical view.



A point array in a textual view.

Example filters having arrays on the outputs:

- ScanMultipleEdges – returns an array of multiple found edge points.
- SplitRegionIntoBlobs – returns an array of multiple blob regions.
- FitLineToPoints_M – creates a line best matching a set of two-dimensional points.

Example filters having arrays on inputs:

- Average – calculates the average of multiple real numbers.

More information about using arrays can be found: here.

Optional

Some filter inputs can be left not connected and not set at all. These are said to be of an optional type. As for arrays, for each type *X* there is a corresponding optional *X** type. The special value corresponding to the special *not-set* case is identified as *Auto*.

Example filters with optional inputs:

- ThresholdImage – has an optional **inRoi** input of the Region type for defining the region-of-interest. If the **Auto** value is provided, the entire image will be processed.
- FitLineToPoints_M – has an optional **inOutlierSuppression** input specifying one of several possible enhancement of line fitting. If the **Auto** value is provided, no enhancement is used.

More information about using optional inputs can be found: here.

Conditional

Conditional types are similar to optional, but their purpose is different – with the *Nil* value (usually read "not detected") they can represent lack of data on filter outputs which results in conditional execution of the filters that follow. For each type *X* there is a corresponding conditional *X?* type.

Example filters with conditional outputs:

- SegmentSegmentIntersection – returns a common point of two segments or **Nil** if no such point exists.
- ReadSingleBarcode – returns a value read from a barcode or **Nil** if no barcode was found.

More information about conditional data can be found: here.

Enumerations

Enumeration types are intended to represent a fixed set of alternative options. For example, a value of the DrawingMode type can be equal to *HighQuality* or *Fast*. This allows for choosing between quality and speed in the image drawing filters.

Composite Types

Array and conditional or optional types can be mixed together. Their names are then read in the reverse order. For example:

- **IntegerArray?** is read as "conditional array of integer numbers".
- **Integer?Array** is read as "array of conditional integer numbers".

Please note, that the above two types are very different. In the first case we have an array of numbers or no array at all (Nil). In the second case there is an array, in which every single element may be present or not.

Built-in Types

A list of the most important data types available in Aurora Vision Studio is here.

Angles

Angles are using Real data type. If you are working with them bear in mind these assumptions:

- All the filters working with angles return their results in degrees.
- The default direction of the measurement between two objects is clockwise. In some of the measuring filters, it is possible

to change that.

Automatic Conversions

There is a special category of filters, Conversions, containing filters that define additional possibilities for connecting inputs and outputs. If there is a filter named *XToY* (where *X* and *Y* stand for some specific type names), then it is possible to create connections from outputs of type *X* to inputs of type *Y*. You can for example connect Integer to Real or Region to Image, because there are filters IntegerToReal and RegionToImage respectively. A conversions filter may only exist if the underlying operation is obvious and non-parametric. Further options can be added with user filters.

A typical example of automatic conversion: a region being used as an image.

Structures

Introduction

A structure is a type of data composed of several predefined elements which we call *fields*. Each field has a name and a type.

Examples:

- Point2D is a simple structure composed of two fields: the X and Y coordinates of type Real.
- DrawingStyle is a more complex structure composed of six fields: DrawingMode, Opacity, Thickness, Filled, PointShape and PointSize.
- Image is actually also a structure, although only some of its fields are accessible: Width, Height, Depth, Type, PixelSize and Pitch.

c++ Structures in Aurora Vision Studio are exactly like structures in C/C++.

Working with Structure Fields

There are special filters "Make" and "Access" for creating structures and accessing their fields, respectively. For example, there is MakePoint which takes two real values and creates a point, and there is AccessPoint which does the opposite. In most cases, however, it is recommended to use Property Outputs and Expanded Input Structures.

Arrays in Aurora Vision Studio

Introduction

An array is a collection composed of zero, one or many elements of the same type. In Aurora Vision Studio for any type *X* there is a corresponding *XArray* type, including *XArrayArray*. For example, there are types: Point2D, Point2DArray, Point2DArrayArray and more.

c++

Arrays are very similar to the *std::vector* class template in C++. As one can have *std::vector* < *std::vector*< *atl::Point2D* > > in C++, there is also Point2DArrayArray in Aurora Vision Studio.

Generic Filters for Arrays

Arrays can be processed using a wide range of generic filters available in the categories:

- Array Basics
- Array Transforms
- Array Composition
- Array Set Operators
- Array Statistics

Each of these filters requires *instantiation* with an appropriate element type to assure that no mistakes are later made when connections are created. Here is an example usage of the TestArrayNotEmpty filter, which has an input of the RegionArray type:

The TestArrayNotEmpty filter used to verify if there is at least one blob.

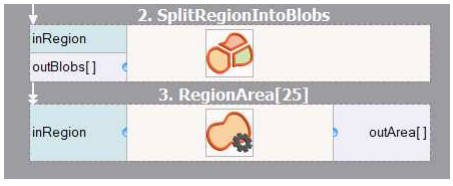
Singleton Connections

When a scalar value, e.g. a Region, is connected to an input of an array type, e.g. RegionArray, an automatic conversion to a single-element array may be performed. This feature is available for selected filters and cannot be used at inputs of macrofilters.

Array Connections

When an array, e.g. a RegionArray, is connected to an input of a scalar type, e.g. Region, the second filter is executed many times, once per each input data element. We call it the array mode. Output data from all the iterations are then merged into a single output array. In the user interface a "[]" symbol is added to the filter name and the output types are changed from *T* to *T(Array)*. The brackets distinguish array types which were created due to array mode from those which are native for the specific filter.

For example, in the program fragment below an array of blobs (regions) is connected to a filter computing the area of a single region. This filter is independently executed for each input blob and produces an array of Integers on its output.



An array connection.

Remarks:

- If the input array connected to a scalar input is empty, then the filter will not be executed at all and empty arrays will be produced on the outputs.
- If there are multiple array connections at a single filter instance, then all of the input arrays must be of the same length. Otherwise a domain error is signaled. For more information on this topic see: Array Synchronization.

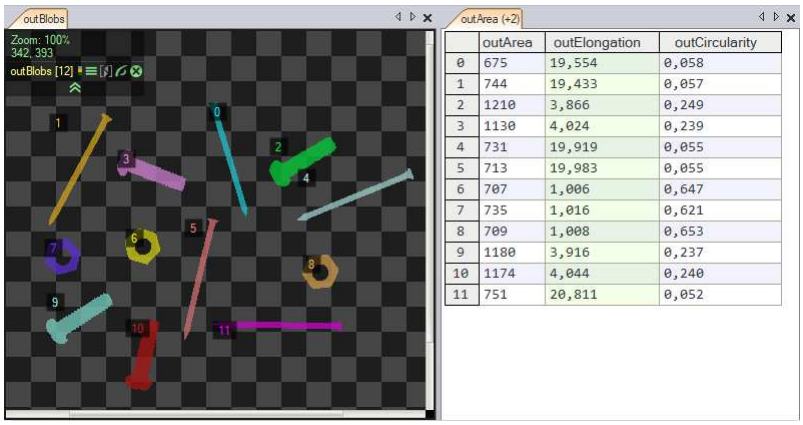
Array Synchronization

Introduction

In many applications several objects are analyzed on a single image. Most typically, in one of the first steps the objects are extracted (e.g. as an array of blobs, edges or occurrences of a template) and then some attributes are computed for all of them. In Aurora Vision Studio these data items are represented as several *synchronized* arrays, with different arrays representing the objects and the individual attributes.

The elements of synchronized arrays correspond one-to-one with each other like the elements of consecutive rows of a table in numerical

spreadsheet applications. Indeed, also in Aurora Vision Studio synchronized arrays of numerical values can be presented as a table, whereas the corresponding graphical objects can be displayed with the corresponding indexes (after checking an appropriate option in the image view menu), that indicate which row they belong to:



	outArea	outElongation	outCircularity
0	675	19,554	0,058
1	744	19,433	0,057
2	1210	3,866	0,249
3	1130	4,024	0,239
4	731	19,919	0,055
5	713	19,983	0,055
6	707	1,006	0,647
7	735	1,016	0,621
8	709	1,008	0,653
9	1180	3,916	0,237
10	1174	4,044	0,240
11	751	20,811	0,052

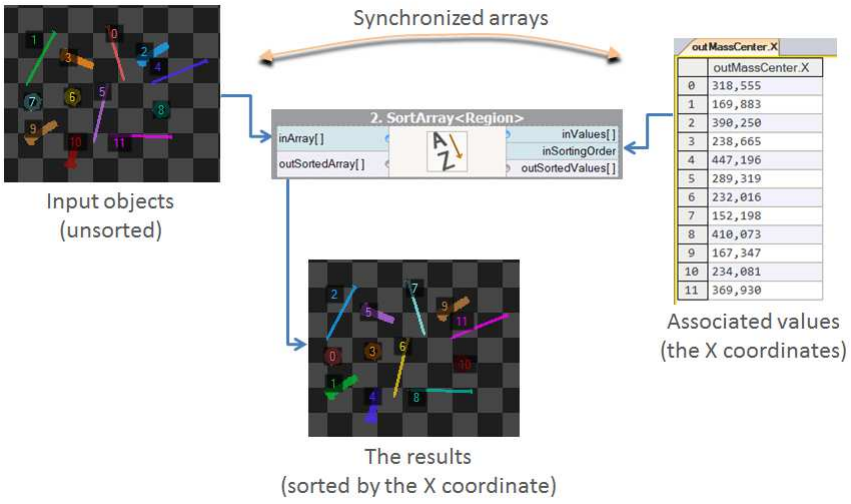
Preview of several synchronized arrays.

Synchronized Inputs

Some filters have several array inputs and require that all the connected arrays are synchronized, so that it is possible to process the elements in parallel. For example, the SortArray filter requires that the array of the objects being sorted and the array of the associated values (defining the requested order) are synchronized. This requirement is visualized with blue semicircles on the below picture:

Visualization of inputs that require synchronized arrays.

For example, we can use the SortArray filter to sort regions by the X coordinates of their mass centers:



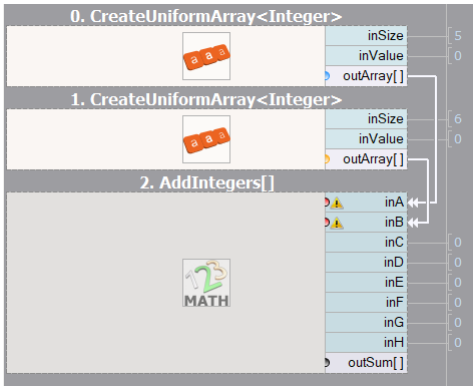
	outMassCenter.X
0	318,555
1	169,883
2	390,250
3	238,665
4	447,196
5	289,319
6	232,016
7	152,198
8	410,073
9	167,347
10	234,081
11	369,930

The principle of synchronized inputs on the example of SortArray.

Other examples of notable filters that require synchronized input arrays include: GetMaximumElement, ClassifyByRange and DrawStrings_SingleColor (the arrays of strings and the corresponding center points must be synchronized).

Static Program Analysis

Aurora Vision Studio performs a static program analysis to detect cases when arrays of possibly different sizes (unsynchronized) are used. If such a case is found, the user is warned about it before the program is executed and the detected issues are marked in the Program Editor with exclamation icons and red semicircles:

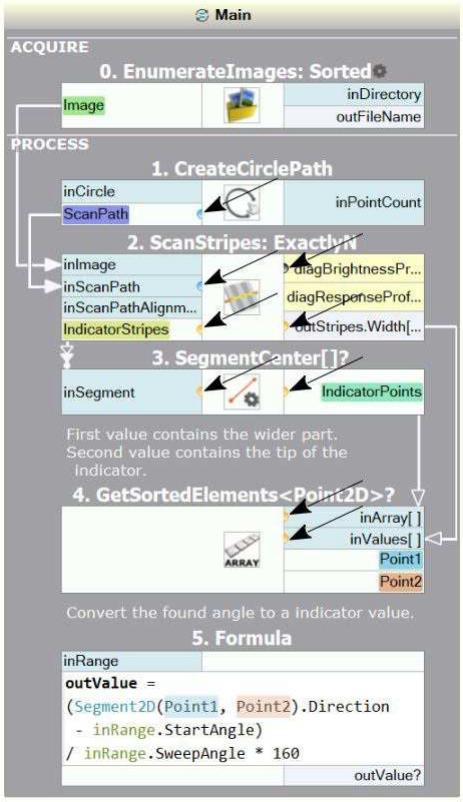


Visualization of an array synchronization problem.

Sometimes, for complicated programs, the static analysis mechanism may generate false warnings. In such cases, the user can manually mark the inputs as synchronized with the "Mark Arrays as Synchronized" command in the filter's input context menu.

Visualization of Related Arrays

The next picture presents a complete visualization of array synchronization in a real program (the "Meter" official example). There are two colors of the semicircles – blue and yellow. By visualizing the array synchronization it is easy to see, which ports contain arrays describing the same entities. In this particular case the blue semicircles are related to the circular scanning path and the yellow circles are related to the founded stripes.



Visualization of example array synchronization.

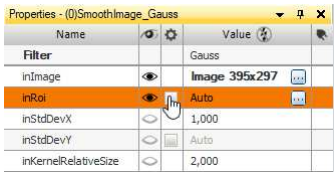
Note: Array synchronization analysis and its visualization is turned on by default, but this can be turned off in the settings of Aurora Vision Studio.

Optional Inputs

Some filters have optional inputs. An optional input can receive some value, but can also be left not set at all. Such an input is then said to have an automatic value. Optional inputs are marked with a star at the end of the type name and are controlled with an additional check-box in the Properties control.

Examples:

- Most image processing filters have an optional **inRoi** input of the Region* type – an optional (or automatic) region of interest. If we do not use this input, then the entire input image will be processed.
- The Pylon_GrabImage filter has an optional **inExposureTime** input of the Real* type. If we do not use this input, then the value previously written to the camera register will be used.



An example of optional ROI in the SmoothImage_Gauss filter.

Optional Inputs vs Conditional Outputs

Besides optional inputs, there are also conditional outputs in some filters. These should not be confused. Optional inputs are designed for additional features of filters that can be turned on or off. Conditional outputs on the other hand create conditional connections, which result in conditional execution. The most important consequence of this model is that connections $T? \rightarrow T*$ are conditional, whereas $T? \rightarrow T?$ are not (where T is a type). The special value of an optional type is called *Auto*, whereas for conditional types it is *Nil*.

Connections

Read before: Introduction to Data Flow Programming.

Introduction

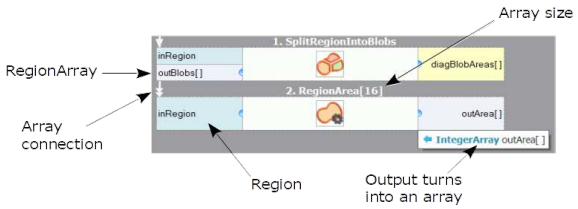
In general, connections in a program can be created between inputs and outputs of compatible types. This does not mean, however, that the types must be exactly equal. You can connect ports of different types if only it is possible to adapt the transferred data values by means of automatic conversions, array decompositions, loops or conditions. The power of this drag and drop programming model stems from the fact that the user just drags a connection and all the logic is added implicitly on the *do what I mean* basis.

Connection Logic

This is probably the most complicated part of the Aurora Vision programming model as it concentrates most of its computing flexibility. You will never define any connection logic explicitly as this is done by the application automatically, but still you will need to think a lot about it.

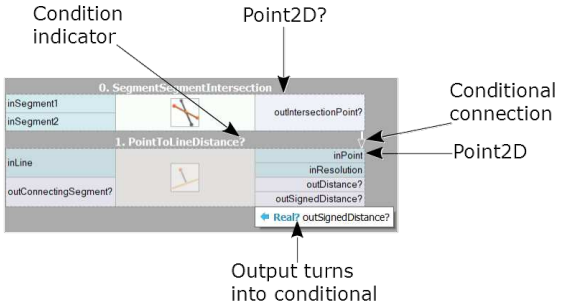
There are 5 basic types of connection logic:

1. **Basic Connection**
   T *T*
   This kind of connection appears between ports of equal types.
2. **Automatic Conversion**
   A *B*
   Automatic conversion is used when the user connects two ports having two different types, e.g. Integer and Real, and there is an appropriate filter in the Conversions category, e.g. IntegerToReal, that can be used to translate the values.
3. **Array Connection**
   TArray *T*
   This logic creates an implicit loop on a connection from an array to a scalar type, e.g. from RegionArray to Region. All the individual results are then merged into arrays and so the outputs of the second filter turn into arrays as well (see also: Arrays).



Example array connection (computing area of each blob).

4. **Singleton Connection**
   T *TArray*
   Conversely, when an output of a scalar type is connected to an array input, e.g. Point2D to Point2DArray, a singleton connection assures that the value will be converted into a single element array. This works only for some filters.
5. **Conditional Connection**
   T? *T*
   The last connection type, conditional connection, appears when conditional data (*T?*) is transferred to a non-conditional input (*T*). This causes the second filter to be executed conditionally depending on whether the value actually exists or not (then it is *NIL*). The filter outputs are changed into conditional outputs accordingly, assuring that consecutive filters will be executed conditionally as well. Conditional execution ends at a filter with a conditional input. More information on this topic can be found in Conditional Execution section.



Example conditional connection (intersection between two segments may not exist).

Mixed Connection Types

For maximum flexibility many combinations of the above logics can appear. Again, this is always inferred automatically.

6. **Conditional Connection with Conversion**
   A? *B*
   If the object exists, then it is converted and passed to the input port. If not, the filter is not invoked.
7. **Array Connection with Conversions**
   AArray *B*
   All elements of the input array are individually converted.
8. **Conditional Array Connections**
   TArray? *T*
   All array elements are processed or none – depending on the condition. The output is of a conditional array type.
9. **Conditional Array Connection with Conversions**
   AArray? *B*
   All array elements are processed and converted individually or none – depending on the condition.
10. **Array Connection with Conditional Elements**
    T?Array *T*
    The second filter is invoked conditionally for each array element and the output types turn into arrays of conditional elements.
11. **Array Connection with Conditional Elements and Conversions**
    A?Array *B*
    As above plus conversions.
12. **Singleton Connection with Conversion**
    A *BArray*
    A single-element array is created from a converted input value.
13. **Conditional Singleton Connection**
    T? *TArray*
    If the object exists, then it is transformed into a single-element array. If not, the second filter is not invoked.
14. **Conditional Singleton Connection with Conversion**
    A? *BArray*
    If the object exists, then it is converted and transformed into a single-element array. If not, the second filter is not invoked.

c++

Each individual connection logic is a realization of one control flow pattern from C/C++. For example, Array Connection with Conditional Elements corresponds to this basic code structure:

```
for (int i = 0; i < arr.Size(); ++i)
{
    if (arr[i] != atl::NIL)
    {
        ...
    }
}
```

Working with Types

As can be seen above, type definitions can have a strong influence on the program logic. For this reason you should pay a lot of attention to define all the arrays and conditionals correctly, especially when changing them in an existing program. This issues arises in the following places:

- on inputs and outputs of macrofilters
- on inputs and outputs of formula blocks
- when instantiating generic filters

In case of a mistake, you may get an incorrect connection in the program

(marked with red), but some tricky mistakes may remain not detected automatically.

Conditional Execution

Introduction

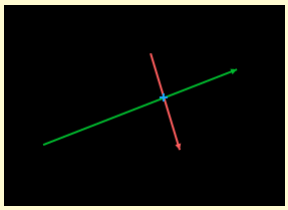
Conditional execution consists in executing a part of a program, or not, depending on whether some condition is met. There are two possible ways to achieve this in Aurora Vision Studio:

1. **Conditional Connections** – more simple; preferred when the program structure is linear and only some steps may be skipped in some circumstances.
2. **Variant Macrofilters** – more elegant; preferred when there are two or more alternative paths of execution.

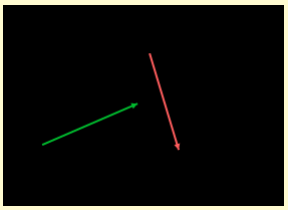
This section is devoted to the former, conditional connections.

Conditional Data

Before conditional connections can be discussed, we need to explain what conditional data is. There are some filters, for which it may be impossible to compute the output data in some situations. For example, the SegmentSegmentIntersection filter computes the intersection point of two line segments, but for some input segments the intersection may not exist. In such cases, the filter returns a special *Nil* value which indicates lack of data (usually read: "not detected").



Intersection exists – the filter returns a point.

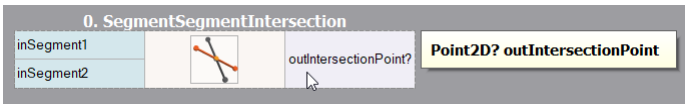


There is no intersection – the filter returns Nil value.

Other examples of filters producing conditional data include:

- ScanSingleEdge – will not find any edge on a plain image.
- DecodeBarcode – will not return any decoded string, if the checksum is incorrect.
- GigEVision_GrabImage_WithTimeout – will not return any image, if communication with the camera times out.
- MakeConditional – explicitly creates a conditional output; returns *Nil* when a specific condition is met.

Filter outputs that produce conditional data can be recognized by their types which have a question mark suffix. For example, the output type in SegmentSegmentIntersection is "Point2D?".



The conditional output of SegmentSegmentIntersection.

c++

The *Nil* value corresponds to the NULL pointer or a special value, such as -1. Conditional types, however, provide more type safety and clearly define, in which parts of the program special cases are anticipated.

Conditional Connections

Conditional connections (▷) appear when a conditional output is connected to a non-conditional input. The second filter is then said to be executed conditionally and has its output types changed to conditional as well. It is actually executed only if the incoming data is not *Nil*. Otherwise, i.e. when a *Nil* comes, the second filter also returns *Nil* on all of its outputs and it becomes subdued (darker) in the Program Editor.

An example: conditional execution of the PointToLineDistance filter.

As output types of conditionally executed filters change into conditional, an entire sequence of conditionally executed filters can easily be created. This sequence usually ends at a filter that accepts conditional data on its input.

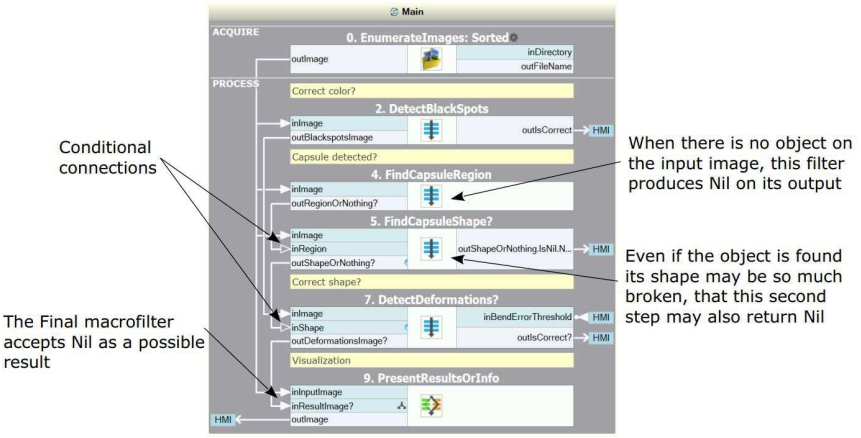
Resolving *Nil* Values

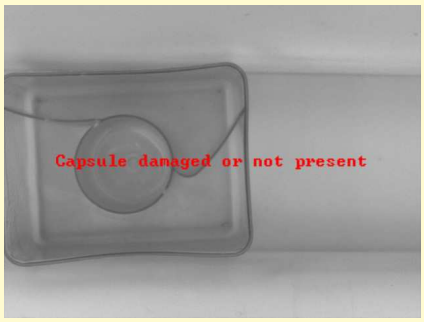
In general, if we have a conditional execution, a *Nil* value has to be anticipated, and it has to be resolved at some point. Here is a list of the most typical solutions:

- First of all, the *Nil* case can be ignored. Some part of the program will be not executed and it will not be signaled in any way.
  Example: Use this approach when your camera is running in a free-run mode and an inspection routine has to be executed only if an object has been detected.
- The most simple way to resolve *Nil* is by using the MergeDefault filter, which replaces the *Nil* with another value, specified in the program.
  Example: Most typically for industrial quality inspection systems, we can replace a *Nil* ("object not detected") with "NOT OK" (*False*) inspection result.
- A conditional value can also be transformed into a logical value by making use of *IsNil* and *IsNil.Not* property outputs. (Before version 4.8 the same could have been done with TestObjectNotNil and TestObjectNil filters).
- If conditional values appear in an array, e.g. because a filter with a conditional output is executed in array mode, then the RemoveNils filter can create a copy of this array with all *Nil*s removed.
- [Advanced] Sometimes we have a Task with conditional values at some point where *Nil* appears only in some iterations. If we are interested in returning the latest non-*Nil* value, a connection to a non-conditional macrofilter output will do just that. (Another option is to use the LastNotNil filter).
- Another possibility is to use a variant macrofilter with a conditional forking input and exactly two variants: "Nil" and "Default". The former will be executed for *Nil*, the latter for any other value.

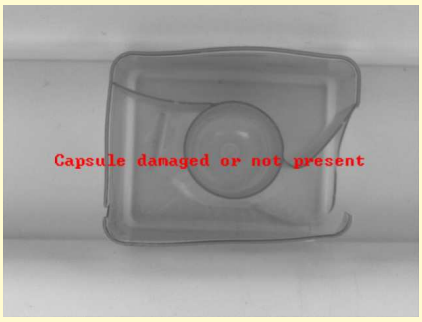
Example

Conditional execution can be illustrated with a simplified version of the "Capsules" example:





Capsule not detected by the FindCapsuleRegion step.



Capsule detected, but shape detection failed in the FindCapsuleShape step.

Other Alternatives to Conditional Execution

Classification

If the objects that have to be processed conditionally belong to an array, then it is advisable to use neither variant macrofilters nor conditional connections, but the classification filters from the Classify group. Each of these filters receives an array on its input and then splits it into several arrays with elements grouped accordingly to one of the three criterions.

Especially the ClassifyByRange filter is very useful when classifying objects on a basis of values of some extracted numeric features.

Conditional Choice

In the simplest cases it is also possible to compute a value conditionally with the filters from the Choose group (ChooseByPredicate, ChooseByRange, ChooseByCase) or with the ternary *<if> ? <then> : <else>* operator in the formula blocks.

Types of Filters

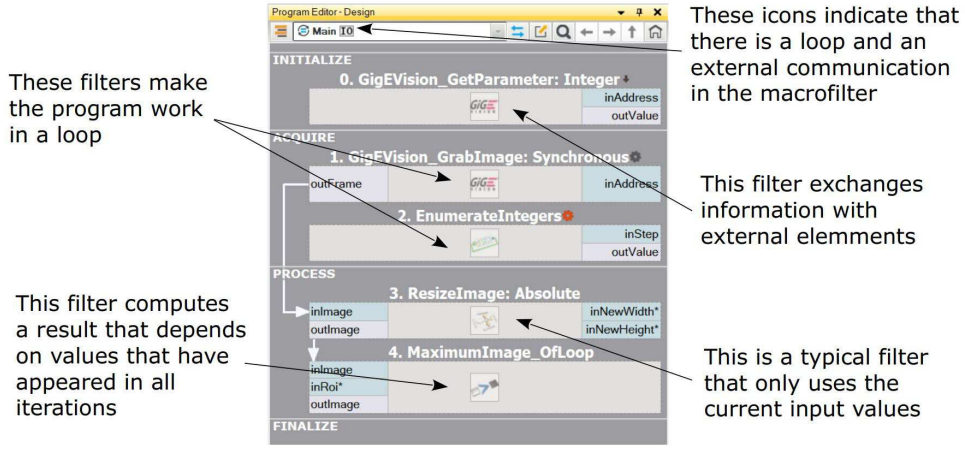
Read before: Introduction to Data Flow Programming.

Most filters are functional, i.e. their outputs depend solely on the input values. Some filters however can store information between consecutive iterations to compute aggregate results, communicate with external devices or generate a loop in the program. Depending on the side effects they cause, filters are classified as:

- **Pure Function**
  A filter which computes the output values always from and only from the input values. Typical examples: AddIntegers, SmoothImage_Gauss.
- **Loop Accumulator**
  A filter which stores information between consecutive iterations. Its output values can be computed from input values that have appeared in this iteration as well as in all previous iterations. Typical examples: AccumulateElements, AddIntegers_OfLoop.
- **I/O Function**
  A filter which exchanges information with elements external to the program. Typical examples: GenICam_SetDigitalOutputs, DAQmx_ConfigureTiming.
- **Loop Generator**
  A filter which inserted to a program makes it work in a loop. Typical examples: EnumerateIntegers, Loop.
- **I/O Loop Generator**
  A filter which inserted to a program makes it work in a loop and, like an I/O Function, exchanges information with external elements. Typical examples: WebCamera_GrabImage, GigEVision_GrabImage.

Visualization

The icon depicting the filter type is displayed in the Program Editor at the top-left corner of the filter's icon:



Visualization of the filter types in the Program Editor.

Remark: The state of *Loop Accumulators* and *Loop Generators* actually belongs to the containing Task macrofilter. The filters are reset only when the execution enters the Task and then it is updated in all iterations.

Constraints

Here is a list of constraints related to some of the filter types:

- Loop generators, loop accumulators and I/O functions cannot be executed in the array mode.
- Loop generators, loop accumulators and I/O functions cannot be re-executed when the program is paused.

Generic Filters

Read before: Introduction to Data Flow Programming.

Introduction

Most of the filters available in Aurora Vision Studio have clearly defined types of inputs and outputs – they are applicable only to data of that specific types. There is however a range of operations that could be applied to data of many different types – for instance the ArraySize filter could work with arrays of any type of the items, and the TestObjectEqualTo filter could compare any two objects having the same type. For this purpose there are *generic filters* that can be *instantiated* for many different types.

c++

Generic filters are very similar to C++ function templates.

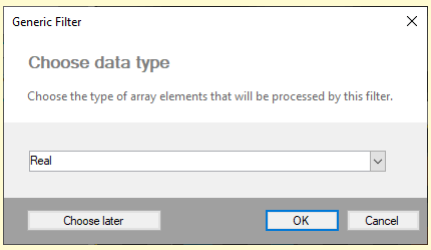
Type Definitions

Each generic filter is defined using a type variable *T* that represents any valid type. For instance, the GetArrayElement filter has the following ports:

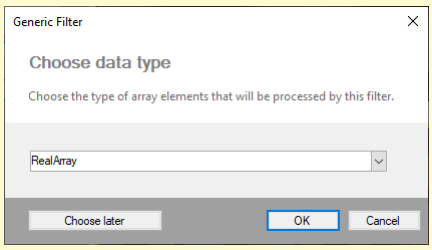
	Port	Type	Description
➡	inArray	**T**Array	Input array
➡	inIndex	Integer	Index within the array
⬅	outValue	**T**	Element from the array

Instantiation

When a generic filter is added to the program, the user has to choose which type should be used as the *T* type for this filter instance. This process is called *instantiation*. There are two variants of the dialog window that is used for choosing the type. They differ only in the message that appears:



This window appears for filters related to arrays, e.g. GetArrayElement or JoinArrays. The user has to choose the type of the array elements.

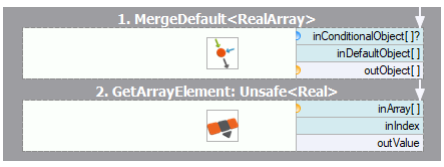


This window appears for filters not related to arrays, e.g. MergeDefault. The user has to choose the type of the entire object.

For example, if the GetArrayElement filter is added to the program and the user chooses the *T* variable to represent the Real type, then the interface of the filter takes the following form:

	Port	Type	Description
➡	inArray	**Real**Array	Input array
➡	inIndex	Integer	Index within the array
⬅	outValue	**Real**	Element from the array

After instantiation the filter appears in the Program Editor with the instantiation type specified by the name of the filter:



An example of two generic filters in the Program Editor: MergeDefault instantiated for replacing Nils in arrays of real numbers and GetArrayElement instantiated for getting an single element from them. Please note that the type parameter specified is different.

A common mistake is to provide the type of the entire object, when only a type of array element is needed. For example, when instantiating the GetArrayElement filter intended for accessing arrays of reals, enter the Real type, which is the type of an element, but NOT the RealArray type, which is the type of the entire object.

Macrofilters

Read before: Introduction to Data Flow Programming.

For information about creating macrofilter in the user interface, please refer to Creating Macrofilters.

This article discusses more advanced details on macrofilter construction and operation.

Macrofilter Structures

There are four possible structures of macrofilters which play four different roles in the programs:

- **Steps**
- **Variant Steps**
- **Tasks**
- **Workers**

Steps

Step is the most basic macrofilter structure. Its main purpose is to make programs clean and organized.

A macrofilter of this structure simply separates a sequence of several filters that can be used as one block in many places of the program. It works exactly as if its filters were expanded in the place of each macrofilter's instance. Consequently:

- The state of the contained filters and registers is preserved between consecutive invocations.
- If a loop generator is inserted to a *Step* macrofilter, then this step becomes a loop generator as the whole.
- If a loop accumulator is inserted to a *Step* macrofilter, then this step becomes a loop accumulator as the whole.

Variant Steps

Variant macrofilters are similar to *Steps*, but they can have multiple alternative execution paths. Each of the paths is called a *variant*. At each invocation exactly one of the variants is executed – the one that is chosen depends on the value of the *forking* input or register (depicted with the icon), which is compared against the *labels* of the variants.

The type of the forking port and the labels can be: Bool, Integer, String or any enumeration type. Furthermore, any conditional or optional types can be used, and then a "Nil" variant will be available. This can be very useful for forking the program execution on the basis on whether some analysis was successful (there exists a proper value) or not (Nil).

All variants share a single external interface – inputs, outputs and also registers are the same. In consecutive iterations different variants might be executed, but they can exchange information internally through the local registers of this macrofilter. From outside a variant step looks like any other filter.

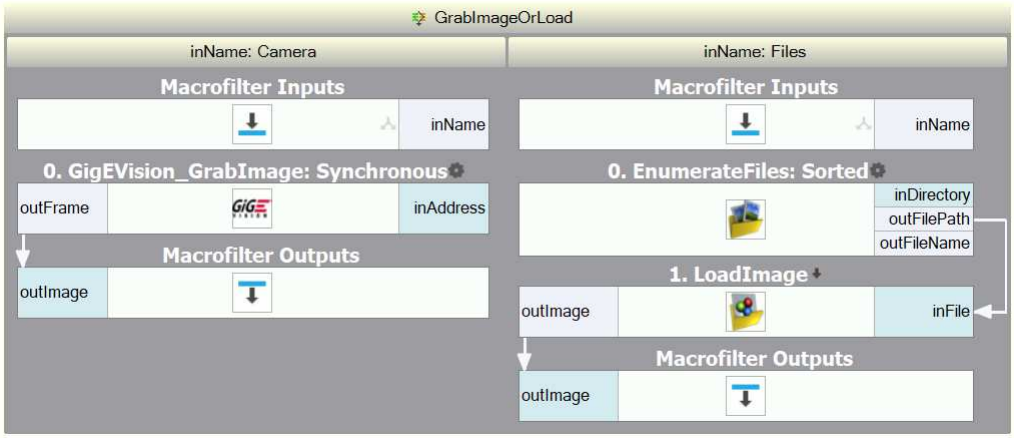
Here are some example applications of variant macrofilters:

- When some part of the program being created can have multiple alternative implementations, which we want to be chosen by the end-user. For example, there might be two different methods of detecting an object, having different trade-offs. The user might be able to choose one of the methods through a combo-box in the HMI or by changing a value in a configuration file.
- When there is an object detection step which can produce several classes of detected objects, and the next step of object recognition should depend on the detected class.
- When an inspection algorithm can have multiple results (most typically: OK and NOK) and we want to execute different communication or visualization filters for each of the cases.
- Finite State Machines. *Variant Step* macrofilters correspond to the *switch* statement in C++.

Example 1

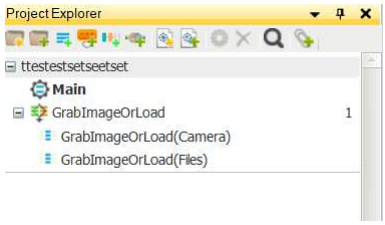
A *Variant Step* can be used to create a subprogram encapsulating image acquisition that will have two options controlled with a String-typed input:

1. Variant "Files": Loading images from files of some directory.
2. Variant "Camera": Grabbing images from a camera.



The two variants of the GrabImageOrLoad macrofilter.

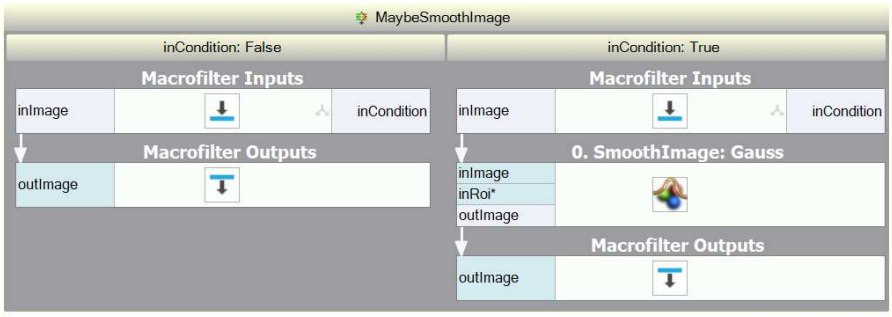
In the Project Explorer, the variants will be accessible after expanding the macrofilter item:



A variant step macrofilter in the Project Explorer

Example 2

Another application of *Variant Step* macrofilters is in creating optional data processing steps. For example, there may be a configurable image smoothing filter in the program and the user may be able to switch it on or off through a CheckBox in the HMI. For that we create a macrofilter like this:



A variant step macrofilter for optional image preprocessing.

NOTE: Using a variant step here is also important due to performance reasons. Even if we set **inStdDev** to 0, the SmoothImage_Gauss filter will still need to copy data from input to output. Also filters such as ChooseByPredicate or MergeDefault perform a full copy. On the other hand, when we use macrofilters, internal connections from macrofilter inputs to macrofilter outputs do not copy data (but they link them). If this is about heavy types of data such as images, the performance benefit can be significant.

Tasks

A *Task* macrofilter is much more than a separated sequence of filters. It is a logical program unit realizing a complete computing process. It can be used as a subprogram, and is usually required in more advanced situations, for example:

- When we need an explicit nested loop in the program which cannot be easily achieved with array connections.
- When we need to perform some computations before the main program loop starts.

Execution Process

The most important difference between *Tasks* and *Steps* is that a *Task* can perform many iterations of its filter sequence before it finishes its own single invocation. This filter sequence gets executed repeatedly until one of the contained loop generating filters signals an end-of-sequence. If there are no loop generators at all, the task will execute exactly one iteration. Then, when the program execution exits the task (returning to the parent macrofilter) the state of the task's filters and registers is destroyed (which includes closing connections with related I/O devices).

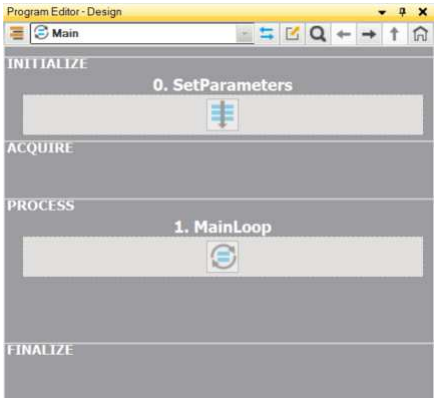
What is also very important, *Tasks* define the notion of an *iteration* of the program. During program execution there is always a single task that is the most nested and currently being executed. We call it *the current* task. When the execution process comes to the end of this task, we say that an iteration of a program has finished. When the user clicks the *Iterate (F6)* button, execution continues to the end of the current task. Also the *Update Data Previews Once an Iteration* refers to the same notion.

c++

Task macrofilters can be considered an equivalent of C/C++ functions with a single *while* loop.

Example: Initial Computations before the Main Loop

A quite common case is when some filters have to be executed before the main loop of the program starts. Typical examples of such initial computations are: setting some camera parameters, establishing some I/O communication or loading some model definitions for external files. You can enclose all these initial operations in a macrofilter and place it in *Initialize* section. Programs of this kind should have the following standard structure consisting of two parts:



One of possible program structures.

When the program is started, all the filters and macrofilters in the *Initialize* section will be executed once and then the loop in the *Acquire* and *Process* section will be executed until the program ends.

Worker Tasks

The main purpose of *Worker Tasks* is to allow users to process data parallelly. They also make several other new things possible:

- Parallel receiving and processing of data coming from different, not synchronized, devices
- Parallel control of output devices that is not dependent on the cycle of the program (e.g. light up a diode every 500ms)
- Dividing our program, splitting parts of the program that need to be processed in real time from the ones that can wait for processing
- Flawless processing of HMI events
- Having additional programs in a project for unit testing of our algorithms.
- Having additional programs in a project for precomputing data that will be later used in the main program.

The number of *Worker Tasks* available for use is dependent on the license factor, however every program will contain at least one *Worker Task*, which will act as "Main".

Creating a Worker Task

Due to its special use you cannot create a *Worker Task* by pressing Ctrl+Space shortcut, or directly in the program editor. The only option to do so is to make one in the Project Explorer, as shown in the Creating Macrofilters article.

Queues

Queues play an important role in communication and synchronization between different *Worker Tasks* in the program. They may pass most types of data, as well as *Arrays* between the threads. Operations on the queues are atomic meaning that once the processor starts processing them it cannot be interrupted or affected in any way. So in case that several threads would like to perform an operation, some of them would have to wait.

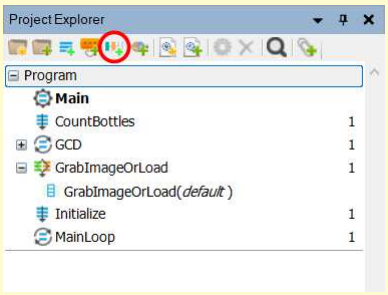
Please note, that different arrays are not synchronized with each other. To process complex data or pass data that needs to be synchronized (like image and its information) you should use user types. We also highly advise not to use queues as an replacement of global parameters.

Creating Queues

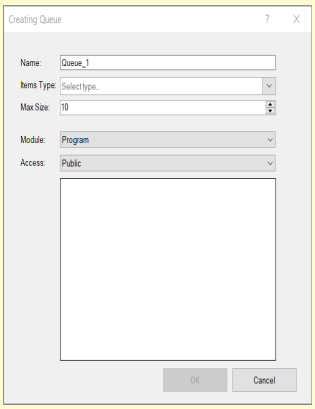
To create a new queue in the Project Explorer click the *Create New Queue...* icon. A new window will appear, allowing you to select the name and parameters of the new queue such as:

- Items Type that specifies the type
- Maximum size of the queue
- Module to which it belongs
- Access - public or private

of data passed by the queue



Creating queues in the Project Explorer.



Selecting the name and the structure of a new queue.

Queue operations

In order to perform queue operations and fully manage queues in *Worker Tasks*, you can use several filters available in our software, namely:

- Queue_Pop - takes value from the queue without copying it. Waits infinitely if the queue is empty. This operation only reads data and does not copy it, so it is performed instantly.
- Queue_Pop_Timeout - takes value from the queue without copying it. Waits for the time specified in Timeout if the queue is empty. This operation only reads data and does not copy it, so it is performed instantly.
- Queue_Peek - returns specified element of the queue without removing it. Waits infinitely if the queue is empty.
- Queue_Peek_Timeout - returns specified element of the queue without removing it. Waits for the time specified in Timeout if the queue is empty.
- Queue_Push - adds element to the queue. This operation copies data, so it may take more time compared to the others.
- Queue_Size - returns the size of the queue.
- Queue_Flush - clears the queue. Does not block data flow.

Macrofilters Ports

Inputs

Macrofilter inputs are set before execution of the macrofilter is started. In the case of *Task* macrofilters the inputs do not change values in consecutive iterations.

Outputs

Macrofilter outputs can be set from connections with filters or with global parameters as well as from constant values defined directly in the outputs.

In the case of *Tasks* (with many iterations) the value of a connected output is equal to the value that was computed the latest. One special case to be considered is when there is a loop generator that exits in the very first iteration. As there is no "the latest" value in this case, the output's default value is used instead. This default value is part of the output's definition. Furthermore, if there is a conditional connection at an output, Nil values cause the previously computed value to be preserved.

Registers

Registers make it possible to pass information between consecutive iterations of a *Task*. They can also be defined locally in *Steps* and in *Variant Steps*, but their values will still be set exactly in the same way as if they belonged to the parent *Task*:

- Register values are initialized to their defaults when the *Task* starts.
- Register values are changed at the end of each iteration.

In case the of variant macrofilters, register values for the next iteration are defined separately in each variant. Very often some registers should just preserve their values in most of the variants. This can be done by creating long connections between the *prev* and *next* ports, but usually a more convenient way is to disable the *next* port in some variants.

Please note, that in many cases it is possible to use accumulating filters (e.g. AccumulateElements, AddIntegers_OfLoop and other OfLoop) instead of registers. Each of these filters has an internal state that stores information between consecutive invocations and does not require explicit connections for that. Thus, as a method this is simpler and more readable, it should be preferred. Registers, however, are more general.

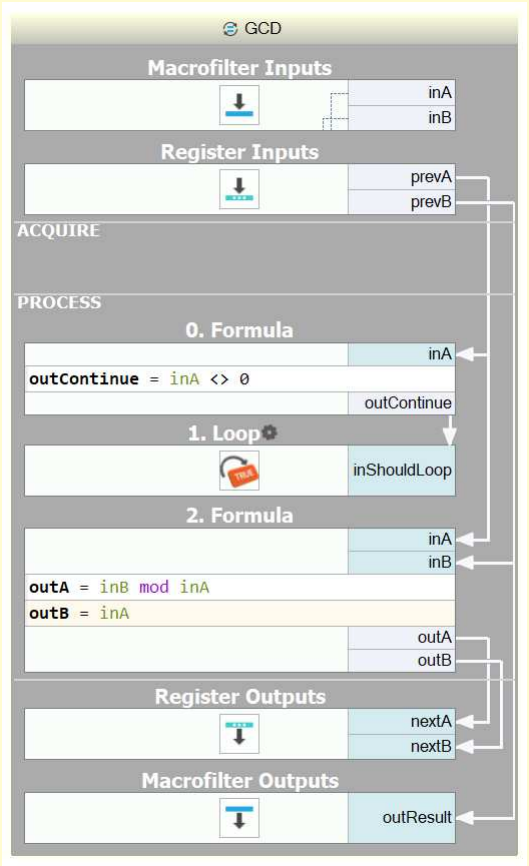
Yet another alternative to macrofilter registers is the *prev* operator in the formula blocks.

c++

Registers correspond to variables in C/C++. In Aurora Vision Studio, however, registers follow the *single assignment rule*, which is typical for functional and data-flow programming languages. It says that a variable can be assigned at most once in each iteration. Programs created this way are much easier to understand and analyze.

Example: Computing Greatest Common Denominator

With *Task* macrofilters and registers it is possible to create very complex algorithms. For example, here is a comparison of how the Greatest Common Denominator function can be implemented in Aurora Vision Studio and in C/C++:



A task with registers computing Greatest Common Denominator of inA and inB.

```
int gcd(int inA, int inB)
    {
  int a = inA;
  int b = inB;
 while (a != 0)
    {
   int tmp = a;
    a = b % a;
     b = tmp;
    }
  return b;
    }
```

Computing Greatest Common Denominator in C++.

Sequence of Filter Execution

It can be assumed that filters are always executed from top to bottom. Internally the order of filter execution can be changed and even some filters may operate in parallel to take advantage of multi-core processors, but this happens only if the top-down execution produces exactly the same results. Specifically, all I/O functions will never be parallelized, so if one I/O operation is above another, it will always be executed first.

Formulas

Introduction

A formula block enables to write down operations on basic arithmetic and logic types in a concise textual form. Its main advantage is the ability to simplify programs, in which calculations may require use of a big number of separate filters with simple mathematical functions.

Formula notations consist of a sequence of operators, arguments and functions and are founded on basic rules of mathematical algebra. Their objective in Aurora Vision Studio is similar to formulas in spreadsheet programs or expressions in popular scripting languages.

A formula block can have any number of inputs and outputs defined by the user. These ports can be of any unrelated types, including various types in one block. Input values are used as arguments (identified by inputs names) in arithmetic formulas. There is a separate formula assigned to each output of a block, the role of such formula is to calculate the value held by given output. Once calculated, a value from output can be used many times in formulas of outputs which are located at a lower position in a block description.

Each of arguments used in formulas has a strictly defined data type, which is known and verified during formula initialization (so called static typing). Types of input arguments arise from explicitly defined block port types. These types are then propagated by operations performed on arguments. Finally, a formula value is converted to the explicitly defined type assigned to a formula output. Compatibility of these types is verified during program initialization (before the first execution of formula block), all incompatibilities are reported as errors.

Formula examples:

```
outValue = inValue + 10
outValue = inValue1 * 0.3 + inValue2 * 0.7
outValue = (inValue - 1) / 3
outValue = max(inValue1, inValue2) / min(inValue1, inValue2)
outRangePos = (inValue >= -5 and inValue <= 10) ? (inValue - -5) / 15 : Nil
outArea = inBox.Width * inBox.Height
```

Data types

Blocks of arithmetic-logic formulas can pass any types from inputs to outputs. The data types mentioned below are recognized by operators and functions available in formulas and therefore they can take part in processing their values.

Data type	Description
Integer	Arithmetic type of integral values in the range of -2,147,483,648...2,147,483,647, it can be used with arithmetic and comparison operators.
Integer? Integer*	Conditional or optional arithmetic type of integral values, it can additionally take value of Nil (empty value). It can be used everywhere, where the type of Integer is accepted. It (if an operator or a function doesn't assume different behavior) causes that the result of an operation performed on it is also a conditional value. It means that an occurrence of Nil value causes that operation execution is aborted and Nil value is returned as operation result.
Real	Real number (floating-point), it can be used with arithmetic and comparison operators.
Real? Real*	Conditional or optional real value, it can additionally take value of Nil (empty value). Similarly to Integer? it can be used everywhere, where the type of Real is accepted causing conditional execution of operators.
Long	Arithmetic type of integral values in the range of -9,223,372,036,854,775,808...9,223,372,036,854,775,807, it can be used with arithmetic and comparison operators.
Long? Long*	Conditional or optional equivalent of Long type, it can additionally take value of Nil (empty value). Similarly to Integer? it can be used everywhere, where the type of Long is accepted causing conditional execution of operators.
Double	Double precision floating-point number, it can be used with arithmetic and comparison operators.
Double? Double*	Conditional or optional equivalent of Double type, it can additionally take value of Nil (empty value). Similarly to Integer? it can be used everywhere, where the type of Double is accepted causing conditional execution of operators.
Bool	Logic type, takes one of the two values: true or false. It can be used with logic and comparison operators or as a condition argument. It's also a result of comparison operators.
Bool? Bool*	Conditional or optional logic type, it can additionally take value of Nil (empty value). It can be used everywhere, where the type of Bool is accepted. It (if an operator or a function doesn't assume different behavior) causes that the result of an operation performed on it is also a conditional value. It means that an occurrence of Nil value causes that operation execution is aborted and Nil value is returned as operation result.
String	Textual type accepting character strings of dynamic length. Text length (in characters) can be determined using String type built-in `.Length` property.
String? String*	Conditional or optional textual type, it can additionally take value of Nil (empty value). It can be used everywhere, where the type of String is accepted. It (if an operator or a function doesn't assume different behavior) causes that the result of an operation performed on it is also a conditional value. It means that an occurrence of Nil value causes that operation execution is aborted and Nil value is returned as operation result.
enum	Any of the enumeration types declared in the type system. An enumeration type defines a limited group of items which can be assigned to a value of given type. Every one of these items (constants) is identified by a unique name among the given enumeration type (e.g. enumeration type *SortingOrder* has two possible values: *Ascending* and *Descending*). As part of formulas it is possible to create a value of enumeration type and to compare two values of the same enumeration type.
enum? *enum**	Any conditional or optional enumeration type, instead of a constant, it can additionally take value of Nil (empty value).
structure	Any structure - a composition of a number of values of potentially different types in one object. Within formulas, it is possible to read separately structure elements and to perform further operations on them within their types.
structure? *structure**	Any conditional or optional structure, instead of field composition, it can additionally take value of Nil (empty value). It is possible to get access to fields of conditional structures on the same basis as in the case of normal structures. However, a field to be read is also of conditional type. Reading fields of a conditional structure of value Nil returns as a result Nil value as well.
<*T*>Array	Any array of values. Within formulas, it is possible to determine array size, read and perform further operations on array elements. Array size can be determined by `.Count` property which is built-in in array types.
<*T*>Array? <*T*>Array*	Any conditional or optional array, instead of elements sequence it can additionally take value of Nil (empty value). It is possible to read sizes and elements of conditional arrays on the same basis as in the case of normal arrays. In this case, values to be read are also of conditional type. Reading an element from a conditional array of Nil value returns as a result Nil value as well.

Literal constants

You can place constant values in a formula by writing them down literally in the specified format.

	Data type	Example
Integer value in decimal notation	Integer	0 150 -10000
	Long	0L 150L -10000L
Integer value in hexadecimal notation	Integer	0xa1c 0x100 0xFFFF
	Long	0xa1cL 0x100L 0xFFFFL
Real value in decimal notation	Real	0.0 0.5 100.125 -2.75
	Double	0.0d 0.5d 100.125d -2.75d
Real value in scientific notation	Real	1e10 1.5e-3 -5e7
	Double	1e10d 1.5e-3d -5e7d
Text	String	"Hello world!" "First line\nSecond line" "Text with \"quote\" inside."

Textual constants

Textual constants are a way to insert data of type String into formula. They can be used for example in comparison with input data or in conditional generating of proper text to output of formula block. Textual constants are formatted by placing some text in quotation marks, e.g.: `"This is text"`; the inner text (without quotation marks) becomes value of a constant.

In case you wish to include in a constant characters unavailable in a formula or characters which are an active part of a formula (e.g. quotation marks), it is necessary to enter the desired character using escape sequence. It consists of backslash character (\) and proper character code (in this case backslash character becomes active and if you wish to enter it as a normal character it requires the use of escape sequence too). It is possible to use the following predefined special characters:

- `\n` - New line, ASCII: 10
- `\r` - Carriage return, ASCII: 13
- `\t` - Horizontal tabulation, ASCII: 9
- `\'` - Apostrophe, ASCII: 39
- `\"` - Quotation mark, ASCII: 34
- `\\` - Backslash, ASCII: 34
- `\v` - Vertical tabulation, ASCII: 11
- `\a` - "Bell", ASCII: 7
- `\b` - "Backspace", ASCII: 8
- `\f` - "Form feed", ASCII: 12

Examples:

```
"This text \"is quoted\""
"This text\nis in new line"
"In Microsoft Windows this text:\r\nis in new line."
"c:\\Users\\John\\"
```

Other characters can be obtained by entering their ASCII code in hexadecimal format, by `\x??` sequence, e.g.: `"send \x06 ok"` formats text containing a character of value 6 in ASCII code, and `"\xce"` formats a character of value 206 in ASCII code (hex: ce).

Predefined constant values

As part of formula, it is possible to obtain access to one of the below-mentioned predefined constant values. In order to use a constant value as operation argument, it is required to enter the name of this constant value.

Name	Data type	Description
true	Bool	Positive logic value.
false	Bool	Negative logic value.
Nil	*Null*	Special value of conditional and optional types, it represents an empty value (no value). This constant doesn't has its own data type (it is represented by the special type of "Null"). Its type is automatically converted to conditional types as part of executed operations.
pi	Real	Real value, it is the approximation of π mathematical constant.
e	Real	Real value, it is the approximation of e mathematical constant.
inf	Real	Represents positive infinity. It is a special value of type Real that is greater than any other value possible to be represented by the type Real. Note that negative infinity can be achieved by "-inf" notation.

Enumeration constants

There are enumeration constants defined in the type system. They have their own types and among these types a given constant chooses one of a few possibilities. E.g. a filter for array sorting takes a parameter which defines the order of sorting. This parameter is of "SortingOrder" enumeration type. As part of this type, you can choose one of two possibilities identified by "Ascending" and "Descending" constants.

As part of formulas, it is possible to enter any enumeration constant into formula using the following syntax:

```
<enum name>.<item name>
```

Full name of a constant used in a formula consists of enumeration type name, of a dot and of name of one of the possible options available for a given type, e.g.:

```
SortingOrder.Descending
BayerType.BG
GradientOperator.Gauss
```

Arithmetic conversions

Binary arithmetic operators are executed as part of one defined common type. In case when the types of both arguments are not identical, an attempt to convert them to one common type is performed, on this type the operation is then executed. Such conversion is called arithmetic conversion and when it's possible to carry it out, the types of arguments are called compatible.

Arithmetic conversion is performed as part of the following principles:

Conditions	Performed conversions
Two different numeric types.	The argument of type with smaller range or precision is converted to the type of second argument (with greater range or precision). Possible argument type conversions includes: Integer to Real, Integer to Long, Integer to Double and Real to Double. The same principle is used in case of conditional counterparts of mentioned types.
One of the types is conditional.	The second type is converted to conditional type.
Two different conditional types.	Arithmetic conversion is performed as part of types on which arguments conditional types are based. As the result, the types remain conditional.
One of the arguments is Nil.	The second argument is converted to conditional type (if it's not already conditional), Nil value is converted to the conditional type of the second argument.

Automatic implicit conversions

When some operation requires an argument of certain type and, instead of it, an argument of different type is used, implicit conversion of such argument type is attempted using the below-mentioned principles.

Conversion	Description
Integer → Real Integer → Double	Conversion of integer to floating-point value. As part of such conversion loss of precision may occur. As the result of it, instead of accurate number representation, an approximated floating-point value is returned. Such conversion is also possible when it comes to conditional counterparts of given types.
Integer → Long Real → Double	Conversion to equivalent type with greater range or precision. No data is lost as a result of such conversion. Such conversion is also possible when it comes to conditional counterparts of given types.
T → *T?*	Assigning conditionality. Any non-conditional type can be converted to its conditional counterpart.
Nil → *T?*	Assigning type to a Nil value. Nil value can be converted to any conditional type. As the result of it, you get an empty value of given type.
*T** → *T?*	As part of formulas, optional and conditional types are handled the same way (in formulas there is no defined reaction for automatic values, they only pass data). If given operation requires argument of conditional type, then it also can, basing on the same principles, take argument of optional type.

Operators

As part of formulas, it is possible to use the following operators:

Negation operator:	`-`
Identity operator:	`+`
Two's complement:	`~`
Logic negation:	`not`
Multiplicative operators:	`* / div mod`
Additive operators:	`+ -`
Bit shift:	`>> <<`
Bitwise operators:	`& | ^`
Logic operators:	`and or xor`
Comparison operator:	`< <= > >=`
Equality testing:	`== <>`
Condition operator:	`if-then-else ?:`
Merge with default value:	`??`
Function call:	*`function`*`()`
Field read:	`.`
Element read:	`[`*`i`*`]`
Explicit array processing:	`[]`
Array creation operator:	`{}`
Previous value of output:	`prev()`
Global parameter read:	`::`

Unary operators

Negation operator (-)

```
- expression
```

Data types: Integer, Real, Long, Double, Vector2D, Vector3D, Matrix

Returns numeric value with negated sign, vector with same length but opposite direction or matrix with all its elements negated.

Identity operator (+)

```
+ expression
```

Data types: Integer, Real, Long, Double, Vector2D, Vector3D, Matrix

Returns value without any changes.

Two's complement operator (~)

```
~ expression
```

Data types: Integer, Long

Returns integer value with negated bits (binary complement).

Logic negation operator (`not`)

```
not expression
```

Data types: Bool

Returns logic value of an opposite state.

Binary operators

Multiplication operator (`*`)

```
expression * expression
```

Data types:

*Integer * Integer*	*→ Integer*
*Long * Long*	*→ Long*
*Real * Real*	*→ Real*
*Double * Double*	*→ Double*
*Vector2D * Vector2D*	*→ Vector2D*
*Vector2D * Real*	*→ Vector2D*
*Real * Vector2D*	*→ Vector2D*
*Vector3D * Vector3D*	*→ Vector3D*
*Vector3D * Real*	*→ Vector3D*
*Real * Vector3D*	*→ Vector3D*
*Matrix * Matrix*	*→ Matrix*
*Matrix * Real*	*→ Matrix*
*Real * Matrix*	*→ Matrix*
*Matrix * Vector2D*	*→ Vector2D*
*Matrix * Point2D*	*→ Point2D*
*Matrix * Vector3D*	*→ Vector3D*
*Matrix * Point3D*	*→ Point3D*

Returns product of two numeric arguments, element-by-element product of two vectors, vector scaled by a factor, multiplies two matrices, multiplies matrix elements by a scalar, or multiplies matrix by a vector or point coordinates.

When multiplying matrix by 2D vector (or 2D point coordinates), a general transformation 2x2 or 3x3 matrix is expected. Multiplication is performed by assuming that the source vector is a 2x1 matrix and that the resulting 1x2 matrix form the new vector or point coordinates:

$$\begin{bmatrix} t_{1,1} & t_{1,2} \\ t_{2,1} & t_{2,2} \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = [x' \quad y']$$

When 3x3 matrix is used the multiplication is performed by assuming that the source vector is a 3x1 matrix (expanded with 1 to three elements) and by normalizing the resulting 1x3 matrix back to two element vector or point coordinates:

$$\begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ t_{3,1} & t_{3,2} & t_{3,3} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = [x' \quad y' \quad w'] \rightarrow \left[\frac{x'}{w'} \quad \frac{y'}{w'}\right]$$

By analogy, when multiplying matrix by 3D vector (or 3D point coordinates) a general transformation 3x3 or 4x4 matrix is expected.

Real division operator (`/`)

```
expression / expression
```

Data types: Real, Double, Vector2D, Vector3D

Returns quotient of two real numbers, element-by-element quotient of two vectors, or vector which length was divided by scalar factor.

Integer division operator (`div`)

```
expression div expression
```

Data types: Integer, Long

Returns quotient of two integer numbers without residue. A Domain Error appears during program runtime when second argument is equal to zero.

Modulo operator (`mod`)

```
expression mod expression
```

Data types: Integer, Long

Returns residue from division of two integer numbers. A Domain Error appears during program runtime when second argument is equal to zero.

Addition operator (`+`)

```
expression + expression
```

Data types:

Integer + Integer	*→ Integer*
Long + Long	*→ Long*
Real + Real	*→ Real*
Double + Double	*→ Double*
Vector2D + Vector2D	*→ Vector2D*
Point2D + Vector2D	*→ Point2D*
Vector3D + Vector3D	*→ Vector3D*
Point3D + Vector3D	*→ Point3D*
Matrix + Matrix	*→ Matrix*
String + String	*→ String*

Returns sum of two numeric values, sum of two vectors, moves a point by a vector or adds two matrices element-by-element.

In case of using it with String type, it performs text concatenation and returns connected string.

Subtraction operator (-)

```
expression - expression
```

Data types:

Integer - Integer	→ Integer
Long - Long	→ Long
Real - Real	→ Real
Double - Double	→ Double
Vector2D - Vector2D	→ Vector2D
Point2D - Vector2D	→ Point2D
Vector3D - Vector3D	→ Vector3D
Point3D - Vector3D	→ Point3D
Matrix - Matrix	→ Matrix

Returns difference of two numeric values, subtracts two vectors, moves a point backwards by a vector or subtracts two matrices element-by-element.

Bit right-shift (>>)

```
expression >> expression
```

Data types: Integer, Long

Returns value of the first argument, in which bits of its binary representation have been shifted right (in the direction of the less meaningful bits) by the number of positions defined by the second argument. After the shift, bits which exceed the type range on the right side are ignored. The lacking bits on the left side are filled in with zeros.

A Domain Error appears during program runtime when second argument is negative. In case of shift value larger than 31 for Integer type, or larger than 63 for Long type, the value of 0 will be returned.

Bit left-shift (<<)

```
expression << expression
```

Data types: Integer, Long

Returns value of the first argument, in which bits of its binary representation have been shifted left (in the direction of the more meaningful bits) by the number of positions defined by the second argument. After the shift, bits which exceed the type range on the left side are ignored. The lacking bits on the right side are filled in with zeros.

A Domain Error appears during program runtime when second argument is negative. In case of shift value larger than 31 for Integer type, or larger than 63 for Long type, the value of 0 will be returned.

Bitwise AND (&)

```
expression & expression
```

Data types: Integer, Long

Returns value of bitwise product of two integer numbers. This operator compares separately each pair of corresponding bits of two arguments. If values of both bits equal 1, then the result bit also equals 1. If value of at least one bit equals 0, then the result bit equals 0.

Bitwise OR (|)

```
expression | expression
```

Data types: Integer, Long

Returns value of bitwise sum of two integer numbers. This operator compares separately each pair of corresponding bits of two arguments. If value of at least one bit equals 1, the result bit also equals 1. If values of both bits equal 0, then the result bit equals 0.

Bitwise exclusive OR (^)

```
expression ^ expression
```

Data types: Integer, Long

Returns value of bitwise exclusive sum of two integer numbers. This operator compares separately each pair of corresponding bits of two arguments. If two bits are equal, the result bit equals 0. If two bits are not equal, then the result equals 1.

Logic AND (and)

```
expression and expression
```

Data types: Bool

This operator merges two logic arguments. If both arguments equal true, then the result also equals true. If at least one of the arguments equals false, then the result equals false.

Logic OR (or)

```
expression or expression
```

Data types: Bool

This operator merges two logic arguments. If at least one of the arguments equals true, then the result equals true. If both arguments equal false, then the result also equals false.

Logic exclusive OR (xor)

```
expression xor expression
```

Data types: Bool

This operator merges two logic arguments. If the arguments are unequal, then the result equals true. If the arguments are equal, then the result equals false.

Less than (<)

```
expression < expression
```

Data types: Integer, Real, Long, Double, String

Result type: Bool

This operator compares two values. It returns value of true only if value of the first argument is smaller than value of the second argument.

In case of using it with String type, the operator performs lexicographic text comparison.

Less than or equal to (<=)

```
expression <= expression
```

Data types: Integer, Real, Long, Double, String

Result type: Bool

This operator compares two values. It returns value of true only if value of the first argument is smaller than or equal to value of the second argument.

In case of using it with String type, the operator performs lexicographic text comparison.

Greater than (>)

```
expression > expression
```

Data types: Integer, Real, Long, Double, String

Result type: Bool

This operator compares two values. It returns value of true only if value of the first argument is larger than value of the second argument.

In case of using it with String type, the operator performs lexicographic text comparison.

Greater than or equal to (>=)

```
expression >= expression
```

Data types: Integer, Real, Long, Double, String

Result type: Bool

This operator compares two values. It returns value of true only if value of the first argument is larger than or equal to value of the second argument.

In case of using it with String type, the operator performs lexicographic text comparison.

Equality test (==)

```
expression == expression
```

Data types: *any compatible types*

Result type: Bool

This operator compares two arguments. If their values are equal, it returns the value of true, otherwise it returns the value of false. It's possible to determine the equality not only between compatible primitive types, but also between objects of any two same types (e.g. structures).

Please note that comparing values of type Real or Double (or structures with those types) may be tricky. It is caused by the fact that very small (unnoticeable) differences in values of such types may lead to unpredictable negative results of the comparison.

The operator treats conditional values in a special way. An occurrence of conditional type in one or both arguments doesn't cause that the whole operator is executed conditionally. Empty values of conditional types take part in comparisons too. It is also possible to determine if an argument carries a conditional empty value by comparing it to Nil constant.

This operator never returns a conditional type.

Inequality test (<>)

```
expression <> expression
```

Data types: *any compatible types*

Result type: Bool

This operator performs an identical comparison to == operator, but it returns an opposite value (true, when arguments are not equal).

Special operators

Condition operator (`if-then-else`, `?:`)

```
if condition_expression then expression_1 else expression_2

if   condition_expression_1 then expression_1
elif condition_expression_2 then expression_2
 (...)
elif condition_expression_n then expression_n_minus_1
else expression_n

condition_expression ? expression_1 : expression_2
```

Data types:

- *condition_expression_x*: Bool
- *expression_1*, *expression_2*, ..., *expression_n*: *any compatible types*

This operator conditionally chooses and returns one of two or more values.

```
if condition_expression then expression_1 else expression_2
```

The first operand, in this case `condition_expression`, is a condition value of Bool data type. Depending on whether condition_expression equals *True* or *False*, it returns either `expression_1` (when *True*) or `expression_2` (when *False*).

```
if   condition_expression_1 then expression_1
elif condition_expression_2 then expression_2
 (...)
elif condition_expression_n then expression_n_minus_1
else expression_n
```

When connecting multiple conditions in a row, the program checks whether any of the consecutive conditions is met. The original condition uses the clause `if...then`, all the following ones use the clause `elif...then`. Should any of these multiple conditions be found to be *True*, then the corresponding expression is returned. However, after both the `if...then` clause and all the `elif...then` clauses are found to be *False*, the operator `else` calls the one expression that must be returned instead.

The operator treats conditional values in a special way. An occurrence of conditional type only in case of the first operand (condition value) causes the whole operator to be executed conditionally. Conditional types in operands of values to choose from do not make any changes in operator execution. Values of these operands are always passed in the same way that operation results are (also in case of an empty value).

The operator result type is determined by combining types of values to choose from (by converting one type to another or by converting two types into yet another common result type), according to the principles of arithmetic conversions. Such types can be changed to a conditional type in case of operator conditional execution.

Merge with default (`??`)

```
expression ?? expression
```

Data types: any compatible types

This operator requires that the first argument is of conditional type. If the first argument returns a non-empty value, then this value becomes the result of the whole operation. If the first argument returns an empty value (Nil), then the value of the second argument becomes the result of the whole operation.

Operation result type is the type proceeding from combining types of two arguments, but with a modified conditionality. If the second argument is of non-conditional type, then the result of the whole operation is non-conditional too. If the second argument is of conditional type, then the result of the whole operation is conditional.

This operator can be used for choosing an alternative value basing on a condition of the empty first argument.

The operator is called merging with default value, because it allows removing conditionality from an argument and replacing its empty value with a default value. In order to do this, the second argument cannot be of conditional type.

This operator is intended to handle conditional types, it means that the operator itself is never executed conditionally.

Function call (`function()`)

```
function(arg_1, arg_2, ..., arg_n)
function<type_name>(arg_1, arg_2, ..., arg_n)
```

Function call is performed by entering function name and a follow-up list of arguments in brackets. Required data types depend on the called function. For functions with generic type it is also optionally possible to explicitly specify the generic type in angular brackets directly after the function name.

See: Functions

Field read (`.`)

```
expression.name
```

This operator is used to read a single structure field or a property supported by an object. Operation result type is compatible with the type of the field which is read.

Any structure (from which a field of any type is read) can be passed as an argument.

name determines name of a field (or property), which should be read.

Object function call (`.function()`)

```
expression.function(arg_1, arg_2, ..., arg_n)
```

This construction is intended to call a function provided by an object (defined by data type of *expression* object).

Access to this function is provided (similar to field read operator) by entering a dot sign and the function name after an argument. After the function name a list of function arguments should be entered in round brackets. The required data types depend on the function which is called.

Element read (`[i]`, `[i, j]`)

```
expression[indexer]
expression[indexer1, indexer2]
```

Data types:

<T>Array[i]	→ *<T>*
Path[i]	→ *Point2D*
Matrix[row, col]	→ *Real*

This operator is used to read a single element from an array, a single point from a Path object, or a single element from a Matrix object. Operator result type is equal to the array elements type. An indexer has to be an argument of type Integer, its value points to an element to be read (value of 0 indicates the first array element). If an indexer returns a value pointing out of 0...ArrayLength-1 range, then the operator causes that a Domain Error appears during program runtime.

Explicit array processing (`[]`)

```
expression[]
```

Suggest that the outer operation (that uses the result of this operator as its argument) should be performed in an array processing mode, using this argument as an array data source.

This operator should be applied on an array data types and it is returning an unchanged array of the same type. The operation has no effect in runtime and is only used to resolve ambiguities of the array processing mode.

Array creation (`{}`)

```
{item_1, item_2, ..., item_n}
```

Creates an array out of the specified elements.

Provided elements must be of the same data type or be convertible to the same type using the rules of arithmetic conversions. Returned value is of type *T*Array, where *T* is a common type of specified elements.

Operator requires to provide at least one item. At least one of the provided items must be different than constant *Nil* (have non-null type).

See also: createArray function

Previous value of output (`prev()`)

```
prev(outputName, defaultValue)
prev(outputName)
```

This operator is used to read the value of a formula block output from the previous iteration of a task loop, inside which the formula block is instantiated. This operator takes an identifier as its first argument - the name of the formula block output. Any output of the parent formula block can be used, including the current formula output and outputs of formulas defined below the current one.

Second argument of the operator defines a default value and can accept any expression.

The purpose of this operator is to create an aggregate or sequence generating formula. In the first iteration of a task, this operator returns the default value (defined by the second argument). In each subsequent task iteration the operator returns the value previously computed by the referenced formula output (defined by the first argument).

Operator result type is determined by combining types of referenced output and default value argument, according to the principles of arithmetic conversions.

Second argument of the operator can be omitted. In such situation *Nil* is used as the default value and the result type is a conditional equivalent of the accessed output type. You can use the merge with default (`??`) operator to handle a *Nil* value.

Global parameter read (`::`)

```
::globalParameterName
```

Reads the value of a program global parameter (referencing parameter by its name).

Global parameter must be accessible from the current program module.

Operator precedence

In case of using more than one operator in the surrounding of the same argument, the precedence of executing these operators depends on their priorities. In the first instance, operators with the highest priority are executed. In case of binary operators with equal priorities, the operators are executed one after another from left to right, e.g.:

```
A + B * C / D
```

In the first instance, arguments B and C are multiplied (multiplication has higher priority than addition and, in this case, is located left from division operator which has the same priority). Multiplication result is divided by D (going from left to right), and then the quotient is added to A.

Unary operators, which are always in form of prefixes, are executed from the closest operator to an argument to the furthest one, that is from right to left.

Ternary condition operators are executed in order from right to left. It enables nesting conditions as part of other condition operator arguments.

Order of operators execution can be changed by placing nested parts of formulas in brackets, e.g.:

```
(A + B) * (C / D)
```

In this case, addition and division are executed in the first instance, then their results are multiplied.

Operator priorities

#	Operator	Description
1	`[i]`	Element read
	`[]`	Explicit array processing
	`()`	Function call
	`.`	Structure field read
	`.name()`	Object function call
2	`+`	Identity
	`-`	Negation
	`~`	Bitwise negation
	`not`	Logic negation
3	`* / div mod`	Multiplicative operators
4	`+ -`	Additive operators
5	`>> <<`	Bit shift
6	`&`	Bitwise operators
7	`^`	
8	`|`	
9	`< <= > >=`	Values comparison
10	`==`	Equality test
	`<>`	Inequality tests
11	`and`	Logic operators
12	`xor`	
13	`or`	
14	`??`	Merge with default value
15	`?:`	Condition operator
16	`if-then-else`	

A lower number means higher priority and earlier execution of operator.

Operator processing modes

Conditional processing

Similarly to filters in a vision application it is possible to invoke formula operators and function calls in conditional mode. When a value provided for the operation argument is of conditional type but the operation is not expecting conditional data type in its place, the whole operation is performed in a conditional mode. In the conditional mode the operation return value is promoted to a conditional type, the operation is executed only when required arguments are non-Nil, and when at least one conditional-mode argument is equal to Nil remaining arguments are ignored and Nil is returned without executing the operation.

For example, the binary operator `+` can be performed on arguments of type `Integer?`. In such case it returns also a value of type `Integer?`. When at least one of its arguments is equal Nil, the other argument is ignored and Nil is returned.

Conditional processing will usually cascade over multiple nested operations resulting in bigger parts of the formula to be performed in a mutual conditional mode. When conditional processing is not desired, or need to be stopped (e.g. when returning conditional data from the formula is not desired) the merge with default operator can be used to remove the condition modifier from the resulting data type and replace Nil with a default value.

Remarks

- Equality checking operators (`==` and `<>`) will not be executed in a conditional mode. Instead those operators will compare the values taking into account conditional types and testing for equality with Nil, even when conditional and non conditional types are mixed.
- The condition operator (`?:`, `if-then-else`) can be executed in a conditional mode when a value of type `Bool?` is used for the condition (first) argument. True and False value arguments (second and third arguments) are not participating in the conditional processing. Their values are passed as the result regardless of their data types.
- The merge with default operator (`??`) will never be executed in a conditional mode as it is designed to handle conditional types explicitly.
- In function call operations the conditional execution mode can be created by any conditional value assigned to a non-conditional argument of the function.
- In generic function call operations conditional execution cannot be implicitly created on generic function arguments, as conditional data type of such arguments will result in automatic generic type deduction to use also a conditional data type. To achieve this the function generic type needs to be specified explicitly.
- Similarly to above, conditional execution cannot be created on arguments of the array creating operator (`{}`) as it will result in creating an array with conditional items. A call to the createArray function with its generic type specified explicitly must be used instead.

Array processing

Also similarly to filters in a vision application it is possible to invoke formula operators and function calls in an array mode in which the formula operations are executed multiple times for each element of a source array, creating an array of results as the output. When an array argument is provided for an operator, or for function argument that does not expect an array (or expect lower rank array) the whole operation is performed in an array mode. In the array mode the operation return value is promoted to an array type (or a higher rank array) and the operation is executed multiple times, one time for each source array element, and the return value is also an array composed from subsequent operation results.

An array mode operation can be performed when only one argument is providing an array, or when multiple different arguments are providing multiple source array. In the latter situation all source arrays must be of equal size (have the same number of elements) and the operation is performed once for each group of equivalent array items from source arrays. Runtime Error is generated during execution when the source array sizes are different.

For example, lets consider a binary addition operation: `a + b`. Below table presents what results will be generated for different data types and values of a and b arguments:

a		b		a + b	
data type	**value**	**data type**	**value**	**data type**	**value**
Integer	10	Integer	5	Integer	15
IntegerArray	{10, 20, 30}	Integer	5	IntegerArray	{15, 25, 35}
IntegerArray	{10, 20, 30}	IntegerArray	{5, 6, 7}	IntegerArray	{15, 26, 37}
IntegerArray	*Empty array*	IntegerArray	*Empty array*	IntegerArray	*Empty array*
IntegerArray	{1, 2, 3}	IntegerArray	{1, 2, 3, 4}	IntegerArray	*Runtime Error*

Array processing is applied automatically when argument types clearly points to the array processing of the operation. In situations when application of the array processing is ambiguous the operation by default is left without array processing. To remove the ambiguity the explicit array processing operator can be applied on the array source arguments of the operation.

Explicit array processing operator needs to be used in the following situations:

- For the element read operator on a double nested array (e.g. `IntegerArrayArray`), the operator in the form `arg[index]` will access the element of the outermost array. The operator in the form `arg[][index]` will access elements of nested arrays in an array processing mode.
- The property `Count` of double nested array types will return the size of the outermost array. The construction `arg[].Count` will return an array of sizes of nested arrays.
- Equality testing operators (`==` and `<>`) will never implicitly start array processing. Instead those operators will always try to compare whole objects and return a scalar Bool value. To compare array elements in an array mode at least one argument needs to be market with the explicit array processing operator.
- In the merge with default operator (`??`) the first argument might need to be marked as explicit array source in some situations when connecting array items with default values from second array (when both arguments are array sources).
- In a generic function call operation values for generic arguments need to be marked as explicit array sources for the automatic generic type deduction to consider the array processing on those arguments (unless generic type is explicitly specified).
- In the array creation operator (`{}`) and the createArray function call all array source arguments always needs to be marked explicitly.

The result of an operator executed in the array mode is considered as an explicit array source for subsequent outer operations. This means that the array processing will cascade in the nested operations without the need to repeat the explicit array processing operator for each nested operation.

For example, considering `a` and `b` to be arrays of type `IntegerArray`, the formula:

```
a == b
```

will check whether both arrays are equal and will return the value of type `Bool`. The formula:

```
a[] == b[]
```

will check equality of array items, element by element, and will return the array of type `BoolArray`.

Remarks

In the condition operator (`?:`, `if-then-else`) the array mode processing can only be started by an array on the condition argument (first argument). However when the operation enters array processing the True and False arguments will also be considered as array sources, allowing to perform element-by-element based conditions in an array mode. It is also possible to mix scalar and array values on the True and False arguments.

Array and conditional processing combination

It is possible to combine conditional and array mode processing on a single operation up to triple nesting, resulting in a conditional-array-conditional processing of the operation.

For example the binary `+` operator, prepared to work on the `Integer` type, can be performed on arguments of types: `Integer`, `Integer?`, `IntegerArray`, `Integer?Array` and `Integer?Array?`. When working on the `Integer?Array?` data type the outermost conditionality modifier results in conditional processing of the whole array, where the inner conditionality modifier result in the conditional processing of array items (where condition is resolved on element-by-element basis).

Innermost condition (array element condition) follows the rules of handling Nil specified by the operation (it is an equivalent of the simple conditional processing when not combined with the array processing). Outermost condition (array condition) does not follow the per-operator rules and always interprets Nil by not executing the operation for the whole array (Nil is returned instead of the array).

Remarks

- The True and False arguments of the condition operator (`?:`, `if-then-else`) can participate in the array mode, but can never participate in the conditional mode. Thus, for condition operator in an array mode it is forbidden to provide True and/or False arguments with conditional arrays.
- The merge with default operator (`??`) has special requirements on its argument data types for complex processing. The following constructs are allowed for the array processing mode:
  - `T?Array ?? T`
  - `T?Array ?? T?`
  - `T?Array? ?? T` (array-conditional processing on first argument)
  - `T?Array? ?? T?` (array-conditional processing on first argument)
  - `T?Array ?? TArray` (requires explicit array mode)
  - `T?Array ?? T?Array` (requires explicit array mode)

  The following constructs will not result in an array processing mode:
  - `T?Array? ?? TArray`
  - `T?Array? ?? T?Array`
  - `T?Array? ?? TArray?`
  - `T?Array? ?? T?Array?`
  - `T?Array ?? TArray?`
  - `T?Array ?? T?Array?`

Functions

As part of formulas, you can use one of the functions listed below by using a function call operator. Each function has its own requirements defined regarding the number and types of individual arguments. Functions with the same names can accept different sets of parameter types, on which the returned value type depends. Each of the functions listed below has its possible signatures defined in form of descriptions of parameters data types and returned value types resulting from them, e.g.:

```
Real foo( Integer value1, Real value2 )
```

This signature describes the "foo" function, which requires two arguments, the first of type Integer and the second of type Real. The function returns a value of type Real. Such function can be used in a formula in the following way:

```
outValue = foo(10, inValue / 2) + 100
```

Arguments passed to a function don't have to exactly match a given signature. In such case, the best fitting function version is chosen. Then an attempt to convert argument types to a form expected by the function is performed (according to the principles of implicit conversions). If choosing a proper function version for argument types is not possible or, if it's not possible to convert one or more arguments to required types, then program initialization ends up with an error of incorrect function parameters set.

A conditional or optional type can be used as an argument of each function. In such case, if function signature doesn't assume some special use of conditional types, the entire function call is performed conditionally. A returned value is then also of a conditional type, an occurrence of Nil value among function arguments results in returning Nil value as function result.

Generic functions

Some functions accept arguments with so called generic type, where the type is only partially defined and the function is able to adapt to the actual type required. Such functions are designated with "<T>" symbol in their signatures, e.g. function minElement:

```
<T> minElement( <T>Array items, RealArray values )
```

This function is designated to process arrays with elements of arbitrary type. Its first argument accepts an array based on its generic type. A value of its generic type is also returned as a result. Generic functions can have only one generic type argument (common for all arguments and return value).

Generic functions can be called in the same way as regular functions. In such case the generic type of the function will be deduced automatically based on the type of specified arguments:

```
outMinElement = minElement(inBoxArrays, inValuesArrays)
```

In this example the value on input *inBoxArrays* is of type *BoxArray*, so the return type of the functions is *Box*.

In situations where the generic type cannot be deduced automatically, or needs to be changed from the automatically deduced one, it is possible to specify it explicitly, e.g. in this call of array function:

```
outNilsArray = array<Box?>(4, Nil)
```

an array of type *Box?Array* is created, containing four Nils.

Functions list

Mathematical functions

sin	cos	tan
asin	acos	atan
exp	ln	log
log2	sqrt	floor
ceil	round	abs
pow	square	hypot
clamp	lerp	integer
real	long	double
toString	parse*Value*	tryParse*Value*
min	max	indexOfMin
indexOfMax	avg	sum
product	variance	stdDev
median	nthValue	quantile
all	any	count
contains	findFirst	findLast
findAll	minElement	maxElement
removeNils	withoutNils	flatten
select	crop	trimEnd
rotate	pick	sequence
array	createArray	join
angleNorm	angleTurn	angleDiff
distance	area	dot
normalize	createVector	createSegment
createLine	translate	toward
scale	rotate	Matrix
identityMatrix	Path	Substring
Trim	ToLower	ToUpper
Replace	StartsWith	EndsWith
Contains	Find	FindLast
IsEmpty		

Conversion functions

Statistic and array processing functions

Geometry processing functions

Complex object creation

Functions of type String

Mathematical functions

sin

```
Real sin( Real )
  Double sin( Double )
```

Returns an approximation of the sine trigonometric function. It takes an angle measured in degrees as its argument.

cos

```
Real cos( Real )
Double cos( Double )
```

Returns an approximation of the cosine trigonometric function. It takes an angle measured in degrees as its argument.

tan

```
Real tan( Real )
Double tan( Double )
```

Returns an approximation of the tangent trigonometric function. It takes an angle measured in degrees as its argument.

asin

```
Real asin( Real )
Double asin( Double )
```

Returns an approximation of the inverse sine trigonometric function. It returns an angle measured in degrees.

acos

```
Real acos( Real )
Double acos( Double )
```

Returns an approximation of the inverse cosine trigonometric function. It returns an angle measured in degrees.

atan

```
Real atan( Real )
Double atan( Double )
```

Returns an approximation of the inverse tangent trigonometric function. It returns an angle measured in degrees.

exp

```
Real exp( Real )
Double exp( Double )
```

Returns an approximated value of the e mathematical constant raised to the power of function argument: exp(x) = e^x

ln

```
Real ln( Real )
Double ln( Double )
```

Returns an approximation of natural logarithm of its argument: ln(x) = $log_e(x)$

log

```
Real log( Real )
Double log( Double )
```

Returns an approximation of decimal logarithm of its argument: log(x) = $log_{10}(x)$

log2

```
Real log2( Real )
Double log2( Double )
```

Returns an approximation of binary logarithm of its argument: log2(x) = $log_2(x)$

sqrt

```
Real sqrt( Real )
Double sqrt( Double )
```

Returns the square root of its argument: sqrt(x) = $\sqrt[2]{x}$

floor

```
Real floor( Real )
Double floor( Double )
```

Rounds an argument down, to an integer number not larger than the argument.

ceil

```
Real ceil( Real )
Double ceil( Double )
```

Rounds an argument up, to an integer number not lesser than the argument.

round

```
Real round( Real, [Integer] )
Double round( Double, [Integer] )
```

Rounds an argument to the closest value with a strictly defined number of decimal places. The first function argument is a real number to be rounded. The second argument is an optional integer number, defining to which decimal place the real number should be rounded. Skipping this argument effects in rounding the first argument to an integer number.

Examples:

```
round(1.24873, 2) → 1.25
round(1.34991, 1) → 1.3
round(2.9812) → 3.0
```

abs

```
Real abs( Real )
Integer abs( Integer )
Double abs( Double )
Long abs( Long )
```

Returns the absolute value of an argument (removes its sign).

pow

```
Real pow( Real, Integer )
Real pow( Real, Real )
Double pow( Double, Integer )
Double pow( Double, Double )
```

Raises the first argument to the power of the second argument. Returns the result as a real number: pow(x, y) = x^y

square

```
Real square( Real )
Double square( Double )
```

Raises the argument to the power of two: square(x) = x^2.

hypot

```
Real hypot( Real a, Real b )
Double hypot( Double a, Double b )
```

Calculates and returns the result of the formula: hypot(a, b) = $\sqrt{a^2 + b^2}$

clamp

```
Integer clamp( Integer value, Integer min, Integer max )
Real clamp( Real value, Real min, Real max )
Long clamp( Long value, Long min, Long max )
Double clamp( Double value, Double min, Double max )
```

Limits the specified value (first argument) to the range defined by min and max arguments. Return value is unchanged when it fits in the range. Function returns the min argument when the value is below the range and the max argument when the value is above the range.

lerp

```
Integer lerp( Integer a, Integer b, Real lambda )
Real lerp( Real a, Real b, Real lambda )
Long lerp( Long a, Long b, Real lambda )
Double lerp( Double a, Double b, Double lambda )
Point2D lerp( Point2D a, Point2D b, Real lambda )
```

Computes a linear interpolation between two numeric values or 2D points. Point of interpolation is defined by the third argument of type Real in the range from 0.0 to 1.0 (0.0 for value equal to a, 1.0 for value equal to b).

Conversion functions

integer

```
Integer integer( Real )
Integer integer( Double )
Integer integer( Long )
```

Converts an argument to Integer type by cutting off its fractional part or ignoring most significant part of integer value.

real

```
Real real( Integer )
Real real( Long )
Real real( Double )
```

Converts an argument to Real type.

long

```
Long long( Real )
Long long( Double )
Long long( integer )
```

Converts an argument to Long type (cuts off fractional part of floating-point values).

double

```
Double double( Integer )
Double double( Long )
Double double( Real )
```

Converts an argument to Double type.

toString

```
String toString( Bool )
String toString( Integer )
String toString( Real )
String toString( Long )
String toString( Double )
```

Converts the argument to readable textual form.

parseInteger, parseLong, parseFloat, parseDouble

```
Integer parseInteger( String )
Long parseLong( String )
Real parseReal( String )
Double parseDouble( String )
```

Takes a number represented as a text and returns its value as a chosen numeric type. It is allowed for the text to have additional whitespace characters at the beginning and the end but it cannot have any whitespace characters or other thousand separator characters in the middle. For Real and Double types both decimal and exponential representations are allowed with dot used as the decimal symbol (e.g. "-5.25" or "1e-6").

This function will generate a DomainError when the provided text cannot be interpreted as a number or its value is outside of the types allowed range. To explicitly handle the invalid input text situation use the tryParse function variants.

tryParseInteger, tryParseLong, tryParseFloat, tryParseDouble

```
Integer? tryParseInteger( String )
Long? tryParseLong( String )
Real? tryParseReal( String )
Double? tryParseDouble( String )
```

Takes a number represented as a text and returns its value as a chosen numeric type. It is allowed for the text to have additional whitespace characters at the beginning and the end but it cannot have any whitespace characters or other thousand separator characters in the middle. For Real and Double types both decimal and exponential representations are allowed with dot used as the decimal symbol (e.g. "-5.25" or "1e-6").

This function returns a conditional value. When the provided text cannot be interpreted as a number or its value is outside of the types allowed range Nil is returned instead. The conditionality of the type needs to be handled later in the formula or the program. When the invalid input value is not expected and does not need to be explicitly handled a simpler parse variants of the function can be used.

Statistic and array processing functions

min

```
Integer min( Integer, ..., Integer )
Real min( Real, ..., Real )
Long min( Long, ..., Long )
Double min( Double, ..., Double )
Integer min( IntegerArray )
Real min( RealArray )
Long min( LongArray )
Double min( DoubleArray )
```

Returns the smallest of input values. This function can take from two to four primitive numeric arguments, from which it chooses the smallest value or an array of primitive numeric values, in which the smallest value is searched for. In case of an attempt to search for a value in an empty array, an operation runtime error is reported.

max

```
Integer max( Integer, ..., Integer )
Real max( Real, ..., Real )
Long max( Long,..., Long )
Double max( Double, ..., Double )
Integer max( IntegerArray )
Real max( RealArray )
Long max( LongArray )
Double max( DoubleArray )
```

Returns the largest of input values. This function can take from two to four primitive numeric arguments, from which it chooses the largest value or an array of primitive numeric values, in which the largest value is searched for. In case of an attempt to search for a value in an empty array, an operation runtime error is reported.

indexOfMin

```
Integer indexOfMin( IntegerArray )
Integer indexOfMin( RealArray )
Integer indexOfMin( LongArray )
Integer indexOfMin( DoubleArray )
```

Returns the (zero based) index of the smallest of the values in the input array. When multiple items in the array match the condition the index of the first one is returned. In case of an attempt to search for a value in an empty array, an operation runtime error is reported.

indexOfMax

```
Integer indexOfMax( IntegerArray )
Integer indexOfMax( RealArray )
Integer indexOfMax( LongArray )
Integer indexOfMax( DoubleArray )
```

Returns the (zero based) index of the largest of the values in the input array. When multiple items in the array match the condition the index of the first one is returned. In case of an attempt to search for a value in an empty array, an operation runtime error is reported.

avg

```
Integer avg( Integer, Integer )
Real avg( Real, Real )
Long avg( Long, Long )
Double avg( Double, Double )
Point2D avg( Point2D, Point2D )
Point3D avg( Point3D, Point3D )
Integer avg( IntegerArray )
Real avg( RealArray )
Long avg( LongArray )
Double avg( DoubleArray )
Point2D avg( Point2DArray )
Point3D avg( Point3DArray )
```

Computes the arithmetic mean of input numeric values or a middle point (center of mass) of input geometric points. This function can take two arguments, which are averaged or an array of values, which is averaged altogether. In case of an attempt to enter an empty array, an operation runtime error is reported.

sum

```
Integer sum( IntegerArray )
Real sum( RealArray )
Long sum( LongArray )
Double sum( DoubleArray )
```

Returns the sum of input values. This function can take an array of primitive numeric values, which will be summed altogether. In case of entering an empty array, the value of 0 will be returned.

Note: in case of summing large values or a big number of real arguments, it's possible to partially lose the result precision and, due to this, to mangle the final result. In case of summing integer numbers of too large values, the result may not fit in the allowed data type range.

product

```
Integer product( IntegerArray )
Real product( RealArray )
Long product( LongArray )
Double product( DoubleArray )
```

Returns the product of entered values. This function can take an array of primitive numeric values, which all will be multiplied. In case of entering an empty array, the value of 1 will be returned.

Note: in case of multiplying large values or a big number of real arguments, it's possible to partially lose the result precision and, due to this, to mangle the final result. In case of multiplying integer numbers of too large values, the result may not fit in the allowed data type range.

variance

```
Real variance( RealArray )
Double variance( DoubleArray )
```

Computes statistic variance from a set of numbers provided as an array in the argument. Result is computed using the formula: $1/n \sum(\bar{x} - x_i)^2$. In case of an attempt to enter an empty array, a domain error is reported.

stdDev

```
Real stdDev( RealArray )
Double stdDev( DoubleArray )
```

Computes statistic standard deviation from a set of numbers provided as an array in the argument. Result is equal to a square root of variance described above. In case of an attempt to enter an empty array, a domain error is reported.

median

```
Integer median( IntegerArray )
Real median( RealArray )
Long median( LongArray )
Double median( DoubleArray )
```

Returns the median value from a set of numbers provided as an array in the argument. In case of an attempt to enter an empty array, a domain error is reported.

nthValue

```
Integer nthValue( IntegerArray, Integer n )
Real nthValue( RealArray, Integer n )
Long nthValue( LongArray, Integer n )
Double nthValue( DoubleArray , Integer n)
```

Returns the n-th value in sorted order from a set of numbers provided as an array in the argument. The zero-based index *n*, of type Integer, is provided as the second argument. In case of an attempt to enter an empty array, a domain error is reported.

quantile

```
Integer quantile( IntegerArray, Real point )
Real quantile( RealArray, Real point )
Long quantile( LongArray, Real point )
Double quantile( DoubleArray , Real point )
```

Returns the specified quantile from a set of numbers provided as an array in the argument. In case of an attempt to enter an empty array, a domain error is reported.

all

```
Bool all( BoolArray predicates )
```

This function takes an array of logical values and returns True when all elements in the array are equal to True. In case of entering an empty array, the value of True will be returned.

any

```
Bool any( BoolArray predicates )
```

This function takes an array of logical values and returns True when at least one element in the array equals True. In case of entering an empty array, the value of False will be returned.

count

```
Integer count( BoolArray predicates )
Integer count( <T>Array items, <T> value )
```

First variant of this function takes an array of logical values and returns the number of items equal to True.

Second variant takes an array of items and counts the number of elements in that array that are equal to the value given in the second argument.

contains

```
Bool contains( <T>Array items, <T> value )
```

Checks if the specified array contains a value.

Returns the value of True when the array given in the first argument contains at least one instance of the value from the second argument.

findFirst

```
Integer? findFirst( <T>Array items, <T> value )
```

Searches the specified array for instances equal to the given value and return the zero based index of the first found item (closest to the beginning of the array). Returns Nil when no items were found.

findLast

```
Integer? findLast( <T>Array items, <T> value )
```

Searches the specified array for instances equal to the given value and return the zero based index of the last found item (closest to the end of the array). Returns Nil when no items were found.

findAll

```
IntegerArray findAll( <T>Array items, <T> value )
```

Searches the specified array for instances equal to the given value and return an array of zero based indices of all found items. Returns an empty array when no items were found.

minElement

```
<T> minElement( <T>Array items, RealArray values )
```

Returns an array element that corresponds to the smallest value in the array of values. When input array sizes does not match or when empty arrays are provided, a domain error is reported.

maxElement

```
<T> maxElement( <T>Array items, RealArray values )
```

Returns an array element that corresponds to the biggest value in the array of values. When input array sizes does not match or when empty arrays are provided, a domain error is reported.

removeNils

```
<T>Array removeNils( <T>?Array items )
```

Removes all Nil elements from an array. Returns a new array with simplified type.

withoutNils

```
<T>Array? withoutNils( <T>?Array items )
```

Returns the source array only when it does not contains any Nil values.

This function accepts an array with conditional items (<T>?Array) and returns a conditional array (<T>Array?). When at least one item in the source array is equal to Nil the source array is discarded and Nil is returned, otherwise the source array is returned with simplified type.

flatten

```
<T>Array flatten( <T>ArrayArray items )
```

Takes an array of arrays, and concatenates all nested arrays creating a single one-dimensional array containing all individual elements.

select

```
<T>Array select( <T>Array items, BoolArray predicates )
```

Selects the elements from the array of items for which the associated predicate is True.

Arrays of items and predicates must have the same number of elements. This function returns a new array composed out of the elements from the items array, in order, for which the elements of the predicates array are equal True.

crop

```
<T>Array crop( <T>Array items, Integer start, Integer length )
```

Selects a continuous subsequence of array elements.

When the specified range spans beyond the source array only the elements intersecting with the requested range are selected (in such case the function will return less than *length* elements).

trimStart

```
<T>Array trimStart( <T>Array items )
<T>Array trimStart( <T>Array items, Integer count )
```

Removes *count* elements from the beginning of the array. By default (when the argument *count* is omitted) removes a single element. When *count* is larger than the size of the array an empty array is returned.

trimEnd

```
<T>Array trimEnd( <T>Array items )
<T>Array trimEnd( <T>Array items, Integer count )
```

Removes *count* elements from the end of the array. By default (when the argument *count* is omitted) removes a single element. When *count* is larger than the size of the array an empty array is returned.

rotate (array)

```
<T>Array rotate( <T>Array items )
<T>Array rotate( <T>Array items, Integer steps )
```

Rotates the elements from the specified array by *steps* places. Rotates right (towards larger indexes) when the *steps* value is positive, and left (towards lower indexes) when the *steps* value is negative. By default (when the *steps* argument is skipped) rotates right by one place.

Rotation operation means that all elements are shifted along the array positions, and the elements that are shifted beyond the end of the array are placed back at the beginning.

pick

```
<T>Array pick( <T>Array items, Integer start, Integer step, Integer count )
```

Picks items from an array at equal spacing.

This function will pick elements from the source array, starting at the *start* position, and moving forward by a *step* amount of elements *count* times. When the specified range spans beyond the source array, only the elements intersecting with the requested range are selected (in such case the function will return less than *count* elements).

The value specified as *step* must be greater that zero.

sequence

```
IntegerArray sequence( Integer start, Integer count )
IntegerArray sequence( Integer start, Integer count, Integer step )
RealArray sequence( Real start, Integer count )
RealArray sequence( Real start, Integer count, Real step )
```

Creates an array of *count* numbers, starting from the value provided in *start* argument and incrementing the value of each item by the value of *step* (or 1 when *step* is not specified).

array

```
<T>Array array( Integer count, <T> item )
```

Creates a uniform array with *count* items by repeating the value of *item*.

createArray

```
<T>Array createArray( <T> arg1, <T> arg2, ..., <T> argN )
```

Creates an array out of arbitrary number of elements provided in the arguments.

This is basically an equivalent of array creation operator ({ }) in function form, allowing for explicit item type specification and empty array creation.

join

```
<T>Array join( <T>[Array] arg1, <T>[Array] arg2, ..., <T>[Array] argN )
```

Joins an arbitrary (at least two) number of arrays and/or scalar values into a single array. Returns a uniform array with elements in the same order as specified in the function arguments.

Geometry processing functions

angleNorm

```
Real angleNorm( Real angle, Real cycle )
```

Normalizes a given angle (or other cyclic value) to the range [0...*cycle*). *cycle* must be a positive value (usually 180 or 360, but any greater than zero value is acceptable).

angleTurn

```
Real angleTurn( Real startAngle, Real endAngle, Real cycle )
```

Calculates a directional difference between two given angles.

Assuming that *startAngle* and *endAngle* are cyclic values in the range [0...*cycle*), this function is calculating the shortest angular difference between them, returning positive values for a forward angle (from *startAngle* to *endAngle*), and a negative value for a backward angle. *cycle* must be a positive value.

angleDiff

```
Real angleDiff( Real angle1, Real angle2, Real cycle )
```

Calculates an absolute difference between two given angles.

Assuming that given angles are cyclic values in the range [0...*cycle*), this function is calculating the shortest absolute angular difference between them. Returned value is non-negative. *cycle* must be a positive value.

distance

```
Real distance( Point2D, Point2D )
Real distance( Point2D, Segment2D )
Real distance( Point2D, Line2D )
Real distance( Point3D, Point3D )
Real distance( Point3D, Segment3D )
Real distance( Point3D, Line3D )
Real distance( Point3D, Plane3D )
```

Calculates the distance between a point and the closest point of a geometric primitive.

area

```
Real area( Box )
Real area( Rectangle2D )
Real area( Circle2D )
Real area( Path )
Integer area( Region )
```

Calculates the surface area of a given geometric primitive.

For Path primitive the path needs to be closed and the surface area enclosed by the path is calculated. The path must not intersect with itself (improper result is returned in that case).

For Region primitive the result (of type Integer) is equivalent to the number of pixels active in the region.

dot

```
Real dot( Vector2D, Vector2D )
Real dot( Vector3D, Vector3D )
```

Calculates the dot product of two vectors.

normalize

```
Vector2D normalize( Vector2D )
Vector3D normalize( Vector3D )
```

Normalizes the specified vector. Returns a vector with the same direction but with length equal 1. When provided with zero length vector returns a zero length vector as a result.

createVector

```
Vector2D createVector( Point2D point1, Point2D point2 )
Vector2D createVector( Real direction, Real length )
```

Creates a two dimensional vector (a *Vector2D* structure). First variant creates a vector between two points (from *point1* to *point2*). Second variant creates a vector pointing towards a given angle (in degrees) and with specified length.

createSegment

```
Segment2D createSegment( Point2D start, Real direction, Real length )
```

Creates a two dimensional segment (a *Segment2D* structure) starting at a given *start* point, pointing towards a given direction (in degrees) and with specified length.

createLine

```
Line2D createLine( Point2D point1, Point2D point2 )
Line3D createLine( Point3D point1, Point3D point2 )
Line2D createLine( Point2D point, Real direction )
```

Creates a line (a *Line2D* or *Line3D* structure). First two variants create a line that contains both of the specified points. Third variant creates a line containing specified point and oriented according to the specified angle (in degrees).

translate

```
Point2D translate( Point2D point, Vector2D translation )
Point2D translate( Point2D point, Real direction, Real distance )
```

Moves a point.

First variant of the function moves a point by the specified translation vector. Second variant moves a point towards a specified direction (defined by angle in degrees) by an absolute distance.

toward

```
Point2D toward( Point2D point, Point2D target, Real distance)
```

Moves a point in the direction of the target point by an absolute distance. The actual distance between point and target does not affect the distance moved. Specifying a negative distance value results in moving the point away from the target.

scale

```
Point2D scale( Point2D point, Real scale )
Point2D scale( Point2D point, Real scale, Point2D origin )
```

Moves a point by relatively scaling its distance from the center of the coordinate system (first function variant) or from the specified origin point (second function variant).

rotate (geometry)

```
Point2D rotate( Point2D point, Real angle, Point2D origin )
```

Moves a point by rotating it around the specified origin point (by an angle specified in degrees).

Complex object creation

Matrix

```
Matrix( Integer rows, Integer cols )
Matrix( Integer rows, Integer cols, Real value )
Matrix( Integer rows, Integer cols, RealArray data )
Matrix( Integer rows, Integer cols, IntegerArray data )
```

Creates and returns a new Matrix object with specified number of rows and columns. When no *value/data* parameter is specified the new matrix is filled with zeros.

A scalar *value* can be specified as the third argument to set all elements of the new matrix to.

An array of values can be specified as the third (*data*) argument to fill the new matrix. Consecutive elements from the array are set into the matrix row-by-row, top-to-bottom, left-to-right. Provided array must have at least as many element as the number of elements in the created matrix.

identityMatrix

```
identityMatrix( Integer size )
```

Creates and returns a new square identity Matrix object with *size* rows and *size* columns.

Path

```
Path( Point2DArray points )
Path( Point2DArray points, Bool closed )
```

Creates and returns a new Path object filled with points provided in an array. By default (when no second argument is specified) the new path is not closed.

Functions of type String

On text arguments it is possible to call functions (according to object function call operator) processing or inspecting a text string, e.g.:

```
outText = inText.Replace( "to-find", inNewText.ToLower() )
```

The type of String provides the following object functions:

Substring

```
String arg.Substring( Integer position )
String arg.Substring( Integer position, Integer length )
```

This function returns the specified part of text. The first argument defines the position (starting from zero) on which the desired part starts in text. The second argument defines the desired length of returned part of text (this length can be automatically shortened when it exceeds the text length). Leaving out the second argument results in returning the part of text from the specified position to the end of text.

Trim

```
String arg.Trim()
```

Returns argument value, from which the whitespace characters have been removed from the beginning and the end.

ToLower

```
String arg.ToLower()
```

Returns argument value, in which all the upper case letters have been changed to their lower case equivalents.

ToUpper

```
String arg.ToUpper()
```

Returns argument value, in which all the lower case letters have been changed to their upper case equivalents.

Replace

```
String arg.Replace( String find, String insert )
```

Searches text for all occurrences of the value entered as the first argument and replaces such occurrences with the value entered as the second argument. Text search is case sensitive.

This function searches the source text consecutively from left to right and leaps the occurrences immediately after finding them. If the sought-after parts overlap in the text, only the complete occurrences found this way will be replaced.

StartsWith

```
Bool arg.StartsWith( String )
```

This function returns True only if text contains at its beginning (on the left side) the value entered as the argument (or if it is equal to it). Text search is case sensitive.

EndsWith

```
Bool arg.EndsWith( String )
```

This function returns True only if text contains at its end (from the right side) the value entered as the argument (or if it is equal to it). Text search is case sensitive.

Contains

```
Bool arg.Contains( String )
```

This function returns True only if text contains (as a part at any position) the value entered as the argument (or if it is equal to it). Text search is case sensitive.

Find

```
Integer arg.Find( String )
Integer arg.Find( String, Integer )
```

This function searches in text for the first occurrence of the part entered as the argument (it performs searching from left to right) and returns the position, at which an occurrence has been found. It returns -1 when the sought-after substring doesn't occur in text. Text search is case sensitive.

Optionally, as the second function argument, one can enter the starting position from which the search should be performed.

FindLast

```
Integer arg.FindLast( String )
Integer arg.FindLast( String, Integer )
```

This function searches in text for the last occurrence of the part entered as the argument (it performs search starting from right) and returns the position, at which an occurrence has been found. It returns -1 when the sought-after substring doesn't occur in text. Text search is case sensitive.

Optionally, as the second function argument, one can enter the starting position from which the search should be performed (and proceed in the direction of text beginning).

IsEmpty

```
Bool arg.IsEmpty()
```

This function returns True if text is empty (its length equals 0).

Structure constructors

As part of formulas, you can pass and access fields of any structures. It's also possible to generate a new structure value, which is passed to further program operations.

In order to generate a structure value, a structure constructor (with syntax identical to the function with the name of the structure type) is used. As part of structure parameters, structure fields values should be entered consecutively. E.g. the constructor below generates a structure of type Box, starting in point 5, 7, with width equal to 100 and height equal to 200:

```
Box(5, 7, 100, 200)
```

It is also possible to create a structure object with default value by calling the constructor with empty parameters list:

```
Box()
```

Not all structure types are available through constructors in formulas. Only structures consisting of fields of primitive data types and not requiring specific dependencies between them are available. E.g. types such as Box, Circle2D or Segment2D consist of arithmetic types and due to this creating them in formulas is possible. Types such as Image or Region have complex data blocks describing primitives and therefore it's not possible to create them in formulas.

Some structures have additional requirements on field values (e.g. Box requires that width and height be non-negative). When such a requirement is not met a Domain Error appears during program runtime.

Typed Nil constructors

Nil symbol, representing an empty value of conditional types, does not provide the data type by itself. In constructions where providing a concrete data type of Nil value is necessary (like array construction or generic function call) it is possible to use a typed Nil construction by calling data type name and proving a single Nil symbol as the argument:

```
Integer(Nil)
Box(Nil)
Segment2D(Nil)
```

Examples

Summing two integer values

Filter inputs:	Filter outputs:
• **inA** of type **Integer** • **inB** of type **Integer**	• **outSum** of type **Integer**

```
outSum = inA + inB
```

Linear interpolation of two integer values

Having given two integer values and the position between them in form of a real number 0...1, the task is to compute the interpolated value in this position.

Filter inputs:	Filter outputs:
• **inA** of type **Integer** • **inB** of type **Integer** • **inPos** of type **Real**	• **outValue** of type **Integer**

```
outValue = integer(round( inA * (1 - inPos) + inB * inPos ))
```

A simple weighted averaging of two given values is performed here. Due to the multiplication by real values, the intermediate averaging result is also a real value. It's necessary then to round and convert the final result back to integer value.

Computing a center of Box primitive

Box is a data structure consisting of four fields of type Integer: X, Y, Width and Height. Having given an object of such primitive, the task is to compute its center in form of two integer coordinates X and Y.

Filter inputs:	Filter outputs:
• **inBox** of type **Box**	• **outX** of type **Integer** • **outY** of type **Integer**

```
outX = inBox.X + inBox.Width div 2
outY = inBox.Y + inBox.Height div 2
```

Generating a structure

In the previous example, data used in a formula is read from a primitive of type Box. Such primitive can also be generated inside of a formula and passed to an output. In this example, we'd like to enlarge a Box primitive by a frame of given width.

Filter inputs:	Filter outputs:
• **inBox** of type **Box** • **inFrame** of type **Integer**	• **outBox** of type **Box**

```
outBox = Box( inBox.X - inFrame, inBox.Y - inFrame, inBox.Width + inFrame*2, inBox.Height + inFrame*2 )
```

Partial conditional execution

In case of working with conditional types, it's possible, as part of formula blocks, to use primitive types and to use conditional filters execution, including conditional execution of entire formula blocks. It's also possible to pass conditional execution inside formulas. This way, only a part of formulas of a block, or even only a part of a single formula, is executed conditionally.

In the simple summing shown below, parameter B is of a conditional type and this conditionality is passed to output.

Filter inputs:	Filter outputs:
• **inA** of type **Real** • **inB** of type **Real?** • **inC** of type **Integer**	• **outValue** of type **Real?**

```
outValue = inA + inB + inC
```

Formula content itself doesn't contain any specific elements related to a conditional parameter. Only formula output is marked with a conditional type. An occurrence of an empty value in input causes aborting execution of further operators and moving an empty value to output, in a similar way to operations on filters.

Replacing an empty value with a default object

If the conditional output type and thereby the conditional execution mode of following filters were not desired in the example above, it's possible to resolve the conditional type still in the same formula by creating a connection with a non-conditional default value.

Filter inputs:	Filter outputs:
• **inA** of type **Real** • **inB** of type **Real?** • **inC** of type **Integer**	• **outValue** of type **Real**

```
outValue = (inA + inB + inC) ?? 0
```

Addition operators still work on conditional types, the entirety of their calculations can be aborted when an empty value occurs. The result is however merged with a default value of 0. If summing doesn't return a non-empty result, then the result value is replaced with such non-empty value. In such case, an output doesn't have to be of conditional type.

Empty array response

In order to find the maximal value in an array of numbers, you can use the max function, it requires although a non-empty array. If an explicit reaction to such case is possible, we can define it by a condition and thereby avoid an error by passing an empty sequence to input.

Filter inputs:	Filter outputs:
• **inElements** of type **IntegerArray**	• **outValue** of type **Integer**

```
outValue = (inElements.Count > 0) ? max(inElements) : 0
```

In the example above a value of zero is entered instead of maximum of an empty array.

Range testing

Let's assume that we test a position of some parameter, which varies in the range of -5...10 and we'd like to normalize its value to the range of 0...1. What's more, the reaction to an error and to range overrun by a parameter is returning an empty value (which means conditional execution of further operations).

Filter inputs:	Filter outputs:
• **inParameter** of type **Real**	• **outRangePos** of type **Real?**

```
outRangePos = (inValue >= -5 and inValue <= 10) ? (inValue + 5) / 15 : Nil
```

Choosing a constant enumeration value

If block results should control operations which take enumeration types as parameters, then it's possible to generate values of such types as part of formulas. In the example below we'd like to determine the sorting direction, having given a flag of type Bool, which determines, if the direction should be opposite to the default one.

Filter inputs:	Filter outputs:
• **inReverse** of type **Bool**	• **outOrder** of type **SortingOrder**

```
outOrder = inReverse ? SortingOrder.Descending : SortingOrder.Ascending
```

Adding integer values of the loop

This example works in a loop, in each iteration there is some value passed to the inValue input and the goal is to sum these values. The prev operator returns 0 in the first iteration (as defined by its second argument). In each iteration the value of inValue input is added to the sum from the previous iteration.

Filter inputs:	Filter outputs:
• **inValue** of type **Integer**	• **outSum** of type **Integer**

```
outSum = prev(outSum, 0) + inValue
```

Testing and Debugging

This article expands on the information given in Running and Analysing Programs.

Execution and Performance

Aurora Vision Studio allows the user to debug and prototype their applications. It employs many features that help the user with designing and testing the program.

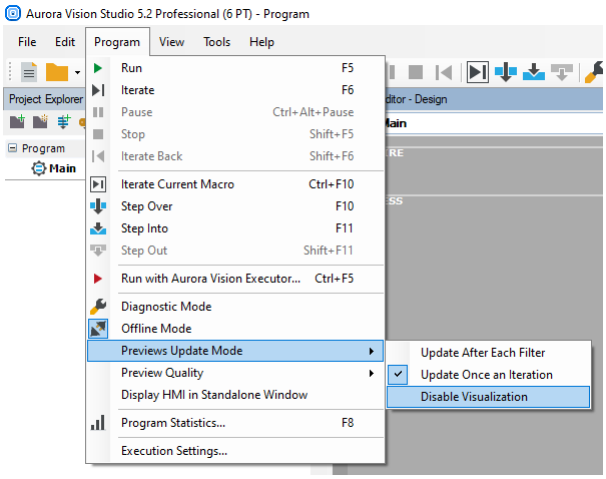
Those features are enabled by default, which makes Aurora Vision Studio behave different than Aurora Vision Runtime. Differences can affect both performance and program flow. As such, those settings are not recommended when testing how the final version works.

Performance-Affecting Settings

There are two settings that affect the performance in a noticeable way. The first one is the *Diagnostic Mode*. Many filters have diagnostic outputs and when the *Diagnostic Mode* is on, those outputs will be populated with additional data that may help during program designing. However, calculating and storing this additional data takes time. Depending on the program, this can cause a considerable slowdown. This mode can be toggled off with the button in the Application toolbar.

Another setting that can hinder the performance compared to Aurora Vision Runtime is *Previews Update Mode*. Ports that are being previewed are updated according to this setting. Updating previews is generally quite fast, but it depends on the amount and complexity of the data being previewed (large instances of Image or Surface will take more time to display than an instance of Integer).

With active previews even having the filter visible in the Program Editor will cause some overhead (tied to how often the filter is executed). This is related to how Aurora Vision Studio shows progress. The option *Program » Previews Update Mode » Disable Visualization* disables the previews altogether. While the overhead is generally small it can be noticeable when the iteration of the program is also short.



Location of the option to disable visualization

Both settings can also be changed in Program Execution settings.

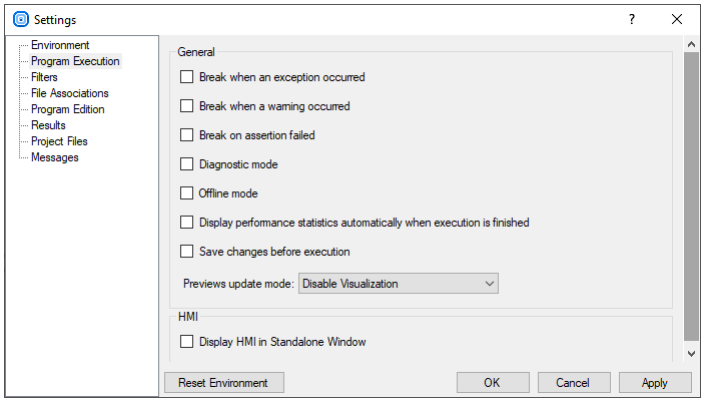
Disabling the *Diagnostic Mode* along with previews makes Studio execute programs nearly as fast as Runtime.

Execution Flow

There is one more difference between Studio and Runtime execution. By default, Studio is set to pause whenever an exception, warning or assertion occurs. Every exception will cause the program to pause, even if it is handled with an error handling.
Runtime continues to run through all of them, when possible - it will only report them in the console.

To make Studio behave like Runtime in that regard, the following settings need to be disabled:

- Break when an exception occurred
  Note that this does not prevent errors from ending the program. This can only be achieved with error handling.
- Break when a warning occurred
- Break on assertion failed



View of program execution settings. The settings above should result in Studio performance as close to Runtime as possible.

At any point during testing it is possible to view statistics of every filter executed until this point.

Operation Mode

There are two modes in which the program can be run during debugging: normal and iteration mode.

- In normal mode, the program will run continuously until the user pauses it or until it ends. In this mode, previews are updated according to the *Previews Update Mode* setting in which it was launched:
- During iteration mode, the user can step into macrofilters, step over them or step to the end of macrofilters or iterate the displayed macrofilters.

Both modes will also pause whenever they encounter one of the things described in the following part.

Execution Pausing

The following can pause the application. The application will pause at any of these regardless of the mode

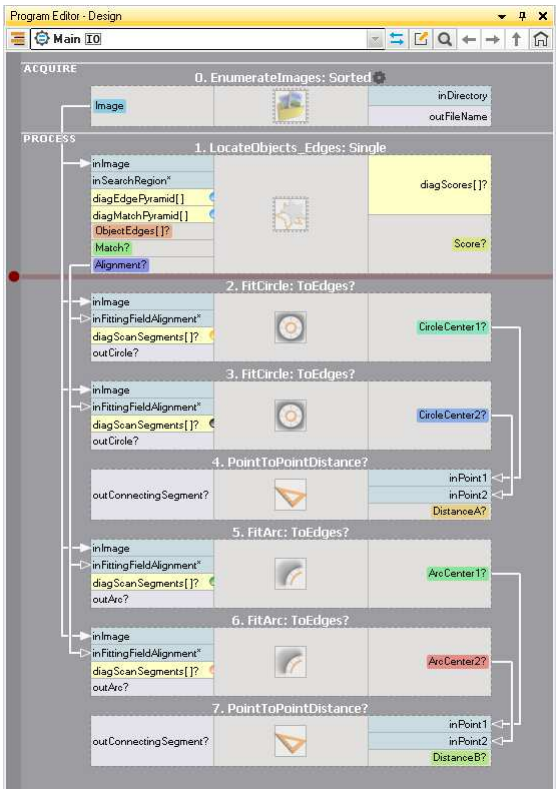
- Breakpoints,
- Warnings (can be changed in settings).
- Ending of the tracked Worker Task,
- Failed Assertions (can be changed in settings),
- Exception (can be changed in settings),

The program can also be stopped at any time using the pause button.

Breakpoints

Breakpoints allow the user to specify points in the program at which the execution will pause. They are effective only in Studio.

- Breakpoints can be placed at any filter or output block by right-clicking and selecting *Toggle Breakpoint*, by pressing F9 or by hovering on the left side of the Program Editor.
- Breakpoints are not saved in the project and they will disappear after loading the program again.
- Breakpoints are shared between all instances of particular macrofilters that contain them. For example, if there are two instances of TestMacro, with a breakpoint inside, the program will pause both when executing the first and the second one.
- Breakpoints can be placed in HMI events.
- Breakpoints affect all Worker threads.



A view of a macrofilter in which a breakpoint was placed.

Single-Threaded Debugging and Testing

An application that features only one Worker Task is a common design pattern. After starting such an application it will run continuously until it ends or is paused.

You can start the application in iteration mode. This will start at the beginning of the application in Main.

Both modes of operation can be used during one run. For example, you can start in the normal mode until a breakpoint, continue in the iteration mode and finish back in the normal mode.

The third way to start the program is to right-click a filter and select *Run Until Here*. The program will start in the normal mode until the specified point and then pause.

- Unlike breakpoints, the point

HMI events can also be debugged. Breakpoints can be placed inside event handling macrofilters and the

specified this way is unique. The program will pause only in the specified instance of the macrofilters, even if there are different instances in the program.

program will pause correctly. Ports in events can be connected to previews. However, it is not possible to use the option *Run Until Here* in events.

It is possible to preview values of ports from different Worker Tasks or HMI events. They will be updated according to the relevant setting.

Multi-Threaded Debugging and Testing

Application with multiple Worker Tasks introduces new concepts. First of them is the Primary Worker Task. It is the Task macrofilter that controls the duration of the program. When the Primary Worker Task ends, the other Worker Tasks end as well, even if they were still running.

Conversely, the program will continue running as long as that Worker Task is running, even if other Worker Tasks have ended. The primary Worker Task is selected by the user and cannot be changed while the program is running. To select a Worker Task macrofilter as the Primary Worker Task macrofilter, you can right-click it and choose *Set as Primary Worker* or select it in the *ComboBox* in the toolbar when the program is not running.

The second concept is Tracked Task. This is the Task that is actively tracked, i.e. it is possible to run it in the iteration mode and its call-stack is visible at the bottom. When the program is paused (for any reason mentioned above) Aurora Vision Studio will automatically make the Worker Task in which the pause happened the Tracked Task. It is also possible to manually change tracked Worker Task - either by right-clicking it and selecting *Track This Worker* or by selecting it in the *ComboBox* when the program is paused.

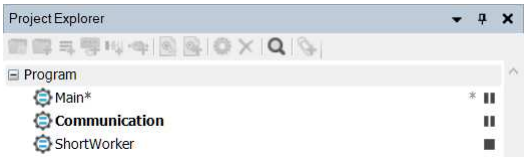
When the program has multiple Worker Tasks, another elements is shown in the toolbar, to the left of the *Run* button. It functions as a ComboBox and allows the user to change the behavior of the *Run* button between executing all Worker Tasks or only the Primary Worker Task.

It is only possible to switch it before the program is started. When the program is paused, all Worker Tasks are paused.

When in the iteration mode only one Worker Task is being run, it is the Tracked Worker. The other Worker Tasks are paused, so if the they exchange information that is important for debugging, the normal mode has to be used at some point.

If the program is started in the iteration mode, the Worker Task that is being viewed at the moment (or that contains the macrofilter being viewed) will be set as the Tracked Worker Task and only this thread will run. At any point, you can run it in normal mode to launch the remaining threads.

The Primary Worker Task is marked with an asterisk attached directly to its name, visible when the program is running. The active Worker Task is emboldened both when the program is off and when it is running.



A view of the Project Explorer of a paused program. The Main Worker Task, which is the Primary Worker Task is marked with an asterisk attached to its name (not to be confused with the asterisk on the right, related to thread consumption). The currently Tracked Worker Task Communication is bold. The ShortWorker has already ended, which is marked with a stop symbol, as opposed to the pause symbol.

Program ComboBox

The *ComboBox* in the Toolbar allows the user to specify both the Primary Worker Task and the Tracked Worker Task. It can also display active threads. Its functionality changes depending on the state of the program.

When the program has not run or has stopped, the *ComboBox* selects the Primary Worker. It is one of two ways of setting the Primary Worker (the other being through right-clicking the desired Worker Task).

When the program has been started and is paused, the *ComboBox* displays all active threads at the moment. This includes HMI events, if the pause happened when an event was being executed. All Worker Tasks that have already ended will not be displayed. The user can set one of the Worker Tasks active in the *ComboBox* as the Worker Task to be tracked.

Iterating Program

Running the program in the iteration mode can be achieved with the buttons present in the Application Toolbar. As mentioned before iterating actions are single-threaded. All Worker Tasks other than the tracked one will be paused when iterating.
Additional information about iterating can be found in Running and Analysing Programs.

Iterate Program

The *Iterate Program* button executes one iteration of the Primary Worker Task.

Iterate Current Macro

Iterate Current Macro launches the program and then pauses it as soon as one iteration of the visible macrofilter instance has finished.

Iterate Back

Under some circumstances it is possible to Iterate Back , that is to reverse the iteration and go back to the previous data. The requirements for this option are:

- The program has to be paused after completing one iteration of a filter (either with *Iterate Current Macro* or *Iterate Program*).
- The filter has to be a Task or Worker Task.
- At least one of the loop generators in the task needs to be deterministic and be a source of data (for example it has to be an enumeration filter). Some examples include:
  - EnumerateIntegers
  - EnumerateImages
  - EnumerateFiles

  Filters such as EnumerateImages and EnumerateFiles create a list of objects before their first iteration and enumerate over it. The list will not reflect later changes to the list (adding/removing images or files).
- The filter has to be the Primary Worker Task or be inside it.
- The Task macrofilter to be iterated back cannot contain other Task macrofilters.

As long as the Task contains at least one enumeration filter is present it is possible to iterate back. Furthermore, there can be other loop generators present, even if they are not enumerators.
Below are examples of filters that do not allow iterating back by themselves:

- Camera grabbing filters, such as GenICam_GrabImage.
- Loop — no data to iterate back.
- Communication filters, such as TcpIp_ReadLine.

If those filters are in a Task that can be iterated back, they will behave as they would during a normal, non-reversed iteration.

Step Buttons

There are three options for the user to move step-by-step through the program. Those are: *Step Over* , *Step Into* , and *Step Out* .
Those buttons are described here.

Iterating Options Conclusion

The following table is describing all iterating options in a concise manner:

Method	Primary Worker	Tracked Worker	Number of workers run	Can be used inside HMI events
Run ( ) *Normal Mode*	Selected by the user prior to starting	Primary Worker	All	Yes
Run ( ) *Normal Mode*	Selected by the user prior to starting	Primary Worker	1 *(Tracked Worker)*	Yes
Run Until Here *(Context menu option)*	Selected by the user prior to starting	The worker in which the option was selected	All	No
Steps	Selected by the user prior to starting	The worker in which the option was selected	1 *(Tracked Worker)*	Yes *(when event is tracked)*
Iterate Program	Selected by user prior to starting	Primary Worker	1 *(Primary Worker)*	No
Iterate Current Macro	Selected by user prior to starting	The worker in which the option was selected	1 *(Tracked Worker)*	No

Error Handling

Introduction

Error handling is a mechanism which enables the application to react to exceptional situations encountered during program execution. The application can continue to run after errors raised in filters.

This mechanism should mainly be used for handling errors of the IO_ERROR kind. Usage for other kinds of errors should be taken with caution. Every application can be created in a manner which precludes DOMAIN_ERROR. This topic is elaborated in a separate article: Dealing with Domain Errors. A frequent cause of this kind of errors is the omission of filters such as SkipEmptyRegion.

Error handling can only be added to Task Macrofilters. A re–execution of the Task Macrofilter following an error results in a reset of stateful filters contained therein. This means in particular a possible reconnect to cameras or other devices. The error handler for each kind of error is executed in the manner of a Step Macrofilter.

A special case is the error handler of the Task Macrofilter set as Startup Program (usually: Main). The program execution is always terminated after the error handler finishes.

There are several kinds of errors which can be handled:

- **IO_ERROR**
  An error during I/O operation or interaction with an external device (camera, network, digital I/O card, storage etc.).
- **SYSTEM_ERROR**
  An error raised by the operating system (e.g. failed memory allocation, missing driver or library).
- **DOMAIN_ERROR**
  Incorrect use of the filter or function, usually from input values (e.g. out of the valid range).
- **ANY_ERROR**
  Includes all of the above kinds as well as unspecified runtime errors arising during program execution.
- **LICENSE_ERROR**
  An error caused by lack of a license required to use the library (AVL).
  The handling proceeds in two distinct ways, depending on the error kind:

  IO_ERROR
  DOMAIN_ERROR
  SYSTEM_ERROR
  After occurrence of one of such errors, the program enters the nearest error handler. After completing the handler, the execution is resumed right after the Task in which the error was raised. The caller (parent Macrofilter) considers this Task to be completed successfully.
- **ANY_ERROR**
  LICENSE_ERROR
  These kinds are critical errors and the **application cannot continue to execute**. After occurrence of such an error the nearest error handler is executed, but after completion of the handler, the error stays "active". Which means, the caller, or any Macrofilter above it, can execute their own handler for this kind of error, and the program will finally be stopped. Handling these kinds of errors can be used to notify the operator about a critical system error and request service attendance.

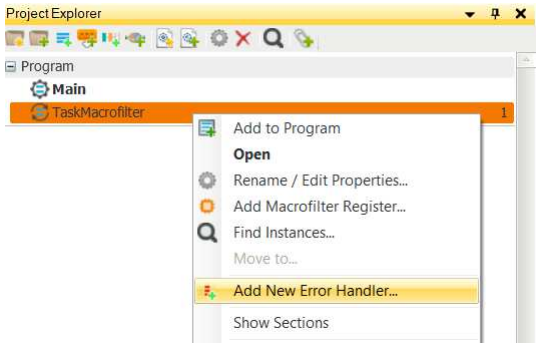
Each Task Macrofilter can have multiple handlers, for different error kinds. If an error occurs, first a handler for the specific kind is checked, and if there is no such handler, the ANY_ERROR handler will be used (if it is defined).

Do not use **ANY_ERROR** handler if you wish your application to continue running after handling the error. Executing this error handler will always cause the application to stop.

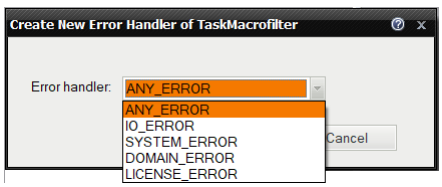
Error handlers for ANY_ERROR should be treated as a way to inform the user about the problem (e.g. write to console, save text logs) before quitting the application.

Adding Error Handlers

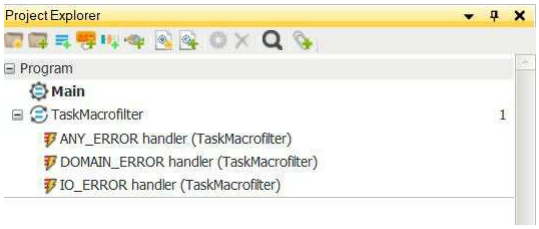
Error handlers can be added by right–clicking a Task Macrofilter in the Project Explorer and choosing "Add New Error Handler ...".



Next, we need to choose the error kind for the handler.



After choosing the kind, it will appear in Project Explorer under the Task Macrofilter to which it was added:

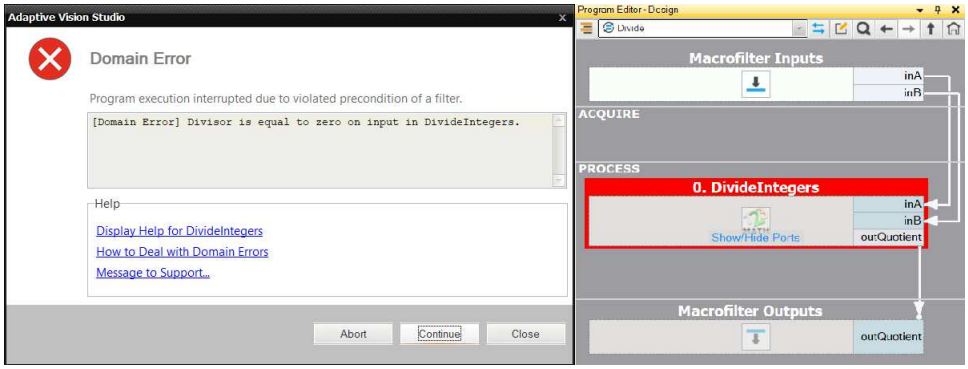


After double–clicking the error handler can be edited.

Program Execution

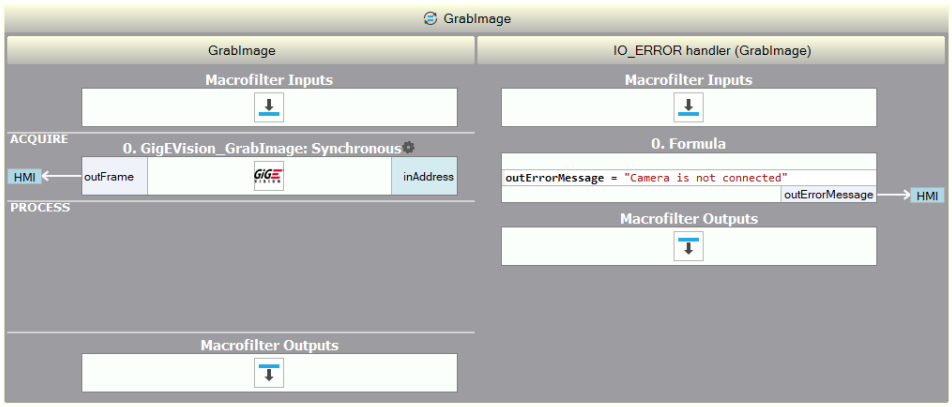
During program execution in the IDE, after an error occurs, there will be a message pop–up, with the option to continue (that is, run the error handler). This message can be turned off in the IDE settings, by changing: *Tools » Settings » Program Execution* : "Break when exception occurred".

In the Runtime environment, understandably, there will be no message and the error handling will be executed immediately.

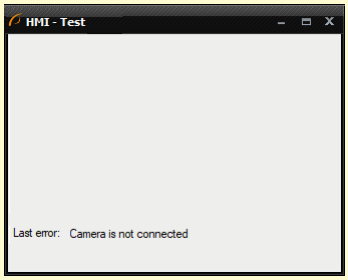


Example

Below is a simple example, acquiring images from a camera in a loop. If the camera is not connected, we enter the error handler, where a message is returned until the error is no longer being raised. Without error handling the program would stop after the first failed image acquisition:



Sample program with error handler. Macrofilter GrabImage has error handler IO_ERROR (right) with error message.



Result after error handler execution.

Result after normal program execution.

Generated C++ Code

In C++ code generation there is an option to generate the error handling, or to leave it up to the user (programmer) to implement it separately. If the error handling option is enabled, the calls of filters in a function (generated from a Task Macrofilter with error handlers) will be enclosed in "try ... catch" blocks, such as:

```
void GrabImage( void )
{
 avl::WebCamera_GrabImageState webCamera_GrabImageState1;
 avl::Image image1;

 try
 {
  for(;;)
  {
   if (!avl::WebCamera_GrabImage(webCamera_GrabImageState1, 0, atl::NIL, image1))
   {
    return;
   }
  }
 }
 catch(const atl::IoError&)
 {
  atl::String string1;

  string1 = atl::String(L"Camera is not connected");
 }
}
```

Offline Mode

1. Worker Tasks
2. HMI Events
3. New, powerful formulas
4. Program Editor Sections and Minimal View
5. Results control
6. Module encryption
7. Elements of the User Interface
8. Program Display
9. The Basic Workflow
10. Data
11. Filters (Tools)
12. Connections
13. Macrofilters
14. Sections
15. Executing Programs
16. Results Control
17. Browsing Macrofilters
18. Execution Breakpoints
19. Knowing Where You Are
20. Toolbox
21. Setting Basic Properties
22. Editing Geometrical Primitives
23. Testing Parameters in Real Time
24. Linking or Loading Data From a File
25. Labeling Connections
26. Invalid Connections
27. Property Outputs
    1. Additional Property Outputs
28. Expanded Input Structures
29. Comment Blocks
30. Extracting Macrofilters (The Quick Way)
31. Creating Macrofilters in the Project Explorer
32. Trick: Configuration File as a Module Not Exported to AVEXE
33. Macrofilter Counter
34. Global Parameters
35. Modules
36. Importing Modules
37. Locking Modules
38. Table of Contents

- Introduction
- Workflow
- Detecting anomalies 1

1. 4. Setting training parameters

- Detecting anomalies 2 (classificational approach)
- Detecting features (segmentation)
- Classifying objects
- Segmenting instances
- Locating points
- Locating objects

1. Overview
2. Common Tasks
3. See Also

- Introduction

1. Prerequisites
2. User Filter Libraries Location
3. Adding New Global User Filter Libraries
4. Adding New Local User Filter Libraries

Introduction

There is a group of filters that require external devices to work correctly, e.g., a camera to grab the images from. The Offline Mode helps to develop vision algorithms without having an access to the real device infrastructure. The ReadFilmstrip makes it possible without even knowing the target device infrastructure and still be able to work on real data.

This article briefly describes what the Offline Mode is, what the Offline Data is and how to access this data and modify it.

Workflow Example

1. Prepare a simple application to collect data directly from the production line using Aurora Vision Studio or import data sets from images stored on the disk.
2. Develop a project using previously prepared data accessing the recorded images with the ReadFilmstrip filter.
3. Replace the ReadFilmstrip filter with the production filter that capture images from the cameras. Replacement is possible with a simple click-and-replace operation.
4. Use of a ready program on the production line. Software is ready to work with real devices (online mode) and with recorded data (offline mode).

At any time during the work, the system maintainer can collect new datasets that can be used to tune the finished system. During the system development developer can playback updated datasets. Algorithms can be tested without any modification of the project's code.

All the data for offline mode is stored in easy to share format. Offline mode is the preferred way to work with big projects with bigger development team.

Online-Only Filters

The filters that require connected external devices can only operate while being connected, thus they are online-only. Great examples of such a filters are:

- GigEVision_GrabImage
- GenICam_GrabImage
- SerialPort_ReadByte

All Online-Only filters are either **I/O Function** filters or **I/O Loop Generator** filters.

The Offline Mode is designed to make the Online-Only filters behave as if they were connected even without the actual connection. For that can happen, the user need to provide the offline data the filter will serve when executed in the Offline Mode.

Although all the Online-Only are I/O filters, the opposite is not always true, i.e. there are I/O filters that can operate in the Offline Mode using their default logic. In most cases this refers to the filters that access the local system resources, e.g.:

- LoadImage
- LoadObject
- GetClockTime

Only Online-Only filters can have their outputs bound to the Dataset Channels. Other filters always execute their default logic.

Accessing the Offline Data

Binding Online-Only Filter Outputs

In Offline Mode, the Online-Only filters are marked `OFFLINE` and are skipped during the execution in the Offline Mode. However, it is possible to make the Online-Only filters execute in the Offline Mode. It is done by binding at least one of their outputs to the Filmstrip channel. The only requirement is that the filter must be inserted into the ACQUIRE section of any Worker Task Macrofilter. The output can then be bound either through the *Bind with Filmstrip* command in the output's context menu or by dragging the channel from the Filmstrip control onto the filter output.

Once the output is bound to the Filmstrip channel, the filter can be executed in the Offline Mode and it's default logic is replaced with loading the data from the disk and forwarding it to the outputs.

All outputs that are not bound, do not provide any data and all connected filters will be skipped during the execution. All macrofilters which contains such filters will be skipped too.

The ReadFilmstrip Filter

Apart from binding the Online-Only filter outputs, the Offline Data may also be accessed with the ReadFilmstrip filter. In that case the Offline Data is exposed by the ReadFilmstrip filter's output, which name corresponds to the assigned Channel name.

The ReadFilmstrip can only be inserted into the ACQUIRE section of the worker macrofilter.

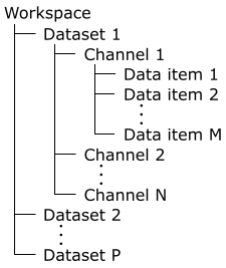
Just like any other filter, the ReadFilmstrip can be inserted into the program with the standard filter lookup and insert facilities like dragging the filter from the Toolbox. Apart from that, the ReadFilmstrip filter can be inserted by drag-and-drop the channel from the Filmstrip control onto the empty space within the Program Editor. This way the inserted filter already assigned to Dataset Channel of choice.

Once the target device infrastructure is known, the filter can be easily replaced with the production-ready, Online-Only filter with the *"Replace..."* command. All the existing connections are reconnected to preserve the current algorithm consistent. All outputs in the replaced filter that are found corresponding to the replaced ReadFilmstrip outputs are bound to the same Dataset Channel, the replaced ReadFilmstrip filter was assigned to.

If the filter ReadFilmstrip was not replaced with an existing I/O filter the execution of this filter will be skipped in Online mode.

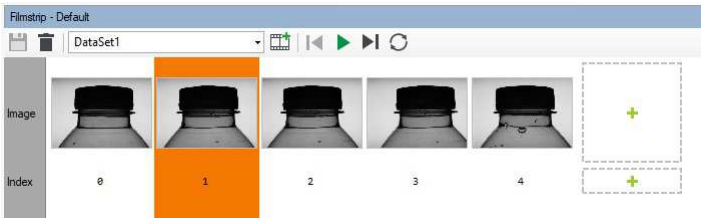
Offline Data Structure

The offline data is organized in the Workspace-Dataset-Channel-Data tree structure:



Workspace structure.

The data items from all channels at the particular index make up the **Sample**. Sample represents a single moment in time in which data was saved.In the Filmstrip rows represent the channels within the current Dataset, whereas columns represent the Samples. In the following screenshot the selected Sample is made up from an image from the *Image* channel and the value "1" from the *Index* channel:



Sample.

At the single iteration of the Worker Task macrofilter, there is one Sample available.

Workspace and Dataset Assignment

Each time an Aurora Vision Studio project is opened or created there is always some Workspace available to that project. It is either the Default Workspace that is created for the user at first application start up or some user-defined Workspace assigned to that project.

There are several rules about the assigning workspace elements to the project elements:

- Single project may have assigned zero or one Workspace.
- One Worker Task may have assigned zero or one Dataset.
- The assignment of the Workspace to the project is performed once any Dataset is used in any Worker Task.
- The assignment of the Dataset to the Worker Task is performed once any Channel is bound to an output or assigned to the ReadFilmstrip filter within that Worker Task.

Note: it is forbidden to use two datasets from different workspaces within the same project. Trying to use a dataset from the other workspace switches datasets in all other Worker Tasks to the datasets with the same names but from the new workspace. It is considered as a project error, if the dataset cannot be found in the new workspace.

Note: Dataset and Channel assignments are name-based. The Worker Task can be successfully switched to any other Dataset as long there is the same set of channels (with the same names and data types) in the new Dataset. Only bound channels count.

In the fresh Aurora Vision Studio installation there is the Default Workspace available for all new projects and for those projects that have not explicitly selected any other Workspace.

Modifying the Offline Data

In general the Offline Data may be considered as some *content* (the channel items) grouped by the workspace, dataset and channel (the structure). Similarly, the Offline Data modifications may be split into two kinds: structural and content.

Structural Modifications

In the workspace structure figure can be seen, that Workspaces, Datasets and Channels are simply collections. Adding and removing workspaces, datasets and channels may be considered as structure modifications. All these operations are available in the Workspaces Window. However, operations common for the current workspace (adding/removing datasets and channels) are also available in the Filmstrip Control so there is no need to jump between controls to perform workspace-related tasks.

By default Aurora Vision Studio starts up with the default Workspace. Initially the default workspace contains a single Dataset with an empty channel of type `Image`. Similarly each new Dataset created in any workspace also contains an empty channel of type `Image`.

Content Modifications

As the Offline content is actually the data that can be used in the filter outputs, specifically the channel items, the easiest way to add a new data is through the big "+" button in the last Filmstrip control's column. The new data editor depends from the type, e.g. for the `Image` type the image files are selected from the disk and the geometrical data, as well as the basic data, is defined through either the appropriate dialog or an inline editor.

Note: When the channel data (e.g. images) is selected from the disk, it is the files at their original locations that are loaded in the offline mode during the execution.

It is also possible to append online data directly to the bound channel. This can be achieved through the *Save* icon at the Filmstrip controls's toolbar:

Saving the sample.

The fundamental types and simple structures are serialized directly in the parent Dataset metadata file (*.dataset). On the other hand, when the acquired online data requires saving to the disk, the channel's storage directory is used and only the path is saved in the *.dataset file. By default the storage directory is the one of the subdirectories within the parent Workspace directory. Though, the user may change the storage directory at any time with the channel's Edit dialog that is available in the Workspaces Window.

Note: All channels within the Dataset have synchronized sizes. That means that manually adding the item to the channel, populates all other channels with data that is default for the channel types. Similarly, saving the online data populates all channels within the dataset either with the data that comes from bound outputs or with the data default for the channel types in case the channel is unbound.

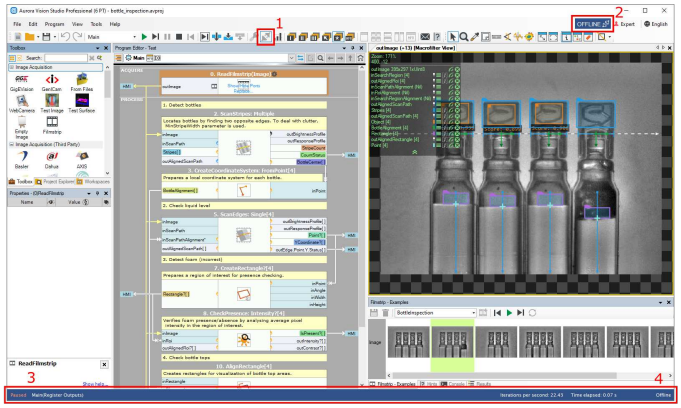
Existing channel items can be modified with the edit button that shows up on hovering the item. The button, once clicked, makes the default editor for the item type opened.:

Editing the channel item.

Activation and Appearance

Main Window

The Aurora Vision Studio main window contains several indicators that signal the current Offline mode and execution state:



Aurora Vision Studio Main Window with marked the Offline mode and execution indicators.

1. The Offline mode button
   
   at the main toolbar. If selected, the Offline mode is active,
2. The button at the top-right corner which visually represents the Mode and Execution:

	Offline Mode	Online Mode
Idle		
Executing		

3. Execution status and the status bar background color. If the program is currently in the execution or paused state, the background color corresponds to the indication button (2) background color,
4. If the Offline mode is active, there is additional *Offline* label at the bottom-right corner of the the window.

Program Editor

The Online-Only filters, unless bound with the Filmstrip channel, are disabled in the Offline mode. It is marked with the `OFFLINE` label over the filter:

Unbound Online-Only filter that is disabled in the Offline mode.

The ReadFilmstrip on the other hand, is disabled in the Online mode and marked with the `NOT IN ONLINE` label:

The ReadFilmstrip filter disabled in the Online mode.

Note: As in case of manually disabled filters, filters disabled due to the current Offline mode make all connected filters disabled too.

See Also

1. Managing Workspaces - extensive description how to manage dataset workspaces in Aurora Vision Studio.
2. Using Filmstrip Control - tool for recording and restoring data from datasets in Aurora Vision Studio.

Summary: Common Terms that Everyone Should Understand

When learning how to create programs in Aurora Vision Studio you will find the below 16 terms appearing in many places. These are important concepts that have specific meanings in our environment. Please, make sure that you clearly understand each of them.

Filter
The basic data processing element. When executed, reads data from its input ports, performs some computations and produces one or several output values. Read more...

Nil
Nothing. We say that an object is Nil when it has not been detected or does not exist. Example: When ReadSingleBarcode filter is executed on an empty image, it will produce Nil value on the output, because there is no barcode that could be found. Read more...

Conditional execution / conditional mode
A mechanism of skipping execution of some filters if the input object has not been detected or does not exist. Read more...

Array
An object representing an ordered collection of multiple elements. All the elements are of the same type. Example: An array of rectangles can be used to represent locations of multiple barcodes detected with the ReadMultipleBarcodes filter. Read more...

Array execution / array mode
A mechanism of executing some filters multiple times in one program iteration, applied when there is an array of objects connected to an input expecting a single object. Read more...

Synchronized arrays
We say that two arrays are synchronized if it is guaranteed that they will always have the same size. Two synchronized arrays usually represent different properties of the same objects. Read more...

Automatic conversion
A mechanism that makes it possible to connect filter ports that have different, but similar types of data. Example: Due to automatic conversions it is possible to use a region as a binary image or an integer number as a real number. Read more...

Optional input
A filter input which can be given the **Auto** value. Example: Image processing filters have an optional inRoi input (region-of-interest). When inRoi is Auto, then the operation is performed on the entire image. Read more...

Global parameter
A named value that can be connected to several filters. When it is changed, all the connected filters will alter their execution. Read more...

Structure
A composite type of data that consist of several named fields. Example: Point2D is a structure consisting of two fields: X (Real) and Y (real). Read more...

Macrofilter
Subprogram. Internally it is a sequence of filters, but from outside it looks like a single filter. Macrofilters are used to: (1) organize bigger programs and (2) to re-use the same sequence of filters in several places. Read more...

Step
Macrofilter representing a logical part of a program consisting of several filters connected and configured for a specific purpose. Read more...

Variant step
Macrofilter representing a logical part of a program consisting of several alternative execution paths. During each execution exactly one of the paths is executed. Read more...

Task
Macrofilter representing a logical part of a program performing some computations in a cycle. A loop. Read more...

Generic filter
A filter that can work with different types of objects. The required type has to be specified by the user when the filter is dropped to the Program Editor. Read more...

Domain error
An error appearing during program execution when a filter receives incorrect input values. Examples: (1) division by zero, (2) computing the mass center of an empty region. Read more...

Summary: Common Filters that Everyone Should Know

There are 16 special filters in Aurora Vision Studio that are commonly used for general purpose calculations, but whose meaning may be not obvious. Please review them carefully and make sure that you understand each of them.

Filters Related to Conditional Data

MakeConditional
Creates a conditional value. Copies the input object to the output if the associated condition is met, or returns Nil otherwise.

MergeDefault
Replaces Nil with a default value while copying the input object to the output. It is used to substitute for a value that is missing due to conditional execution.

RemoveNils
Gets an array with conditional elements and removes all the Nil values from it.

Filters Related to Arrays

CreateArray
Creates an array from up to eight individual elements.

CreateUniformArray, CreateIntegerSequence, CreateRealSequence
These filters create simple arrays of the specified size.

ClassifyByRange, ClassifyByPredicate, ClassifyRegions, ClassifyPaths
These filters get one array on the input and split the elements into two or more output arrays depending on some condition.

GetArrayElement, GetArrayElements, GetMultipleArrayElements
These filters get an array of elements and output one or several elements from specified indices.

GetMinimumElement, GetMaximumElement, GetMinimumRegion, GetMaximumRegion, GetMinimumPath, GetMaximumPath
These filters get an array of elements and output one element depending on some condition.

FlattenArray (JoinArrays_OfArray)
Used when you have a two-dimensional array (e.g. Point2DArrayArray), but you need a flat list of all individual elements (e.g. Point2DArray).

Filters Related to Loops

EnumerateIntegers, EnumerateReals, EnumerateElements, EnumerateFiles, EnumerateImages
These filters create loops of consecutive numbers, array elements or files on disk. The loop is finished when the end of the specified collection is reached.

Loop
Creates a loop that ends when the **inShouldLoop** input gets False.

LastTwoObjects
Returns two values: one from the current iteration and one from the previous one.

AccumulateElements, AccumulateArray
These filters create arrays by joining single elements or entire arrays that appear in consecutive iterations.

CountConditions
Calculates how many times some condition was **True** across all iterations.

Other

CopyObject
Does a very simple thing – copies the input object to the output. Useful for creating values that should be send to the HMI at some point of the program.

ChooseByPredicate
Gets two individual elements and outputs one of them depending on a condition.

7. Programming Tips

Table of content:

Formulas Migration Guide to version 5.0

Introduction

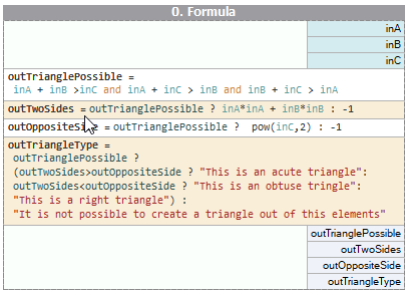
Aurora Vision Studio 5.0 comes with many new functions available directly in the formula block. Moreover, many operations, which were previously done using filters, can now be accomplished directly in the formulas. This way, the complexity of the inspection program can be reduced significantly. This document focuses on pointing out some of the key differences between formulas in Aurora Vision Studio 5.0 and its previous versions. For all functions please refer to the Formulas article in the Programming Reference section of this documentation.

Conditional operator

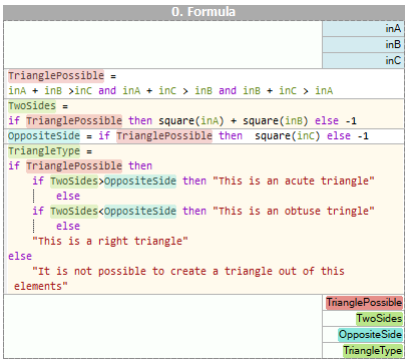
Before 5.0, a ternary operator **`?:`** was used to perform a conditional execution inside a formula. This compact syntax, however, might prove quite challenging and hard to understand, especially to users, who are not familiar with textual programming. In order to overcome this issue, a more user friendly **`if-then-else`** syntax has been introduced, while also keeping the option of using the ternary operator.

In the example below, apart from the new conditional syntax, a new **`square`** function has also been used, instead of `inA*inA`.

Before 5.0



From 5.0

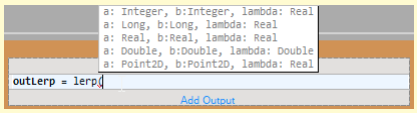


Mathematical functions

- Lerp - computes a linear interpolation between two numeric values.

Before 5.0

From 5.0



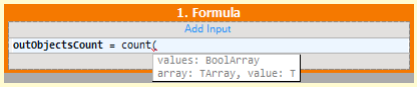
- Square - raises the argument to the power of two.
- Hypot - calculates the hypotenuse of triangle.
- Clamp - limits the specific value to the range.

Statistic and array processing functions

- indexOfMin - returns the index of the smallest of the values in the input array.
- indexOfMax - returns the index of the largest of the values in the input array.
- Count value in array - depending on the chosen variant, returns the number of items equal to True or counts the number of elements in the array that are equal to the value given in the second argument.

Before 5.0

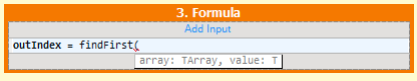
From 5.0



- contains - checks if the specified array contains a value.
- Find - searches the specified array for instances equal to the given value and return the index of the first found item.

Before 5.0

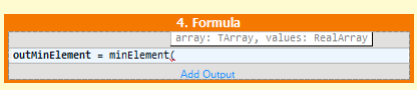
From 5.0



- findLast - searches the specified array for instances equal to the given value and return the index of the last found item.
- findAll - searches the specified array for instances equal to the given value and return an array of indices of all found items.
- minElement - returns an array element that corresponds to the smallest value in the array of values.

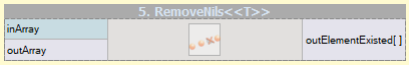
Before 5.0

From 5.0



- maxElement - returns an array element that corresponds to the biggest value in the array of values.
- removeNils - removes all Nil elements from an array.

Before 5.0

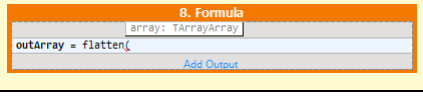


From 5.0

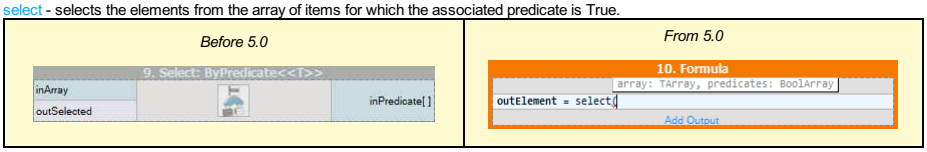
- flatten - takes an array of arrays, and concatenates all nested arrays creating a single one-dimensional array containing all individual elements.

Before 5.0

From 5.0

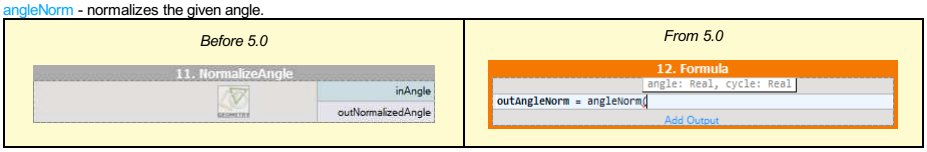


- select - selects the elements from the array of items for which the associated predicate is True.

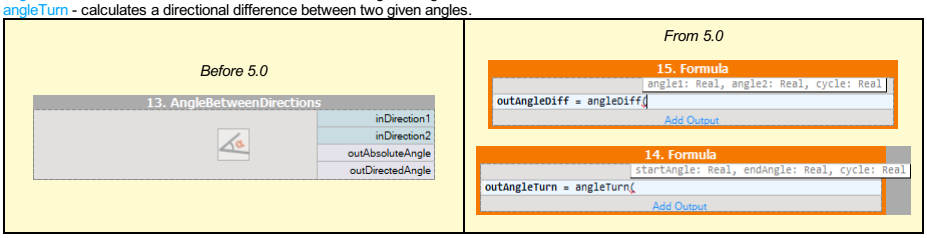


Geometry processing functions

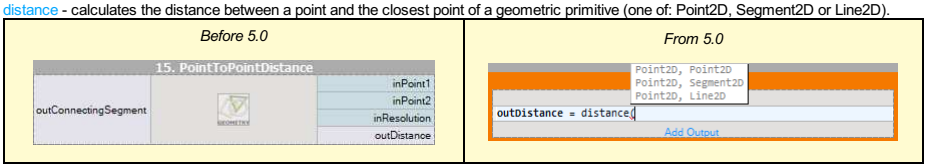
- angleNorm - normalizes the given angle.



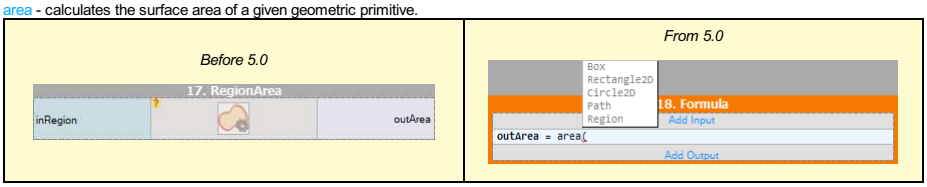
- angleDiff - calculates an absolute difference between two given angles.
  angleTurn - calculates a directional difference between two given angles.



- distance - calculates the distance between a point and the closest point of a geometric primitive (one of: Point2D, Segment2D or Line2D).



- area - calculates the surface area of a given geometric primitive.



- dot - calculates the dot product of two vectors.
- normalize - normalizes the specified vector.

For complete information about the syntax and semantics please refer to the Formulas article in the Programming Reference section of this documentation.

Dealing with Domain Errors

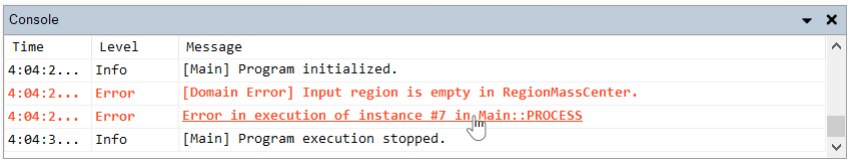
Introduction

Domain Errors stop the program execution when a precondition of a filter is not met. In general it is not possible to predict all possible *Domain Errors* automatically and this is responsibility of the user to assure that they will not appear in the final version of the program.

Example *Domain Errors*:

- The SelectChannel filter trying to get the second channel of a monochromatic image will raise "[Domain Error] Channel index out of range in SelectChannel.".
- The DivideIntegers filter when invoked with **inB** = 0 will raise "[Domain Error] Divisor is equal to zero on input in DivideIntegers.".
- The RegionMassCenter filter invoked on an empty region will raise "[Domain Error] Input region is empty in RegionMassCenter.".

Domain Errors are reported in the Console window. You can highlight the filter instance that produced a *Domain Error* by clicking on the link that appears there:



A link to a filter in the Console window.

Dealing with Empty Objects

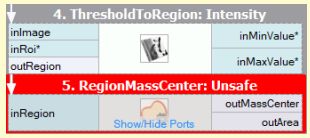
One of the most common sources of *Domain Errors* are the filters whose outputs are undefined for the empty object. Let us take the RegionMassCenter filter under consideration: where is the center of an empty region's mass? The answer is: it is undefined. Being unable to compute the result, the filter throws a *Domain Error* and stops the program execution.

If a program contains such filters and it is not possible to assure that the precondition will always be met, then the special cases have to be explicitly resolved. One way to deal with empty objects is to change the filter variant from *Unsafe* to *OrNil*. It will simply skip the execution and assign the Nil value to the filter's outputs. You can find more information on *OrNil* filter variants here.

Another possible solution is to use one of the *guardian* filters, which turn empty objects into conditional processing. These filters are: SkipEmptyArray, SkipEmptyRegion, SkipEmptyPath, SkipEmptyProfile, SkipEmptyHistogram and SkipEmptyDataHistogram.

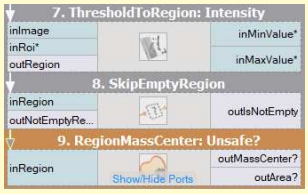
Examples

An example of a possibly erroneous situation is a use of the ThresholdToRegion filter followed by RegionMassCenter. In some cases, the first filter can produce an empty region and the second will then throw a *Domain Error*. To prevent that, the *OrNil* variant should be used to create an appropriate data flow.



RegionMassCenter throwing a Domain Error on empty input region. The program execution is stopped.

RegionMassCenter properly used in *OrNil variant.*



RegionMassCenter properly preceded with SkipEmptyRegion. Conditional processing appears.

Another typical example is when one is trying to select an object having the maximum value of some feature (e.g. the biggest blob). The GetMaximumElement filter can be used for this purpose, but it will throw a *Domain Error* if there are no objects found (if the array is empty). In this case using the *OrNil* option instead of the *Unsafe* one solves the problem.

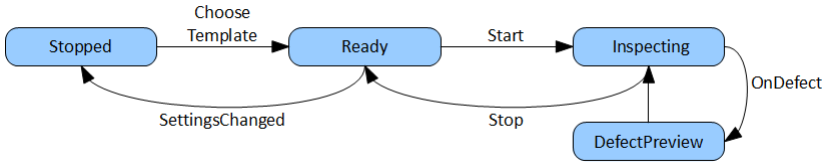
Programming Finite State Machines

Introduction

Industrial applications often require the algorithm to work in several different modes. Two most typical examples are:

- An application that has several user interface modes, e.g. the inspection mode and the model definition mode.
- An application that guides a robot arm with modes like "searching for an object", "picking an object" etc.

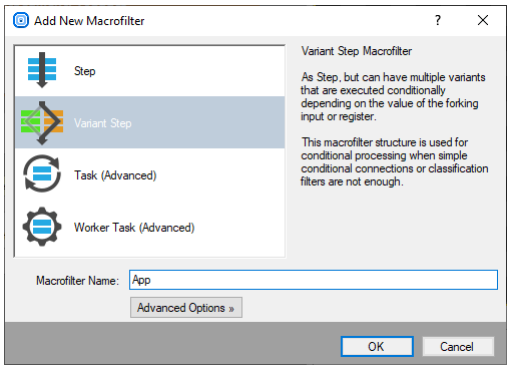
Such applications are best described and programmed as Finite State Machines (FSM). Below is an example diagram depicting one in a graphical way:



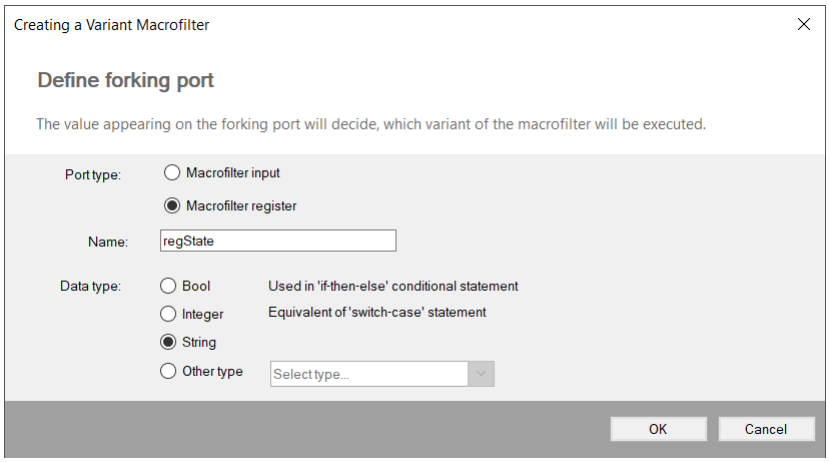
Instructions

In Aurora Vision Studio, Finite State Machines can be created with variant macrofilters and registers. The general program schema consists of a *main loop* macrofilter (a task, usually the "Main" macrofilter) and a variant macrofilter within it with variants corresponding to the states of the Finite State Machine. Individual programs may vary in details, but in most cases the following instructions provide a good starting point:

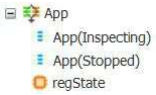
1. Create a *Variant Step* macrofilter (e.g. "App") for the State Machine with variants corresponding to individual states.



2. Use a forking register (e.g. "regState") of the String type, so that you can assign clear names to each state.



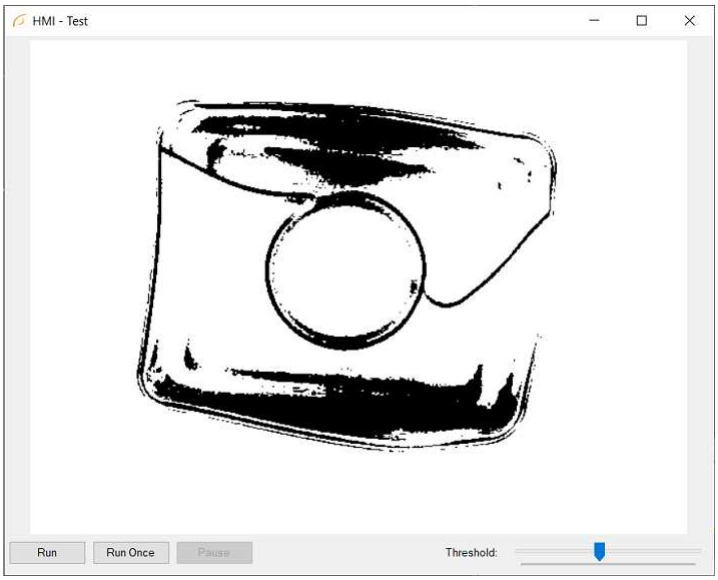
3. At first, you will have only one default variant. Remove it and add one variant with a meaningful label for each state.



4. Do not forget to set an appropriate initial value for the forking register (the initial state).
5. Add macrofilter inputs – usually for the input image and major parameters.
6. Add macrofilter outputs that will contain information about the results. In each state these outputs can be computed in a different way.
7. Create an instance of this variant macrofilter in some task with a loop, e.g. in the "Main" macrofilter.
8. In each state compute the value of the next state and connect the result to the *next* port of the forking register. Typically, an HMI button's outputs are used here as inputs to formula blocks.
9. Optional: Create a formula in the main loop task for enabling or disabling individual controls, depending on the current state (also requires exposing the current state value as the variant step's output).

Example

For a complete example, please refer to the "HMI Start-Stop" example program. It is based on two states: "Inspecting" and "Stopped". In the first state the input images are processed, in the second they are not. Two buttons, *Start* and *Stop*, allow the user to control the current state.



An example application with a simple Finite State Machine

Note: There is also a standard control, ProgramControlBox, which provides "Start", "Iterate" and "Pause" buttons. It is very easy to use, but it is not customizable. Finite State Machines allow for creating any custom set of states and transitions.

Blocking Filters

Please note that while using the Finite State Machine approach to create more complex program logic, you should avoid the use of image acquisition filters configured for the triggered mode (e.g. GigEVision_GrabImage). These filters are *blocking*, which means that the program execution will halt waiting for the trigger and no other computations are performed during this time. Events, such as button clicks, will not get processed until the next image is acquired.

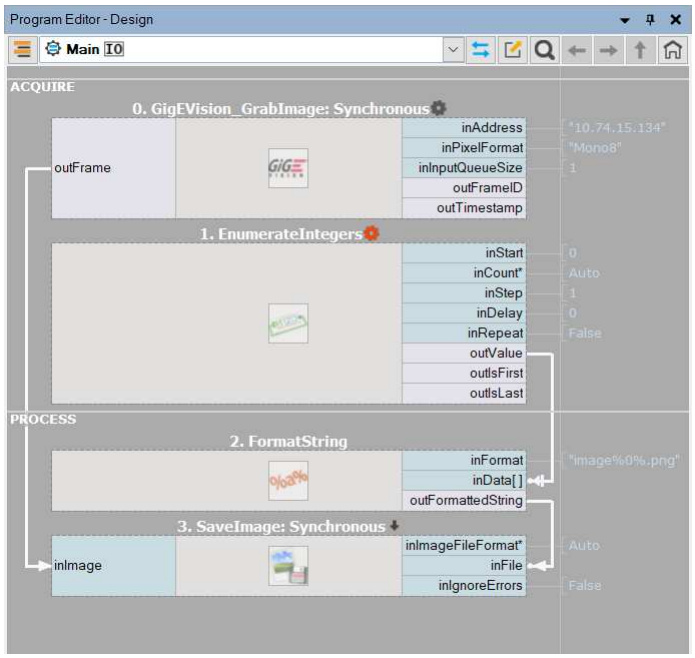
If your application is using a camera or multiple cameras in the triggered mode, then it is advisable to use filters with timeout – GigEVision_GrabImage_WithTimeout or GenICam_GrabImage_WithTimeout. These filters return *Nil* after the time is out and no image has been acquired. It is thus possible to use them in a loop, in which events can be processed and the inspection part is executed conditionally – only when there is a new input image.

See also: Handling Events in Low Frame-Rate Applications.

Recording Images

Continuous Recording

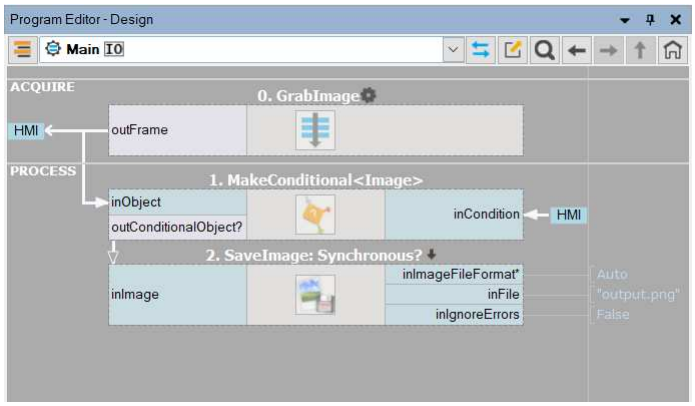
Images can be recorded to a disk with this simple program:



The RecordAllImages program.

Recording Single Images

Sometimes it might be useful to have a simple HMI with a button triggering the action of saving the current image to a file. This can be done by conditional execution of the SaveImage filter:



The RecordSingleImages program.

The HMI for this program consists of an image preview and an impulse button:

HMI for recording single images.

Sorting, Classifying and Choosing Objects

Introduction

Sorting, classifying and choosing objects are three types of general programming tasks that are often an important part of machine vision inspections. For example, it might be asked how to sort detected objects by the Y coordinate, how to decide which objects are correct on the basis of a computed feature, or how to select the region having the highest area. In Aurora Vision Studio, by design, there are no filters that do just that. Instead, for maximum flexibility such tasks are divided into two steps:

1. Firstly, an array of values (features) describing the objects of interest has to be created.
2. Secondly, both the array of objects, and the array of the corresponding values, have to be passed to an appropriate sorting, classifying or choosing filter.

This approach makes it possible to use arbitrary criteria, also ones that are very specific to a particular project and have not been anticipated by the original creators of the software.

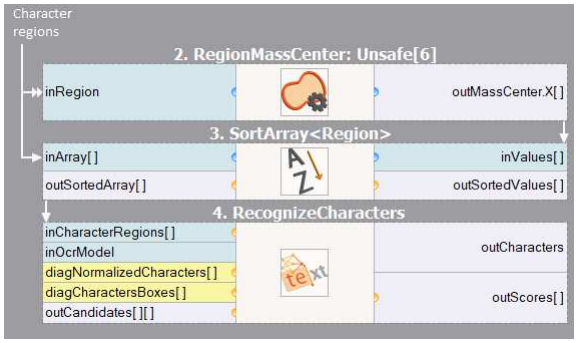
Sorting Elements of an Array

Let us consider the question of how to sort an array of objects by one of the coordinates or by some other computed feature. We will show how to achieve this on an example with manual sorting of characters from left to right for an OCR. Here is the input image with an overlay of the detected character regions:

Segmented characters which might be out of order.

After passing this array of character regions to the RecognizeCharacters filter with the **inCharacterSorting** input set to *None*, we will get the result "MAEPXEL", which has all characters recognized correctly, but in a random order.

Here is a sample program fragment that sorts the input regions by the X coordinates of their mass centers:



Sorting regions from left to right before passing them to OCR.

The final result is "EXAMPLE", which is both correct and in order.

Classifying Elements of an Array

An example of object classification has already been shown in the very first program example, in the First Program: Simple Blob Analysis article, where blobs were classified by the elongation feature with a dedicated filter, ClassifyRegions. Classification is a common step in many programs that process multiple objects and there is a group Classify of general filters which can classify objects by various criteria. The general scheme of object classification is:

1. Detect objects – for example using Template Matching or segmentation (e.g. ThresholdToRegion + SplitRegionIntoBlobs).
2. Compute some features – for example the area, elongation, mean brightness etc.
3. Classify the objects – using one of the Classify filters.

The general idea of how to use the ClassifyByRange filter is shown below:

The usage of the ClassifyByRange filter.

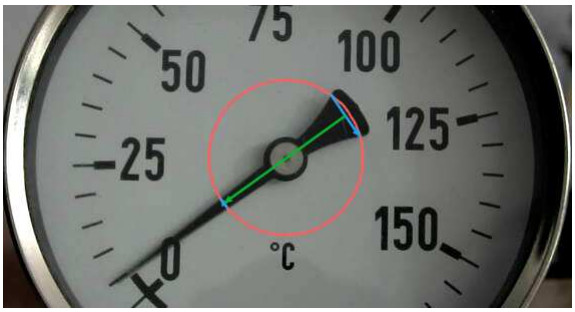
There are four elements:

- The objects to be classified are connected to the **inArray** input.
- The values corresponding to the objects are connected to the **inValues** input.
- The **inMinimum** and **inMaximum** inputs define the acceptable ranges for the values.
- The 3 outputs contain 3 arrays – containing the objects whose values were respectively: in the range, below the range and above the range.

Other filters that can be used for classification are: ClassifyByPredicate – when instead of associated real values we have associated booleans (*True* / *False*) and ClassifyByCase – when instead of associated real values we have associated indices of classes that the corresponding objects belong to.

Choosing an Element out of an Array

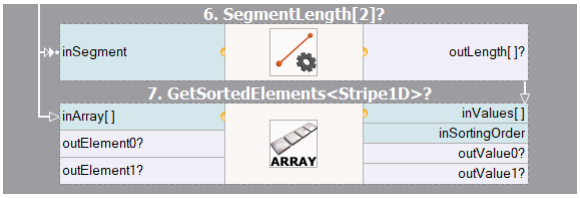
Choosing differs from sorting and classification mainly in that it is intended to produce a single value instead of an array. Let us consider the example of finding the orientation of a meter's needle:



The task of finding the orientation of a meter's needle.

There are two segments (blue) detected with the ScanExactlyNStripes filter along a circle (red). At this stage we have an array of two segments, but we do not know, which is the smaller and which is the bigger. This information is necessary to determine the orientation (green).

To solve this problem we need to compute an array of the segment's lengths using the SegmentLength filter and then get the appropriate elements with the GetSortedElements filter. This filter will produce the smaller element on the first output and the bigger element on the second output (see the picture below). Alternatively, we could use two separate filters: GetMinimumElement and GetMaximumElement.



The solution for choosing the smaller segment and the bigger segment.

Choosing one Object out of Several Individual Objects

Sometimes we have several individual objects to choose from instead of an array. For example we might want to choose one of two images for display in the HMI depending on whether some check-box is checked or not. In such case it is advisable to use the ChooseByPredicate filter, which requires two objects on the inputs and returns one of them on the output. Which one is chosen is determined by the **inCondition** input.

Warning: For efficiency reasons you should use the ChooseByPredicate filter only if the two input objects can be obtained without lengthy computations. If there are two alternative ways to compute a single value, then variant macrofilters should be used instead.

c++

The ChooseByPredicate filter is very similar to the ternary (?:) operator from the C/C++ programming language.

Choosing an Object out of the Loop

Yet another case is choosing an object out of objects that appear in different iterations. As with arrays, also here we need an associated criterion, which can be real numbers, boolean values or status of a conditional value. The available filters are:

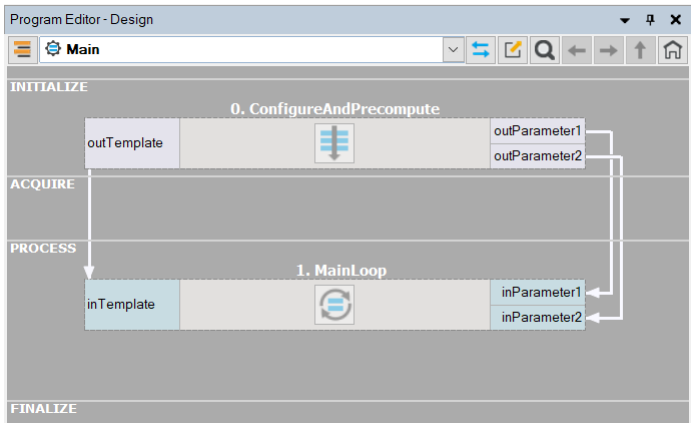
- LoopMaximum – chooses the object whose associated value was the highest.
- LoopMinimum – chooses the object whose associated value was the lowest.
- LastMarkedObject – chooses the most recent object whose associated boolean value was *True*.
- LastNotNil – chooses the most recent object which actually existed (was different than *Nil*).

Optimizing Image Analysis for Speed

General Rules

Rule #1: Do not compute what you do not need.

- Use image resolution well fitted to the task. The higher the resolution, the slower the processing.
- Use the **inRoi** input of image processing to compute only the pixels that are needed in further processing steps.
- If several image processing operations occur in sequence in a confined region then it might be better to use CropImage at first.
- Do not overuse images of types other than UInt8 (8-bit).
- Do not use multi-channel images when there is no color information being processed.
- If some computations can be done only once, move them before the main program loop, or even to a separate program. Below is an example of a typical structure of the "Main" macrofilter that implements this advice. There are two macrofilters: the first one is responsible for once-only computations, and the second is a Task implementing the main program loop:



Typical program structure separating precomputing from the main loop.

Rule #2: Prefer simple solutions.

- Do not use Template Matching if more simple techniques as Blob Analysis or 1D Edge Detection would suffice.
- Prefer pixel-precise image analysis techniques (Region Analysis) and the *Nearest Neighbour* (instead of *Bilinear*) image interpolation.
- Consider extracting higher level information early in the program pipeline – for example it is much faster to process Regions than Images.

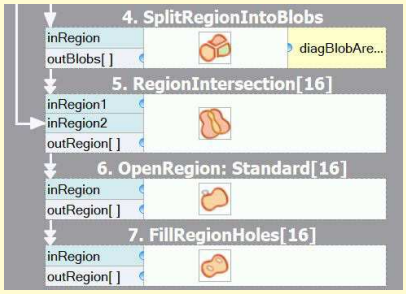
Rule #3: Mind the influence of the user interface.

- Note that in the development environment displaying data on the preview windows takes much time. Choose *Program » Previews Update Mode » Disable Visualization* to get performance closer to the one you can expect in the runtime environment.
- In the runtime environment use the VideoBox control for image display. It is highly optimized and can display hundreds of images per second.
- Using the VideoBox controls, prefer the setting of **SizeMode: Normal**, especially if the image to be displayed is large. Also consider using DownsampleImage or ResizeImage.
- Prefer the *Update Data Previews Once an Iteration* option.
- Mind the Diagnostic Mode. Turn it off whenever you need to test speed.
- Pay attention to the information provided by the Statistics window. Before optimizing the program, make sure that you know what really needs optimizing.

Rule #4: Mind the influence of the data flow model.

Data flow programming allows for creating high speed machine vision applications nearly as well as the standard C++ programming. This, however, requires meeting an assumption that we are using high-level tools and image analysis is the main part. On the other hand, for low level programming tasks – like using many simple filters to process high numbers of pixels, points or small blobs – all interpreted languages will perform significantly slower than C++.

- For performance-critical low-level programming tasks consider User Filters.
- Prefer formula blocks over arithmetic filters like AddIntegers or DivideReals.
- Use a lower number of higher level filters (e.g. RotatePath) instead of a big number of low level filters or formulas (e.g. calculating coordinates of all individual points of the path).
- Avoid using low-level filters (such as MergeDefault or ChooseByPredicate) with non-primitive types such as Image or Region. Filters perform full copying of at least one of the input objects. Prefer using Variant Step Macrofilters instead.
- Mind the connections with conversions (the arrow head with a dot) – there are additional computations, which is some cases (e.g. RegionToImage) might take some time. If the same conversion is used many times, then it might be better to use the converting filter directly.
- The sequence of filters with array connections may produce a lot of data on the outputs. If only the final result is important, then consider extracting a macrofilter that will be executed in array mode as a whole and inside of it all the connections will be basic. For example:



Before optimizing: There are several arrays of intermediate results.

After optimizing: Only the final array is computed, reducing memory consumption. Please note that the macrofilter input is of Region, not RegionArray type.

Common Optimization Tips

Apart from the above general rules, there are also some common optimization tips related to specific filters and techniques. Here is a check-list:

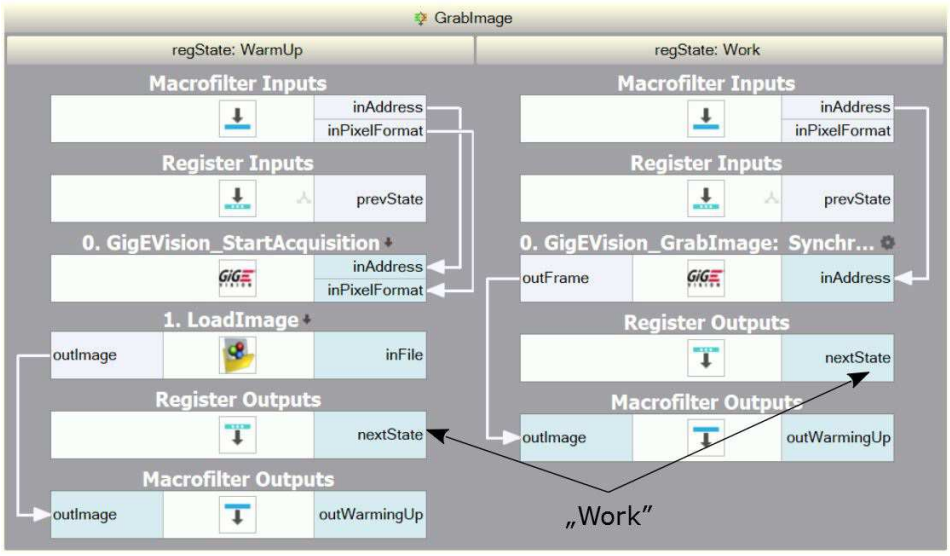
- Template Matching: Do not mark the entire object as the template region, but only mark a small part having a unique shape.
- Template Matching: Prefer high pyramid levels, i.e. leave the **inMaxPyramidLevel** set to *Auto*, or to a high value like between *4* and *6*.
- Template Matching: Prefer **inEdgePolarityMode** set not to Ignore and **inEdgeNoiseLevel** set to Low.
- Template Matching: Use as high values of the **inMinScore** input as possible.
- Template Matching: If you process high-resolution images, consider setting the **inMinPyramidLevel** to *1* or even *2*.
- Template Matching: When creating template matching models, try to limit the range of angles with the **inMinAngle** and **inMaxAngle** inputs.
- Template Matching: Do not expect high speed when allowing rotations and scaling at the same time. Also model creation can take much time or even fail with an "out of memory" error.
- Template Matching: Consider limiting **inSearchRegion**. It might be set manually, but sometimes it also helps to use Region Analysis techniques before Template Matching.
- Template Matching: Decrease **inEdgeCompleteness** to achieve higher speed at the cost of lower reliability. This might be useful when the pyramid cannot be made higher due to loss of information.
- Do not use these filters in the main program loop: CreateEdgeModel1, CreateGrayModel, TrainOcr_MLP, TrainOcr_SVM.
- If you always transform images in the same way, consider filters from the Image Spatial Transforms Maps category instead of the ones from Image Spatial Transforms.
- Do not use image local transforms with arbitrary shaped kernels: DilateImage_AnyKernel, ErodeImage_AnyKernel, SmoothImage_Mean_AnyKernel. Consider the alternatives without the "_AnyKernel" suffix.
- SmoothImage_Median can be particularly slow. Use Gaussian or Mean smoothing instead, if possible.

Application Warm-Up (Advanced)

An important practical issue in industrial applications with triggered cameras is that the first iteration of a program must often already be executed at the full speed. There are however additional computations performed in the first iterations that have to be taken into account:

- Memory buffers (especially images) for output data are allocated.
- Memory buffers get loaded to the cache memory.
- External DLL libraries get delay-loaded by the operating system.
- The modern CPU mechanics, like branch prediction, get trained.
- Connections with external devices (e.g. cameras) get established.
- Some filters, especially ones from 1D Edge Detection and Shape Fitting, precompute some data.

These are things that result from both the simplified data-flow programming model, as well as from the modern architectures of computers and operating systems. Some, but not all, of them can be solved with the use of Aurora Vision Library (see: When to use Aurora Vision Library?). There is however, an idiom that might be useful also with Aurora Vision Studio – it is called "Application Warm-Up" and consists in performing one or a couple of iterations on test images (recorded) before the application switches to the operational stage. This can be achieved with the following "GrabImage" variant macrofilter:



An example "GrabImage" macrofilter designed for application warming-up.

The "GrabImage" variant macrofilter shown above is an example of how application warm-up can be achieved. It starts its operation in the "WarmUp" variant, where it initializes the camera and produces a test image loaded from a file (which has exactly the same resolution and format as the images acquired from the camera). Then it switches to the "Work" variant, where the standard image acquisition filter is used. There also an additional output **outIsWarmingUp** that can be used for example to suppress the output signals in the warming-up stage.

Configuring Parallel Computing

The filters of Aurora Vision Studio internally use multiple threads to utilize the full power of multi-core processors. By default they use as many threads as there are physical processors. This is the best setting for majority of applications, but in some cases another number of threads might result in faster execution. If you need maximum performance, it is advisable to experiment with the ControlParallelComputing filter with both higher and lower number of threads. In particular:

- If the number of threads is **higher** than the number of physical processors, then it is possible to utilize the Hyper-Threading technology.
- If the number of threads is **lower** than the number of physical processors (e.g. 3 threads on a quad-core machine), then the system has at least one core available for background threads (like image acquisition, GUI or computations performed by other processes), which may improve its responsiveness.

Configuring Image Memory Pools

Among significant factors affecting filter performance is memory allocation. Most of the filters available in Aurora Vision Studio re-use their memory buffers between consecutive iterations which is highly beneficial for their performance. Some filters, however, still allocate temporary image buffers, because doing otherwise would make them less convenient in use. To overcome this limitation, there is the filter ControlImageMemoryPools which can turn on a custom memory allocator for temporary images.

There is also a way to pre-allocate image memory before first iteration of the program starts. For this purpose use the InspectImageMemoryPools filter at the end of the program, and – after a the program is executed – copy its **outPoolSizes** value to the input of a ChargeImageMemoryPools filter executed at the beginning. In some cases this will improve performance of the first iteration.

Using GPGPU/OpenCL Computing

Some filters of Aurora Vision Studio allow to move computations to an OpenCL capable device, like a graphics card, in order to speed up execution. After proper initialization, OpenCL processing is performed completely automatically by suitable filters without changing their use pattern. Refer to "Hardware Acceleration" section of the filter documentation to find which filters support OpenCL processing and what are their requirements. Be aware that the resulting performance after switching to an OpenCL device may vary and may not always be a significant improvement relative to CPU processing. Actual performance of the filters must always be verified on the target system by proper measurements.

To use OpenCL processing in Aurora Vision Studio the following is required:

- a processing device installed in the target system supporting OpenCL C language in version 1.1 or greater,
- a proper and up-to-date device driver installed in the system,
- a proper OpenCL runtime software provided by its vendor.

OpenCL processing is supported for example in the following filters: RgbToHsi, HsiToRgb, ImageCorrelationImage, DilateImage_AnyKernel.

To enable OpenCL processing in filters an InitGPUProcessing filter must be executed at the beginning of a program. Please refer to that filter documentation for further information.

When to use Aurora Vision Library?

Aurora Vision Library is a separate product for the C++ programmers. The performance of the functions it provides is roughly the same as of the filters provided by Aurora Vision Studio. There are, however, some important cases when the overall performance of the compiled code is better.

Case 1: High number of simple operations

There is an overhead of about 0.004 ms on each filter execution in Studio. That value may seem very little, but if we consider an application which analyzes 50 blobs in each iteration and executes 20 filters for each blob, then it may sums up to a total of 4 ms. This may already be not negligible. If this is only a small part of a bigger application, then User Filters might be the right solution. If, however, this is how the entire application works, then the library should be used instead.

Case 2: Memory re-use for big images

Each filter in Aurora Vision Studio keeps its output data on the output ports. Consecutive filters do not re-use this memory, but instead create new data. This is very convenient for effective development of algorithms as the user can see all intermediate results. However, if the application performs complex processing of very big images (e.g. from 10 megapixel or line-scan cameras), then the issue of memory re-use might become critical. Aurora Vision Library may then be useful, because only at the level of C++ programming the user can have the full control over the memory buffers.

Aurora Vision Library also makes it possible to perform in-place data processing, i.e. modifying directly the input data instead of creating new objects. Many simple image processing operations can be performed in this way. Especially the Image Drawing functions and image transformations in small regions of interest may get a significant performance boost.

Case 3: Initialization before first iteration

Filters of Aurora Vision Studio get initialized in the first iteration. This is for example when the image memory buffers are allocated, because before the first image is acquired, the filters do not know how much memory they will need. Sometimes, however, the application can be optimized for specific conditions and it is important that the first iteration is not any slower. On the level of C++ programming this can be achieved with preallocated memory buffers and with separated initialization of some filters (especially for 1D Edge Detection and Shape Fitting filters, as well as for image acquisition and I/O interfaces). See also: Application Warm-Up.

Understanding OrNil Filter Variants

What are OrNil filter variants

Filter variants distinguish various approaches to a single task and are packed into a single task related group. Most of the time filter variants will use a distinct algorithm or a different set of input data to achieve the same result at the end. In case of *OrNil* filter variants the difference lies in the error handling output data type. *OrNil* filter variants are available when a filter expects a non-empty data collection on input and make it possible to skip the execution instead of stopping it with a Domain Error in the Unsafe variant. When this happens all affected outputs are set to the Nil value.

Why were OrNil filter variants introduced

Filters like RegionMassCenter, GetArrayElement or PathBoundingRectangle expect a non-empty data collection on input in order to extract certain attributes based on the values. When the collection is empty and the Unsafe variant is used, the execution is stopped with a Domain Error. If it is beneficial to simply skip such operation instead of stopping the program, SkipEmpty(Collection) filters can be used to convert execution to conditional processing and skip it entirely when the collection is empty. *OrNil* filter variants are a natural progression as they require less work and provide the same capability.

When should OrNil filter variants be used

Use *OrNil* filter variants when there's a need to skip the execution after invalid input data was fed to a filter instead of stopping it.

When do OrNil filter variants skip the execution

The conditions vary depending on the filter. Most of the time the main condition is an empty data collection on input. Some *OrNil* filter variants may have additional conditions described on their help pages. Do note though that OrNil filter variants will still throw Domain Error exceptions if the values provided are outside of the domain, e.g. negative array index.

RegionMassCenter throwing a Domain Error on empty input region. The program execution is stopped.

RegionMassCenter not being executed due to conditional processing introduced with SkipEmptyRegion. Program execution continues.

RegionMassCenter in the OrNil variant safely executed and introduces conditional processing on its output. Program execution continues.

Working with XML Trees

Table of contents

Introduction

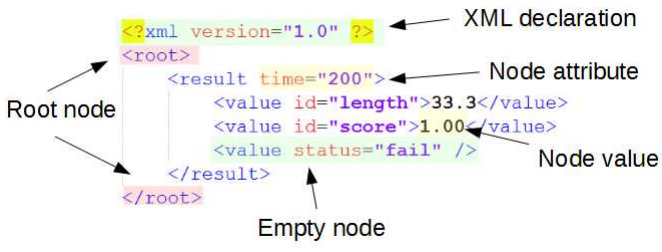
Extensible Markup Language (XML) is a markup language used for storing data in human readable text-format files. XML standard is very popular in exchanging data between two systems. XML provides a clear way to create text files which can be readable both by a computer application and a human.

An XML file is organized into a tree structure. The basic element of this tree structure is called a node. A node has a name, attributes and can have a text value. The example below shows a node with the name 'Value' and with two attributes 'a', 'b'. The example node contains text '10.0'.

```
<Value a="1.0" b="2.0">10.0</Value>
```

An XML node can also contain other nodes which are called children. The example below shows a node named 'A' with two children: 'B' and 'C'.

```
<A>
  <B />
  <C />
</A>
```



Descriptions of XML elements.

The XML technology is commonly used for:

- Configuration files
- Data exchange format between applications
- Structured data sets
- Simple file-based databases

To load or store data in the XML format, two filters are necessary: Xml_LoadFile and Xml_SaveFile. Also two conversions are possible: XmlNodeToString and StringToXmlNode. These filters are commonly used to perform operations on data received from different sources like Serial Port or TCP/IP.

Accessing XML Nodes

After loading an XML file all node properties can be accessed with AccessXmlNode filter.

Xml_GetChildNode and filters from group Xml_GetAttribute are used to access directly a node's element using its known number or name.

To perform more complex access operations on elements using specified criteria, please read: Selecting Elements Using XPath Query.

Creating an XML Document

The process of creating XML documents is organized in a bottom-up approach. At the beginning, the lowest nodes in the tree must be created. Then, such nodes can be added to a node at a higher level in the tree and that node can be added to some yet higher node and so on. The node without a parent which is the highest level in hierarchy is called root.

To create an XML node Xml_CreateNode filter should be used. Then, the newly created node should be added to the root node. The new node can have text value assigned.

Created XML nodes should be added to the tree using Xml_AddChildNodes and Xml_AddChildNodes_OfArray filters. New nodes will be added to the end of the list of children.

Also attributes can be added to a node using Xml_SetAttributes filter.

Selecting Elements Using XPath Query

To perform other operations on more complex XML trees filters from group Xml_SelectNodeValues can be used. Modification of the node's properties can be also done by using Xml_SetAttributes and Xml_SetNodeValues filters.

Selecting and altering filters is done using XPath criteria. XPath is a query language used to selecting sub-trees and attributes from an XML tree. All selecting and setting filters that are using XPath support the XPath 1.0 Specification.

The table below shows basic examples of XPath usage of filter Xml_SelectMultipleNodes:

Example	XPath	Description
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root**	Selecting all nodes of type 'root' that are at the first level of the tree.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root/result[2]**	Selecting the second child of 'root' node which is of type 'result'.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root/result/value**	Selecting all children of type 'value' of parent nodes '/root/result' of a specified XML node.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root/result[2]/value[2]**	Selecting specified second child of type 'root' and the second child of 'result'.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**//value**	Selecting all value nodes regardless of their position in the XML tree.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**//*[@status]**	Selecting all nodes which have 'status' attribute name.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**//*[@status='fail']**	Selecting all nodes which have 'status' of value 'fail'.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**//*[starts-with(name(),'res') and (@time > 200)]**	Selecting all nodes with names starting with 'res' and having attribute 'time' greater than '200'.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**//*[contains(text(),'10')]**	Selecting all nodes whose text (value) contains '10'.

Notice: All indexes are counted starting from [1].

The table below shows basic examples of XPath usage of Xml_SelectMultipleNodeValues_AsStrings filter:

Example	XPath	Description
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root[1]/result[2]/value[2]**	Selecting value from a specified node.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail"></value> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root[1]/result[1]/value**	Get all node values from a specific node. *Notice that empty XML nodes empty contains a default type value.*
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail"></value> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root[1]/result/value[@type='length']**	Get values of all nodes with attribute **length**.
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail"></value> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**/root[1]/result/value[@type='length']|/root[1]/result/value[@type='score']**	Get values of all nodes with attribute **length** and nodes with **score** attribute.

The table below shows basic examples of XPath usage of Xml_SelectMultipleAttributes_AsStrings filter:

Example	XPath	Description
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**//*/@type**	Selecting all attributes named 'type' .
<?xml version="1.0" ?> <root> <result time="200"> <value type="length">33.3</value> <value type="score">1.00</value> <value status="fail" /> </result> <result time="250"> <value type="length">10.6</value> <value type="score">0.30</value> <value status="pass" /> </result> </root>	**//*[./*[@status='fail']]/@time**	Selecting all attributes 'time' of nodes whose one of the children has status fail.

Common Practices (tips)

- Creating an XML tree by appending new nodes is not always necessary. If the tree structure is constant, the whole XML tree can be stored in a Global Parameter and when it is necessary new values of attributes or node text can be set using Xml_SetAttributes and Xml_SetNodeValues filters.
- XML files are very handy to store program settings.
- Most of Aurora Vision Studio objects can be serialized to text and stored in an XML file.
- XmlNodeToString filter can be used to send XML via TCP/IP or Serial Port.

8. Technical Issues

Table of content:

Using TCP/IP Communication

Introduction

Aurora Vision Studio has a set of filters for communication over the TCP/IP protocol. They are located in the Program I\O category of the Toolbox.

TCP/IP is actually a *protocol stack*, where TCP is on top of IP. These protocols are universally used from local to wide area networks and are fundamental to the communication over the Internet. They also constitute a basis for other protocols, like popular HTTP, FTP, as well as industrial standards Modbus TCP/IP or Modbus RTU/IP.

The TCP/IP protocol is a transport level communication protocol. It provides a bi-directional raw data stream, maintains a connection link and ensures data integrity (by keeping data in order and attempting to retransmit lost network packets). However raw TCP/IP protocol is usually not enough to implement safe and stable long term communication for a concrete situation and an additional communication protocol designed for a specific system must be formed on top of it. For example, when implementing a master-slave commands system with correct command execution check a command and acknowledge transaction must be used (first sending a command, than receiving with a timeout a command acknowledge where the timeout expiration informs that the command was not processed correctly). Single send operation is not able to detect whether data was properly received and processed by a receiver, because it is only putting data into a transmission buffer. Other situations may require different approach and usually required protocol will be imposed by a system on the other side of the connection. Because of the above basic TCP/IP and communication protocols knowledge is required to properly implement a system based on TCP/IP.

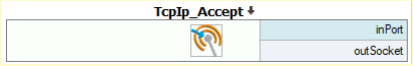
The communication is made possible with the use of *Sockets*. A network socket is an abstract notion of a bidirectional endpoint of a connection. A socket can be written to or read from. Data written to a socket on one end of the connection can be read from the opposite end. The mechanism of sockets is present in many programming languages, and is also the tool of choice for general network programming in Aurora Vision Studio.

Establishing a Connection

To communicate using the TCP/IP protocol stack, first a connection needs to be established. There are two ways of doing this: (1) starting a connection to a server and (2) accepting connections from clients. The sockets resulting from both of the methods are later indistinguishable - they behave the same: as a bidirectional communication utility.

A connection, once created, is accessed through its socket. The returned socket must be connected to all the following filters, which will use it for writing and reading data, and disconnecting.

Usually, a connection is created before the application enters its main loop and it remains valid through multiple iterations of the process. It becomes invalid when it is explicitly closed or when an I/O error occurs.

	Connects to a specific port on a remote host. The party to perform this operation is the client.
	Opens a local TCP port and waits for incoming connections. The party to perform this operation is the server.

Writing Data to Sockets

The filters for sending data take a SocketId value, which identifies an open connection and data, which is to be transferred to the other endpoint. The data can be of three different kinds: textual, binary or serialized objects.

The write operation is usually fast, but it can become problematic, when the other party is too slow in consuming the data, or if we attempt to write data faster than the available network throughput. The write operation is limited to putting data into a transmission buffer, without waiting for data delivery or receive confirmation. When data are being written faster than they can be delivered or processed by a receiver the amount of data held in a transmission buffer will start growing, causing significant delay in communication, and eventually the transmission buffer will overflow causing the write operation to rise and error.

	Writes plain text in UTF-8 encoding to a socket. An optional suffix* can be appended to the text.
	Writes arbitrary binary data, given as a ByteBuffer, to a socket.
	Writes a serialized object to a socket. The resulting data can only be read in another instance of Aurora Vision Studio/Executor with the TcpIp_ReadObject filter described below, and with using the same instantiation type.

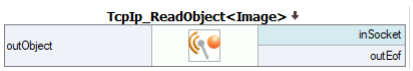
Reading Data from Sockets

Reading data requires an open connection and can return the received data in either of ways:

- as String, using UTF-8 encoding
- as ByteBuffer
- as a de-serialized object

The time necessary to receive data can be dependent on the network RTT (round-trip time), transfer bandwidth and amount of received data, but much more on the fact, whether the other side of the connection is sending the data at all. If there is no data to read, the filter has to wait for it to arrive. This is, where the use of the **inTimeout** parameter makes the most sense.

The filters for reading can optionally inform that the connection was closed on the other side and no more data can be retrieved. This mechanism can be used when connection closing is used to indicate the end of transmission or communication. The **outEof** output is used to indicate this condition. When the end of stream is indicated the socket is still valid and we should still close it on our side.

	Reads text in UTF-8 encoding until reaching a specified delimiter. The delimiter* can be either discarded, returned at the end of output, or left in the buffer to be processed by a subsequent read operation.
	Reads a fixed-length chunk of binary data.
	Reads a serialized object. The data can only come from another instance of Aurora Vision Studio/Executor executing the TcpIp_WriteObject filter described above, using the same type parameter.
	Reads all text, until EOF (until other side closes the connection).
	Reads all data, until EOF (until other side closes the connection).

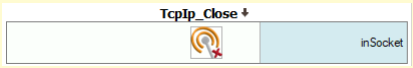
* - the delimiter and the suffix are passed as *escaped strings*, which are special because they allow for so called *escape sequences*. These are

combinations of characters, which have special meaning. The most important are "\n" (newline), "\r" (carriage return), and "\\" - a verbatim backslash. This allows for sending or receiving certain characters, which cannot be easily included in a String.

Closing Connections after Use

When a give socket is not needed anymore it should be passed to the socket closing filter, which will close the underlying connection and release the associated system resources. Every socket should be explicitly closed using the socket closing filter, even when the connection was closed on the other side or the connection was broken.

Typically, connections are being closed after the application exits its main loop macrofilter.

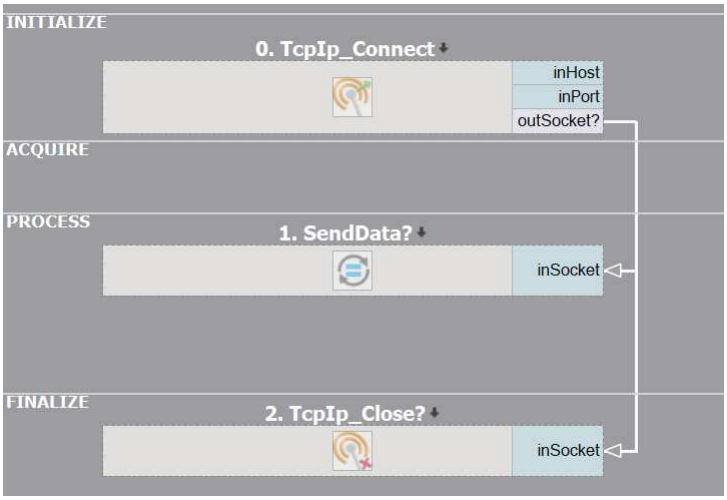
	Closes the connection gracefully and releases the socket.

Application Structure

The most typical application structure consist of three elements:

1. A filter creating a socket (Tcplp_Connect or Tcplp_Accept) executed.
2. A macrofilter realizing the main loop of the program (task).
3. A filter closing the socket (Tcplp_Close).

If the main loop task may exit due to an I/O error, as described in the next section, there should be also a fourth element, a Loop filter (possibly together with some Delay), assuring that the system will attempt reconnection and enters the main loop again.



For more information see the "IO Simple Tcplp Communication" official example.

Error Handling and Recovery

In systems where an error detection is critical it is usually required to implement communication correctness checks explicitly based on used communication protocol and capabilities of a communication peer. Such checks will usually consists of transmitting operation acknowledge messages and receiving expected transmissions with limited time. To implement this a timeout system of receiving filters can be used.

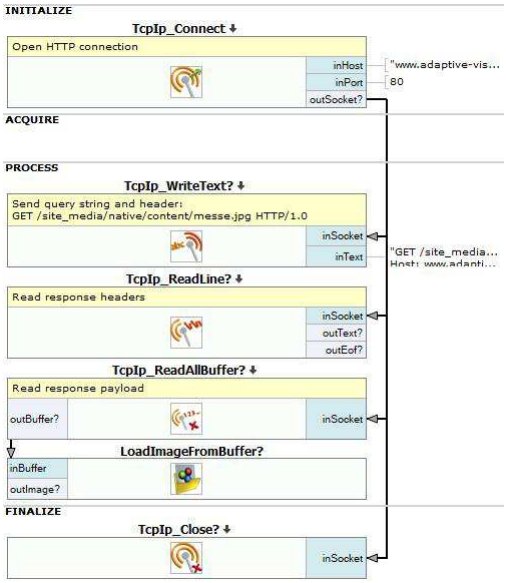
Additionally all TCP/IP filters can notify about unexpected situations or internal communication errors by rising an IoError. Such errors might result out of connection closing by a peer (when it is not expected in a given filter), system errors, network errors, or a broken connection state. The broken connection is usually the most common source of a communication error. Connection enters a broken state as a result of underlying system detecting that the other side of the communication is not confirming that it is receiving data (in some system defined time). After the system marks the connection as broken the next TCP/IP filter (that attempts to use this connection) will notify about that by rising an IoError. Relaying on detecting a broken connection is usually not the most reliable way of verifying valid connection state in error critical systems, as its detection may take a long time, detection time may vary on different systems and in some situations (like uni-directional transmission) it may not be detected at all. The TCP/IP Keep-Alive mechanism (available to activate in Tcplp_Connect and Tcplp_Accept) may help in increasing chances of detecting a broken connection.

IoError raised by filters can be handled using the macrofilter Error Handling mechanism (by putting TCP/IP filters into a separate Task). When a TCP/IP filter is terminated by an error its underlying connection is left in an undefined state, the connection should not be used anymore and the socket should be closed with a Tcplp_Close filter. The application can then attempt to recover from TCP/IP errors by initializing the connection from scratch. Specific possibilities and requirements of recovering from communication errors depends on a used protocol and a communication peer capabilities.

Using Tcp/Ip Communication with XML Files

Very often Tcp/Ip communication is used to exchange information structured as XML documents. For this type of communication, use filters for reading or writing text together with the filters from the System :: XML category for parsing or creating XML trees.

Example: Reading Data from HTTP



A sequence of filters to download an image through HTTP

This example consists of the following steps:

1. A connection is made, to port 80 (the HTTP standard port) of *www.adaptive-vision.com* host.
2. A HTTP query string is sent, to request an image from a specific URL.
3. The headers, which end with "\r\n\r\n" (double newline) are read (and not processed).
4. All the remaining data - the HTTP response body - is read and returned as binary data.
5. The bytes are converted to an image, using a utility function.

General Image Acquisition

Camera Acquisition Thread

In Aurora Vision image acquisition is done in the background. Due to that, regardless of what the vision program is doing a new image can be received from the camera as soon as possible.

When the communication with the camera is started (for example by using **GigEVision_StartAcquisition**, but this applies to different vendors as well) Aurora Vision creates a special background thread. This thread handles all communications with that particular camera, and in case of received images stores them in a special queue. While not exactly the same, in principle that queue is similar queues available to user in Aurora Vision. The size of the queue is determined by the filter (some camera interfaces do not support sizes larger than 1).

When the program executes a grabbing filter (e.g. **GigEVision_GrabImage:Synchronous**) that filter in fact grabs an image from the queue made by the background acquisition thread, not directly from the camera.

If there are no images in the queue, the grabbing filter waits until a new image arrives either for an indefinite amount of time (**synchronous variant**) or for a specified amount of time (**asynchronous variant**). This behavior is analogous to how the filters **Queue_Pop** and **Queue_Pop_Timeout** behave, respectively. After getting an image, it is removed from the queue. If the queue is full and the camera sends a new image, then the oldest image in the queue is replaced.

Images will remain in the queue until they are grabbed, are replaced by a newer image, or their camera thread was closed.

What is important to remember is that if the queue size is larger than 1, the grabbing filter will return the oldest image available. Depending on the application structure this may or may not be the desired behavior.

- If the application has to analyze every image, but the iteration time may be longer than the period between images the queue should be large enough to store all images until the application can catch up. For example, a camera is triggered multiple time in the burst, followed by a period of inactivity, the queue size should be equal to the number of triggers in one burst.
- Conversely, if the application does not need to inspect every image (common in application with a free-running camera) the queue size can usually be limited to one. This ensures the lowest possible lag between image acquisition and results.

As mentioned before, the background camera thread handles all communication with its assigned camera. If no thread for the specified camera exists, it will be created as soon as any camera filter is used. Camera can only have one thread assigned to it, so filters will always use the existing thread if one is available.

Camera threads also handle setting and reading parameters. Depending on the camera interface there may be optimizations, such as writing new parameter values only if the value has changed.

It is important to remember that the threads will be closed if the task in which they were created ends. When the thread is closed all data in it (including images in the queue) is removed.

Working with Gen TL devices

Introduction

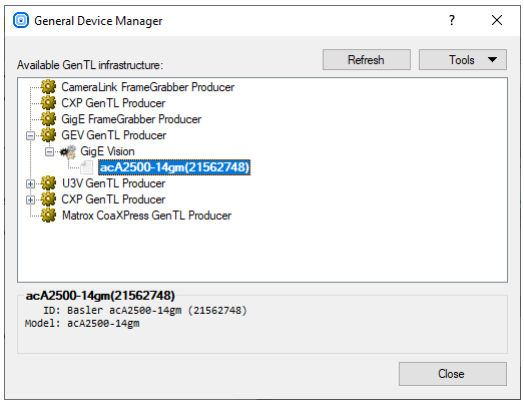
Aurora Vision Studio allows to communicate with devices which use Gen TL. For this purpose, you need to install the software provided by device vendor in your operating system (this step is highly dependant on device vendor so please refer to the device documentation, there is no general recipe for it). You can check if the installed software has added Gen TL provider to your system in the following way:

1. There is a definition of *GENICAM_GENTL32_PATH* and/or *GENICAM_GENTL64_PATH* in the environment variables section.
2. Catalogs created by the installed device vendor software should contain some .cti (Common Transport Interface) files.

Please make sure that the platform (32/64-bit) which the vendor software is intended for is the same as the version (32/64-bit) of Aurora Vision Studio you use.

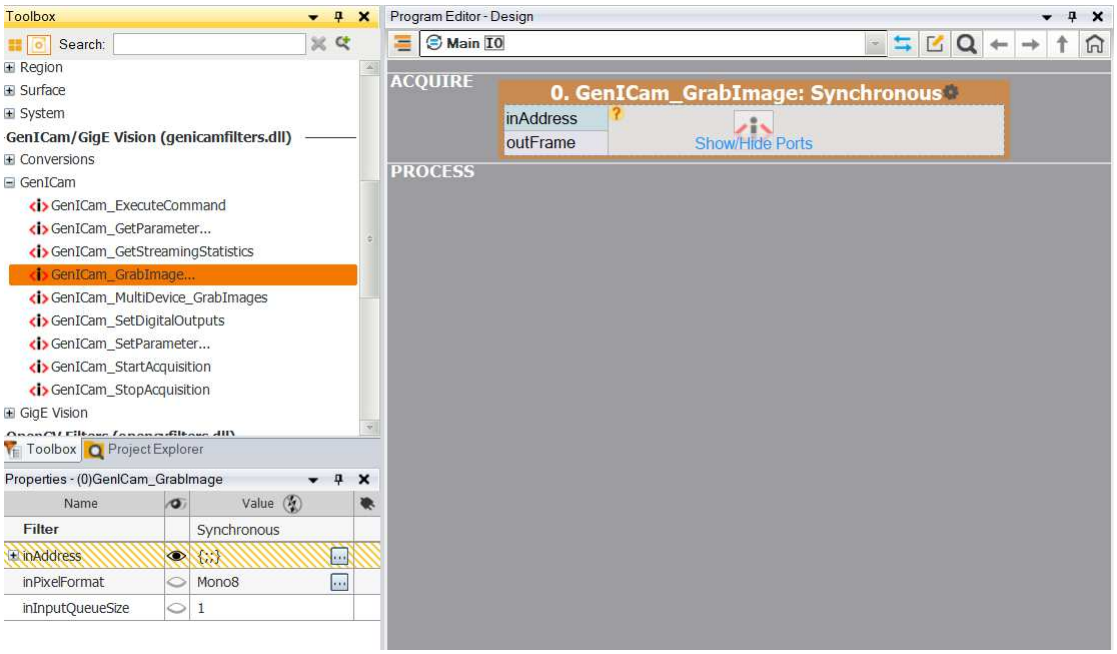
Using GenICam Device Manager

Once you have completed the steps above, you should be able to see your GenICam devices connected to the computer in the GenICam Device Manager (if they are not visible there, the cause is usually that device vendor's .cti files or proper entries in environment variables section are missing), like in the image below:

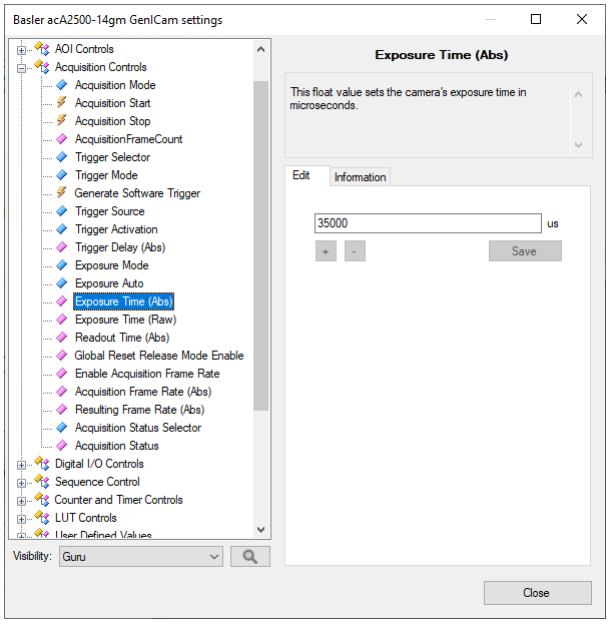


You can access the GenICam Device Manager in one of the two ways:

- Choose *Tools » Manage GenICam Devices...* from the Main Menu.
- Add one of GenICam filters to program and open the GenICam Device Manager directly from filter properties.

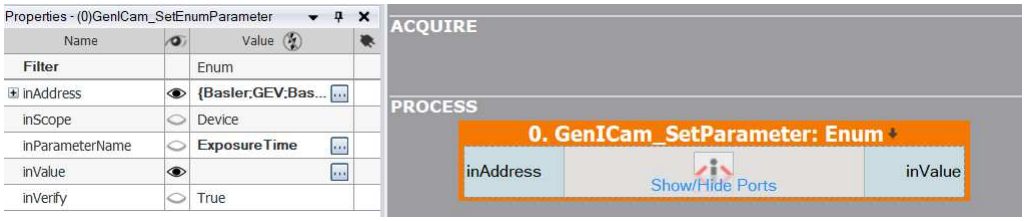


After choosing a device you can go to its settings tree by clicking *Tools » Device Settings* in the GenICam Device Manager. You should see device parameters grouped by category there. Some parameters are read-only, other are both readable and writable. You can set the writable parameters directly in the GenICam Settings Tree.



Another way to read or write parameters is by using the filters from the GenICam category. In order to do this you should:

1. Check of which type is the parameter you'd like to read/write (you can check it in the GenICam Settings Tree).
2. Add a proper filter from the GenICam category to the program.
3. Choose a device, define parameter name and new value (when you want to set the parameter's value) in the filter properties.
4. Execute the filter.



Known Issues Regarding Specific Camera Models

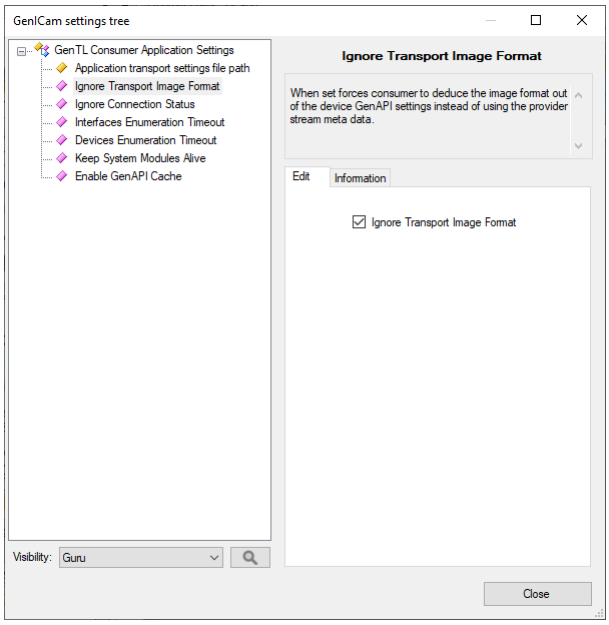
In this section you will find solutions to known issues that we have came across while testing communication between Aurora Vision products and different camera models through GenICam.

Manta G032C

We have encountered a problem with image acquisition (empty preview window with no image from camera) while testing this camera with default parameters. The reason for this is that the packet size in this camera is set by default to 8,8 kB. You need a network card supporting Jumbo Packets to work with packets of such size (the Jumbo Packets option has to be turned on in such case). If you don't have such network card, you need to change the packet size to 1.5 kB (maximal UDP packet size) in the GenICam Settings Tree for the Manta camera. In order to do this, please open the GenICam Device Manager, choose your camera and then click *Tools » Device Settings* (go to the GenICam Settings Tree). The parameter which needs to be changed is *GevSCPSPPacketSize* in the GigE category, please set its value to 1500 and save it. After doing this, you should be able to acquire images from your Manta G032C camera through GenICam.

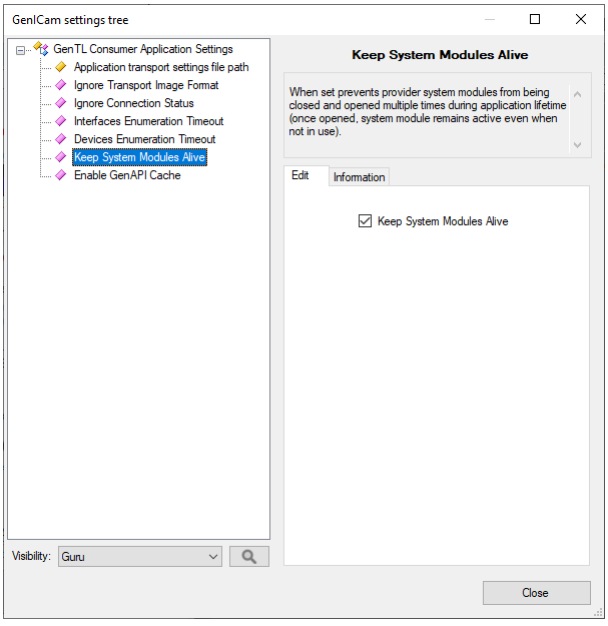
Adlink NEON

On devices with early version of the firmware, there are problems with starting image acquisition. The workaround is simple: just set the IgnoreTransportImageFormat setting to "yes". This option is available in the GenICam Device Manager (Tools → Application Settings...):



Daheng cameras

1. Open GentTL settings: *Tools » Manage GenICam Devices... » Tools » Application Settings*
2. Turn on "Keep System Modules Alive" option



3. Close the window and restart Aurora Vision Studio

It is possible to get IO Error: "Unable to open GenTL Provider System Module" after stopping and starting the application again. To solve that problem you should: Note, that it is a local setting, thus it needs to be repeated again on every PC on which the application would be run.

Working with National Instruments devices

Introduction

Aurora Vision Studio allows for communication with i/o devices from National Instruments. This is done by using filters from the Hardware Support :: National Instruments category.

Working with Tasks

The entire support for National Instruments devices is based on *tasks* (do not confuse with Task macrofilters). A task in Aurora Vision Studio is a single channel. Every task is associated with an ID, which is necessary for configuring tasks, reading states from inputs or writing values to device outputs.

Creating Tasks

To start working with National Instruments' devices, a task that will perform a specific operation must be created.
For this purpose one of the filters from list shown below should be selected.

- DAQmx_CreateDigitalPort
- DAQmx_CreateAnalogChannel
- DAQmx_CreatePulseChannelFreq
- DAQmx_CreateCountEdgesChannel

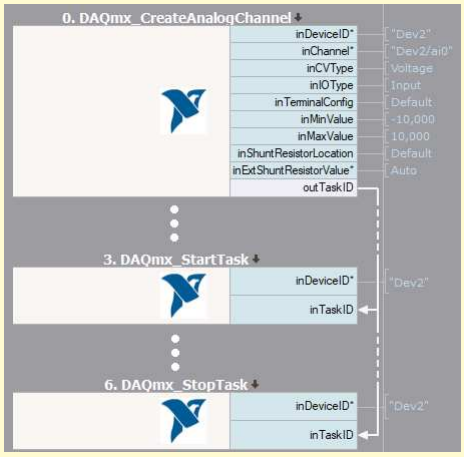
These filters return *outTaskID*, which is the identifier of a created task.

Starting Tasks

To start a task, filter DAQmx_StartTask should be used.

Finishing Tasks

There are two ways of finishing tasks: by using the DAQmx_StopTask filter or by waiting for the entire program to finish. It should be noted that you cannot start two tasks using the same port or channel in a device.



Basic scheme of program

Reading and Writing Values

Aurora Vision Studio provides two kinds of filters for reading and writing values. DAQmx_WriteAnalogArray, DAQmx_WriteDigitalArray and DAQmx_ReadAnalogArray, DAQmx_ReadDigitalArray, DAQmx_ReadCounterArray filters allow to work with multiple values. For reading or writing a single value, DAQmx_WriteAnalogScalar, DAQmx_WriteDigitalScalar and DAQmx_ReadAnalogScalar, DAQmx_ReadDigitalScalar, DAQmx_ReadCounterScalar filters should be used.

Configuring Tasks

After creating a task, it can be configured using the filters DAQmx_ConfigureTiming, DAQmx_ConfigAnalogEdgeTrigger or DAQmx_ConfigDigitEdgeTrigger. Note that some configuration filters should be used before the task is started.

Sample Application

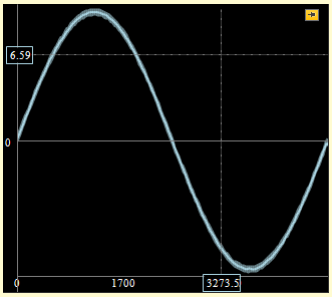
The following example illustrates how to use DAQmx filters. The program shown below reads 5000 samples of voltage from analog input. However, data is acquired on rising edge of a digital signal.

At first, analog channel is created. In this sample, input values will be measured, so **inCVType** input must be set to *Voltage* and **inIOType** should be set to *Input*. The rest of inputs depends on used device.

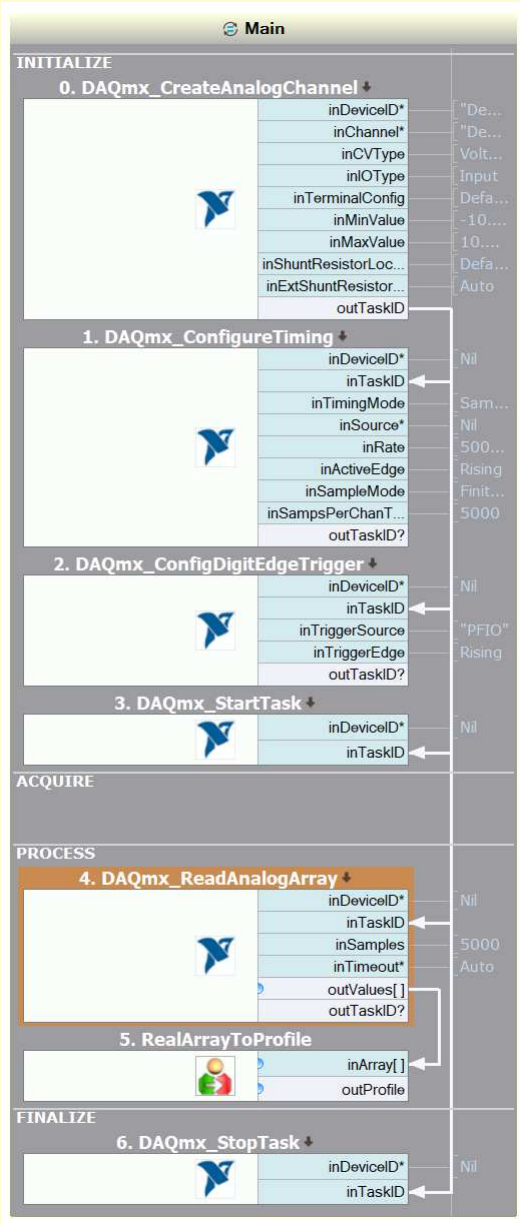
Afterwards, sample clock to task is assigned. **inDeviceID** input might be set to *Auto*, because it is the only one device used in this program. Other parameters could be set according to user's camera.

Sample application should start acquiring data after a digital edge occurs. Because an active edge of digital signal should be rising, **inTriggerEdge** input must be set to *Rising* (for falling edge, value *Falling* must be chosen). Task is ready to start.

Last step in presented program is to acquire multiple data from selected device. Filter DAQmx_ReadAnalogArray could be used for this. DAQmx_StopTask is not required, because this program doesn't use one channel in two different tasks. Acquired array of values might be represented e.g. as a profile (shown below).



Acquired data from DAQ device



Sample application for acquiring data

Working with Modbus TCP Devices

Introduction

The filters for Modbus over TCP communication are located in the System :: Modbus TCP

Establishing Connection

To communicate with a ModbusTCP device, first a connection needs to be established. The ModbusTCP_Connect filter should be used for establishing the connection. The default port for ModbusTCP protocol is 502. Usually, a connection is created before the application enters its main loop and it remains valid through multiple iterations of the process. It becomes invalid when it is explicitly closed using ModbusTCP_Close filter or when an I/O error occurs. The Modbus filters work on the same socket as TCP IP filters. If Modbus device supports Keep-Alive mechanism it is recommended to enable it, to increase chance of detecting a broken connection. For more details please read Using TCP/IP Communication.

Reading and Writing Values

Aurora Vision Studio provides set of filters to manage Modbus objects. The following is table of object types provided by a modbus:

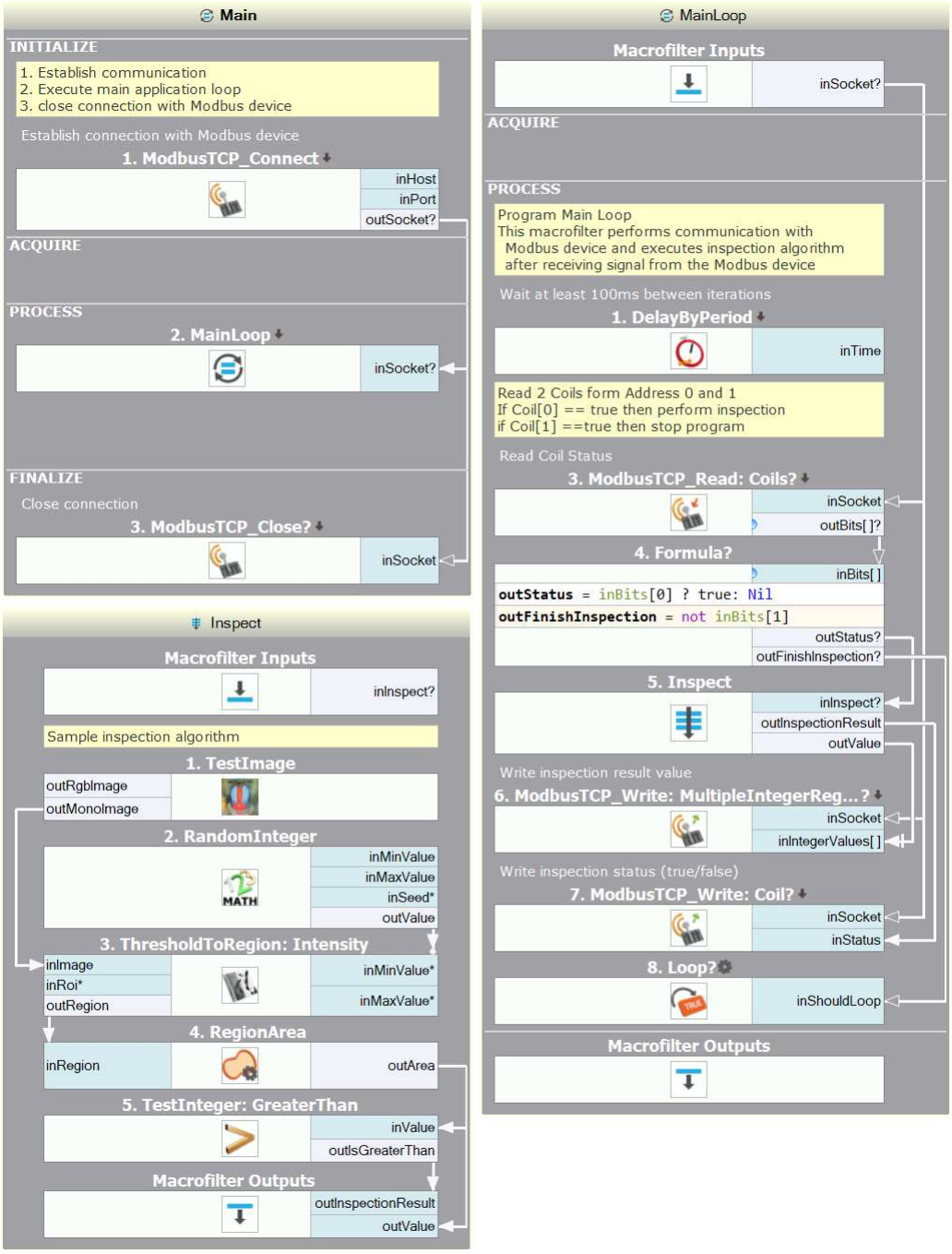
Object Type	Access	Size
Coil	Read-Write	1-bit
Discrete Input	Read-Only	1-bit
Input Register	Read-Only	16-bits
Holding Register	Read-Write	16-bits

The Coils, Inputs and Registers in ModbusTCP protocol are addressed starting at zero. The table below lists filters to control object types mentioned above.

Object Type	Read	Write
Coil	ModbusTCP_ReadCoils	ModbusTCP_WriteCoil ModbusTCP_ForceMultipleCoils
Discrete Input	ModbusTCP_ReadDiscreteInputs	Read-Only
Input Register	ModbusTCP_ReadInputRegisters_AsByteBuffer ModbusTCP_ReadInputIntegerRegisters ModbusTCP_ReadInputRealRegisters	Read-Only
Holding Register	ModbusTCP_ReadMultipleRegisters_AsByteBuffer ModbusTCP_ReadMultipleIntegerRegisters ModbusTCP_ReadMultipleRealRegisters	ModbusTCP_WriteSingleRegister ModbusTCP_WriteMultipleRegisters_AsByteBuffer ModbusTCP_WriteMultipleIntegerRegisters ModbusTCP_WriteMultipleRealRegisters

Sample Application

The following example illustrates how to use ModbusTCP filters.



Main macrofilter - establish and close connection

Main Loop - communication with Modbus device

Inspection - sample inspection algorithm

Custom Functions

To support device specific function Aurora Vision Studio provides ModbusTCP_SendBuffer filter. This filter allows to create a custom Modbus frame.

Project Files

When you save a project, several files are created. Most of them have textual or xml formats, so they can be edited also in a notepad and can be easily managed with version control systems.

A project may consist of the following file types:

- Single ***.avproj** file (XML) – this is the main file of the project; plays a role of the table of content.
- Single ***.avcode** file (textual) – this is the main source code file.
- Zero or more ***.avlib** files (textual) – these are additional source code modules.
- Zero or more ***.avdata** files (binary) – these files contain binary data edited with graphical tools, such as Regions or Paths.
- Single ***.avview** file (XML) – this file stores information about the layout of Data Previews panels.
- Zero or one ***.avhmi** file (XML) – this file stores information about the HMI Design.

C++ Code Generator

- Introduction
- User interface
- Generated Code Organization

Introduction

A program accepted by Aurora Vision Studio and Aurora Vision Executor is executed in a virtual machine environment. Such a machine consecutively executes (with preservation of proper rules) filters from the source program. The C++ Code Generator allows to automatically create a C++ program, which is a logical equivalent of the job performed by the virtual machine of Aurora Vision Studio executor. As part of such programs, consecutive filter calls are changed to calls of functions from Aurora Vision Library.

To generate, compile and run programs in this way it is necessary to own also a Aurora Vision Library license.

User interface

The C++ Code Generator functionality is available in the Main Menu in *File » Generate C++ Code...*. After choosing this command, a window with additional generator options and parameters (divided into tabs) will be opened.

In the *Output* tab basic generator parameters can be set:

- **Program name** - it determines the name of the generated C++ program (it doesn't have to be related to Aurora Vision Studio project name). It is a base name for creating names of source files and (after choosing a proper option) it is also the name of the newly created project.
- **Output directory** - it determines the folder in which a generated program and project files will be saved. Such folder has to exist before generation is started. The path can be either absolute or relative (when an Aurora Vision Studio project is already saved) starting from the project directory. All program and project files will be saved in this folder with no additional subfolders. Before overwriting any files in the output directory, a warning message with a list of conflicted files will be displayed.
- **Code namespace** - an optional namespace, in which the whole generated C++ code will be contained. A namespace can be nested many times using the "::" symbol as a separator (e.g. "MyNamespace" or "MyProject::MyLibrary::MyNamespace"). Leaving the code namespace field empty will result in generating code to the global namespace.
- **Create sample Microsoft Visual Studio solution** - enabling this option will result in generating a new sample tentatively configured solution for compiling generated code in the Microsoft Visual Studio environment. Each time when this option is enabled and code is generated (e.g. when generating code again after making some changes in an Aurora Vision project), a new solution will be created and any potential changes made in an already existing solution with the same path will be overwritten.
  This option is enabled by default when generating code for the first time for the selected folder. It is disabled by default when code is generated again (updated) in the selected output folder.
  Details regarding Microsoft Visual Studio solution configuration required by generated code are described in this document below.

In the *Modules* tab there is a list of project modules which can participate in code generation. It is possible to choose any subset of modules existing in an Aurora Vision Studio project, as long as this doesn't cause breaking the dependencies existing in them. Selecting a subset of modules will cause that only macrofilters and global parameters present in the selected modules will be generated to C++ code. If disabling a module causes breaking dependencies (e.g. a macrofilter being generated refers to an other macrofilter which exists in a module which is disabled in code generation), it will be reported with an error during code generation.

In the *Options* tab there are additional and advanced options modifying the behavior of the C++ Code Generator:

- **Include instances in diagnostic mode** - diagnostic instances (instances processing data from diagnostic outputs of preceding filter instances) are meant to be used to analyze program execution during program development, that's why they aren't included by default in generated programs. In order to include such instances in a generated program this option has to be enabled (**Note: in order to make diagnostic outputs of functions work correctly, enabling diagnostic mode in Aurora Vision Library has to be taken into account in a C++ program**).
- **Generate macrofilter inputs range checks** - the virtual machine controls data on macrofilter inputs in terms of assigned to them allowed ranges. Generated code reproduces this behavior by default. If such control is not necessary in a final product, it is possible to uncheck this option to remove it from a C++ program.
- **Enable function block merging** - language constructions of conditional blocks and loops, which map virtual machine functions in terms of conditional and array filter execution, are placed in generated code. The C++ Code Generator uses optimization which places, where possible, a couple of filter calls in common blocks (merging their function blocks). Unchecking this option will disable such optimization. This functionality is meant to help solve problems and usually should be kept enabled.

The *Generate* button at the bottom of the window starts code generation according to the chosen configuration (in case of a need to overwrite existing files an additional window to confirm such action will be displayed) and when the generation operation is completed successfully it closes the window and saves the parameters. The *Close* button closes the window and saves the chosen configuration. The *Cancel* button closes the window and ignores the changes made to the configuration.

All parameters and generation options are saved together with an Aurora Vision Studio project. At the next code generation (when the configuration window is opened again), the parameters chosen previously for the project will be restored.

In order to keep names between Aurora Vision Studio project files and C++ program files synchronized, it is advised to save a project each time before generating C++ code.

Generated Code Organization

A program is generated to C++ code preserving program's split into modules. For each program module, chosen for generation, there are two files (.cpp and .h) created. For the main module (or when there is only one module) these files have names equal to the program name (with appropriate extensions) which was chosen in generation options. For other modules file names are created according to the template `program-name.module-name.cpp/h`.

Macrofilters contained in modules are generated in form of C++ functions. Each public macrofilter, or a private macrofilter used by an other macrofilter contained in generated code, is included in a .cpp file as a function definition. For each public macrofilter its declaration is included also in an .h file. Analogical rules are applied to global parameters. Macrofilter visibility level (public/private) can be set in its properties.

Other project elements, e.g. an HMI structure, don't have their correspondents in generated code.

Generated function interface corresponds to a set of macrofilter inputs and outputs. In the first place inputs are consecutively mapped to function arguments (sometimes using constant reference type), next arguments of reference types from outputs (function output arguments) are also mapped, through them a function will assign results to objects passed to a function from the outside. E.g. for a macrofilter containing two inputs (inValue1, inValue2) and one output (outResult) a function interface may look like this:

```
void MyMacro( int inValue1, int inValue2, int& outResult )
```

When a macrofilter contains facilities requiring preserving their states in consecutive macrofilter iterations (e.g. a Step macrofilter containing registers, loop generators or accumulators), a state argument will be added to the function interface on the first position:

```
bool MyMacro( MyMacroState& state, int inValue1, int inValue2, int& outResult )
```

Data types of inputs, outputs and connections will be mapped in code to types, structures and constructions based on the templates (e.g. `atl::Array<>` or `atl::Conditional<>`) coming from Aurora Vision Library. Filter calls will be mapped to function calls coming also from Aurora Vision Library. In order to get an access to proper implementations and declarations, there are Aurora Vision Library headers included in generated code. Such includes can appear as well in .cpp files, as in .h files (because references to Aurora Vision Library types appear also in function signatures generated from macrofilters).

In generated program there are also static constants, including constant compound objects, which require initial initialization or loading from a file (e.g. if in the IDE there is a hand-edited region on a filter input, then such region will be saved in an .avdata file together with generated code). A program requires preparing constants before using any of its elements. In order to do this, in the main module in generated code an additional function `Init` with no arguments is added, as part of this function compound constants are prepared. This function has to be called before using any element from generated code (also before constructing a state object).

Generated Code Usage

The basis of generated code are file pairs emerging from modules with the structure described above. These modules can be used as part of a vision program, e.g. as a set of functions implementing the algorithms designed in the Aurora Vision Studio environment.

Before calling any generated function, constructing a function state object or accessing a global parameter, it is necessary to call the `Init()` function added to the main module code. The Init function must be called once at the beginning of a program. This function is added to generated code even when a program does not contain any compound constants which require initialization, in order not to have to modify the schema of generated code after modifying and updating the program.

Stateful functions

As described above, sometimes a function may have an additional first argument named *state*, passed by reference. Such object preserves a function state between consecutive iterations (each call of such function is considered as an iteration). In such objects there are, among others, kept generator positions (e.g. the current position of filters of *Enumerate** type), register states of Step macrofilters or internal filter data (e.g. a state of a connection with a device in filters which acquire images from cameras).

A state object type is a simple class with a parameterless constructor (which initializes the state for the first iteration) and with a destructor which frees state resources. It is the task of applications using functions with a state to prepare a state object and to pass it through a parameter to a function. **An application may construct a state object only after calling the `Init()` function.** An application should not modify such object by itself. One instance of a state object is intended for a function call as part of one and the same task (it is an equivalent of a single macrofilter instance in an Aurora Vision Studio program). Lifetime of a single object should be sustained for the time when such task is performed (e.g. you should not construct a new state object to perform the same operation on consecutive video frames). A single state object cannot be shared among different tasks (e.g. if as part of one iteration the same function is called twice, then both calls should use different instances of a state object).

State object type name is generated dynamically and it can be found in the declaration of a generated function. The C++ Code Generator, if possible, creates names of state types according to the template *Macrofilter-name*State.

Freeing state resources is performed in a state class destructor. If a function established connections with external devices, then a state object destructor is used also to close them. The recommended way of state handling is using a local variable on the stack, in such way that a destructor is called automatically after stepping out of a program block.

Error handling

The functions of Aurora Vision Library and generated code report the same errors which can be observed in the Aurora Vision Studio environment. On the C++ code level error reporting is performed by throwing an exception. To throw an exception, an object of type derived from the `atl::Error` class is used. Methods of this class can be used to get the description of reported problems.

Note: the `Init()` function can also throw an exception (e.g. when an external .avdata file containing a static constant could not be loaded).

Global parameters

Simple global parameters (only read by the vision application) are generated as global variables in the C++ program (with the same name as the global parameter). It is possible to modify those variables from the user code in order to change the application configuration, however such modification is **not thread safe**.

When thread safe modification of the global parameters is needed, global parameters should only be accessed with WriteParameter and ReadParameter filter blocks in the vision application. In order to allow access from user code to such operations, appropriate read/write should be encapsulated in a public macrofilter participating in the code generation. This will give access to the parameters from the user code in form of a function call.

Compiling a program with generated code

Generated C++ code is essentially a project basing on Aurora Vision Library and has to follow its guidelines. Compilation requires this product to be installed in order to get the access to proper headers and .lib files. Aurora Vision Library requires that a project is compiled in the Microsoft Visual Studio environment.

A program being compiled requires the following settings:

- The header file search path is set to the *include* folder of the Aurora Vision Library package.
- The search paths of .lib files for respective platforms are set to the corresponding subfolders in the *lib* directory of the Aurora Vision Library package (e.g. the Release|Win32 configuration compiled in the Microsoft Visual Studio package has to use the libraries from the *lib\Win32* subfolder).
- The AVL.lib library is linked to a project (in order to link a dynamic library - AVL.dll).

Running a compiled program

A program created basing on generated code and the Aurora Vision Library product requires the following components to run:

- All .avdata files generated with C++ code (if any were generated at all), located in the current application folder during the `Init()` function call.
- If in the Aurora Vision Studio project (from which code is generated) there are any filters reading external data (e.g LoadImage) or enclosures of data files to filter inputs, then all such files have to be available during program execution. The files will be searched according to the defined path, in case of a relative path the search will begin starting from the current application folder.
- A redistributable package of Visual C++ (from the version of Microsoft Visual Studio used to compile the program and Aurora Vision Library) installed in the destination system.
- The AVL.dll file ready to work with the used platform. This module can be found in a subfolder of the *bin* directory of the Aurora Vision Library package (analogically to the use of the search paths of .lib files).
- If the generated code uses third party packages (e.g. cameras, I/O cards, serial ports), then additional .dll files (intermediating in the access to the drivers of such devices) are required. Such files are identified with names ending with the "_Kit" suffix, they can be found in the same directory as the used AVL.dll file (also in this case it is necessary that this file is compatible with given platform). In case when a required file is missing, the program being executed will report an error pointing the name of the missing module in a message. Such libraries are loaded dynamically during program execution. In such case, for correct execution it is also necessary to install proper redistributable packages or SDKs for the used devices.
- A valid license for Aurora Vision Library activated in the destination system.

Generated sample Microsoft Visual Studio solution

After choosing a proper option in the *Output* tab, the C++ Code Generator, apart from modules' code, will also create a sample tentatively configured Microsoft Visual Studio project together with sample user code using generated code. The configuration of such project follows the compilation requirements described above, including attaching headers and .lib files from the Aurora Vision Library package (it uses the AVL_PATHXX environment path created by the installer of Aurora Vision Library). This means that to compile a sample project, a properly installed package of Aurora Vision Library is required.

To make the generated project's structure clear and distinguish its elements, there are two filters (solution directories) created in such project:

- **Generated Code** - here are located files deriving from generation of project modules. Such elements are not supposed to be further manually modified by the user, because in case of project update and code re-generation such modifications will be overwritten.
- **User Code** - here are located files from the sample user application. It's assumed that such code should be created by the user and implement a final application which uses generated code.

Project configuration covers:

- Including paths to the headers of the Aurora Vision Library package.
- Linking with the AVL.lib library.
- Copying the AVL.dll file to the destination folder.

The configuration does not cover providing proper _Kit files (if they are required).

As sample user code there is one simple "main.cpp" file created, it performs the following actions:

- Calling the Init function.
- Activating the diagnostic mode of Aurora Vision Library (if in project options usage of diagnostic instances is enabled for code generation).

- Calling the Main macrofilter's function (if it took part in code generation during creation of the main.cpp file).
- Catching all exceptions thrown by the program and Aurora Vision Library and reporting error messages on the standard output before closing the program.

A project and sample user code are generated once. It means that neither project configuration nor user code which calls generated code will be updated during project update (code re-generation). They can be only completely overwritten in case of choosing again the option to create a sample Microsoft Visual Studio solution. A project and sample code are created basing on names and options from the code generation configuration in Aurora Vision Studio.

In case of developing an application using generated code, the duty to adjust a project to changes in generated code is assigned to the user. To such tasks belong among others:

- Adding to the project and removing from it files with code of generated modules and adjusting `#include` directives in case of modifying modules in an Aurora Vision Studio project.
- Adjusting the code which calls generated code in case of modifications of interfaces of public macrofilters.
- Providing proper _Kit files in the scope of the resulting program.

Please note that projects generated from Aurora Vision Studio up to version 3.2 and thus prepared for Aurora Vision Library up to version 3.2 required linking with additional library named "AvCodeGenRuntime.lib". This library is not required anymore and can be removed from project configuration when updating to newer version of Aurora Vision Library.

.NET Macrofilter Interface Generator

Introduction

Generation of macrofilter interfaces allows to design complex applications without loosing comfort of visual programming which comes from the environment of Aurora Vision Studio.

The most common reasons people choose .NET Macrofilter Interfaces are:

- Creating applications with very complex or highly interactive HMI.
- Creating applications that connect to external databases or devices which can be accessed more easily with .NET libraries.
- Creating flexible systems which can be modified without recompiling the source code of an application.

Macrofilters can be treated as mini-programs which can be run independently of each other. Aurora Vision Studio enables to use those macrofilters in such programming languages as C# or C++/CLI and execute them as regular class methods. Because those methods are just interfaces to the macrofilters, there is no need to re-generate the assembly after each change made to the AVCODE (the graphical program).

Requirements

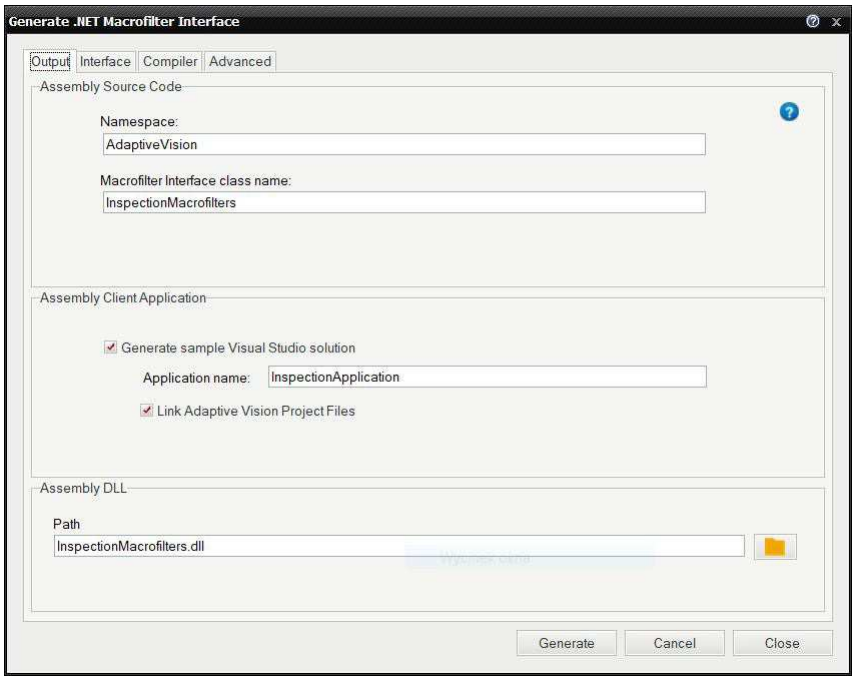
In order to build and use a .NET Macrofilter Interface assembly, the following applications must be present in the user's machine:

Building	Running
Aurora Vision Professional 5.3	**Aurora Vision Professional 5.3** or **Aurora Vision Runtime 5.3**
Microsoft Visual Studio 2015 (or greater)	**Microsoft Visual C++ Redistributable Package** of the same bitness (32/64) as the generated assembly and the same version as Microsoft Visual Studio used to build the assembly.

.NET Macrofilter Interface Assembly Generator

Macrofilters may be interfaced for use in such managed languages as C# or Visual Basic. Such interface is generated into a Dynamic Link Library (dll), which can be then referenced in a Visual C# project. To generate a dll library for the current Aurora Vision Studio project, fill in all necessary options in the .NET Macrofilter Interface Generator form that can be opened from *File » Generate .NET Macrofilter Interface...*

Output page



Namespace
Defines the name of the main library class container.

Macrofilter Interface Class Name
Defines the name of the class, where all macrofilters will be available as methods (with the same signatures as macrofilters).

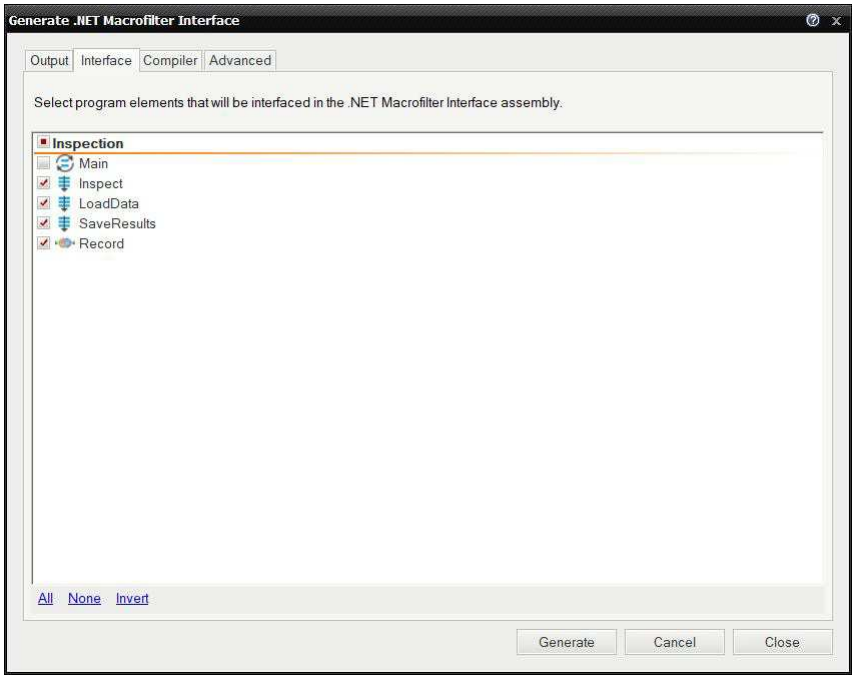
Generate sample Microsoft Visual Studio solution
Generates empty Microsoft Visual Studio C# WinForms project that uses to-be created Macrofilter .NET Interface assembly.

Link Aurora Vision Project Files
Includes current Aurora Vision project files to the C# project as links. This way Aurora Vision project files (*.avproj, *.avcode, *.avlib) are guaranteed to be easily accessible from the application output directory, e.g. with following line of code (assuming the project is Inspection.avproj):

```
macrofilters = InspectionMacrofilters.Create(@"auroravision\Inspection.avproj");
```

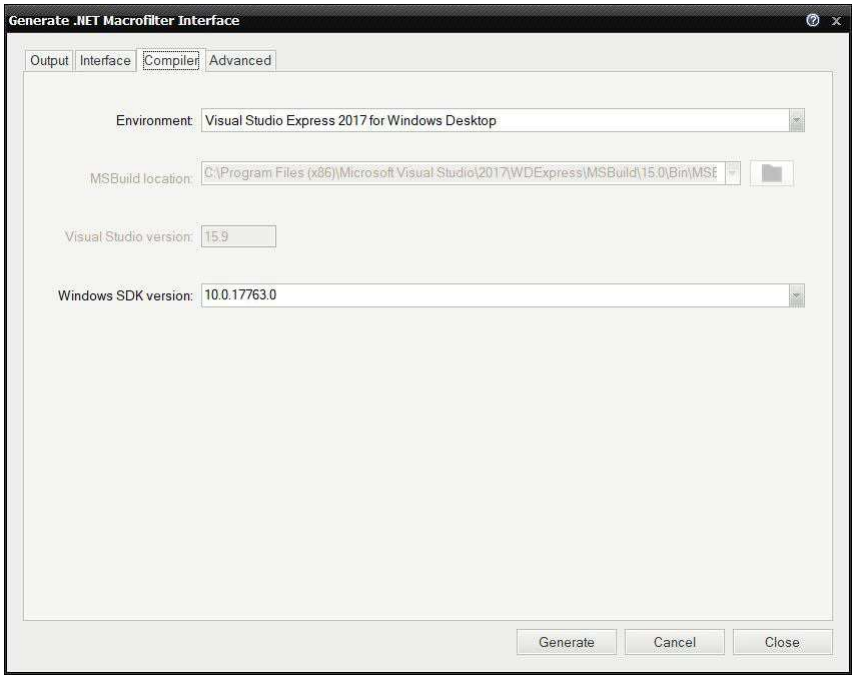
Path to the generated dll library
Location of the to-be generated assembly.

Interface page



Check box list
List of all macrofilters and User Types defined in the current Aurora Vision project that a .NET Interface may be generated for.

Compiler page



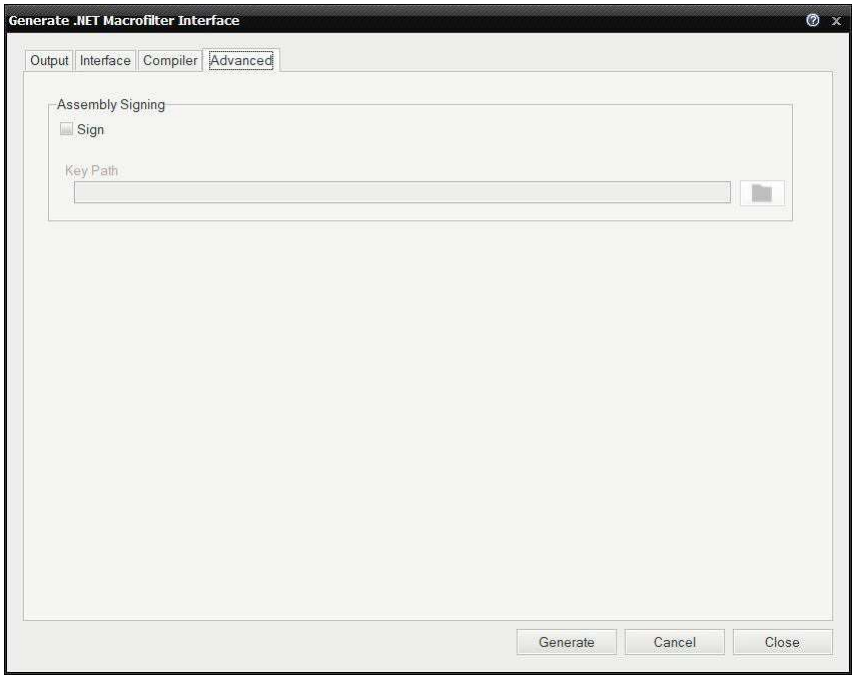
Environment
Selection of which Microsoft Visual Studio build tools should be used to build an assembly. The drop down list is populated with all detected compatible tools in the system, including Microsoft Visual Studio and Microsoft Visual Studio Build Tools environments. If none of detected are suitable, custom environment may be used. Then, MSBuild.exe location and target Microsoft Visual Studio version properties need to be defined manually.

MSBuild location
Shows the path to the actual MSBuild.exe according to the selected environment. Editable, when Custom environment is selected.

Microsoft Visual Studio version
Defines the generated sample C# project format (sln and csproj files).

Windows SDK version
Allows choosing the appropriate SDK version when generating Macrofilter .Net Interface. The list contains all detected SDK versions in the system.

Advanced page



Assembly Signing
Enables signing the generated Macrofilter .NET Assembly with given private key and makes it a Strong-Named assembly. Keys may be generated e.g. in Strong Name Tool or in Microsoft Visual Studio IDE (C# project properties page).

.NET Macrofilter Interface Assembly Usage

Initialization

Once a generated library is referenced in a Visual C# project, macrofilters chosen in the .NET Macrofilter Interface Assembly Generator form are available as instance methods of the class defined in the Macrofilter Interface Class Name text box (`MacrofilterInterfaceClass`), in the Output page. However, in order to successfully execute these methods, a few Aurora Vision libraries must be initialized, i.e. Executor.dll and available Filter Libraries. It is done in a static method named `MacrofilterInterfaceClass.Create`, so no additional operations are necessary. To achieve the best performance, such initialization should be performed only once in application lifetime, and only one instance of the `MacrofilterInterfaceClass` class should exist in an application.

`MacrofilterInterfaceClass.Create` static method receives a path to either *.avcode, *.avproj or *.avexe path, for which the dll library was generated. It is suggested to wrap `MacrofilterInterfaceClass` instantiating with a `try-catch` statement, since the `Create` method may throw exceptions, e.g. when some required libraries are missing or are corrupted.

Finalization

`MacrofilterInterfaceClass` class implements the IDisposable interface, in which the Dispose() method, libraries' releasing and other cleaning ups are performed. It is a good practice to clean up on application closure.

Bitness

Generated assembly bitness is the same as the used Aurora Vision Studio bitness. That is why the user application needs to have its target platform adjusted to avoid libraries' format mismatch. For example, if Aurora Vision Studio 64-bit was used, the user application needs to have its target platform switched to the *x64* option.

Diagnostic mode

The macrofilter execution mode can be modified with `DiagnosticMode` property of the generated Macrofilter .NET Interface class. It enables both checking and enabling/disabling the diagnostic mode.

Dialogs

It is possible to edit geometrical primitives the same way as in Aurora Vision Studio. All that need to be done is to use appropriate classes from the Avl.NET.Designers.dll assembly. Dialog classes are defined in AvlNet.Designers namespace. For more info see the AVL.NET Dialogs article.

Full AVL.NET

With Aurora Vision Library installed one can take advantage of full AVL.NET functionality in the applications that use Macrofilter .NET Interface assemblies. Just follow the instructions from Getting Started with Aurora Vision Library .NET.

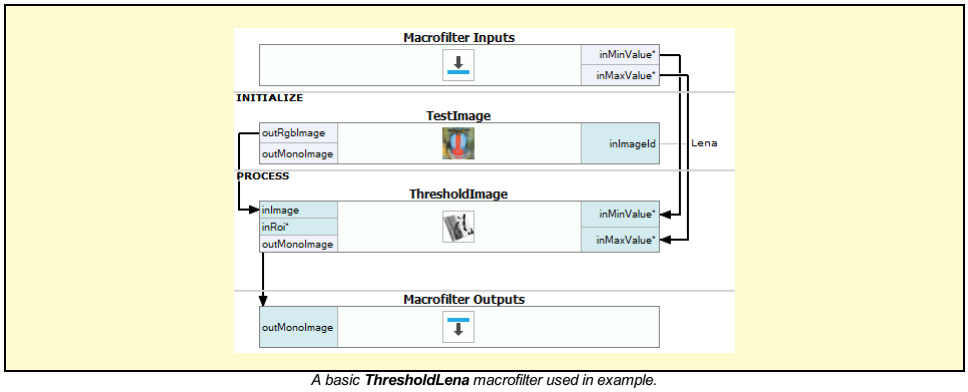
Example

Introduction

This example is a step-by-step demonstration of the .NET Macrofilter Interface Assembly generation and usage. To run this example at least Aurora Vision Studio Professional 5.3 and Microsoft Visual Studio 2015 are required. Visual C# will be used to execute macrofilters.

Creating Aurora Vision Project

We have to have some Aurora Vision Studio project, which macrofilter we want to use in our application. For demonstrative purposes the project will be as simple as possible, just thresholding Lena's image with parameters provided in a Visual C# application's GUI. The macrofilter we would like to run in a C# application will be **ThresholdLena** (see: Creating Macrofilters). The whole macrofilter consists of two filters as in the image below:

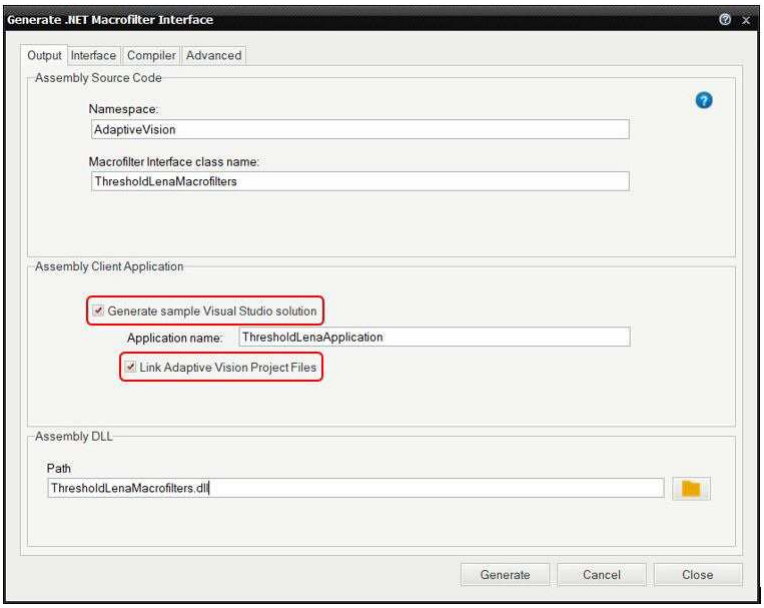


*A basic **ThresholdLena** macrofilter used in example.*

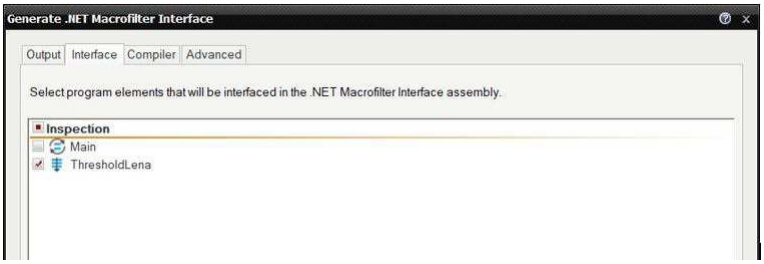
Generating .NET Macrofilter Interface Assembly

Having a ready Aurora Vision Studio project, a .NET Macrofilter Interface assembly may be generated from *File » Generate .NET Macrofilter Interface...*. If the project has not been saved yet, you will be prompted to do it now.

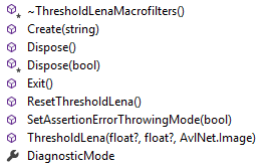
In the Output page of the .NET Macrofilter Interface dialog check the *Generate Visual Studio Solution* option to create a default C# project with required configuration properly set for the current Aurora Vision Studio installation. It is suggested to check also the *Link Aurora Vision Project Files* check box to include necessary files to the C# project.



In the Interface page check the ThresholdLena to generate .NET Macrofilter Interface for the macrofilter. This macrofilter will be accessible as C# method later on.



When ready, click *Generate* to generate an assembly, which will allow us to run the ThresholdLena macrofilter in a Visual C# application.



Methods provided by macrofilter interface class.

Using .NET Macrofilter Interface Assembly in a Visual C# application

Having generated the ExampleMacrofilters.dll we may include it into Visual C# application references and have access to the only type exposed by the library, i.e. `AuroraVision.ExampleMacrofilters` class (if you have checked the *Generate Visual Studio Solution* option in the *Generate .NET Macrofilter Interface* dialog, you may proceed to the next paragraph since all references are already set, as well as a ready-to-run starter application). Aside from generated ExampleMacrofilters.dll there is also the Avl.NET.TS.dll assembly which has to be referenced if any of the AVL types are used in the application. In this example the `AvlNet.Image` type will be used to obtain an image from ThresholdLena macrofilter so Avl.NET.TS.dll is essential here.

As it was mentioned before, instantiation of this class should be performed once during application lifetime. As so, `Form1`'s constructor is a perfect place for such an initialization. Instance of the `ExampleMacrofilters` should be then kept in the `Form1` class as a private member:

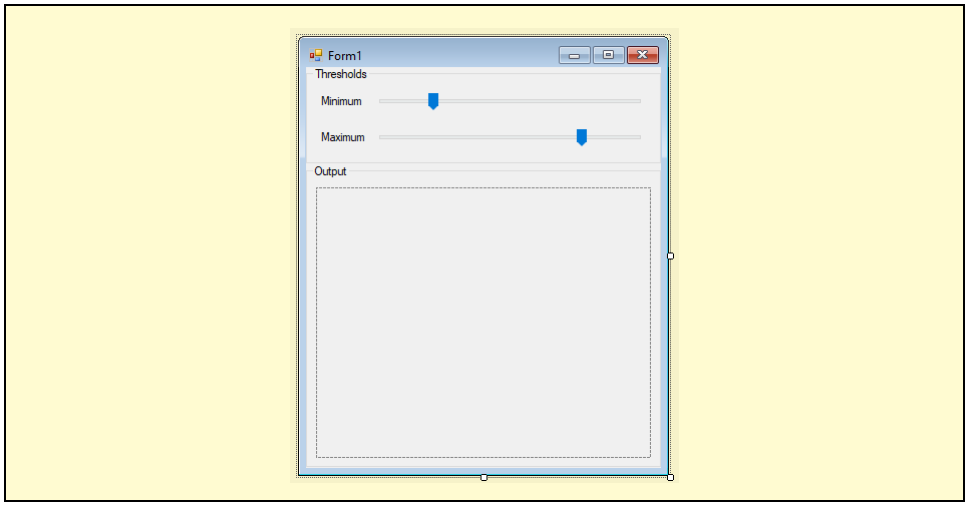
```
using AvlNet;
using AuroraVision;
    public partial class Form1 : Form
    {

        /// <summary>
        /// Object that provides access to the macrofilters defined in the Aurora Vision Studio project.
        /// </summary>
        private readonly ThresholdLenaMacrofilters macros;

        public Form1()
        {
            InitializeComponent();
            try
            {
                string avsProjectPath = @"auroravision\ThresholdLena.avproj";
                macros = ThresholdLenaMacrofilters.Create(avsProjectPath);
            }
            catch (Exception e)
            {
                MessageBox.Show(e.Message);
            }
        }
```

`ExampleMacrofilters` class does not provide a public constructor. Instead, instances of the `ExampleMacrofilters` class can only be obtained through it's static method `Create` accepting a path to either *.avcode, *.avproj or *.avexe file. In this example it takes the path to the *.avcode file with the definition of ThresholdLena macrofilter. *Example.avcode* passed to the `Create` method means, that the runtime will look for the *.avcode file in the application output directory. To guarantee that this file will be found, it should be included in the project and its "Copy to Output Directory" property should be set to either "Copy always" or "Copy if newer".

The C# project is prepared to run the macrofilters as methods. Since ThresholdLena outputs an image taking two optional float values, which stand for threshold's minimum and maximum values, let's add one `PictureBox` and two `TrackBar` controls with value range equal to 0-255:



Changing a value of either of track bars calls the `UpdateImage()` method, where ThresholdLena macrofilter is executed and calculated image is printed in the PictureBox control:

```
private void UpdateImage()
{
    try
    {
        //create an empty image buffer to be populated in the ThresholdLena macrofilter
        using (var image = new Image())
        {
            //call macrofilter
            macros.ThresholdLena(minTrackBar.Value, maxTrackBar.Value, image);

            //dispose previous background if necessary
            pictureBox1.Image?.Dispose();

            //get System.Drawing.Bitmap object from resulting AvlNet.Image
            pictureBox1.Image = image.CreateBitmap();
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(
            ex.Message,
            "Macrofilter Interlace Application",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

On application closing, all resources loaded in `ExampleMacrofilters.Create(...)` method, should be released. It is achieved in `ExampleMacrofilters.Dispose()` instance method, which should be called during disposing of containing form, in `Form1.Dispose(bool)` override:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        if (components != null)
            components.Dispose();

        //Release resources held by the Macrofilter .NET Interface object
        if (macros != null)
            macros.Dispose();
    }

    base.Dispose(disposing);
}
```

Hints

- Generated macrofilter interface class offers also **`Exit()`** method which allows user to disconnect from Aurora Vision environment.
- Every step macrofilter contains specific resetting method, which resets an internal state. For example method **`ResetLenaThreshold()`** resets internal register values and any iteration states.
- Best way to create an application using macrofilter interfaces is to use encrypted avexe files. It secures application code from further modifications.
- Diagnostic mode can be switched on and off with IsDiagnosticModeEnabled static property of the AvlNet.Settings class.
- The application using Macrofilter .NET interface may also reference the AVL's AvlNet.dll assembly to enable direct AVL function calls from the user code (see Getting Started with Aurora Vision Library .NET for referencing the AvlNet.dll in the user applications).

Remote USB License upgrade

Introduction

This guide will show steps, which are to be taken, when dongle license upgrade is considered. This may happen in various situations, like purchase of new products or new major software release.

To perform remote license upgrade following will be needed:

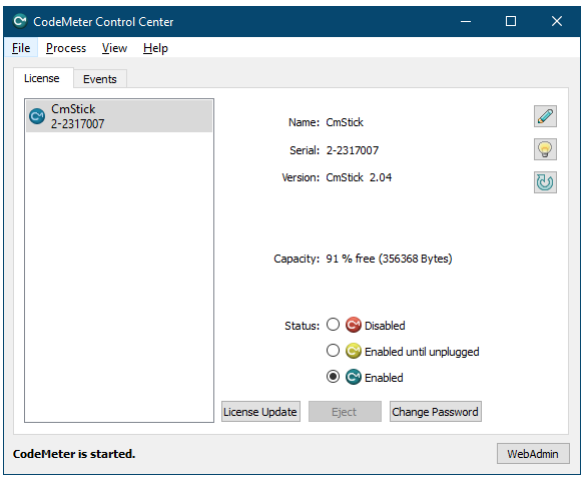
- Computer with Wibu's CodeMeter runtime. This can be installed with Aurora Vision Studio or downloaded from https://www.wibu.com/support/user/downloads-user-software.html.
- Access to Internet.
- Dongle, which will be upgraded.

Process of remote update consists of three steps:

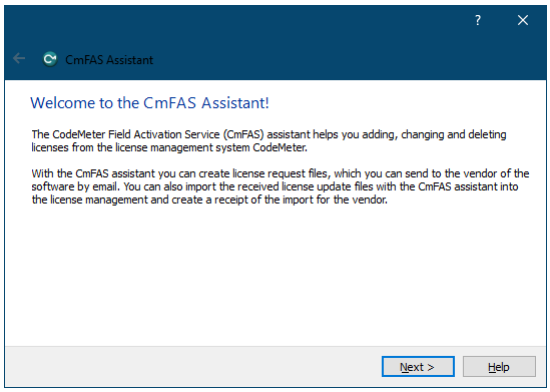
1. Creation of WibuCmRac file, which identifies your dongle.
2. Sending created file to Aurora Vision team.
3. Receiving WibuCmRau file, which is used to update dongle.

WibuCmRac file creation

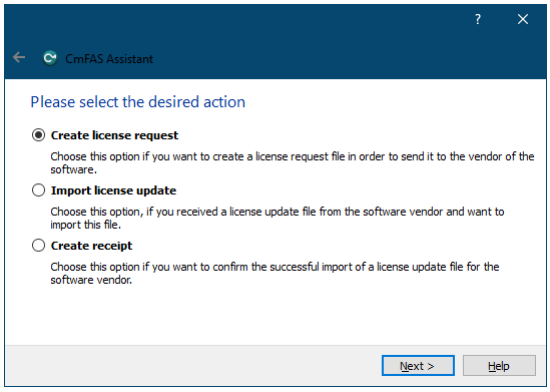
First step is to open CodeMeter Control Center, by clicking on its icon in systems' tray. Opened window should like below:



Next step is to choose dongle, that is about to be upgraded and then click on "License Update" button, which will bring CmFas Assistant window:

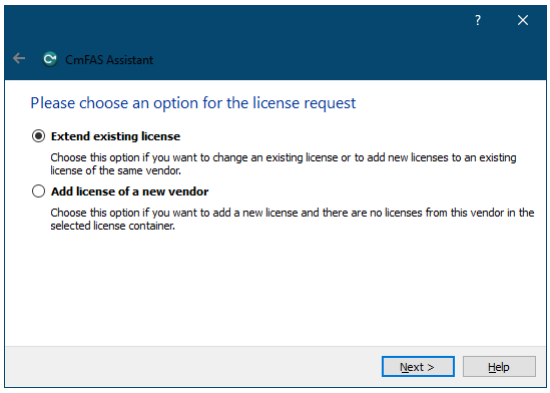


If "Next" button is chosen, window with possible operations will appear:

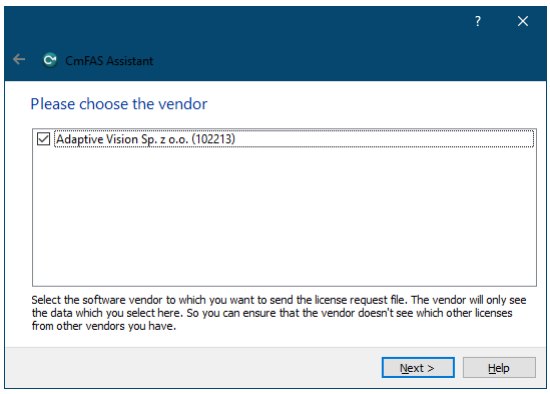


To create WibuCmRac file, which is necessary for process of update, first option should be selected, "Create license request".

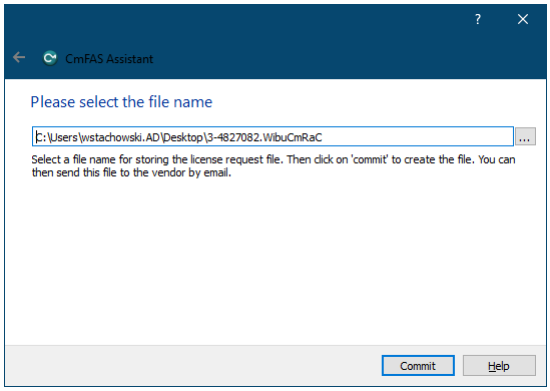
Because there is already Aurora Vision Studios' license in dongle being upgraded, in next step "Extend existing license" should be chosen.



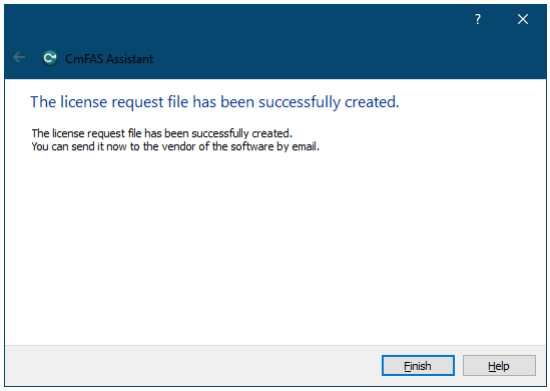
In next window one can choose which vendors' license is wanted to be updated. "Adaptive Vision Sp. z o.o." should be selected.



The last step of creation WibuCmRac file is to choose its name (default is fine) and localization.

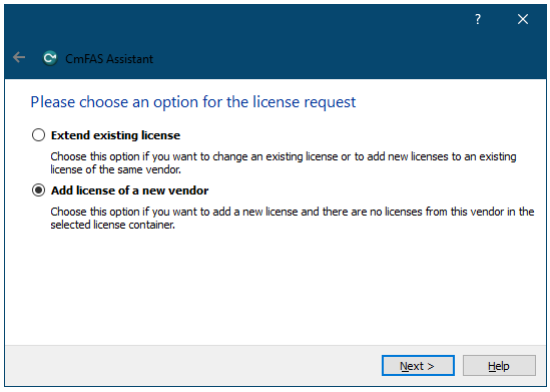


All settings need to be applied by clicking "Commit". If everything went well, this window will appear:

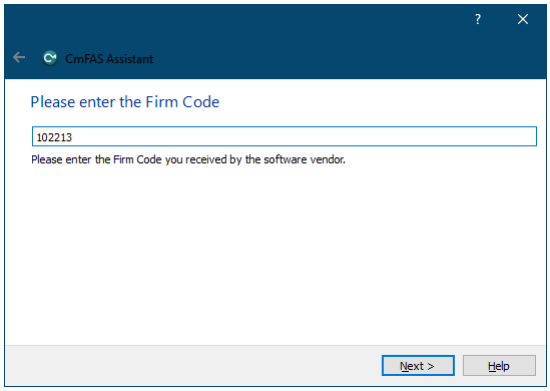


Now its time to send created file to Aurora Vision team.

If your dongle happens to be empty (no prior Aurora Vision license is programmed, and list with available producers is empty), you need to select "Add license of a new vendor" option in CmFAS Assistant:



In next window one will be prompted for entering the FirmCode of vendor. In case of "Adaptive Vision Sp. z o.o." one should type 102213, as shown on image below:



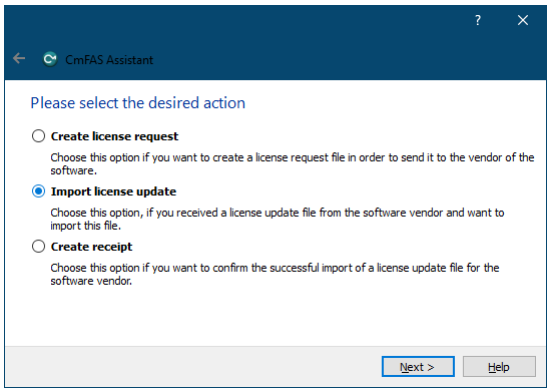
After committing these steps, request file will be generated as described earlier in this section.

Using WibuCmRau file to upgrade USB dongle

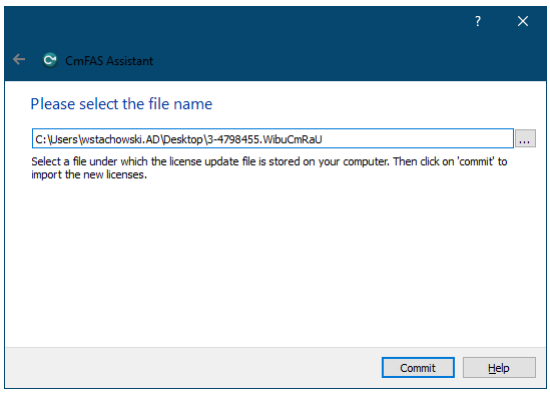
When WibuCmRau file is received, one can transfer it to USB dongle, which has to be present in one's computer's USB port.

Using CodeMeter Control Center to upgrade USB dongle

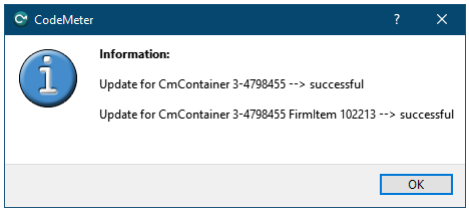
As previously, Control Center needs to be opened by clicking on its icon in tray. Dongle needs to be selected, and "License update" should be clicked. Note that WibuCmRau file can only be used once, and will work only with dongle, from which the corresponding WibuCmRac was created.



This time "Import license update" needs to be chosen.

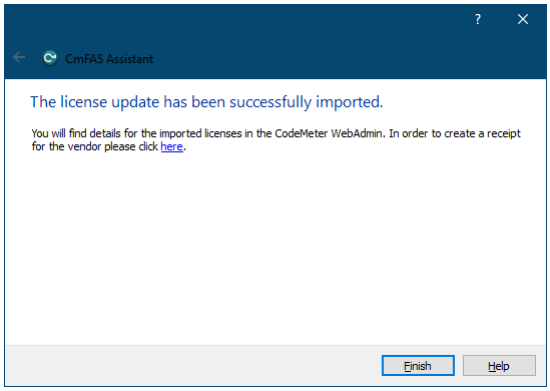


Navigate to file, which was received from Aurora Vision team, and click "Commit".



If everything went well, similar information will pop out.

Last window of assistant contains "Finish" button, which need to be clicked in order to exit.

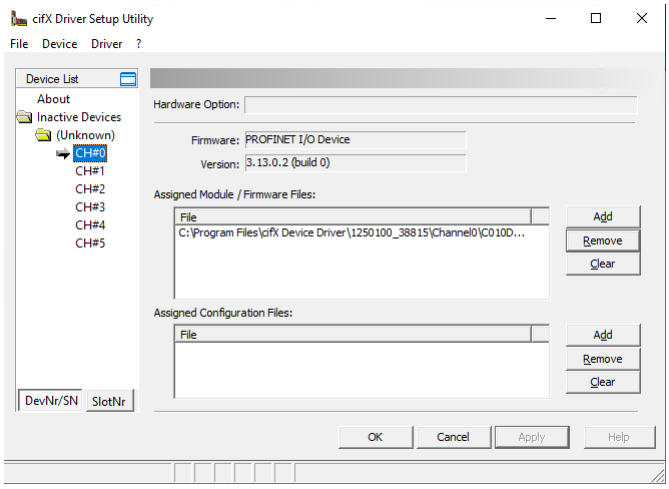


Working with Hilscher Devices

Introduction

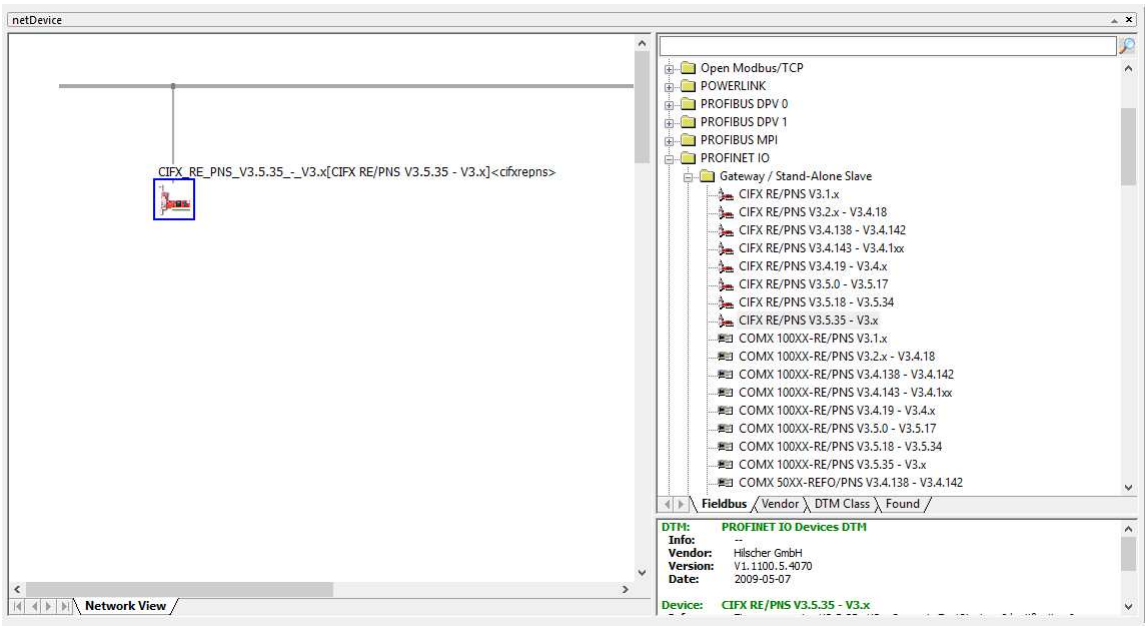
To work with a Hilscher device make sure that you have the recommended driver installed, newest SYCON.net, and firmware prepared. Profinet requirements are written down in Hilscher_Channel_Open_Profinet documentation.

The easiest way to install firmware to the cifX Driver is using the cifX Driver Setup software - the Windows Search in the start menu should find this tool, otherwise it is located in `C:\Program Files\ cifX Device Driver` directory. In the cifX Driver Setup Utility select the channel you want to configure (usually channel 0 - CH#0) and remove preexisting firmware by clicking "Clear". To add a new firmware file click "Add", navigate to the downloaded profinet slave firmware and click "Open". Finally, click "Apply" - the button should turn disabled.

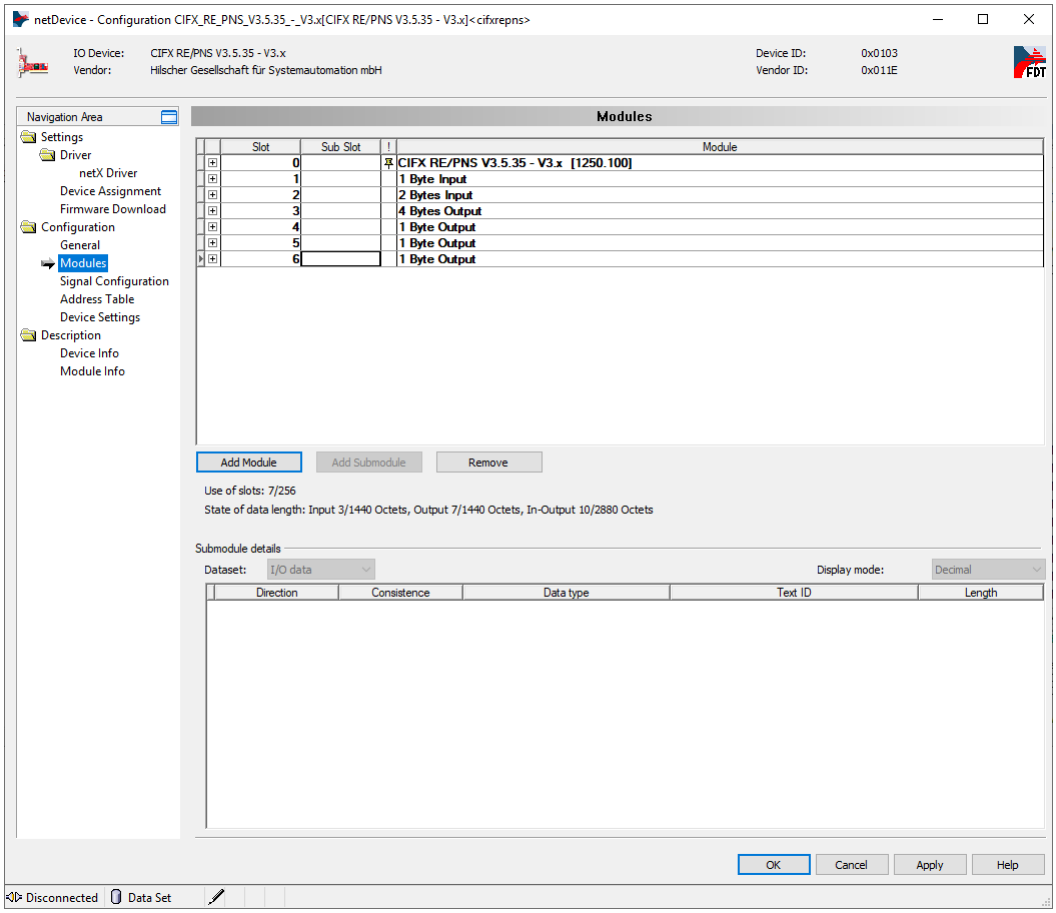


Generating channel configuration files

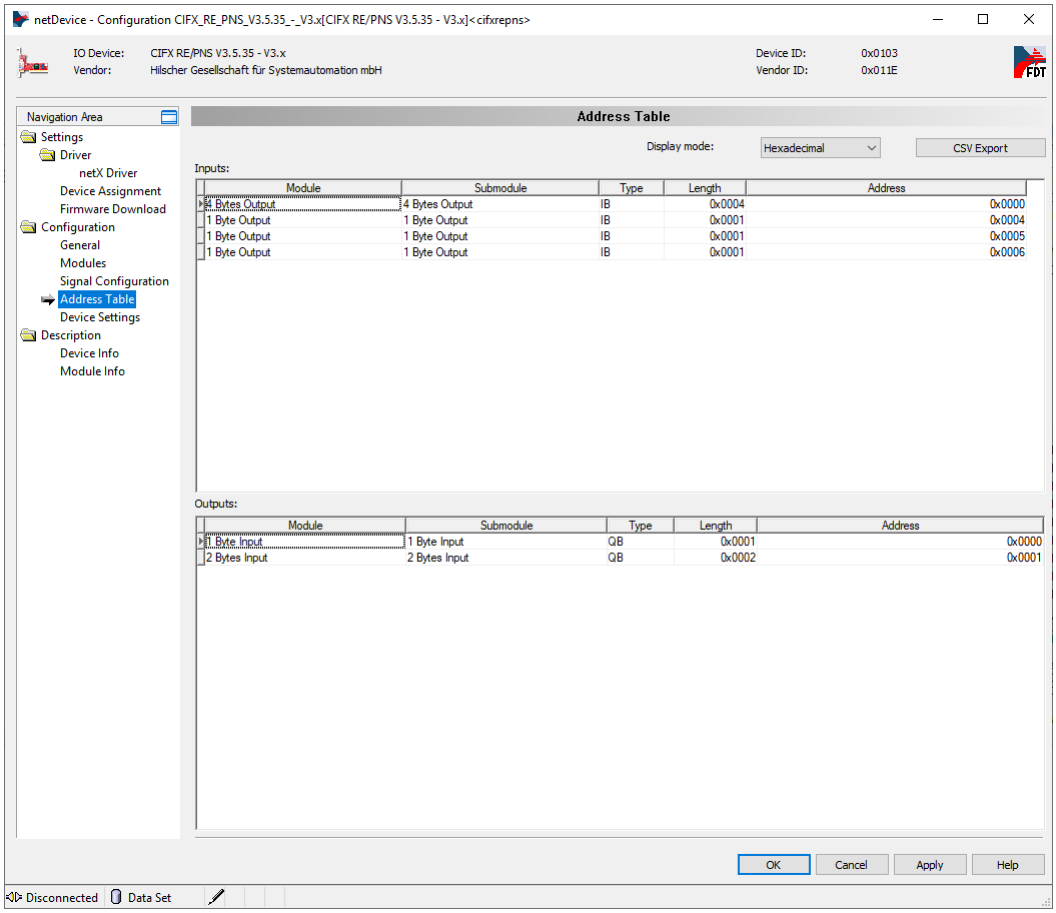
Channel configuration files are generated in SYCON.net software. Firstly, pick a proper slave device using a PCI card, here we use the Profinet Stand-Alone Slave, and drag it to the gray bus on the "Network View".



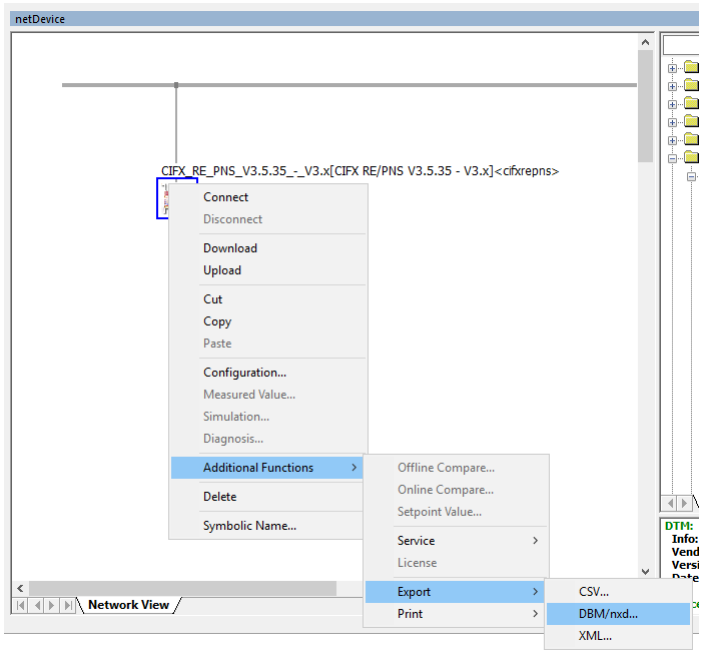
To open card configuration, double click on its icon and select the "Configuration/Modules" category in the Navigation Area on the left side of the dialog. In the main window you can add individual modules. **Remember that in Profinet, Slot configuration must match the configuration from your master device, otherwise the connection will not work.** For boolean indicators we recommend "1 Byte Output" or "1 Byte Input".



You can see the final address table (addresses on the Hilscher device where input or output data will be available) in the "Configuration/Address table" category. If you plan to use IORead and IOWrite filters, for example Hilscher_Channel_IORead_SInt8, note the addresses. You can switch the display mode to decimal, as Aurora Vision Studio accepts decimal addressing, not hexadecimal. Aurora Vision Studio implementation of Profinet checks whether the address and data size match. In the sample configuration below, writing a byte to address 0x002 would not work, because that module address starts at 0x001 and spans 2 bytes. Moreover, Aurora Vision Studio prohibits writing with a SlotWrite filter to input areas, and reading with a SlotRead filter from output areas. Click "OK" when you have finished the proper configuration.



The final step is to generate configuration files for Aurora Vision Studio. You can to this by right clicking on the device icon, then navigating to "Additional Functions->Export->DBM/nxd...", entering your configuration name and clicking "Save". You can now close SYCON.net for this example, **remember to save your project before, so that it is easier to add new slots.**



Filters in Aurora Vision Studio

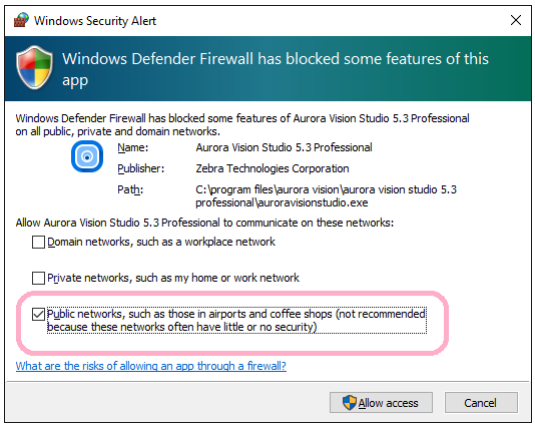
At the beginning of every program where you want to use filters intended for communication over a Hilscher card you need to add Hilscher_Channel_Open_Profinet filter. The configuration files generated in the previous step are now required in inConfig (`xxx.nxd`) and inNwid (`xxx_nwid.nxd`) properties of that filter. These files are used when there is no connection between a card and a Profinet master. In that case Aurora Vision Studio updates the card configuration and starts the communication. For IO, we recommend SlotRead and SlotWrite filters, as they are more convenient. For example, the Hilscher_Channel_SlotWrite_SInt8 filter writes 8 bytes of signed data to the selected slot. Slot numbers match those in the "Configuration/Modules" category of card configuration in SYCON.net program.

9. Working with GigE Vision® Devices

Table of content:

Enabling Traffic in Firewall

Standard windows firewall or other active firewall applications should prompt for confirmation on enabling incoming traffic upon first start of an Aurora Vision program that is using a video streaming filter. Sample prompt message from standard Windows 7 Firewall is shown on the image bellow.



Please note that **a device connected directly to computer's network adapter** in Windows Vista and Windows 7 will become an element of an unidentified network. Such device will be treated by default as **Public network**. In order to communicate with such a device you must allow for traffic also in Public networks as shown on the above image.

Clicking on *Allow access* will enable application to stream video from a device. Because of a delay caused by the firewall dialog first run of a program may fail with a timeout error. In such situations just try running program again after enabling access.

For information about changing settings of your firewall application search how to allow a program to communicate through this firewall in a Windows help or a third party application manual. GigE Vision® driver requires that incoming traffic is enabled on all UDP ports for Aurora Vision Studio and Aurora Vision Executor (by default located in *C:\Program Files\Aurora Vision\Aurora Vision Studio Professional\AuroraVisionStudio.exe* and *C:\Program Files\Aurora Vision\Aurora Vision Studio Runtime\AuroraVisionExecutor.exe*).

Enabling Jumbo Packets

Introduction

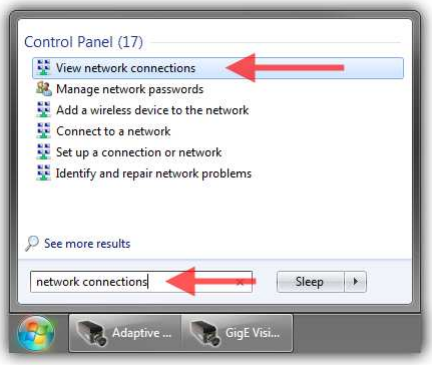
Jumbo Packet is an extension of network devices that allows for transmission of packets bigger than 1.5kB. Enabling Jumbo Packets can significantly increase video streaming performance.

Note that not all network devices support Jumbo Packets. To activate a big packet size, all devices from a network adapter through network routing equipment to a camera device must support and have enabled big packet sizes. Most suitable situation for using Jumbo Packets is when the device is connected directly to computer's network adapter with a crossed Ethernet cable.

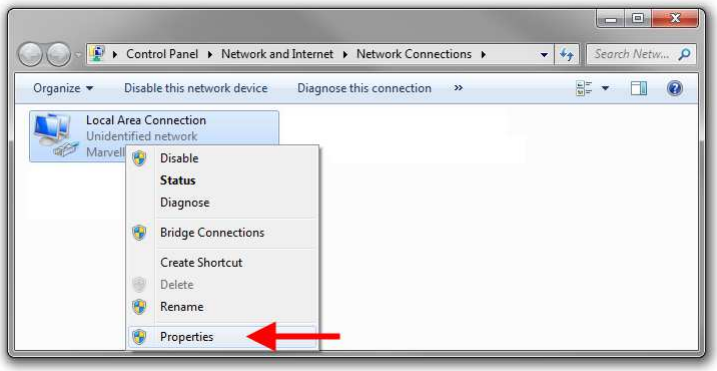
Do not enable Jumbo Packets when the device is connected through complicated network infrastructure with more that one routing path as maximum allowed packet sizes detected at application start can change later in the process.

Enabling Jumbo Packets in Windows Vista/7

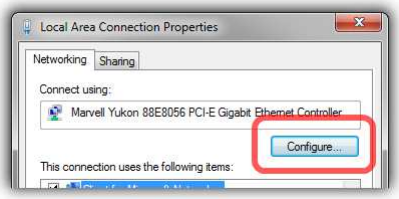
1. Open network connections applet from the control panel.



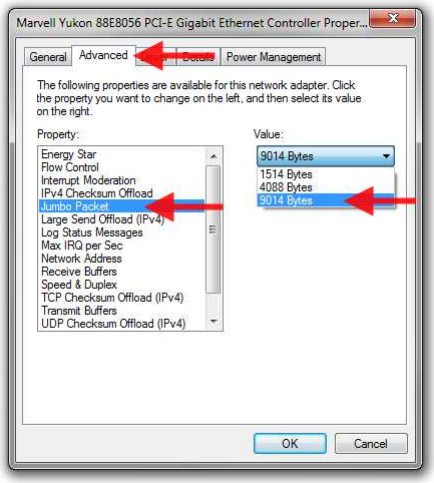
2. Right click a network adapter that have a connection with a device and open its properties (an administrator password may be needed).



3. In network adapter's properties click on *Configure*.



4. From Advanced tab select *Jumbo Packet* property and increase its value up to 9014 Bytes (9k Bytes). This step might look differently depending on the network card vendor. For some vendors this property might have different similar name (e.g. *Large Packet*). When there is no property for enabling/setting large packet size this card does not support jumbo packets.



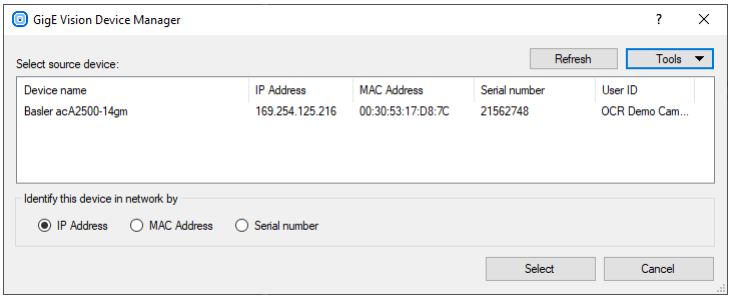
5. Click OK.

GigE Vision® Device Manager

The GigE Vision Device Manager is available from *Tools* section within the Main Window Menu. It also appears when a user starts to edit some properties of GigE Vision®-related filters.

Device Manager Functions

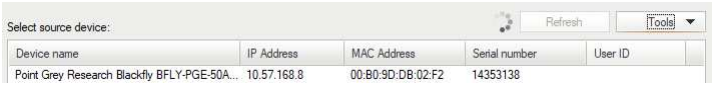
Typical state of the Device Manager is shown on image below. Note that the window may change its appearance depending on its purpose (like selecting a device address in a filter).



At first the manager will search local network for active devices. All found devices will be shown in list with the following information: manufacturer name and device name, current IP address, network interface hardware address (MAC address), serial number (if supported), user specified name (saved in the device memory; if supported by device). Informations like MAC address and serial number should be printed on the device casing for easy identification. Sometimes, when a device has more than one interface, is may appear in list more that once. In this situation every entry in the list identifies another device feature.

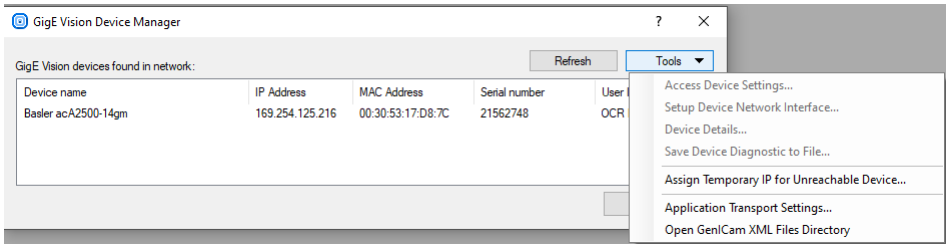
Refresh

Refresh button performs a new search in the network. Use this function when the network configuration has been changed, a new device has been plugged in or when your device has not been found at startup.



Tools

Tools button opens a menu with functions designed for device configuration. Some of these functions are device dependant and require the user to selected a device on the list first (they are also available in a device context menu).



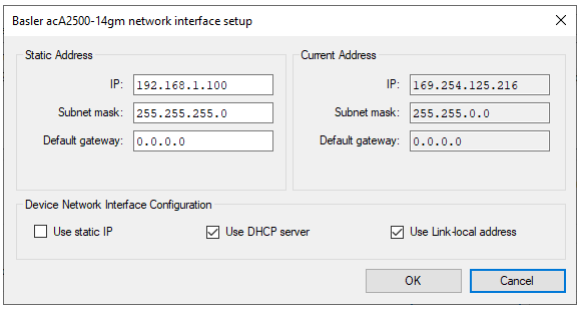
Tool: Access Device Settings...

This tool allows to access device-specific parameters prepared by its manufacturer and available through GenICam interface.

See: Device Settings Editor

Tool: Setup Device Network Interface...

This tool is intended to manage network configuration of a device network adapter.

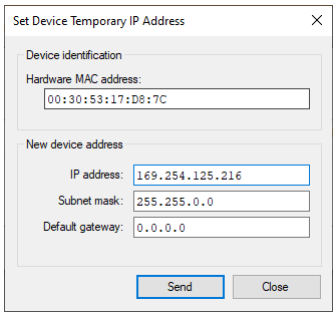


- **Static address** — this field allows to set a static (persistent) network configuration saved in device non-volatile memory. Use this setting when the device is identified by IP address that cannot change or when automatic address configuration is not available. This field has no effect when *Use static IP* field is not checked.
- **Current address** — this read-only field shows current network configuration of a device, for example the address assigned to it by a DHCP server.
- **Device IP configuration** — this field allows to activate or deactivate specified methods of acquiring addresses by a device on startup. Some of this options can be not available (grayed) when the device is not supporting specified mode.
  - **Use static IP** — Device will use address specified in *Static address* field.
  - **Use DHCP server** — When a DHCP server is available in the network, devices will acquire automatically assigned address from it.
  - **Use Link-local address** — When there is no other method available a device will try to find a free address from 169.254.-.- range. When using this method (for example on a direct connection between the device and a computer) the device will take significantly more time to become available in network after startup.

After clicking OK the new configuration will be send to a device. Configuration can be changed only when the device is not used by another application and/or is not streaming video. New configuration may be not available until the device is restarted or reconnected.

Tool: Assign Temporary IP for Unreachable Device

This tool is intended for situations when a device cannot be accessed because of its invalid or unspecified network configuration (note that this should be a very rare case and usually the device should appear in list). The tool allows to immediately change network address of an idle device (thus realizing GigE Vision® FORCE IP function).



This tool requires a user to specify device hardware network adapter MAC address (should be printed on device casing). After that a new IP configuration can be specified. The address can be changed only when the device is idle (is not connected to other application and not streaming video). The new address will be available immediately after successful send operation.

Tool: Application Transport Settings...

This tool allows to access and edit application settings related with driver transport layer, like connection attempts and timeouts. Settings are saved and used at whole application level. Changes affects only newly opened connections.

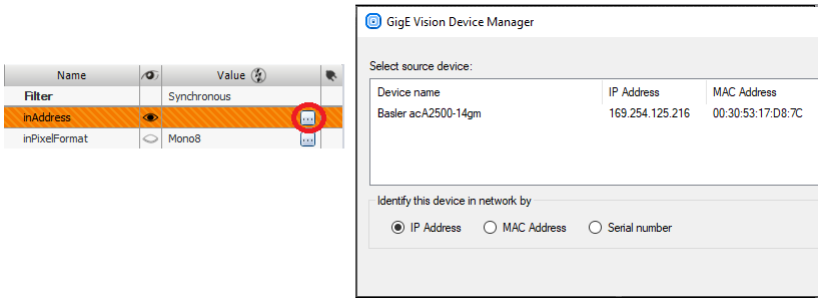
Tool: Open GenICam XML Directory

GigE Vision® devices are implementing GenICam standard. GenICam standard requires that a device must be described by a special XML file that defines all device parameters and capabilities. This file is usually obtained automatically by the application from the device memory or from manufacturer's internet web page. Sometimes the XML file can be supplied by manufacturer on a separate disk. Aurora Vision Studio and Aurora Vision Executor use a special directory for these files which is located in the user data directory. Use this tool to open that directory.

Device description files should be copied into this directory without changing their name, extension and content. File can also be supplied as a ZIP archive — do not decompress such file nor change its extension.

Selecting Device Address for Filter

Device Manager can be opened as a helper tool for editing device address in filter properties.



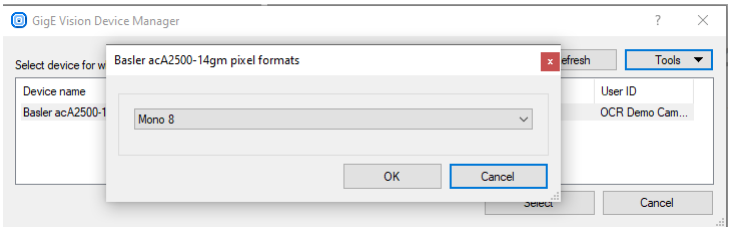
In such mode the manager allows to select one device from list (by selecting it on a list and clicking Select or by double clicking on the list). Below the list shown are options determining how a device will be identified in program:

- **IP Address** - (Default) Current device IP address will be saved in program. This mode allows for faster start, but when the device is using automatic IP configuration its address can change in future and device will become unavailable for the program.
- **MAC Address** - Device hardware MAC address will be saved in a program. Upon every program start application will search for the device to obtain most recent IP address. This mode allows to identify device even when its IP is changed.
- **Serial number** - Device serial number will be saved in a program. Use this mode only when device serial number identification is supported by device. Upon every program start application will search for the device to obtain most recent IP address. This mode allows to identify the device even when its IP has changed.

After device selection, address in the chosen format is inserted into a filter's property value.

Selecting Pixel Format

Device manager can be opened as helper tool for editing device pixel format for GigEVision_GrabImage filter parameter.



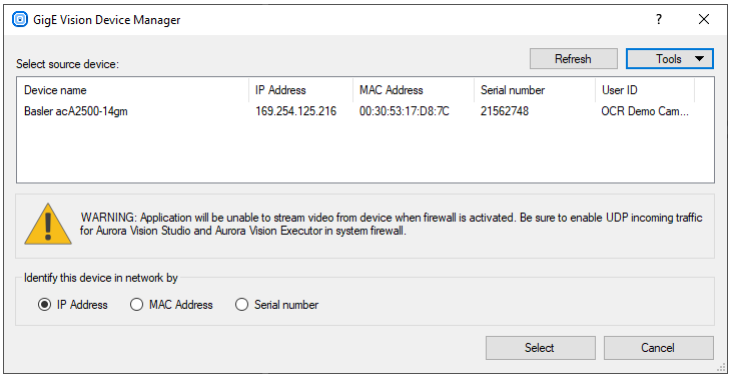
In this mode additional dialog is shown with a drop-down list of pixel formats supported by a device. When there is more that one device in the network you must first select a device for which format will be selected. When there is no device available in network a list of all standard pixel formats will be shown.

Pixel format list contains all formats signaled by a device and uses a name provided by the device. If it is a device-specific format, if may be not possible to decode it.

After selecting a format and clicking Ok the format name will be inserted into a filter's property value.

Firewall Warning

In some situations device manager might show a warning message about Windows firewall.



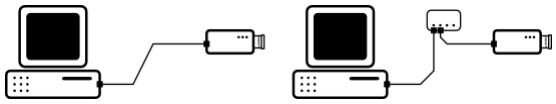
This warning is shown when the application has determined that Windows firewall will block any type of UDP traffic for application. For some devices it is required that incoming UDP traffic is explicitly allowed in firewall. Lack of this warning does not indicate that firewall will not block connection (for example if there is a 3rd party firewall in the system or when the Windows firewall allows traffic in Private domain but a device is in an unidentified network).

To clear out this message you must enable incoming traffic on all UDP ports in Windows firewall. For more information about enabling traffic in Windows Firewall see: Enabling Traffic in Firewall.

Connecting Devices

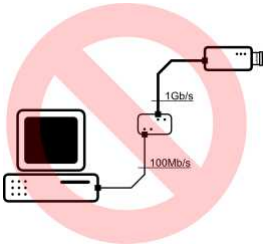
Connecting a GigE Vision device to a computer means plugging both into the same Ethernet network.

It is recommended that the connection is as simple as possible. To achieve best performance use direct connection with a crossed Ethernet cable or connect the camera and the computer to the same Ethernet switch (without any other heavy traffic routed through the same switch).



The device and the computer must reside in a single local area network and must be set up for the same subnet.

GigE Vision® is designed for 1 Gb/s networks, but it is also possible to use 100 Mb/s connection as long as the entire network connection have an uniformed speed (some custom device configuration might be required when the device is not able to detect connection speed automatically). It is recommended however to avoid connecting a device to a network link which is faster than the maximum throughput of the whole network route. Such configurations require manual setting of the device's transmission speed limit.



Firewall Issues

GigE Vision® protocol produces a specific type of traffic that is not *firewall friendly*. Typical firewall software is unable to recognize that video streaming traffic is initialized by a local application and will block this connection. Aurora Vision's GigE driver attempts to overcome this problem using *firewall traversal mechanism*, but not all devices support this.

It is thus required to enable incoming traffic on all UDP ports for Aurora Vision Studio and Aurora Vision Executor in a firewall on your local computer.

For information how to enable such traffic in Windows Firewall see: Enabling Traffic in Firewall.

Configuring IP Address of a Device

In most situations a GigE Vision device is able to automatically obtain an IP address without user action (using DHCP server or automatic local link address). It is however recommended to set a static IP address for both local network card and device whenever possible. In some cases (e.g. when preparing the device for operation in an industrial network) it might be required to access and set/change the device's network configuration for a proper static IP address. Most suitable for this purpose will be a software and a documentation provided by the device manufacturer. When these are not available Aurora Vision Studio offers universal configuration tools available from the GigE Vision Device Manager (see: Device Manager section).

Packet Size

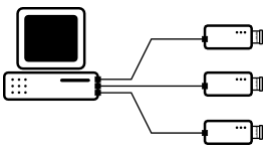
Network video stream is divided into packets of a specified size. The packet size is limited by the Ethernet standard but some network cards support an extension called *jumbo packet* that increases allowed packet size. Because a connection is more efficient when the packet size is bigger, the application will attempt to negotiate biggest possible network packet size for current connection, taking advantage of enabled jumbo packets.

For information how to enable jumbo packets see: Enabling Jumbo Packets

Connecting Multiple Devices to a Single Computer

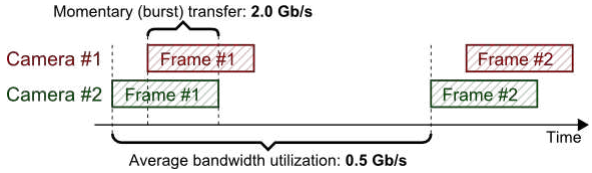
It is possible to connect multiple GigEVision cameras to a single computer and to perform image processing based on multiple video streams (e.g. observing objects from multiple sides), however it can introduce multiple technical challenges that must be considered.

For the best performance it is recommended to connect all devices directly to the computer using multiple gigabit network cards:



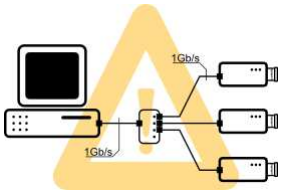
In such configuration it is required for the computer hardware to handle concurrent gigabit streams at once. Even when separate network cards are able to receive the network streams there still may be problems for the computer hardware to transfer data from the network adapters to the system memory. Attention must be paid when choosing hardware for such applications. When given requirements are not met the system may observe excessive packets loss leading to the video stream frames loss.

Even when the cameras framerate is low and the resulting average network throughput is relatively low the system still **may drop packets during network bursts** when the momentary data transfer exceeds the system capabilities. Such burst may appear when multiple cameras transmit a single frame at the same time. By default GigEVision camera is transferring a single frame with maximum available speed and lowering framerate is only increasing the gaps between frame transfers:



Although diagnostic tools will report network throughput utilization to be well below system limits it is still possible for short burst transfers to temporary exceed the system limits resulting in packets drop. To overcome such problems it is required to not only ensure camera framerates to be below proper limit, but also to limit the maximum network transfer speed of the device network adapters. Refer to the device documentation for details about how to limit the network transfer speed in specific device. Usually this can be achieved by decreasing value of parameters such as *DeviceLinkThroughputLimit* or *StreamBytesPerSecond* (in bytes per second), or by introducing delays in between the network packets by increasing parameters such as *PacketDelay*, *InterPacketDelay* or *GevSCPD* (measured in internal device timer ticks - must be calculated individually for device using device timer frequency).

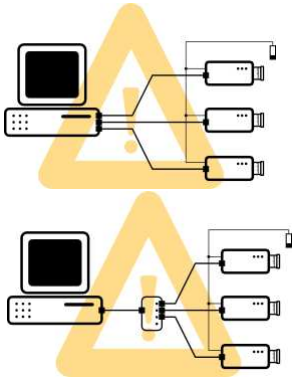
Above requirements are especially important when cameras are connected to the computer using a single network card and a shared network switch:



Special care must be taken to assure that all the cameras connected to the switch (when all transmitting at once) do not exceed transfer limits of the connection between the switch and the computer, both average transfer (by limiting the framerate) as well as temporary burst transfer (by limiting network transfer speed). Network switch will attempt to handle burst transfers by storing the packets in its internal buffer and transmitting packets stored in the buffer after the burst, but when the amount of data in burst transfer exceeds the buffer size the network packets will be dropped. Thus it is required for the switch buffer to be large enough to store all camera frames captured at once, or to limit transmission speed for the switch buffer to not overflow.

It is important to note that **the maximum performance of the multi-camera system with shared network switch is limited by the throughput of the link between the switch and the computer**, and usually it will not be possible to achieve the maximum framerate and/or resolution of the cameras.

A common case of using multiple cameras at once is to capture multiple photos of a object from a single trigger source (with synchronous triggering):

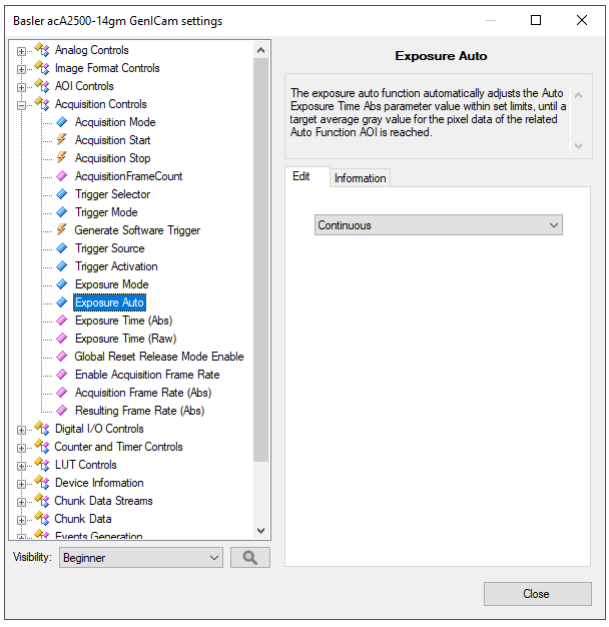


All above recommendations must be considered for such configuration. Because under synchronous triggering all the cameras will always be transferring images at the same time the problem of momentary burst transfer is especially present. Care must be taken to limit the maximum network transmission speed in the cameras to the system limits and to give enough time between trigger events for the cameras to finish the transfer.

Device Settings Editor

GigE Vision® compliant devices are implementing GenICam standard that describes camera internal parameters and a way how to access them. Device Manager allows a user to access and edit device settings through a Settings Editor tool (available from *Tools » Access Device Settings*).

Example appearance of the Device Settings Editor is shown on the image below.



On the left side of the window is a tree representation of device parameters split into categories. All these parameters and their organization is device dependent, which means that different devices can produce different sets of parameters, with different meanings. Parameter's friendly name and a brief explanation (also provided by a device) is shown on the right side of the window after the parameter is highlighted in tree. For more information about specific parameter functions refer to a device documentation.

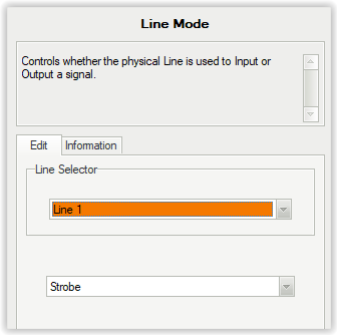
When editing selected parameter is possible and supported, an editor of the parameter value will be displayed below its explanation. Different editors are provided for the following parameter types:

- **Integer** - Plain number or hexadecimal number (indicated by *Hex* label on the left side of the text box). Values are limited by their maximum, minimum and allowed step. Numbers that does not fulfill this rules are corrected automatically upon confirmation. After clicking on *Save* button (or pressing *Enter*) new value will be validated and sent to the device.
- **Float** - Real number with fractional part. Values are limited by their maximum and minimum. Numbers that do not match this range are corrected automatically upon confirmation. A parameter can also have suggested step added after clicking in +/- buttons. After clicking on *Save* button (or pressing Enter) new value will be validated and sent to the device.
- **String** - Text of a limited length.
- **Boolean** - Single Yes/No value represented by check box. A value is sent to the device immediately after check state is changed.
- **Enumeration** - Parameter that accepts one of several predefined values. Predefined values are represented as list of their friendly names. Parameter is edited by choosing one of its values from a drop-down list. New values are sent to the device immediately after their selection in the list.
- **Command** - This is a special parameter that is represented only by a single button. Clicking on button will execute related activity in the device (for example Saving current parameter set to non-volatile memory).

Depending on situation, editor can be disabled (grayed), which means that this parameter is currently locked (for example parameter describing image format when the camera streaming is active). Editor can be read only (Save button grayed, grayed drop-down list or unchangeable check-box), which means that this parameter is read-only (for example informational parameters like manufacturer name).

Instead of an editor can there also be a displayed text: **"This parameter is currently not available"**. This means that the parameter can not be accessed or edited in the current device state or due to other parameter states. For example parameter describing acquisition frame rate value, when the user selection of frame rate is disabled (by parameter like *Enable Acquisition Framerate*).

Sometimes with the parameter editor displayed will be an additional editor, named selector.



In such situation selected parameter is connected with one of categories (slots) described by selector. In the example on the above image, parameter is determining whether the physical Line is used to Input or Output a signal. This device has two lines and both have its own separate values to choose from. Selector will pick which line we want to edit and bottommost editor will change it purpose. This means that there are actually two different *Line Modes* parameters in device.

Please note that selector will not always be displayed above editor. You must follow a device documentation and search parameters tree for selectors and other parameters on which this parameter is dependent.

The Device Settings Editor can be used to identify device capabilities and descriptions or to set up a new device. Device Editor can be also used when a program is running and the camera is streaming. In this situation changes should be immediately visible in the camera output.

Settings Editor gives a user an unlimited access to the device parameters and, **when used improperly, can put device in an invalid state** in which the device will become inaccessible by applications or can cause transitional errors in the program execution.

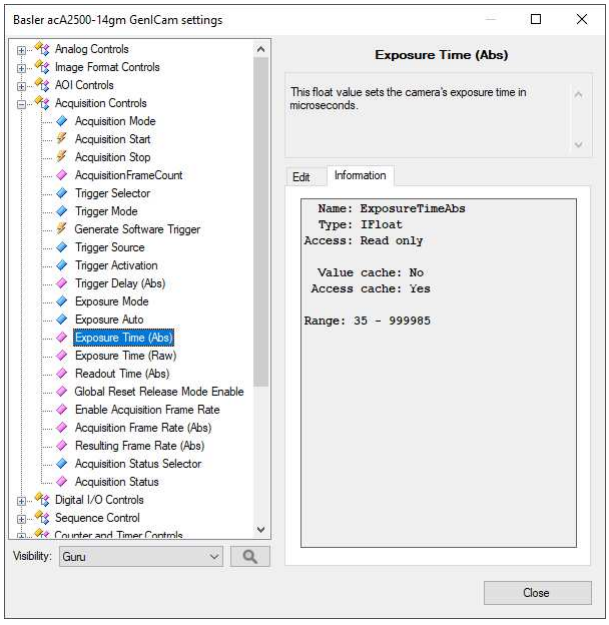
Saving Device Configuration

Most parameters available in the Settings Editor are stored by devices in a volatile memory and will be lost (reset to default) after device reset or power down.

A device should offer functions to save parameters set in *Configuration Sets* section of parameters tree. Refer to a device documentation for more information about configuration set saving and loading.

Selecting Parameter Name for Filter

The Settings Editor can be also used to select parameter names for filters in Aurora Vision Studio. You can access this mode by clicking ... button in filter properties list. When there is more that one device in the network one must be first selected in the manager. The tool will show parameters of the specified device for its current state.



In this mode the window appearance is changed. Instead of editors, on the right side of the window displayed are useful informations for a program developer, including:

- **Name** - parameter internal name. Note that parameter tree and descriptions are using human friendly name, not parameter ID. This field shows a proper parameter ID that must be used in a filter property. You can easily transfer this name to property list by clicking Select button (or double clicking parameter in tree).
- **Type** - parameter type name. This type must be consistent with the filter value type.
- **Access** - allowed access to parameter. Parameter must be writable to be set by program.
- **Range** - for numeric parameters this field shows the allowed range. Range of some parameters can change dynamically during its operation.
- **Value cache** - when GenAPI cache is enabled this field indicates if device allows to store this parameter value in local memory to reduce network operations.
- **Access cache** - when GenAPI cache is enabled this field indicates if device allows to store access mode of this parameter in local memory to reduce network operations on controlling parameter accessibility.
- **Available entries** - for enumeration parameters this field will list currently available values for a parameter. The field shows proper internal IDs that should be used when setting the parameter (note that editor's drop-down lists are using human friendly names).

Known Issues

In this section you will find solutions to known issues that we have came across while testing communication between Aurora Vision products and different camera models through GigE Vision.

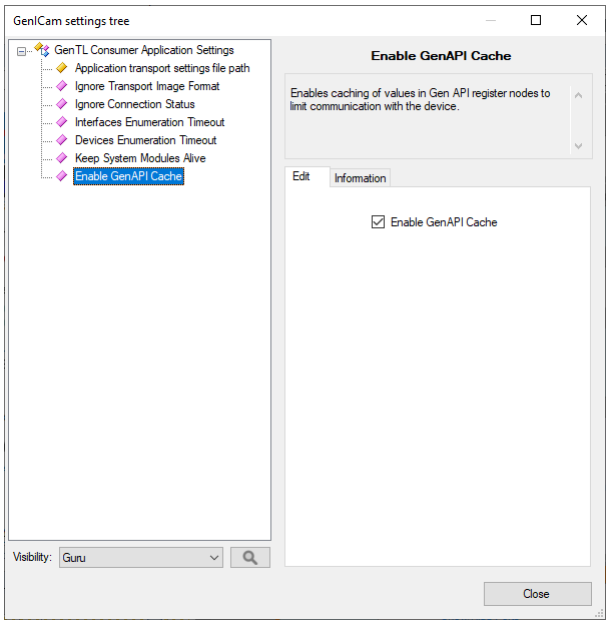
Imaging Source Cameras

There might be problems with image acquisition from Imaging Source cameras through GigE. It's caused by the implementation (regarding caching and packet size) of GigE Vision standard in those cameras and as a result no image can be seen in Aurora Vision Studio (the previews are empty during program execution).

To resolve this issue, a camera restart (this has to be done only once, after you encounter the problem with image acquisition) and changing some parameters in Aurora Vision GenAPI configuration are required. Parameters which should be changed are:

- Enable *GenAPI Cache* (should be set to *False*),
- Disable *Packet Size Negotiation* (should be set to *True*),
- Enable *Constant Packet Size* (should be set to *False*).

You can change these parameters in the GigEVision Application Settings Tree (*Tool » Manage GigE Vision Devices... » Tools » Application Transport Settings*). Make sure that each of the three parameters mentioned above is set to the proper value in this tree.



Flir Cameras

There might be problems with image acquisition from Flir cameras through GigE Vision. It is caused by the implementation (regarding image trailer data) of the GigE Vision standard in those cameras and as a result no image can be seen in Aurora Vision Studio (the previews are empty during program execution).

To resolve this issue a parameter change in Aurora Vision GenAPI configuration is required. Parameter which should be changed is:

- Enable *Ignore Gev Image Trailer data* (should be set to True).

You can change these parameters in the GigE Vision Application Settings Tree (Tool » Manage GigE Vision Devices... » Tools » Application Transport Settings). Remember to revert this changes before working with different devices as it can prevent some cameras from working properly in some modes.

10. Machine Vision Guide

Table of content:

Image Processing

Introduction

There are two major goals of Image Processing techniques:

1. To enhance an image for better human perception
2. To make the information it contains more salient or easier to extract

It should be kept in mind that in the context of computer vision only the second point is important. Preparing images for human perception is not part of computer vision; it is only part of information visualization. In typical machine vision applications this comes only at the end of the program and usually does not pose any problem.

The first and the most important advice for machine vision engineers is: **avoid image transformations designed for human perception when the goal is to extract information**. Most notable examples of transformations that are not only not interesting, but can even be highly disruptive, are:

- JPEG compression (creates artifacts not visible by human eye, but disruptive for algorithms)
- CIE Lab and CIE XYZ color spaces (specifically designed for human perception)
- Edge enhancement filters (which improve only the "apparent sharpness")
- Image thresholding performed before edge detection (precludes sub-pixel precision)

Examples of image processing operations that can really improve information extraction are:

- Gaussian image smoothing (removes noise, while preserving information about local features)
- Image morphology (can remove unwanted details)
- Gradient and high-pass filters (highlight information about object contours)
- Basic color space transformations like HSV (separate information about chromaticity and brightness)
- Pixel-by-pixel image composition (e.g. can highlight image differences in relation to a reference image)

Regions of Interest

The image processing tools provided by Aurora Vision have a special *inRoi* input (of Region type), that can limit the spatial scope of the operation. The region can be of any shape.

An input image and the inRoi.

Result of an operation performed within inRoi.

Remarks:

- The output image will be black outside of the *inRoi* region.
- To obtain an image that has its pixels modified in *inRoi* and copied outside of it, one can use the ComposeImages filter.
- The default value for *inRoi* is *Auto* and causes the entire image to be processed.
- Although *inRoi* can be used to significantly speed up processing, it should be used with care. The performance gain may be far from proportional to the *inRoi* area, especially in comparison to processing the entire image (*Auto*). This is due to the fact, that in many cases more SSE optimizations are possible when *inRoi* is not used.

Some filters have a second region of interest called *inSourceRoi*. While *inRoi* defines the range of pixels that will be written in the output image, the *inSourceRoi* parameter defines the range of pixels that can be read from the input image.

Image Boundary Processing

Some image processing filters, especially those from the Image Local Transforms category, use information from some local neighborhood of a pixel. This causes a problem near the image borders as not all input data is available. The policy applied in our tools is:

- Never assume any specific value outside of the image, unless specifically defined by the user.
- If only partial information is available, it is better not to detect anything, than detect something that does not exist.

In particular, the filters that use information from a local neighborhood just use smaller (cropped) neighbourhood near the image borders. This is something, however, that has to be taken into account, when relying on the results – for example results of the smoothing filters can be up to 2 times less smooth at the image borders (due to half of the neighborhood size), whereas results of the morphological filters may "stick" to the image borders. If the highest reliability is required, the general rule is: **use appropriate regions of interest to ignore image processing results that come from incomplete information** (near the image borders).

Toolset

Image Combinators

The filters from the Image Combinators category take two images and perform a pixel-by-pixel transformation into a single image. This can be used for example to highlight differences between images or to normalize brightness – as in the example below:

Input image with high reflections.

Image of the reflections (calibrating).

The result of applying DivideImages with inScale = 128 (inRoi was used).

Image Smoothing

The main purpose of the image smoothing filters (located in the Image Local Transforms category) is removal of noise. There are several different ways to perform this task with different trade-offs. On the example below three methods are presented:

1. Mean smoothing – simply takes the average pixel value from a rectangular neighborhood; it is the fastest method.
2. Median smoothing – simply takes the median pixel value from a rectangular neighborhood; preserves edges, but is relatively slow.
3. Gauss smoothing – computes a weighted average of the pixel values with Gaussian coefficients as the weights; its advantage is isotropy and reasonable speed for small kernels.

Input image with some noise.

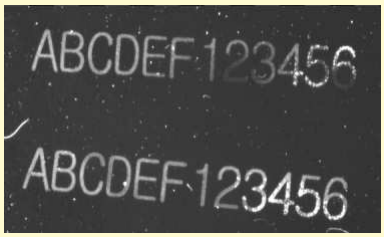
Result of applying SmoothImage_Mean.

Result of applying SmoothImage_Gauss.

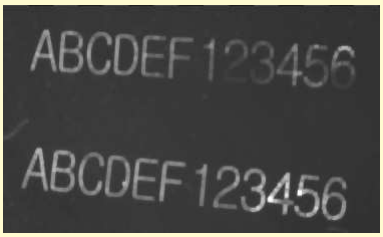
Result of applying SmoothImage_Median.

Image Morphology

Basic morphological operators – DilateImage and ErodeImage – transform the input image by choosing maximum or minimum pixel values from a local neighborhood. Other morphological operators combine these two basic operations to perform more complex tasks. Here is an example of using the OpenImage filter to remove salt and pepper noise from an image:



Input image with salt-and-pepper noise.



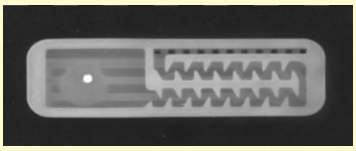
Result of applying OpenImage.

Gradient Analysis

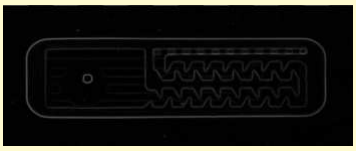
An image gradient is a vector describing direction and magnitude (strength) of local brightness changes. Gradients are used inside of many computer vision tools – for example in object contour detection, edge-based template matching and in barcode and DataMatrix detection.

Available filters:

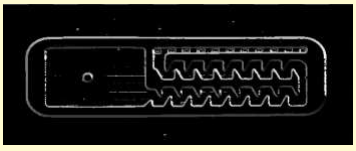
- GradientImage – produces a 2-channel image of signed values; each pixel denotes a gradient vector.
- GradientMagnitudeImage – produces a single channel image of gradient magnitudes, i.e. the lengths of the vectors (or their approximations).
- GradientDirAndPresenceImage – produces a single channel image of gradient directions mapped into the range from 1 to 255; 0 means no significant gradient.



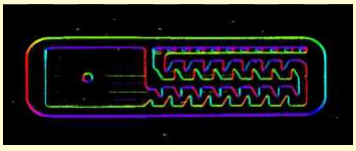
An input image.



Result of GradientMagnitudeImage.



Result of GradientDirAndPresenceImage.



Diagnostic output of GradientImage showing hue-coded directions.

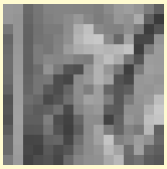
Spatial Transforms

Spatial transforms modify an image by changing locations, but not values, of pixels. Here are sample results of some of the most basic operations:

Result of MirrorImage.

Result of RotateImage.

Result of ShearImage.



Result of DownsampleImage.

Result of TransposeImage.

Result of TranslateImage.

Result of CropImage.

Result of UncropImage applied to the result of CropImage.

There are also interesting spatial transform tools that allow to transform a two dimensional vision problem into a 1.5-dimensional one, which can be very useful for further processing:

An input image and a path.

Result of ImageAlongPath.

Spatial Transform Maps

The spatial transform tools perform a task that consist of two steps for each pixel:

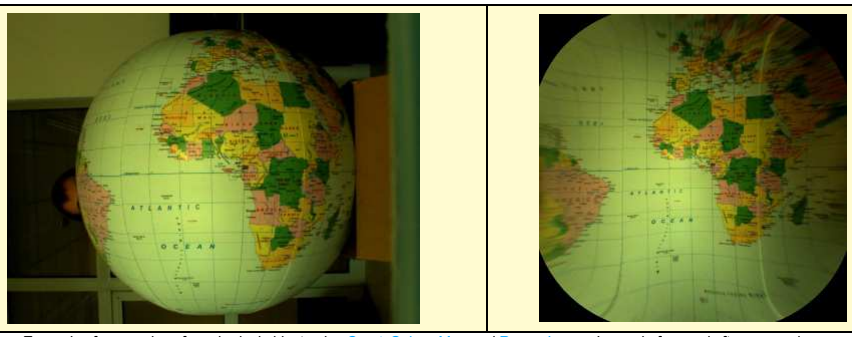
1. compute the destination coordinates (and some coefficients when interpolation is used),
2. copy the pixel value.

In many cases the transformation is constant – for example we might be rotating an image always by the same angle. In such cases the first step – computing the coordinates and coefficients – can be done once, before the main loop of the program. Aurora Vision provides the Image Spatial Transforms Maps category of filters for exactly that purpose. When you are able to compute the transform beforehand, storing it in the SpatialMap type, in the main loop only the RemapImage filter has to be executed. This approach will be much faster than using standard spatial transform tools.

The SpatialMap type is a map of image locations and their corresponding positions after given geometric transformation has been applied.

Additionally, the Image Spatial Transforms Maps category provides several filters that can be used to flatten the curvature of a physical object. They can be used for e.g. reading labels glued onto curved surfaces. These filters model basic 3D objects:

1. **Cylinder** (CreateCylinderMap) – e.g. flattening of a bottle label.
2. **Sphere** (CreateSphereMap) – e.g. reading a label from light bulb.
3. **Box** (CreatePerspectiveMap_Points or CreatePerspectiveMap_Path) – e.g. reading a label from a box.
4. **Circular objects (polar transform)** (CreateImagePolarTransformMap) - e.g. reading a label wrapped around a DVD disk center.



Example of remapping of a spherical object using CreateSphereMap and RemapImage. Image before and after remapping.

Furthermore custom spatial maps can be created with ConvertMatrixMapsToSpatialMap.


An example of custom image transform created with ConvertMatrixMapsToSpatialMap. Image before and after remapping.

Image Thresholding

The task of Image Thresholding filters is to classify image pixel values as foreground (white) or background (black). The basic filters ThresholdImage and ThresholdToRegion use just a simple range of pixel values – a pixel value is classified as foreground iff it belongs to the range. The ThresholdImage filter just transforms an image into another image, whereas the ThresholdToRegion filter creates a region corresponding to the foreground pixels. Other available filters allow more advanced classification:

- ThresholdImage_Dynamic and ThresholdToRegion_Dynamic use average local brightness to compensate global illumination variations.
- ThresholdImage_RGB and ThresholdToRegion_RGB select pixel values matching a range defined in the RGB (the standard) color space.
- ThresholdImage_HSx and ThresholdToRegion_HSx select pixel values matching a range defined in the HSx color space.
- ThresholdImage_Relative and ThresholdToRegion_Relative allow to use a different threshold value at each pixel location.

Input image with uneven light.

Result of ThresholdImage – the bars can not be recognized.

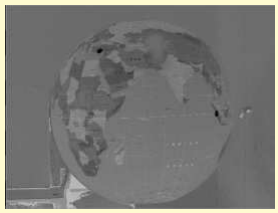
Result of ThresholdImage_Dynamic – the bars are correct.

There is also an additional filter SelectThresholdValue which implements a number of methods for automatic threshold value selection. It should, however, be used with much care, because there is no universal method that works in all cases and even a method that works well for a particular case might fail in special cases.

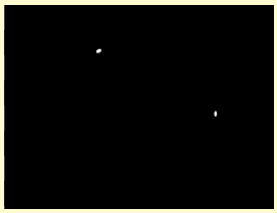
Image Pixel Analysis

When reliable object detection by color analysis is required, there are two filters that can be useful: ColorDistance and ColorDistanceImage. These filters compare colors in the RGB space, but internally separate analysis of brightness and chromaticity. This separation is very important, because in many cases variations in brightness are much higher than variations in chromaticity. Assigning more significance to the latter (high value of the *inChromaAmount* input) allows to detect areas having the specified color even in presence of highly uneven illumination:

Input image with uneven light.



Result of ColorDistanceImage for the red color with inChromaAmount = 1.0. Dark areas correspond to low color distance.



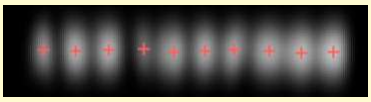
Result of thresholding reveals the location of the red dots on the globe.

Image Features

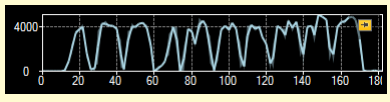
Image Features is a category of image processing tools that are already very close to computer vision – they transform pixel information into simple higher-level data structures. Most notable examples are: ImageLocalMaxima which finds the points at which the brightness is locally the highest, ImageProjection which creates a profile from sums of pixel values in columns or in rows, ImageAverage which averages pixel values in the entire region of interest. Here is an example application:

Input image with digits to be segmented.

Result of preprocessing with CloseImage.



Digit locations extracted by applying SmoothImage_Gauss and ImageLocalMaxima.

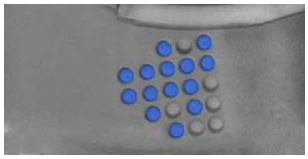


Profile of the vertical projection revealing regions of digits and the boundaries between them.

Blob Analysis

Introduction

Blob Analysis is a fundamental technique of machine vision based on analysis of consistent image regions. As such it is a tool of choice for applications in which the objects being inspected are clearly discernible from the background. Diverse set of Blob Analysis methods allows to create tailored solutions for a wide range of visual inspection problems.



Main advantages of this technique include high flexibility and excellent performance. Its limitations are: clear background-foreground relation requirement (see Template Matching for an alternative) and pixel-precision (see 1D Edge Detection for an alternative).

Concept

Let us begin by defining the notions of *region* and *blob*.

- *Region* is any subset of image pixels. In Aurora Vision Studio regions are represented using Region data type.
- *Blob* is a connected region. In Aurora Vision Studio blobs (being a special case of region) are represented using the same Region data type. They can be obtained from any region using a single SplitRegionIntoBlobs filter or (less frequently) directly from an image using image segmentation filters from category Image Analysis techniques.

		
An example image.	*Region of pixels darker than 128.*	*Decomposition of the region into array of blobs.*

The basic scenario of the Blob Analysis solution consists of the following steps:

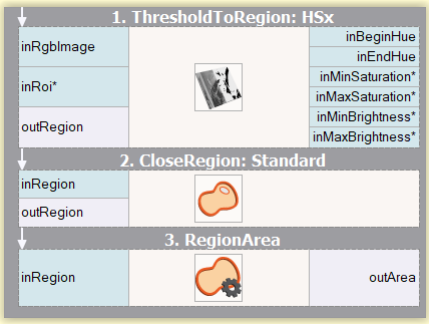
1. **Extraction** - in the initial step one of the Image Thresholding techniques is applied to obtain a region corresponding to the objects (or single object) being inspected.
2. **Refinement** - the extracted region is often flawed by noise of various kind (e.g. due to inconsistent lightning or poor image quality). In the Refinement step the region is enhanced using region transformation techniques.
3. **Analysis** - in the final step the refined region is subject to measurements and the final results are computed. If the region represents multiple objects, it is split into individual blobs each of which is inspected separately.

Examples

The following examples illustrate the general schema of Blob Analysis algorithms. Each of the techniques represented in the examples (thresholding, morphology, calculation of region features, etc.) is inspected in detail in later sections.

Rubber Band

In this, idealized, example we analyze a picture of an electronic device wrapped in a rubber band. The aim here is to compute the area of the visible part of the band (e.g. to decide whether it was assembled correctly).

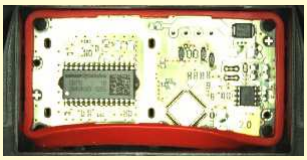


In this case each of the steps: Extraction, Refinement and Analysis is represented by a single filter.

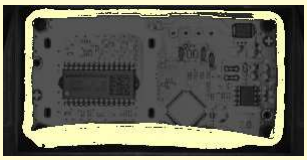
Extraction - to obtain a region corresponding to the red band a Color-based Thresholding technique is applied. The ThresholdToRegion_HSx filter is capable of finding the region of pixels of given color characteristics - in this case it is targeted to detect red pixels.

Refinement - the problem of filling the gaps in the extracted region is a standard one. Classic solutions for it are the region morphology techniques. Here, the CloseRegion filter is used to fill the gaps.

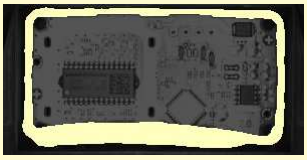
Analysis - finally, a single RegionArea filter is used to compute the area of the obtained region.



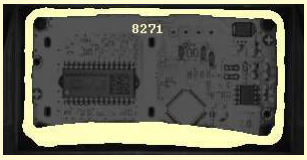
Initial image



Extraction



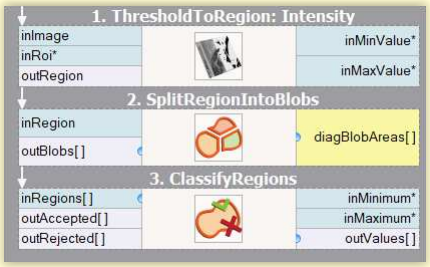
Refinement



Results

Mounts

In this example a picture of a set of mounts is inspected to identify the damaged ones.



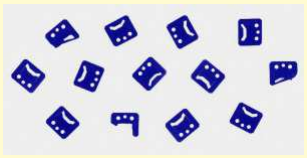
Extraction - as the lightning in the image is uniform, the objects are consistently dark and the background is consistently bright, the extraction of the region corresponding to the objects is a simple task. A basic ThresholdToRegion filter does the job, and does it so well that no **Refinement** phase is needed in this example.

Analysis - as we need to analyze each of the blobs separately, we start by applying the SplitRegionIntoBlobs filter to the extracted region.

To distinguish the bad parts from the correct parts we need to pick a property of a region (e.g. area, circularity, etc.) that we expect to be high for the good parts and low for the bad parts (or conversely). Here, the area would do, but we will pick a somewhat more sophisticated rectangularity feature, which will compute the similarity-to-rectangle factor for each of the blobs.

Once we have chosen the rectangularity feature of the blobs, all that needs to be done is to feed the regions to be classified to the ClassifyRegions filter (and to set its **inMinimum** value parameter). The blobs of too low rectangularity are available at the **outRejected** output of the classifying filter.

Input image



Extraction

Analysis

Results

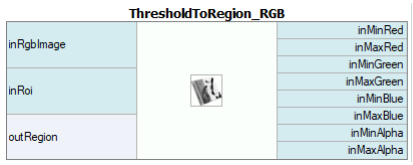
Extraction

There are two techniques that allow to extract regions from an image:

- **Image Thresholding** - commonly used methods that compute a region as a set of pixels that meet certain condition dependent on the specific operator (e.g. region of pixels brighter than given value, or brighter than the average brightness in their neighborhood). Note that the resulting data is always a single region, possibly representing numerous objects.
- **Image Segmentation** - more specialized set of methods that compute a set of blobs corresponding to areas in the image that meet certain condition. The resulting data is always an array of connected regions (blobs).

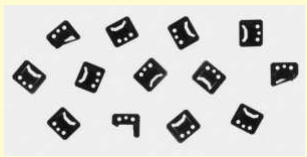
Thresholding

Image Thresholding techniques are preferred for common applications (even those in which a set of objects is inspected rather than a single object) because of their simplicity and excellent performance. In Aurora Vision Studio there are six filters for image-to-region thresholding, each of them implementing a different thresholding method.

Brightness-based (basic)		
Brightness-based (additional)		
Color-based		

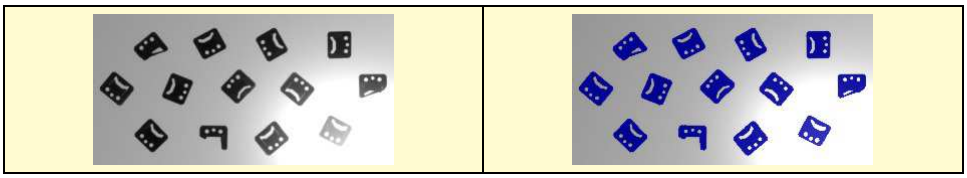
Classic Thresholding

ThresholdToRegion simply selects the image pixels of the specified brightness. It should be considered a basic tool and applied whenever the intensity of the inspected object is constant, consistent and clearly different from the intensity of the background.



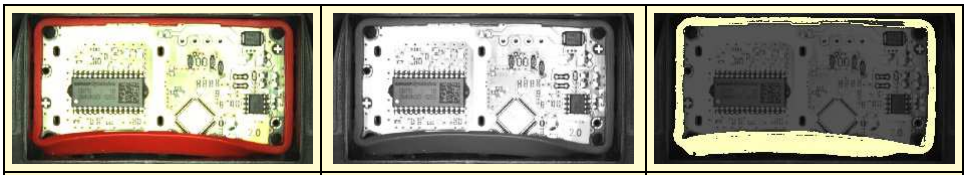
Dynamic Thresholding

Inconsistent brightness of the objects being inspected is a common problem usually caused by the imperfections of the lightning setup. As we can see in the example below, it is often the case that the objects in one part of the image actually have the same brightness as the background in another part of the image. In such case it is not possible to use the basic ThresholdToRegion filter and ThresholdToRegion_Dynamic should be considered instead. The latter selects image pixels that are *locally* bright/dark. Specifically - the filter selects the image pixels of the given *relative local brightness* defined as the difference between the pixel intensity and the average intensity in its neighborhood.



Color-based Thresholding

When inspection is conducted on color images it may be the case that despite a significant difference in color, the brightness of the objects is actually the same as the brightness of their neighborhood. In such case it is advisable to use Color-based Thresholding filters: ThresholdToRegion_RGB, ThresholdToRegion_HSx. The suffix denote the color space in which we define the desired pixel characteristic and not the space used in the image representation. In other words - both of these filters can be used to process standard RGB color image.

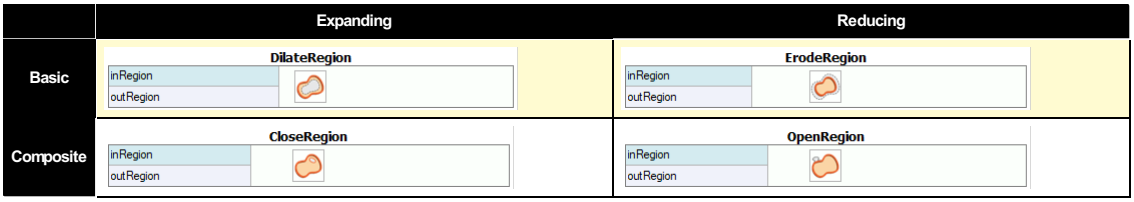


An example image.	*Mono equivalent of the image depicting brightness of its pixels.*	*Result of the color-based thresholding targeted at red pixels.*

Refinement

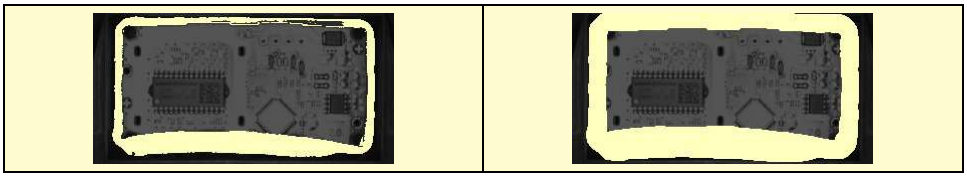
Region Morphology

Region Morphology is a classic technique of region transformation. The core concept of this toolset is the usage of a *structuring element* also known as the *kernel*. The kernel is a relatively small shape that is repeatedly centered at each pixel within dimensions of the region that is being transformed. Every such pixel is either added to the resulting region or not, depending on operation-specific condition on the minimum number of kernel pixels that have to overlap with actual input region pixels (in the given position of the kernel). See description of **Dilation** for an example.

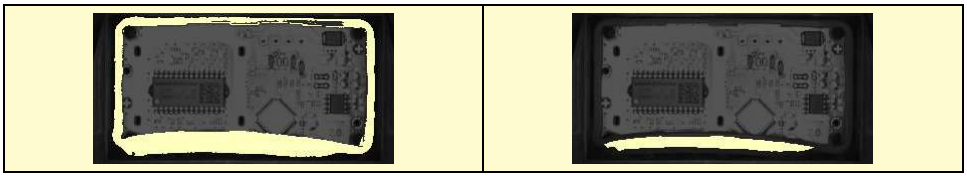


Dilation and Erosion

Dilation is one of two basic morphological transformations. Here each pixel **P** within the dimensions of the region being transformed is added to the resulting region if and only if the structuring element centered at **P** overlaps with *at least* one pixel that belongs to the input region. Note that for a circular kernel such transformation is equivalent to a uniform expansion of the region in every direction.

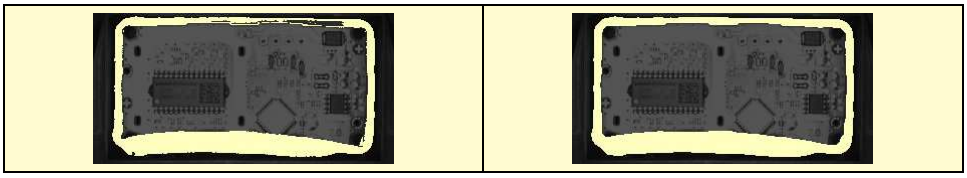


Erosion is a dual operation of **Dilation**. Here, each pixel **P** within the dimensions of the region being transformed is added to the resulting region if and only if the structuring element centered at **P** is *fully contained* in the region pixels. Note that for a circular kernel such transformation is equivalent to a uniform reduction of the region in every direction.

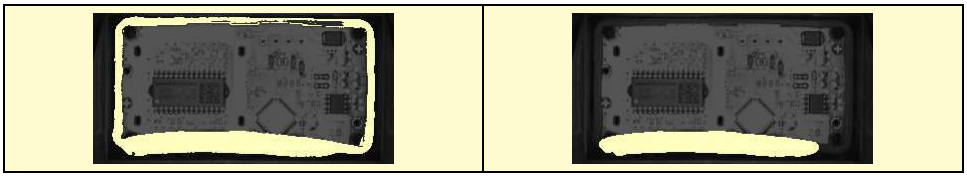


Closing and Opening

The actual power of the **Region Morphology** lies in its *composite operators* - **Closing** and **Opening**. As we may have recently noticed, during the blind region expansion performed by the **Dilation** operator, the gaps in the transformed region are filled in. Unfortunately, the expanded region no longer corresponds to the objects being inspected. However, we can apply the **Erosion** operator to bring the expanded region back to its original boundaries. The key point is that the gaps that were completely filled during the dilation will stay filled after the erosion. The operation of applying **Erosion** to the result of **Dilation** of the region is called **Closing**, and is a tool of choice for the task of filling the gaps in the extracted region.



Opening is a dual operation of **Closing**. Here, the region being transformed is initially eroded and then dilated. The resulting region preserves the form of the initial region, with the exception of thin/small parts, that are removed during the process. Therefore, **Opening** is a tool for removing the thin/outlying parts from a region. We may note that in the example below, the **Opening** does the - otherwise relatively complicated - job of finding the segment of the rubber band of excessive width.



Other Refinement Methods

Analysis

Once we obtain the region that corresponds to the object or the objects being inspected, we may commence the analysis - that is, extract the information we are interested in.

Region Features

Aurora Vision Studio allows to compute a wide range of numeric (e.g. area) and non-numeric (e.g. bounding circle) region features. Calculation of the measures describing the obtained region is often the very aim of applying the blob analysis in the first place. If we are to check whether the rectangular packaging box is deformed or not, we may be interested in calculating the *rectangularity factor* of the packaging region. If we are to check if the chocolate coating on a biscuit is broad enough, we may want to know the *area* of the coating region.

It is important to remember, that when the obtained region corresponds to multiple image objects (and we want to inspect each of them separately), we should apply the SplitRegionIntoBlobs filter before performing the calculation of features.

Numeric Features

Each of the following filters computes a number that expresses a specific property of the region shape.

Annotations in brackets indicate the range of the resulting values.

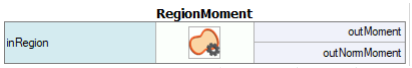
Size of the region (0 - ∞)

Similarity to a circle (0.0 - 1.0)

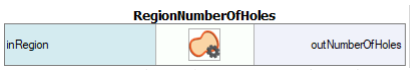
Similarity to own convex hull (0.0 - 1.0)

Similarity to a rectangle (0.0 - 1.0)

Similarity to a line (0.0 - ∞)

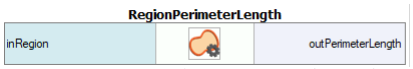


Moments of the region (0.0 - ∞)



Count of the region holes (0 - ∞)

Orientation of the main region axis (0.0 - 180.0)



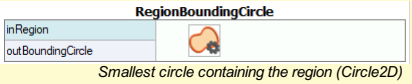
Length of the region contour (0.0 - ∞)

Non-numeric Features

Each of the following filters computes an object related to the shape of the region. Note that the primitives extracted using these filters can be made subject of further analysis. For instance, we can extract the holes of the region using the RegionHoles filter and then measure their areas using the RegionArea filter.

Annotations in brackets indicate Aurora Vision Studio's type of the result.

Smallest axis-aligned rectangle containing the region (Box)



Smallest circle containing the region (Circle2D)

Smallest any-orientation rectangle containing the region (Rectangle2D)

Boundaries of the region (PathArray)

Longest segment connecting two points inside the region (Segment2D)

Array of blobs representing gaps in the region (RegionArray)

Skeleton of the region (PathArray)

Case Studies

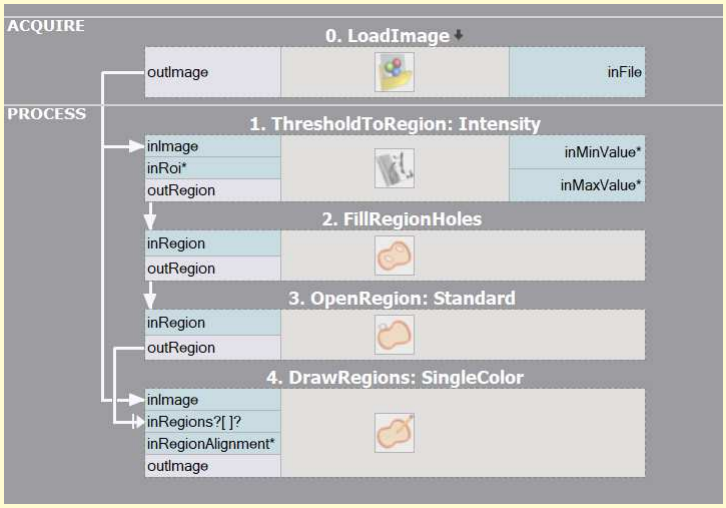
Capsules

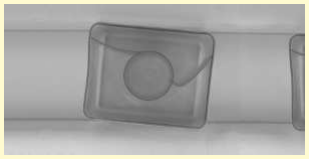
In this example we inspect a set of washing machine capsules on a conveyor line. Our aim is to identify the deformed capsules.

We will proceed in two steps: we will commence by designing a simple program that, given picture of the conveyor line, will be able to identify the region corresponding to the capsule(s) in the picture. In the second step we will use this program as a building block of the complete solution.

FindRegion Routine

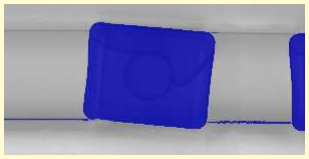
In this section we will develop a program that will be responsible for the **Extraction** and **Refinement** phases of the final solution. For brevity of presentation in this part we will limit the input image to its initial segment.



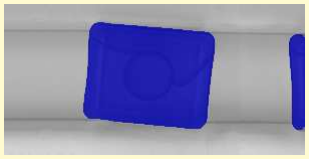


Initial image

ThresholdToRegion



FillRegionHoles



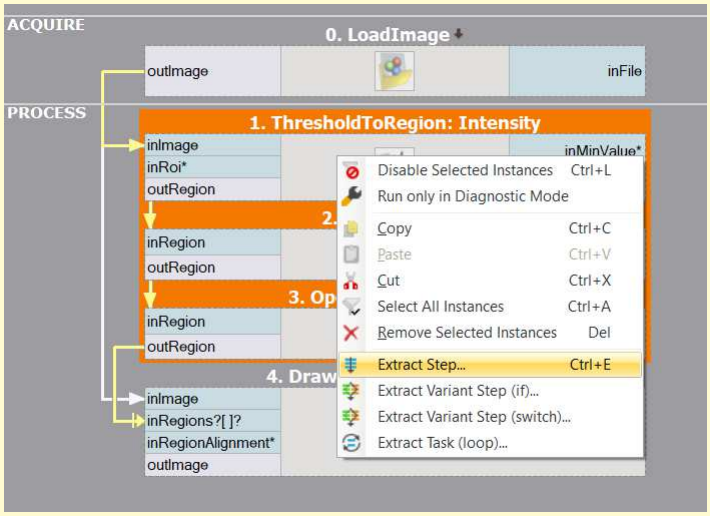
OpenRegion

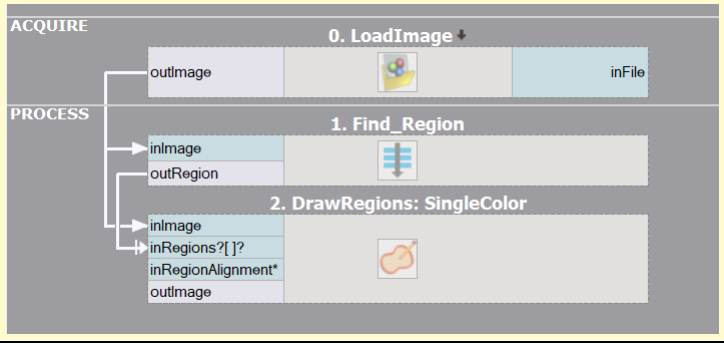
After a brief inspection of the input image we may note that the task at hand will not be trivial - the average brightness of the capsule body is similar to the intensity of the background. On the other hand the border of the capsule is consistently darker than the background. As it is the border of the object that bears significant information about its shape we may use the basic ThresholdToRegion filter to extract the darkest pixels of the image with the intention of filling the extracted capsule border during further refinement.

The extracted region certainly requires such refinement - actually, there are two issues that need to be addressed. We need to fill the shape of the capsule and eliminate the thin horizontal stripes corresponding to the elements of the conveyor line setup. Fortunately, there are fairly straightforward solutions for both of these problems.

FillRegionHoles will extend the region to include all pixels enclosed by present region pixels. After the region is filled all that remains is the removal of the thin conveyor lines using the classic OpenRegion filter.

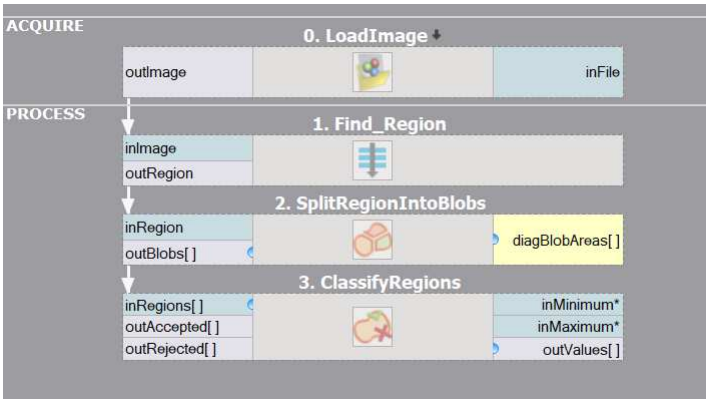
Our routine for **Extraction** and **Refinement** of the region is ready. As it constitutes a continuous block of filters performing a well defined task, it is advisable to encapsulate the routine inside a macrofilter to enhance the readability of the soon-to-be-growing program.





Complete Solution

Our program right now is capable of extracting the region that directly corresponds to the capsules visible in the image. What remains is to inspect each capsule and classify it as a correct or deformed one.

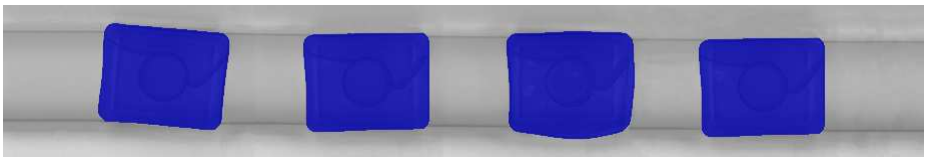


As we want to analyze each capsule separately, we should start with decomposition of the extracted region into an array of connected components (blobs). This common operation can be performed using the straightforward SplitRegionIntoBlobs filter.

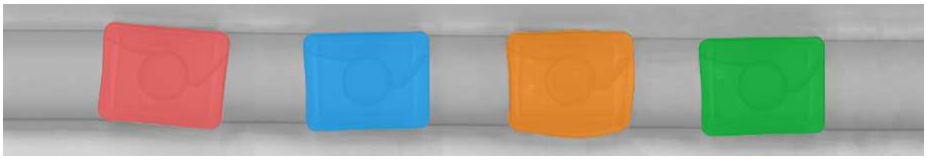
We are approaching the crucial part of our solution - how are we going to distinguish correct capsules from deformed ones? At this stage it is advisable to have a look at the summary of numeric region features provided in **Analysis** section. If we could find a numeric region property that is correlated with the nature of the problem at hand (e.g. it takes low values for a correct capsules and high values for a deformed one, or conversely), we would be nearly done.

Rectangularity of a shape is defined as the ratio between its area and area of its smallest enclosing rectangle - the higher the value, the more the shape of the object resembles a rectangle. As the shape of a correct capsule is almost rectangular (it is a rectangle with rounded corners) and clearly *more* rectangular than the shape of deformed capsule, we may consider using rectangularity feature to classify the capsules.

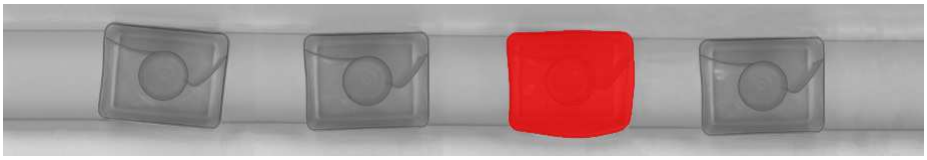
Having selected the numeric feature that will be used for the classification, we are ready to add the ClassifyRegions filter to our program and feed it with data. We pass the array of capsule blobs on its **inRegions** input and we select Rectangularity on the **inFeature** input. After brief interactive experimentation with the *inMinimum* threshold we may observe that setting the minimum rectangularity to 0.95 allows proper discrimination of correct (available at **outAccepted**) and deformed (**outRejected**) capsule blobs.



Region extracted by the FindRegion routine.



Decomposition of the region into individual blobs.

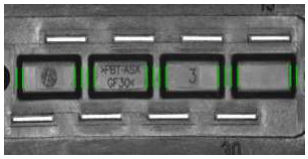


Blobs of low rectangularity selected by ClassifyRegions filter.

1D Edge Detection

Introduction

1D Edge Detection (also called *1D Measurement*) is a classic technique of machine vision where the information about image is extracted from one-dimensional profiles of image brightness. As we will see, it can be used for measurements as well as for positioning of the inspected objects.



Main advantages of this technique include sub-pixel precision and high performance.

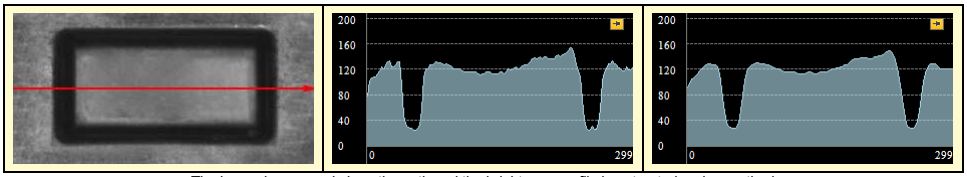
Concept

The 1D Edge Detection technique is based on an observation that any edge in the image corresponds to a rapid brightness change in the direction perpendicular to that edge. Therefore, to detect the image edges we can scan the image along a path and look for the places of significant change of intensity in the extracted brightness profile.

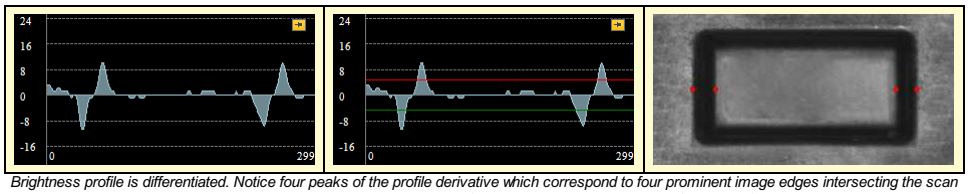
The computation proceeds in the following steps:

1. **Profile extraction** – firstly the profile of brightness along the given path is extracted. Usually the profile is smoothed to remove the noise.
2. **Edge extraction** – the points of significant change of profile brightness are identified as *edge points* – points where perpendicular edges intersect the scan line.
3. **Post-processing** – the final results are computed using one of the available methods. For instance ScanSingleEdge filter will select and return the strongest of the extracted edges, while ScanMultipleEdges filter will return all of them.

Example



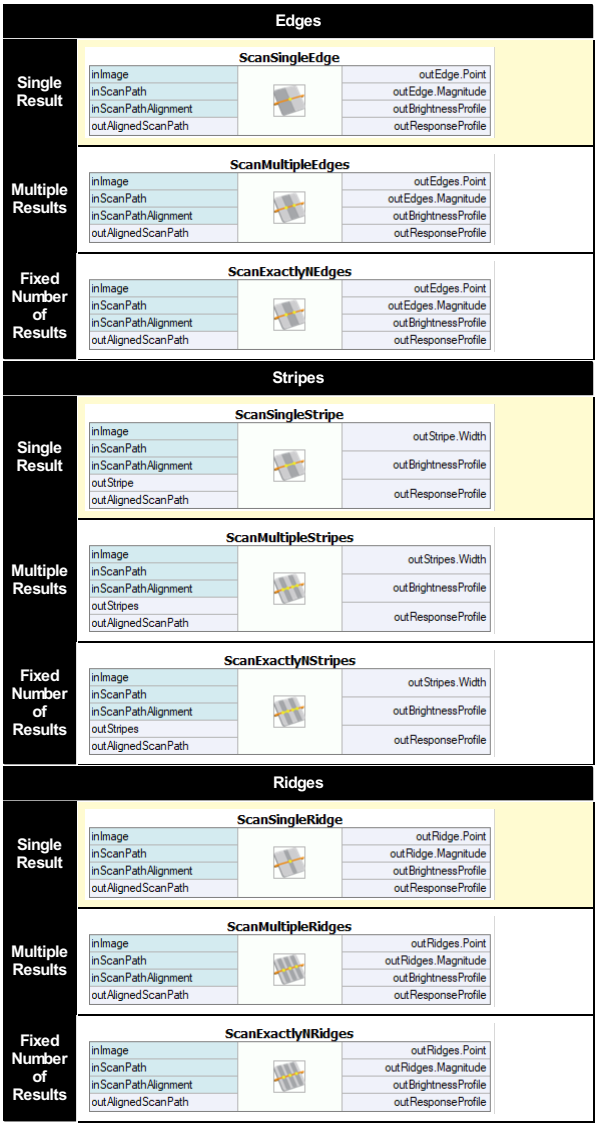
The image is scanned along the path and the brightness profile is extracted and smoothed.



Brightness profile is differentiated. Notice four peaks of the profile derivative which correspond to four prominent image edges intersecting the scan line. Finally the peaks stronger than some selected value (here minimal strength is set to 5) are identified as edge points.

Filter Toolset

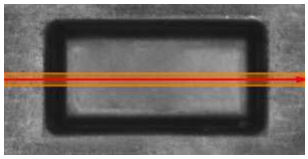
Basic toolset for the 1D Edge Detection-based techniques scanning for edges consists of 9 filters each of which runs a single scan along the given path (**inScanPath**). The filters differ on the structure of interest (edges / ridges / stripes (edge pairs)) and its cardinality (one / any fixed number / unknown number).



Note that in Aurora Vision Library there is the **CreateScanMap** function that has to be used before a usage of any other 1D Edge Detection function. The special function creates a scan map, which is passed as an input to other functions considerably speeding up the computations.

Parameters

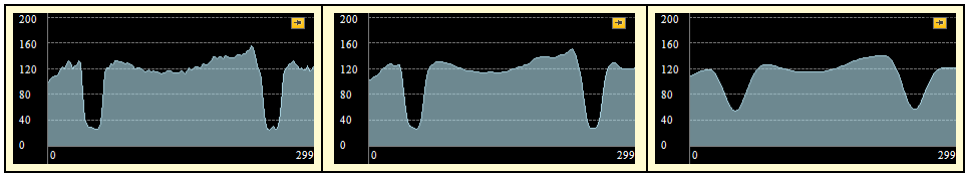
Profile Extraction



In each of the nine filters the brightness profile is extracted in exactly the same way. The stripe of pixels along **inScanPath** of width **inScanWidth** is traversed and the pixel values across the path are accumulated to form one-dimensional profile. In the picture on the right the stripe of processed pixels is marked in orange, while **inScanPath** is marked in red.

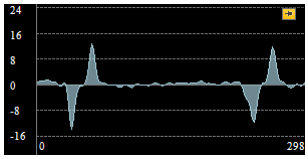
The extracted profile is smoothed using Gaussian smoothing with standard deviation of **inSmoothingStdDev**. This parameter is important for the robustness of the computation - we should pick the value that is high enough to eliminate noise that could introduce false / irrelevant extrema to the profile derivative, but low enough to preserve the actual edges we are to detect.

The **inSmoothingStdDev** parameter should be adjusted through interactive experimentation using **outBrightnessProfile** output, as demonstrated below.



*Too low **inSmoothingStdDev** - too much noise*	*Appropriate **inSmoothingStdDev** - low noise, significant edges are preserved*	*Too high **inSmoothingStdDev** - significant edges are attenuated*

Edge Extraction

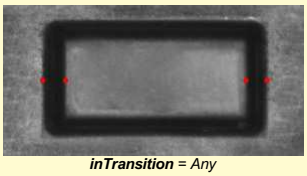


After the brightness profile is extracted and refined, the derivative of the profile is computed and its local extrema of magnitude at least **inMinMagnitude** are identified as edge points. The **inMinMagnitude** parameter should be adjusted using the **outResponseProfile** output.

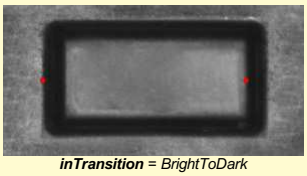
The picture on the right depicts an example **outResponseProfile** profile. In this case the significant extrema vary in magnitude from 11 to 13, while the magnitude of other extrema is lower than 3. Therefore any **inMinMagnitude** value in range (4, 10) would be appropriate.

Edge Transition

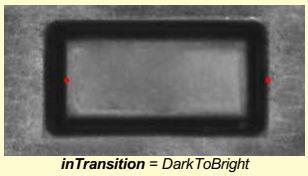
Filters being discussed are capable of filtering the edges depending on the kind of transition they represent - that is, depending on whether the intensity changes from bright to dark, or from dark to bright. The filters detecting individual edges apply the same condition defined using the **inTransition** parameter to each edge (possible choices are *bright-to-dark*, *dark-to-bright* and *any*).



inTransition = *Any*



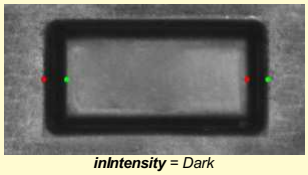
inTransition = *BrightToDark*



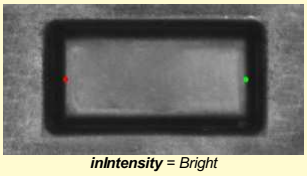
inTransition = *DarkToBright*

Stripe Intensity

The filters detecting stripes expect the edges to alternate in their characteristics. The parameter **inIntensity** defines whether each stripe should bound the area that is brighter, or darker than the surrounding space.



inIntensity = *Dark*

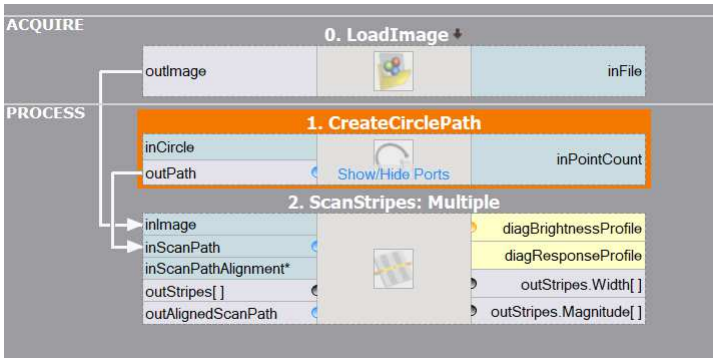


inIntensity = *Bright*

Case Study: Blades

Assume we want to count the blades of a circular saw from the picture.

We will solve this problem running a single 1D Edge Detection scan along a circular path intersecting the blades, and therefore we need to produce appropriate circular path. For that we will use a straightforward CreateCirclePath filter. The built-in editor will allow us to point & click the required **inCircle** parameter.



The next step will be to pick a suitable measuring filter. Because the path will alternate between dark blades and white background, we will use a filter that is capable of measuring stripes. As we do not now how many blades there are on the image (that is what we need to compute), the ScanMultipleStripes filter will be a perfect choice.

We expect the measuring filter to identify each blade as a single stripe (or each space between blades, depending on our selection of **inIntensity**), therefore all we need to do to compute the number of blades is to read the value of the **outStripes.Count** property output of the measuring filter.

The program solves the problem as expected (perhaps after increasing the **inSmoothingStdDev** from default of 0.6 to bigger value of 1.0 or 2.0) and detects all 30 blades of the saw.

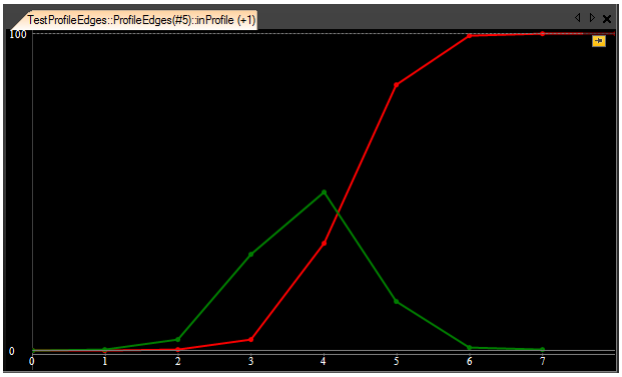
1D Edge Detection – Subpixel Precision

Introduction

One of the key strengths of the 1D Edge Detection tools is their ability do detect edges with precision higher than the pixel grid. This is possible, because the values of the derivative profile (of pixel values) can be interpolated and its maxima can be found analytically.

Example: Parabola Fitting

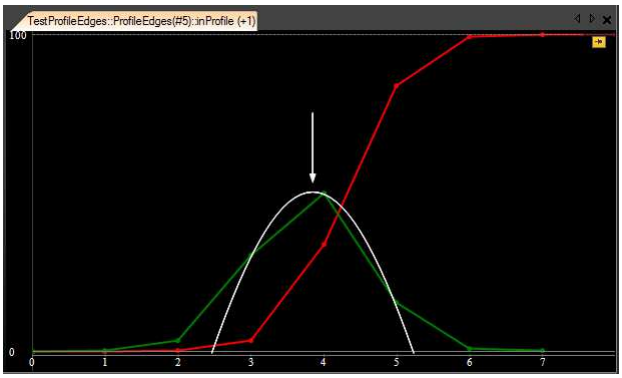
Let us consider a sample profile of pixel values corresponding to an edge (red):



Sample edge profile (red) and its derivative (green). Please note, that the derivative is shifted by 0.5.

The steepest segment is between points 4.0 and 5.0, which corresponds to the maximum of the derivative (green) at 4.5. Without the subpixel precision the edge would be found at this point.

It is, however, possible to consider information about the values of the neighbouring profile points to extract the edge location with higher precision. The simplest method is to fit a parabola to three consecutive points of the derivative profile:



Fitting a parabola to three consecutive points.

Now, the edge point we are looking for can be taken from the maximum of the parabola. In this case it will be 4.363, which is already a subpixel-precise result. This precision is still not very high, however. We know it from an experiment – this particular profile, which we are considering in this example, has been created from a perfect gaussian edge located at the point 430 and downsampled 100 times to simulate a camera looking at an edge at the point 4.3. The error that we got, is 0.063 px. From other experiments we know that in the worst case it can be up to 1/6 px.

Advanced: Methods Available in Aurora Vision

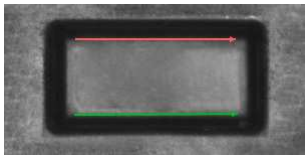
More advanced methods can be used that consider not three, but four consecutive points and which employ additional techniques to assure the highest precision in presence of noise and other practical edge distortions. In Aurora Vision Studio they are available in a form of 3 different profile interpolation methods:

- *Linear* – the simplest method that results in pixel-precise results,
- *Quadratic3* – an improved fitting of parabola to 3 consecutive points,
- *Quadratic4* – an advanced method that fits parabola to 4 consecutive points.

The precision of these methods on perfect gaussian edges is respectively: 1/2 px, 1/6 px and 1/23 px. It has to be added, however, that the *Quadratic4* method differs significantly in its performance on edges which are only slightly blurred – when the image quality is close to perfect, the precision can be even higher than 1/50 px.

Shape Fitting

Introduction



Shape Fitting is a machine vision technique that allows for precise detection of objects whose shapes and rough positions are known in advance. It is most often used in measurement applications for establishing line segments, circles, arcs and paths defining the shape that is to be measured.

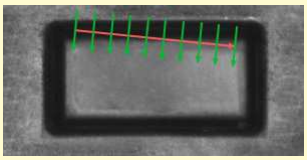
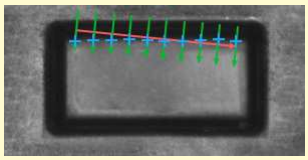
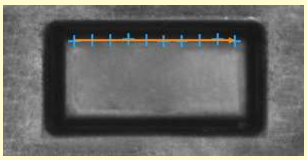
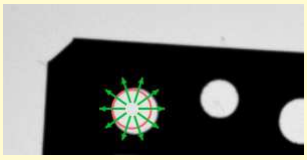
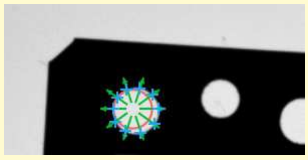
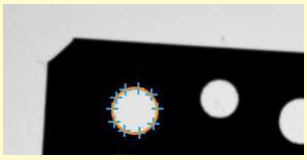
As this technique is derived from 1D Edge Detection, its key advantages are similar – including sub-pixel precision and high performance.

Concept

The main idea standing behind Shape Fitting is that a continuous object (such as a circle, an arc or a segment) can be determined using a finite set of points belonging to it. These points are computed by means of appropriate 1D Edge Detection filters and are then combined together into a single higher-level result.

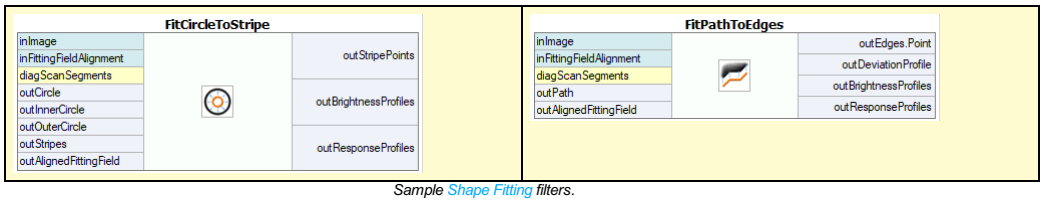
Thus, a single Shape Fitting filter's work consists of the following steps:

1. **Scan segments preparation** – a series of segments is prepared. The number, length and orientations of the segments are computed from the filter's parameters.
2. **Points extraction** – points that should belong to the object being fitted are extracted using (internally) a proper 1D Edge Detection filter (e.g. ScanSingleEdge in FitCircleToEdges) along each of the scan segments as the scan path.
3. **Object fitting** – the final result is computed with the use of a technique that allows fitting an object to a set of points. In this step, a filter from Geometry 2D Fitting is internally used (e.g. FitCircleToPoints in FitCircleToEdges). An exception to the rule is path fitting. No Geometry 2D Fitting filter is needed there, because the found points serve themselves as the output path characteristic points.

		
*The scan segments are created according to the fitting field and other parameters (e.g. **inScanCount**).*	*ScanSingleEdge (or another proper 1D Edge Detection filter) is performed.*	*A segment is fitted to the obtained points.*
		
*The scan segments are created according to the fitting field and other parameters (e.g. **inScanCount**).*	*ScanSingleEdge (or another proper 1D Edge Detection filter) is performed.*	*A segment is fitted to the obtained points.*

Toolset

The whole toolset for Shape Fitting consists of several filters. The filters differ on the object being fitted (a circle, an arc, a line segment or a path) and the 1D Edge Detection structures extracted along the scan segments (edges, ridges or stripe, all of them clearly discernible on the input image).

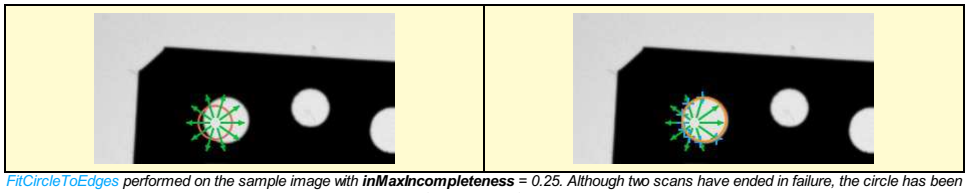


Sample Shape Fitting filters.

Parameters

Because of the internal use of 1D Edge Detection filters and Geometry 2D Fitting filters, all parameters known from them are also present in Shape Fitting filters interfaces.

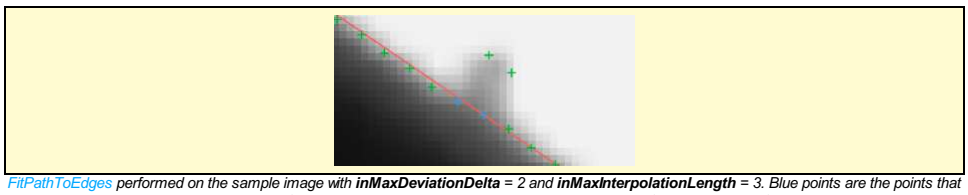
Beside these, there are also a few parameters specific to the subject of shape fitting. The **inScanCount** parameter controls the number of the scan segments. However, not all of the scans have to succeed in order to regard the whole fitting process as being successful. The **inMaxIncompleteness** parameter determines what fraction of the scans may fail.



*FitCircleToEdges performed on the sample image with **inMaxIncompleteness** = 0.25. Although two scans have ended in failure, the circle has been fitted successfully.*

The path fitting functions have some additional parameters, which help to control the output path shape. These parameters are:

- **inMaxDeviationDelta** – it defines the maximal allowed difference between deviations of consecutive points of the output path from the corresponding input path points; if the difference between deviations is greater, the point is considered to be not found at all.
- **inMaxInterpolationLength** – if some of the scans fail or if some of found points are classified to be wrong according to another control parameters (e.g. **inMaxDeviationDelta**), output path points corresponding to them are interpolated depending on points in their nearest vicinity. No more than **inMaxInterpolationLength** consecutive points can be interpolated, and if there exists a longer series of points that would have to be interpolated, the fitting is considered to be unsuccessful. The exception to this behavior are points which were not found on both ends of the input path. Those are not part of the result at all.



*FitPathToEdges performed on the sample image with **inMaxDeviationDelta** = 2 and **inMaxInterpolationLength** = 3. Blue points are the points that were interpolated. If **inMaxInterpolationLength** value was less than 2, the fitting would have failed.*

Template Matching

Introduction

Template Matching is a high-level machine vision technique that identifies the parts on an image that match a predefined template. Advanced template matching algorithms allow to find occurrences of the template regardless of their orientation and local brightness.

Template Matching techniques are flexible and relatively straightforward to use, which makes them one of the most popular methods of object localization. Their applicability is limited mostly by the available computational power, as identification of big and complex templates can be time-consuming.

Concept

Template Matching techniques are expected to address the following need: provided a reference image of an object (the *template image*) and an image to be inspected (the *input image*) we want to identify all *input image* locations at which the object from the *template image* is present. Depending on the specific problem at hand, we may (or may not) want to identify the rotated or scaled occurrences.

We will start with a demonstration of a naive Template Matching method, which is insufficient for real-life applications, but illustrates the core concept from which the actual Template Matching algorithms stem from. After that we will explain how this method is enhanced and extended in advanced **Grayscale-based Matching** and **Edge-based Matching** routines.

Naive Template Matching

Imagine that we are going to inspect an image of a plug and our goal is to find its pins. We are provided with a *template image* representing the reference object we are looking for and the *input image* to be inspected.

	
Template image	*Input image*

We will perform the actual search in a rather straightforward way – we will position the *template* over the image at every possible location, and each time we will compute some numeric measure of similarity between the template and the image segment it currently overlaps with. Finally we will identify the positions that yield the best similarity measures as the probable template occurrences.

Image Correlation

One of the subproblems that occur in the specification above is calculating the *similarity measure* of the aligned template image and the overlapped segment of the input image, which is equivalent to calculating a similarity measure of two images of equal dimensions. This is a classical task, and a numeric measure of image similarity is usually called *image correlation*.

Cross-Correlation

The fundamental method of calculating the image correlation is so called *cross-correlation*, which essentially is a simple sum of pairwise multiplications of corresponding pixel values of the images.

Though we may notice that the correlation value indeed seems to reflect the similarity of the images being compared, cross-correlation method is far from being robust. Its main drawback is that it is biased by changes in global brightness of the images - brightening of an image may sky-rocket its cross-correlation with another image, even if the second image is not at all similar.

Image1	Image2	Cross-Correlation
		19404780
		23316890
		24715810

$$\text{Cross-Correlation}(\text{Image1}, \text{Image2}) = \sum_{x,y} \text{Image1}(x,y) \times \text{Image2}(x,y)$$

Normalized Cross-Correlation

Normalized cross-correlation is an enhanced version of the classic *cross-correlation* method that introduces two improvements over the original one:

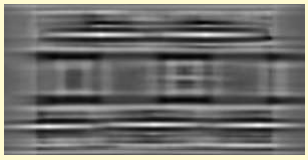
- The results are invariant to the global brightness changes, i.e. consistent brightening or darkening of either image has no effect on the result (this is accomplished by subtracting the mean image brightness from each pixel value).
- The final correlation value is scaled to [-1, 1] range, so that NCC of two identical images equals 1.0, while NCC of an image and its negation equals -1.0.

Image1	Image2	NCC
		-0.417
		0.553
		0.844

$$\text{NCC}(\text{Image1}, \text{Image2}) = \frac{1}{N\sigma_1\sigma_2} \sum_{x,y} (\text{Image1}(x,y) - \overline{\text{Image1}}) \times (\text{Image2}(x,y) - \overline{\text{Image2}})$$

Template Correlation Image

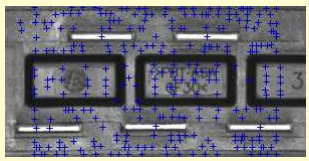
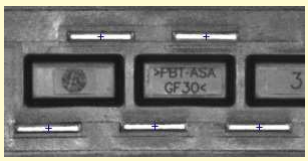
Let us get back to the problem at hand. Having introduced the Normalized Cross-Correlation - robust measure of image similarity - we are now able to determine how well the template fits in each of the possible positions. We may represent the results in a form of an image, where brightness of each pixels represents the NCC value of the template positioned over this pixel (black color representing the minimal correlation of -1.0, white color representing the maximal correlation of 1.0).

		
Template image	*Input image*	*Template correlation image*

Identification of Matches

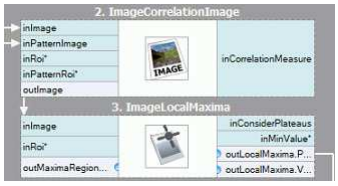
All that needs to be done at this point is to decide which points of the *template correlation image* are good enough to be considered actual matches. Usually we identify as matches the positions that (simultaneously) represent the template correlation:

- stronger that some predefined threshold value (i.e stronger that 0.5)
- locally maximal (stronger that the template correlation in the neighboring pixels)

		
Areas of template correlation above 0.75	*Points of locally maximal template correlation*	*Points of locally maximal template correlation above 0.75*

Summary

It is quite easy to express the described method in Aurora Vision Studio - we will need just two built-in filters. We will compute the template correlation image using the ImageCorrelationImage filter, and then identify the matches using ImageLocalMaxima - we just need to set the **inMinValue** parameter that will cut-off the weak local maxima from the results, as discussed in previous section.



Though the introduced technique was sufficient to solve the problem being considered, we may notice its important drawbacks:

- Template occurrences have to preserve the orientation of the reference template image.
- The method is inefficient, as calculating the template correlation image for medium to large images is time consuming.

In the next sections we will discuss how these issues are being addressed in advanced template matching techniques: **Grayscale-based Matching** and **Edge-based Matching**.

Grayscale-based Matching, Edge-based Matching

Grayscale-based Matching is an advanced Template Matching algorithm that extends the original idea of correlation-based template detection enhancing its efficiency and allowing to search for template occurrences regardless of its orientation. **Edge-based Matching** enhances this method even more by limiting the computation to the object edge-areas.

In this section we will describe the intrinsic details of both algorithms. In the next section (**Filter toolset**) we will explain how to use these techniques in Aurora Vision Studio.

Image Pyramid

Image Pyramid is a series of images, each image being a result of downsampling (scaling down, by the factor of two in this case) of the previous element.

		
Level 0 (input image)	*Level 1*	*Level 2*

Pyramid Processing

Image pyramids can be applied to enhance the efficiency of the correlation-based template detection. The important observation is that the template depicted in the reference image usually is still discernible after significant downsampling of the image (though, naturally, fine details are lost in the process). Therefore we can identify match candidates in the downsampled (and therefore much faster to process) image on the highest level of our pyramid, and then repeat the search on the lower levels of the pyramid, each time considering only the template positions that scored high on the previous level.

At each level of the pyramid we will need appropriately downsampled picture of the reference template, i.e. both input image pyramid and template image pyramid should be computed.

Level 0 (template reference image)	*Level 1*	*Level 2*

Grayscale-based Matching

Although in some of the applications the orientation of the objects is uniform and fixed (as we have seen in the plug example), it is often the case that the objects that are to be detected appear rotated. In Template Matching algorithms the classic pyramid search is adapted to allow *multi-angle* matching, i.e. identification of rotated instances of the template.

This is achieved by computing not just one *template image* pyramid, but a set of pyramids - one for each possible rotation of the template. During the pyramid search on the input image the algorithm identifies the pairs *(template position, template orientation)* rather than sole template positions. Similarly to the original schema, on each level of the search the algorithm verifies only those *(position, orientation)* pairs that scored well on the previous level (i.e. seemed to match the template in the image of lower resolution).

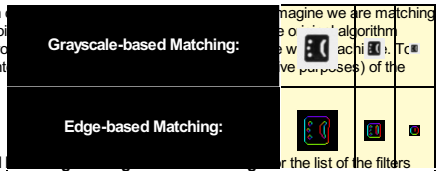
		
Template image	*Input image*	*Results of multi-angle matching*

The technique of pyramid matching together with *multi-angle* search constitute the **Grayscale-based Template Matching** method.

Edge-based Matching

Edge-based Matching enhances the previously discussed Grayscale-based Matching using one crucial observation - that the shape of any object is defined mainly by the shape of its edges. Therefore, instead of matching of the whole template, we could extract its edges and match only the nearby pixels, thus avoiding some unnecessary computations. In common applications the achieved speed-up is usually significant.

Matching object edges instead of an object as a whole requires slight modification [illegible] imagine we are matching an object of uniform color positioned over uniform background. All of object edge pi[illegible] algorithm would match the object anywhere wherever there is large enough blob of the appro[illegible] achi[illegible]. To resolve this problem, in Edge-based Matching it is the *gradient direction* (represent[illegible]ve purposes) of the edge pixels, not their intensity, that is matched.



Different kinds of template pyramids used in Template Matching algorithms.

Filter Toolset

Aurora Vision Studio provides a set of filters implementing both **Grayscale-based** [illegible]r the list of the filters see Template Matching filters.

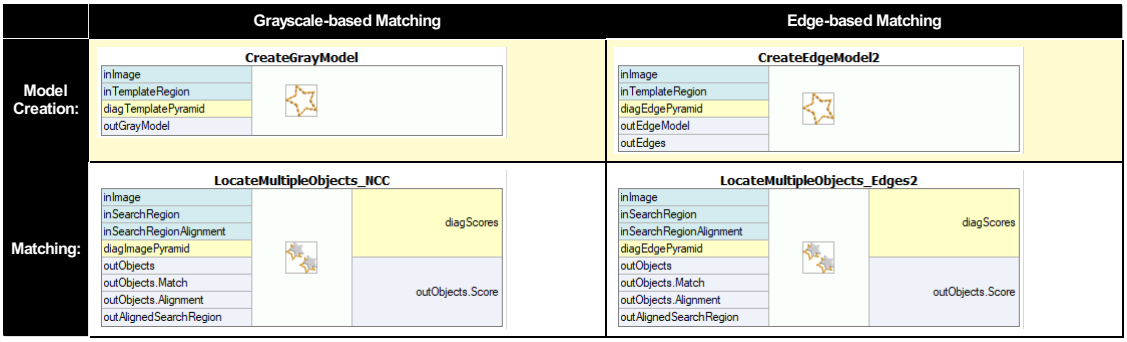
As the template image has to be preprocessed before the pyramid matching (we need to calculate the template image pyramids for all possible rotations and scales), the algorithms are split into two parts:

- **Model Creation** - in this step the *template image* pyramids are calculated and the results are stored in a *model* - atomic object representing all the data needed to run the pyramid matching.
- **Matching** - in this step the *template model* is used to match the template in the *input image*.

Such an organization of the processing makes it possible to compute the *model* once and reuse it multiple times.

Available Filters

For both Template Matching methods two filters are provided, one for each step of the algorithm.



Please note that the use of CreateGrayModel and CreateEdgeModel2 filters will only be necessary in more advanced applications. Otherwise it is enough to use a single filter of the **Matching** step and create the *model* by setting the *inGrayModel* or *inEdgeModel* parameter of the filter. For more information see Creating Models for Template Matching. The CreateEdgeModel2 and LocateMultipleObjects_Edges2 filters are preferred over CreateEdgeModel1 and LocateMultipleObjects_Edges1 because they are newer, more advanced versions with more capabilities.

The main challenge of applying the Template Matching technique lies in careful adjustment of filter parameters, rather than designing the program structure.

Advanced Application Schema

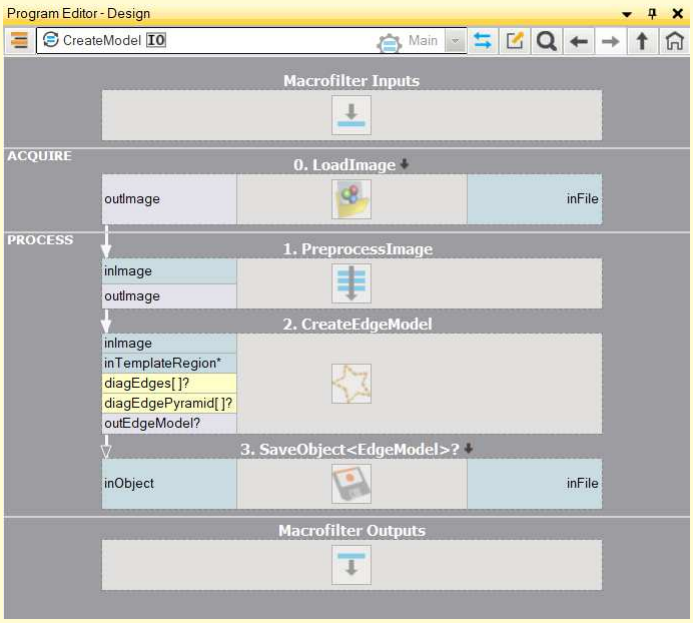
There are several kinds of advanced applications, for which the interactive GUI for Template Matching is not enough and the user needs to use the CreateGrayModel or CreateEdgeModel2 filter directly. For example:

1. When creating the model requires non-trivial image preprocessing.
2. When we need an entire array of models created automatically from a set of images.
3. When the end user should be able to define his own templates in the runtime application (e.g. by making a selection on an input image).

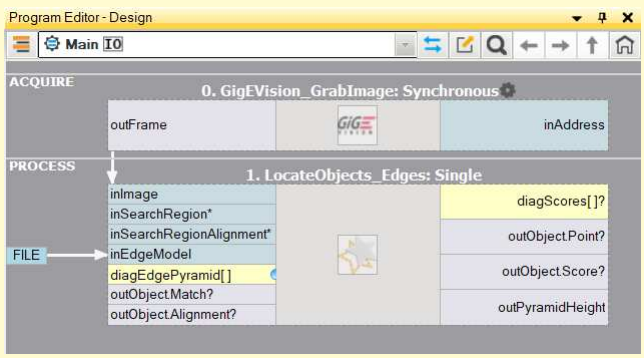
Schema 1: Model Creation in a Separate Program

For the cases 1 and 2 it is advisable to implement model creation in a separate *Task* macrofilter, save the model to an AVDATA file and then link that file to the input of the matching filter in the main program:

Model Creation:



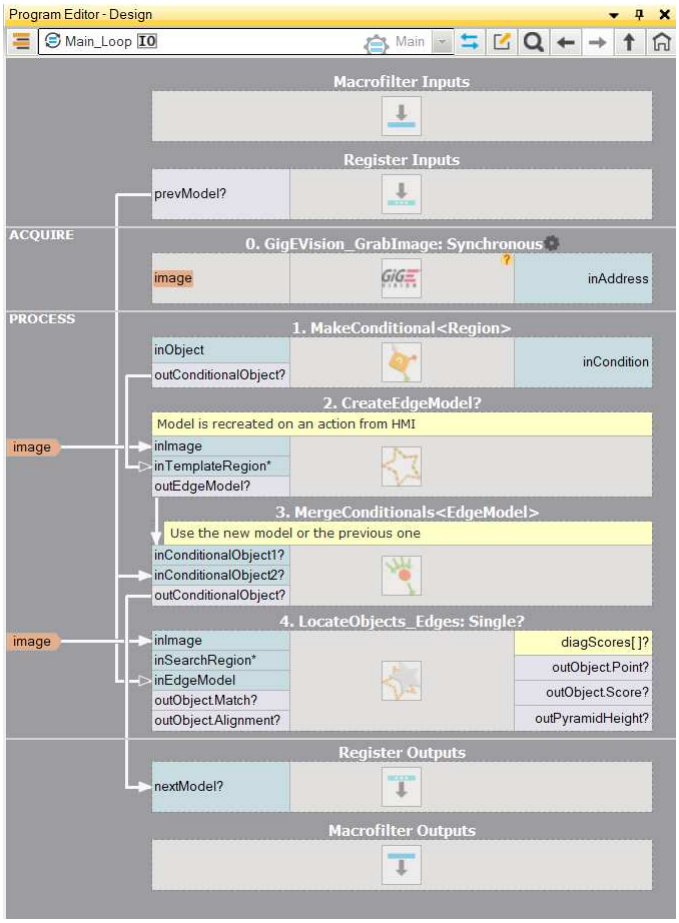
Main Program:



When this program is ready, you can run the "CreateModel" task as a program at any time you want to recreate the model. The link to the data file on the input of the matching filter does not need any modifications then, because this is just a link and what is being changed is only the file on disk.

Schema 2: Dynamic Model Creation

For the case 3, when the model has to be created dynamically, both the model creating filter and the matching filter have to be in the same task. The former, however, should be executed conditionally, when a respective HMI event is raised (e.g. the user clicks an ImpulseButton or makes some mouse action in a VideoBox). For representing the model, a register of EdgeModel2? type should be used, that will store the latest model (another option is to use LastNotNil filter). Here is an example realization with the model being created from a predefined box on an input image when a button is clicked in the HMI:



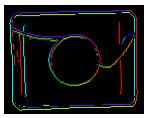
Model Creation

Height of the Pyramid

The **inMaxPyramidLevel** parameter determines the number of levels of the pyramid matching and should be set to the largest number for which the template is still recognizable on the highest pyramid level. This value should be selected through interactive experimentation using the diagnostic output **diagTemplatePyramid** (Grayscale-based Matching) or **diagEdgePyramid** (Edge-based Matching).

The **inMinPyramidLevel** parameter determines the lowest pyramid level that is generated during creation phase and the lowest pyramid level that the occurrences are tracked to during location phase. If the parameter is set to lower value in location than in creation, the missing levels are generated dynamically by the locating filter. This approach leads to much faster creation, but a bit slower location.

In the following example the **inMaxPyramidLevel** value of 4 would be too high (for both methods), as the structure of the template is entirely lost on this level of the pyramid. Also the value of 3 seems a bit excessive (especially in case of Edge-based Matching) while the value of 2 would definitely be a safe choice.

	Level 0	Level 1	Level 2	Level 3	Level 4
Grayscale-based Matching (diagTemplatePyramid):					
Edge-based Matching (diagEdgePyramid):					

Angle Range

The **inMinAngle**, **inMaxAngle** parameters determine the range of template orientations that will be considered in the matching process. For instance (values in brackets represent the pairs of **inMinAngle**, **inMaxAngle** values):

- (-180.0, 180.0): all rotations are considered (default value)
- (-15.0, 15.0): the template occurrences are allowed to deviate from the reference template orientation at most by 15.0 degrees (in each direction)
- (0.0, 0.0): the template occurrences are expected to preserve the reference template orientation

Wide range of possible orientations introduces significant amount of overhead (both in memory usage and computing time), so it is advisable to limit the range whenever possible, especially if different scales are also involved. The number of rotations created can be further manipulated with **inAnglePrecision** parameter. Decreasing it results in smaller models and smaller execution times, but can also lead to objects that are slightly less accurate.

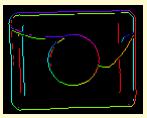
Scale Range

The **inMinScale**, **inMaxScale** parameters determine the range of template scales that will be considered in the matching process. It enables locating objects that are slightly smaller or bigger than the object used during model creation.

Wide range of possible scales introduces significant amount of overhead (both in memory usage and computing time), so it is advisable to limit the range whenever possible. The number of scales created can be further manipulated with **inScalePrecision** parameter. Decreasing it results in smaller models and smaller execution times, but can also lead to objects that are slightly less accurate.

Edge Detection Settings (only Edge-based Matching)

The **inEdgeThreshold**, **inEdgeHysteresis** parameters of CreateEdgeModel2 filter determine the settings of the hysteresis thresholding used to detect edges in the template image. The lower the **inEdgeThreshold** value, the more edges will be detected in the template image. These parameters should be set so that all the significant edges of the template are detected and the amount of redundant edges (noise) in the result is as limited as possible. Similarly to the pyramid height, edge detection thresholds should be selected through interactive experimentation using the **outEdges** output and the diagnostic output **diagEdgePyramid** - this time we need to look only at the picture at the lowest level.

		
(15.0, 30.0) - excessive amount of noise	*(40.0, 60.0) - OK*	*(60.0, 70.0) - significant edges lost*

The CreateEdgeModel2 filter will not allow to create a model in which no edges were detected at the top of the pyramid (which means not only *some* significant edges were lost, but all of them), yielding an error in such case. Whenever that happens, the height of the pyramid, or the edge thresholds, or both, should be reduced.

Matching

The **inMinScore** parameter determines how permissive the algorithm will be in verification of the match candidates - the higher the value the less results will be returned. This parameter should be set through interactive experimentation to a value low enough to assure that all correct matches will be returned, but not much lower, as too low value slows the algorithm down and may cause false matches to appear in the results.

Tips and Best Practices

How to Select a Method?

For vast majority of applications the **Edge-based Matching** method will be both more robust and more efficient than **Grayscale-based Matching**. The latter should be considered only if the template being considered has smooth color transition areas that are not defined by discernible edges, but still should be matched.

How to even further upgrade the results of Edge-based Matching?

You can use EnhanceMultipleObjectMatches filter or EnhanceSingleObjectMatch filter to fine-tune the results. A great example of usage is presented in the CreateGoldenTemplate2 filter.

Using Local Coordinate Systems

Introduction

Local coordinate systems provide a convenient means for inspecting objects that may appear at different positions on the input image. Instead of denoting coordinates of geometrical primitives in the absolute coordinate system of the image, local coordinate systems make it possible to use coordinates local to the object being inspected. In an initial step of the program the object is located and a local coordinate system is set accordingly. Other tools can then be configured to work within this coordinate system, and this makes them independent of the object translation, rotation and scale.

Two most important notions here are:

- **CoordinateSystem2D** – a structure consisting of *Origin* (Point2D), *Angle* (real number) and *Scale* (real number), defining a relative Cartesian coordinate system with its point (0, 0) located at the *Origin* point of the parent coordinate system (usually an image).
- **Alignment** – the process of transforming geometrical primitives from a local coordinate system to the coordinates of an image (absolute), or data defining such transformation. An alignment is usually represented with the **CoordinateSystem2D** data type.

Creating a Local Coordinate System

There are two standard ways of setting a local coordinate system:

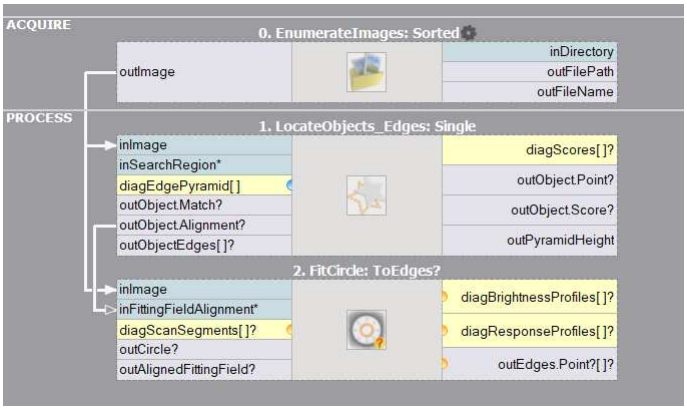
1. With Template Matching filters it is straightforward as the filters have **outObjectAlignment(s)** outputs, which provide local coordinate systems of the detected objects.
2. With one of the CreateCoordinateSystem filters , which allow for creating local coordinate systems manually at any location, and with any rotation and scale. In most typical scenarios of this kind, the objects are located with 1D Edge Detection, Shape Fitting or Blob Analysis tools.

Using a Local Coordinate System

After a local coordinate system is created it can be used in the subsequent image analysis tools. The high level tools available in Aurora Vision Studio have an **inAlignment** (or similar) input, which just needs to be connected to the port of the created local coordinate system. At this point, you should first run the program at least to the point where the coordinate system is computed, and then the geometrical primitives you will be defining on other inputs, will be automatically aligned with the position of the inspected object.

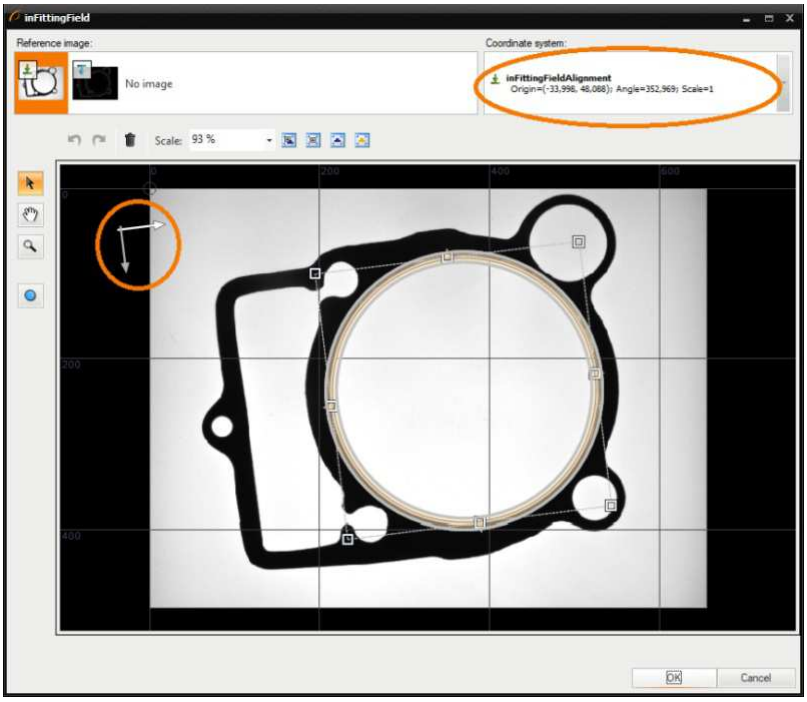
Example 1: Alignment from Template Matching

To use object alignment from a Template Matching filter, you need to connect the **Alignment** ports:



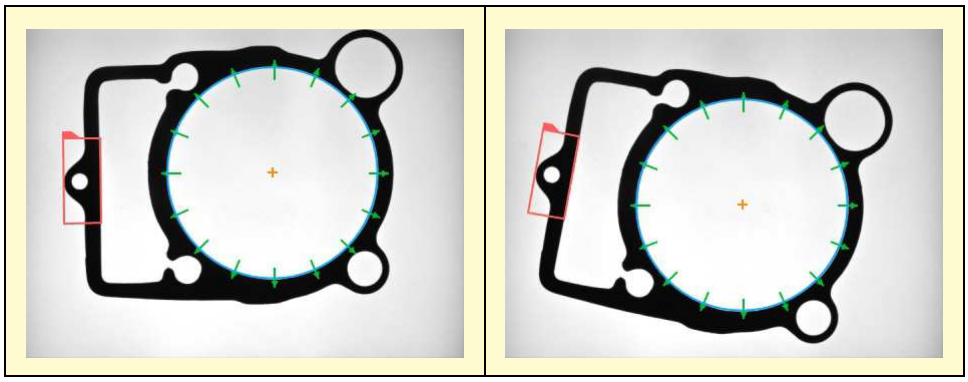
Template Matching and an aligned circle fitting.

When you execute the template matching filter and enter the editor of the **inFittingField** input of the FitCircleToEdges filter, you will have the local coordinate system already selected (you can also select it manually) and the primitive you create will have relative coordinates:



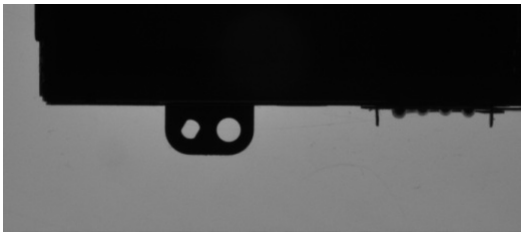
Editing an expected circle in a local coordinate system.

During program execution this geometrical primitive will be automatically aligned with the object position. Moreover, you will be able to adjust the input primitive in the context of any input image, because they will be always displayed aligned. Here are example results:



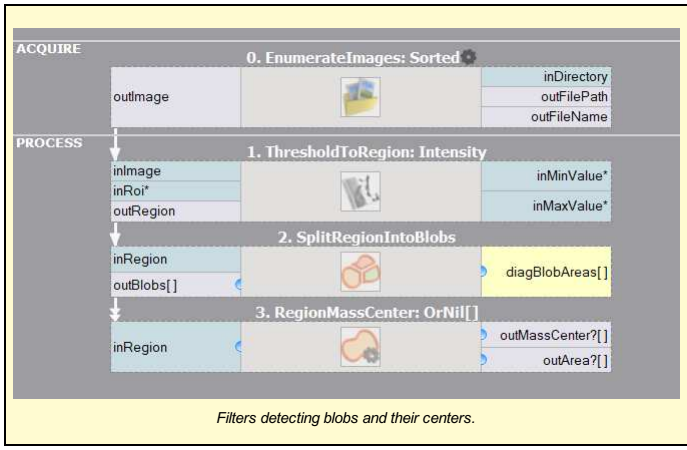
Example 2: Alignment from Blob Analysis

In many applications objects can be located with methods simpler and faster than Template Matching – like 1D Edge Detection, Shape Fitting or Blob Analysis. In the following example we will show how to create a local coordinate system from two blobs:

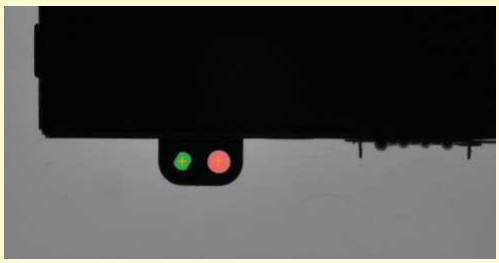


Two holes clearly define the object location.

In the first step we detect the blobs (see also: Blob Analysis) and their centers:



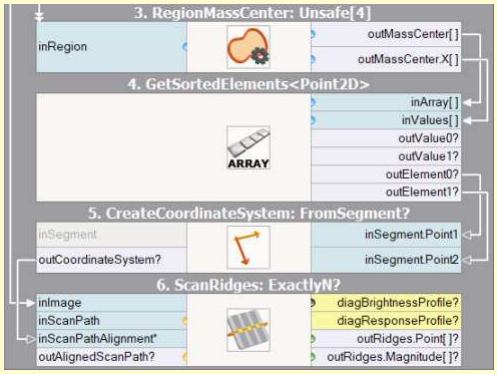
Filters detecting blobs and their centers.



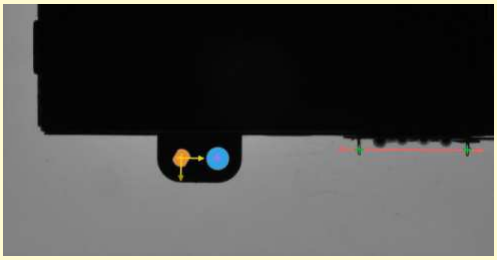
The result of blob detection.

In the second step we sort the centers by the X coordinate and create a coordinate system "from segment" defined by the two points (CreateCoordinateSystemFromSegment). The segment defines both the origin and the orientation. Having this coordinate system ready, we connect it to the **inScanPathAlignment** input of ScanExactlyNRidges, which will measure the distance between two insets. The measurement will work

correctly irrespective of the object position (mind the expanded structure inputs and outputs):



Filters creating a coordinate systems and performing an aligned measurement.



Created local coordinate system and an aligned measurement.

Manual Alignment

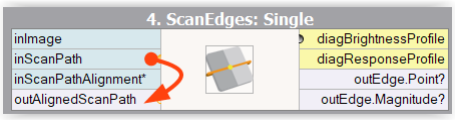
In some cases the filter you will need to use with a local coordinate system will have no appropriate **inAlignment** input. In such cases the solution is to transform the primitive manually with filters like AlignPoint, AlignCircle, AlignRectangle. These filters accept a geometrical primitive defined in a local coordinate system, and the coordinate system itself, and return the same primitive, but with absolute coordinates, i.e. aligned to the coordinate system of an image.

A very common case is with ports of type Region, which is pixel-precise and, while allowing for creation of arbitrary shapes, cannot be directly transformed. In such cases it is advisable to use the CreateRectangleRegion filter and define the region-of-interest at **inRectangle**. The filter, having also the **inRectangleAlignment** input connected, will return a region properly aligned with the related object position. Some ready-made tools, e.g. CheckPresence_Intensity, use this approach internally.

Not Mixing Local Coordinate Systems

It is important to keep in mind that geometrical primitives that appear in different places of a program may belong to different coordinate systems. When such different objects are combined together (e.g. with a filter like SegmentSegmentIntersection) or placed on a single data preview, the results will be meaningless or at least confusing. Thus, only objects belonging to the same coordinate system should be combined. In particular, when placing primitives on a preview on top of an image, only aligned primitives (with absolute coordinates) should be used.

As a general rule, image analysis filters of Aurora Vision Studio accept primitives in local coordinate systems on inputs, but outputs are always aligned (i.e. in the absolute coordinate system). In particular, many filters that align input primitives internally also have outputs that contain the input primitive transformed to the absolute coordinate system. For example, the ScanSingleEdge filters has a **inScanPath** input defined in a local coordinate system and a corresponding **outAlignedScanPath** output defined in the absolute coordinate system:

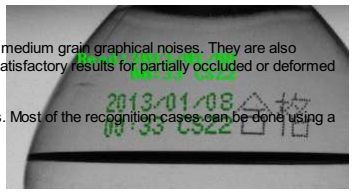


*The ScanSingleEdge filter with a pair of ports: **inScanPath** and **outAlignedScanPath**, belonging to different coordinate systems.*

Optical Character Recognition

Introduction

Optical Character Recognition (OCR) is a machine vision task consisting in extracting textual information from images.



Result of data extraction using OCR.

State of the art techniques for OCR offer high accuracy of text recognition and invulnerability to medium grain graphical noises. They are also applicable for recognition of characters made using dot matrix printers. This technology gives satisfactory results for partially occluded or deformed characters.

Efficiency of the recognition process mostly depends on the quality of text segmentation results. Most of the recognition cases can be done using a provided set of recognition models. In other cases a new recognition model can be easily prepared.

Concept

OCR technology is widely used for automatic data reading from various sources. It is especially used to gather data from documents and printed labels.

In the first part of this manual usage of high level filters will be described.

The second part of this manual shows how to use standard OCR models provided with Aurora Vision Studio. It also shows how to prepare an image to get best possible results of recognition.

The third part describes the process of preparing and training OCR models.

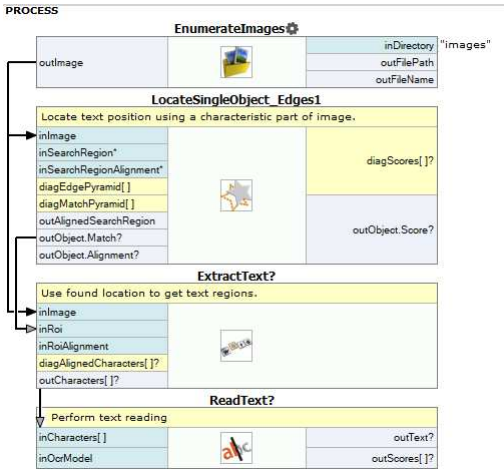
The last part presents an example program that reads text from images.

Using high level Optical Character Recognition filters

Aurora Vision Studio offers a convenient way to extract a text region from an image and then read it using a trained OCR classifier.

The typical OCR application consists of the following steps:

1. **Find text position** – locate the text position using template matching,
2. **Extract text** – use the filter ExtractText to distinct the text form the background and perform its segmentation,
3. **Read text** – recognizing the extracted characters with the ReadText filter.



Example OCR application using high level filters.

Aurora Vision Studio provides convenient graphical tools for configuring the OCR process. More informations can be found in the following articles:

1. Creating Text Segmentation Models
2. Creating Text Recognition Models

Details on Optical Character Recognition technique

Reading text from images

In order to achieve the most accurate recognition it is necessary to perform careful text extraction and segmentation. The overall process of acquiring text from images consists of the following steps:

1. Getting text location,
2. Extracting text from the background,
3. Segmenting text,
4. Using prepared OCR models,
5. Character recognition,
6. Interpreting results,
7. Verifying results.

The following sections will introduce methods used to detect and recognize text from images. For better understanding of this guide the reader should be familiar with basic blob analysis techniques.

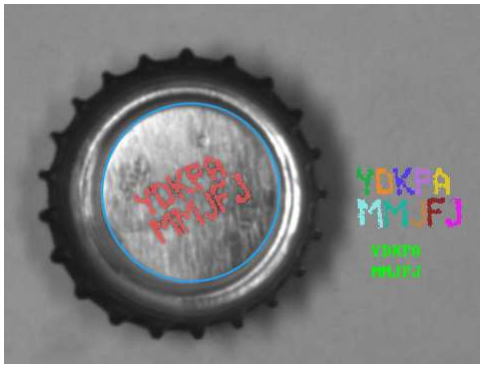
Getting text location

In general, text localization tasks can be divided into three cases:

1. The location of text is fixed and it is described by boxes called masks. For example, the personal identification card is produced according to the formal specification. The location of each data field is known. A well calibrated vision system can take images in which the location of the text is almost constant.

An example image with text masks.

2. Text location is not fixed, but it is related to a characteristic element on the input images or to a special marker (an optical mark). To get the location of the text the optical mark has to be found. This can be done with template matching, 1D edge detection or other technique.
3. The location of text is not specified, but characters can be easily separated from the background with image thresholding. The correct characters can then be found with blob analysis techniques.



Getting text from a bottle cap.

When the text location is specified, the image under analysis must be transformed to make text lines parallel to the X-axis. This can be done with RotateImage, CropImageToRectangle or ImageAlongPath filters.

Extracting text from the background

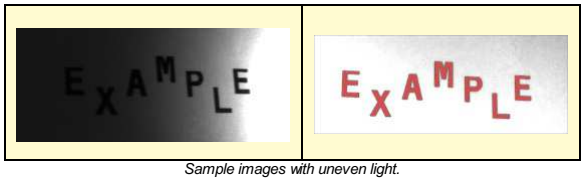
A major complication during the process of text extraction may be uneven light. Some techniques like light normalization or edge sharpening can help in finding characters. The example of light normalization can be found in the example project *Examples\Tablets*. The presentation of image sharpening using the Fourier transform can be found in the *Examples\Fourier* example.

Original image.

Image after light normalization.

Image after low-frequency image damping using the Fourier transform.

Text extraction is based on image binarization techniques. To extract characters, filters like ThresholdToRegion and ThresholdToRegion_Dynamic can be used. In order to avoid recognizing regions which do not include characters, it is advisable to use filters based on blob area.



Sample images with uneven light.

Results of ThresholdToRegion and ThresholdToRegion_Dynamic on images with uneven light.

At this point the extracted text region is prepared for segmentation.

Segmenting text

Text region segmentation is a process of splitting a region into lines and individual characters. The recognition step is only possible if each region contains a single character.

Firstly, if there are multiple lines of text, separation into lines must be performed. If the text orientation is horizontal, simple region dilation can be used followed by splitting the region into blobs. In other cases the text must be transformed, so that the lines become horizontal.

The process of splitting text into lines using region morphology filters.

When text text lines are separated, each line must be split into individual characters. In a case when characters are not made of diacritic marks and characters can be separated well, the filter SplitRegionIntoBlobs can be used. In other cases the filter SplitRegionIntoExactlyNCharacters or SplitRegionIntoMultipleCharacters must be used.

Character segmentation using SplitRegionIntoBlobs.

Character segmentation using SplitRegionIntoMultipleCharacters.

Next, the extracted characters will be translated from graphical representation to textual representation.

Using prepared OCR models

Standard OCR models are typically located in the disk directory *C:\ProgramData\Aurora Vision\{Aurora Vision Product Name}\PretrainedFonts*.

The table below shows the list of available font models:

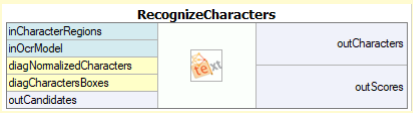
Font name	Font typeface	Set name	Characters
OCRA	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
OCRB	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
MICR	monospaced	ABC09	ABC0123456789
Computer	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
DotMatrix	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ+-./
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ+-01234556789./
		09	01234556789.+-/
Regular	proportional	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+

Character recognition

Aurora Vision Library offers two types of character classifiers:

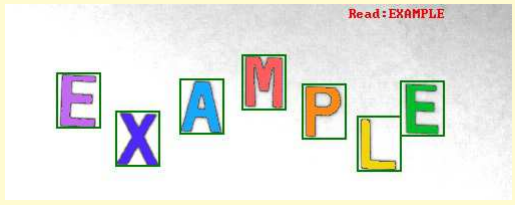
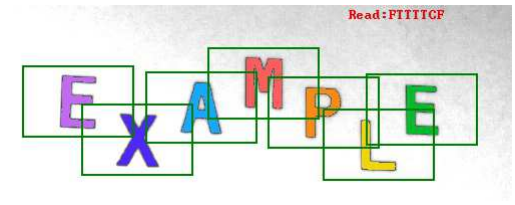
1. Classifier based on multi-layer perceptron (MLP).
2. Classifier based on support vector machines (SVM).

Both of the classifiers are stored in the OcrModel type. To get a text from character regions use the RecognizeCharacters filter, shown on the image below:



The first and the most important step is to choose the appropriate character normalization size. The internal classifier recognizes characters using their normalized form. More information about character normalization process will be provided in the section describing the process of classifier training.

The character normalization allows to classify characters with different sizes. The parameter **inCharacterSize** defines the size of a character before the normalization. When the value is not provided, the size is calculated automatically using the character bounding box.

Character presentation	Characters after normalization	Description
		The appropriate character size is chosen.
		The size of character is too small.
		Too much information about a character is lost because of too large size has been selected .

Next, character sorting order must be chosen. The default order is from left to right.

If the input text contains spaced characters, the value of **inMinSpaceWidth** input must be set. This value indicates the minimal distance between two characters between which a space will be inserted.

Character recognition provides the following information:

1. the read text as a string (**outCharacters**),
2. an array of character recognition scores (**outScores**),
3. an array of recognition candidates for each character (**outCandidates**).

Interpreting results

The table below shows recognition results for characters extracted from the example image. An unrecognized character is colored in red.

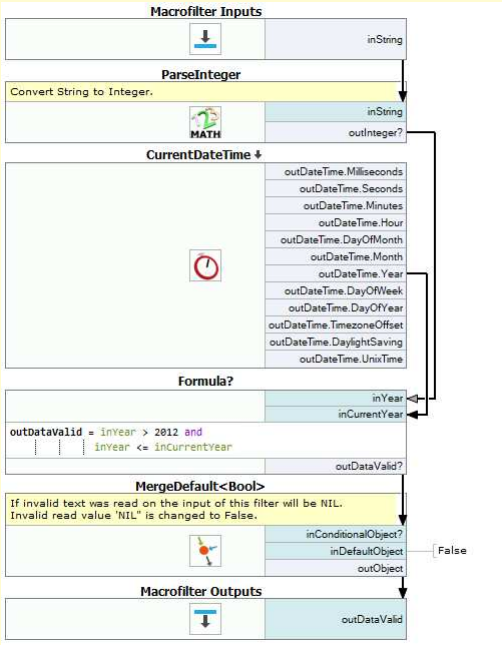
Original character	Recognized character	Score	Candidates (character and accuracy)	
	(outCharacters)	(outScores)	(outCandidates)	
E	E	1.00	E: 1.00	
X	X	1.00	X: 1.00	
A	A	1.00	A: 1.00	
M	M	1.00	M: 1.00	
P	**R**	0.50	R: 0.90	B: 0.40
L	L	1.00	L: 1.00	
E	E	1.00	E: 1.00	

In this example the letter *P* was not included in the training set. In effect, the OCR model was unable to recognize the representation of the *P* letter. The internal classifier was trying to select most similar known character.

Verifying results

The results of the character recognition process can be validated using basic string manipulation filters.

The example below shows how to check if a read text contains a valid year value. The year value should be greater than 2012 (e.g. production started in this year) and must not be greater than the current year.



The short year value validation.

This example consists of two parts. In the first part an input string is converted to an Integer value. The second one checks if year number is valid.

The filter ParseInteger was used to convert a text into a number of the Integer type. ParseInteger can return Nil value so MergeDefault filter should be used to decide what is a validation result when invalid year number was provided.

Next, the year number is checked using a formula. To get the current year the filter CurrentDateTime is used. If all conditions are met, the validation passes.

For more complex validation a user defined filter is recommended. For further information read the documentation on creating user filters.

Preparation of the OCR models

An OCR model consists of an internal statistical tool called a classifier and a set of character data. There are two kinds of classifiers used to recognize characters. The first classifier type is based on the multilayer perceptron classifier (MLP) and the second one uses support vector machines (SVM). For further details please refer to the documentation of the MLP_Init and the SVM_Init filters. Each model must be trained before it can be used.

The process of OCR model training consists of the following steps:

1. preparation of the training data set,
2. selection of the normalization size and character features,
3. setup of the OCR model,
4. training of the OCR model,
5. saving results to a file.

When these steps are performed, the model is ready to use.

Preparation of the training data set

Each classifier needs character samples in order to begin the training process. To get the best recognition accuracy, the training character samples should be as similar as possible to those which will be provided for recognition. There are two possible ways to obtain sample characters: (1) extraction of characters from real images or (2) generation of artificial characters using computer fonts.

Synthetic characters generated by means of a computer font.

In the perfect world the model should be trained using numerous real samples. However, sometimes it can be difficult to gather enough real character samples. In this case character samples should be generated by deforming the available samples. A classifier which was trained on a not big enough data set can focus only on familiar character samples at the same time failing to recognize slightly modified characters.

Character samples acquired from a real usage.

Example operations which are used to create new character samples:

1. region rotation (using the RotateRegion filter),
2. shearing (ShearRegion),
3. dilatation and erosion (DilateRegion, ErodeRegion),
4. addition of a noise.

The set of character samples deformed by: the region rotation, morphological transforms, shearing and noises.

Note: Adding too many deformed characters to a training set will increase the training time of a model.

Note: Excessive deformation of character shape can result in classifier inability to recognize the learnt character base. For example: if the training set contains a *C* character with too many noises, it can be mistaken for *O* character. In this case the classifier will be unable to determine the base of a newly provided character.

Each character sample must be stored in a structure of type CharacterSample. This structure consists of a character region and its textual representation. To create an array of character samples use the MakeCharacterSamples filter.

Selection of normalization size and character features

The character normalization allows for reduction of the amount of data used in the character classification. The other aim of normalization is to enable the classification process to recognize characters of various sizes.

During normalization each character is resized into a size which was provided during the model initialization. All further classifier operations will be performed on the resized (normalized) characters.

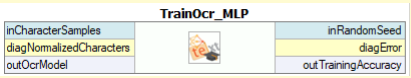
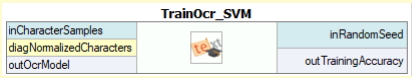
Various size characters before and after the normalization process.

Selection of too large normalization size will increase training time of the OCR classifier. On the other hand, too low size will result in loss of important character details. The selected normalization size should be a compromise between classification time and the accuracy of recognition. For the best results, a character size after normalization should be similar to its size before normalization.

During the normalization process some character details will be lost, e.g. the aspect ratio of a character. In the training process, some additional information can be added, which can compensate for the information loss in the normalization process. For further information please refer to the documentation of the TrainOcr_MLP filter.

Training of the OCR model

There are two filters used to train each type of an OCR classifier. These filters require parameters which describe the classifier training process.

TrainOcr_MLP	TrainOcr_SVM
	
Training of MLP classifier using TrainOcr_MLP.	*Training of SVM classifier using TrainOcr_SVM.*

Saving the training results

After successful classifier training the results should be saved for future use. The filter SaveObject should be used.

Example: Reading numbers from masks

The purpose of this example is to read fields from ID cards. The input data is extracted from real cases. The example consists of two parts:

1. Creation of data set for classifier training is presented in "TrainCharacters" macrofilter.
2. This macrofilter is designed to acquire data from provided masks. It is implemented in the "Main" macrofilter.

The example project is available in *Examples\Ocr Read Numbers* directory.

Training the classifier

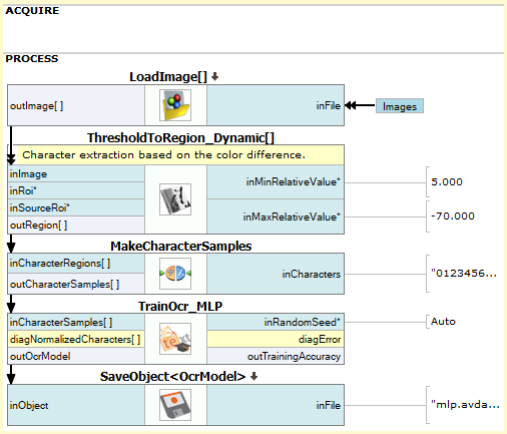
First, this filter LoadImage loads the character images from separate files. File names are stored in a global parameter called *FileNames*.

Next, each image is thresholded and the character samples are made.

Then an MLP classifier is created.

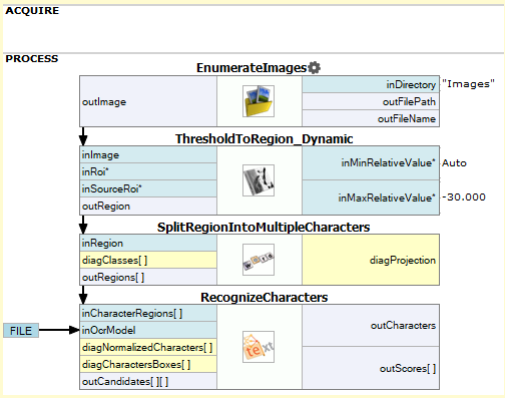
After initialization the classifier is trained using available character samples.

Finally, the created classifier is saved to a file.



Macrofilter definition which performs OCR training.

Reading data from images



Macrofilter which uses the trained classifier to acquire text from images.

First, the example image is loaded from a file.

Next, the text is extracted from the background by ThresholdToRegion_Dynamic.

Then, the text is split into separated characters with SplitRegionIntoMultipleCharacters.

Finally, the characters are recognized.

Camera Calibration and World Coordinates

Camera Calibration

Camera calibration, also known as camera resectioning, is a process of estimating parameters of a camera model: a set of parameters that describe the internal geometry of image capture process. Accurate camera calibration is is essential for various applications, such as multi-camera setups where images relate to each other, removing geometric distortions due to lens imperfections, or precise measurement of real-world geometric properties (positions, distances, areas, straightness, etc.).

The model to be used is chosen depending on the camera type (e.g. projective camera, telecentric camera, line scan camera) and accuracy requirements. In a case of a standard projective camera, the model (known as pinhole camera model) consists of focal length, principal point location and distortion parameters.

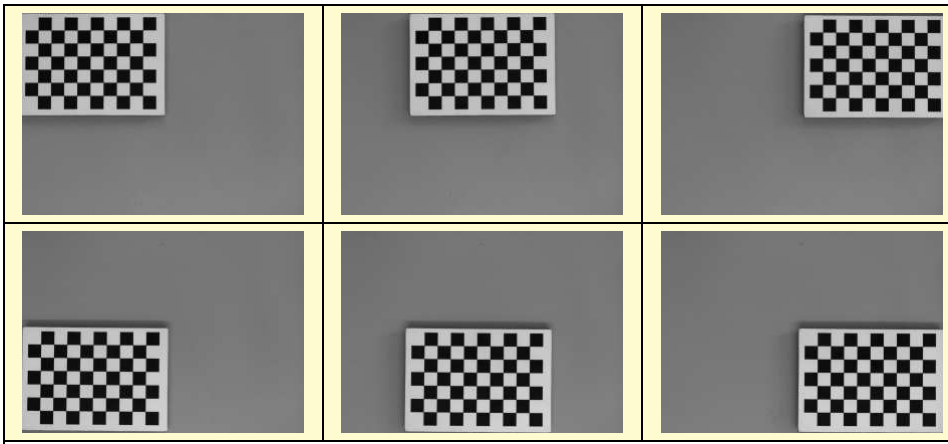
A few distortion model types are supported. The simplest - divisional - supports most use cases and has predictable behaviour even when calibration data is sparse. Higher order models can be more accurate, however they need a much larger dataset of high quality calibration points, and are usually needed for achieving high levels of positional accuracy across the whole image - order of magnitude below 0.1 pix. Of course this is only a rule of thumb, as each lens is different and there are exceptions.

The area scan camera models (pinhole or telecentric) contain only *intrinsic* camera parameters, and so it does not change with camera repositioning, rotations, etc. Thanks to that, there is no need for camera calibration in the production environment, the camera can be calibrated beforehand. As soon as the camera has been assembled with the lens and lens adjustments (zoom/focus/f-stop rings) have been tightly locked, the calibration images can be taken and camera calibration performed. Of course any modifications to the camera-lens setup void the calibration parameters, even apparently minor ones such as removing the lens and putting it back on the camera in seemingly the same position.

On the other hand the line scan model contains parameters of whole imaging setup, i.e. camera and a moving element (usually a conveyor belt). Such approach, in contrast with area scan camera calibration, is necessary as the moving element of line scan camera system is tightly bound within the image acquisition geometry.

Camera model can be directly used to obtain an *undistorted* image (an image, which would have been taken by a camera with the same basic parameters, but without lens distortion present), however for most use cases the camera calibration is just a prerequisite to some other operation. For example, when camera is used for inspection of planar surfaces (or objects lying on such surface), the camera model is needed to perform a *World Plane* calibration (see *World Plane - measurements and rectification* section below).

In Aurora Vision Studio user will be prompted by a GUI when a camera calibration is needed to be performed. Alternatively, filters responsible for camera calibration may be used directly: CalibrateCamera_Pinhole, CalibrateCamera_Telecentric, CalibrateCamera_LineScan.



A set of grid pictures for basic calibration. Note that high accuracy applications require denser grids and higher amount of pictures. Also note that all grids are perpendicular to the optical axis of the camera, so the focal length won't be calculated by the filter.

World Plane - Measurements and Rectification

Vision systems which are concerned with observation and inspection of planar (flat) surfaces, or objects lying on such surfaces (e.g. conveyor belts) can take advantage of the *image to world plane transform* mechanism of Aurora Vision Studio, which allows for:

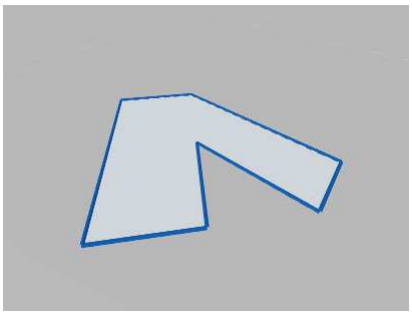
- Calculation of real world coordinates from locations on original image. This is crucial, for example, for interoperability with external devices, such as industrial robots. Suppose a object is detected on the image, and its location needs to be transmitted to the robot. The detected object location is given in image coordinates, however the robot is operating in real world with different coordinate system. A common coordinate system is needed, defined by a *world plane*.
- Image *rectification* onto the *world plane*. This is needed when performing image analysis using original image is not feasible (due to high degree of lens and/or perspective distortion). The results of analysis performed on a rectified image can also be transformed to real-world coordinates defined by a world plane coordinate system. Another use case is a multi-camera system – *rectification* of images from all the cameras onto common *world plane* gives a simple and well defined relation between those rectified images, which allows for easy superimposing or mosaic stitching.

The image below shows the image coordinate system. Image coordinates are denoted in pixels, with the origin point (0, 0) corresponding to the top-left corner of the image. The X axis starts at the left edge of an image and goes towards the right edge. The Y axis starts at the top of the image towards image bottom. All image pixels have nonnegative coordinates.

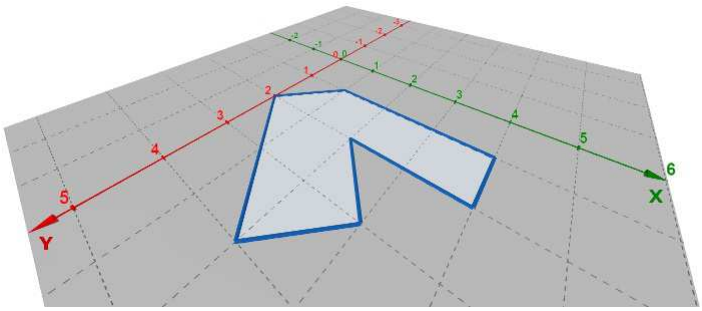
Directions and pixel positions in image coordinates.

The *world plane* is a distinguished flat surface, defined in the real 3D world. It may be arbitrarily placed with respect to the camera. It has a defined origin position and XY axes.

Images below present the concept of a *world plane*. First image presents an original image, as captured by a camera that has not been mounted quite straight above the object of interest. The second image presents a *world plane*, which has been aligned with the surface on which the object is present. This allows for either calculation of world coordinates from pixel locations on original image, or image rectification, as shown on the next images.



Object of interest as captured by an imperfectly positioned camera.



World plane coordinate system superimposed onto the original image.

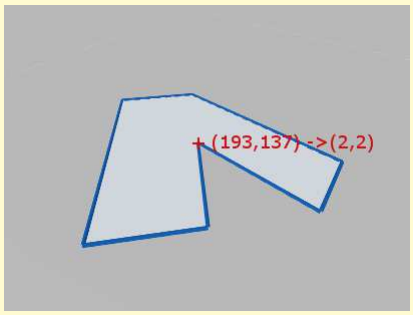


Image to world plane coordinate calculation.

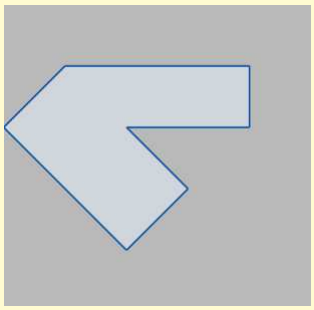


Image rectification, with cropping to an area from point (0,0) to (5,5) in world coordinates.

In order to use the *image to world plane transform* mechanism of Aurora Vision Studio, appropriate UI wizards are supplied:

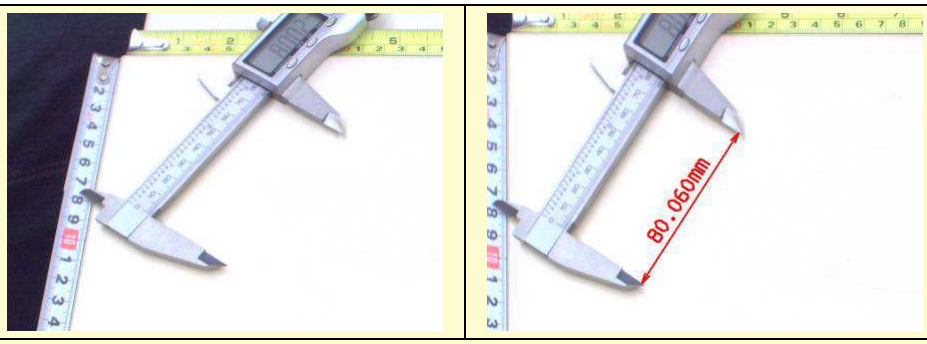
- For calculation of real world coordinates from locations on original image – use a wizard associated with the **inTransform** input of ImagePointToWorldPlane filter (or other from ImageObjectsToWorldPlane group).
- For image *rectification* onto the *world plane* – use a wizard associated with the **inRectificationMap** input of RectifyImage filter.

Although using UI wizards is the recommended course of action, the most complicated use cases may need a direct use of filters, in such a case following steps are to be performed:

1. Camera calibration – this step is highly recommended to achieve accurate results, although not strictly necessary (e.g. when lens distortion errors are insignificant).
2. World plane calibration – the CalibrateWorldPlane filters compute a RectificationTransform, which represents *image to world plane relation*
3. The *image to world plane relation* then can be used to:
    - Calculate of real world coordinates from locations on original image, and vice versa, see ImagePointToWorldPlane, WorldPlanePointToImage or similar filters (from ImageObjectsToWorldPlane or WorldPlaneObjectsToImage groups).
    - Perform image *rectification* onto the world plane, see CreateRectificationMap filters.

There are different use cases of world coordinates calculation and image rectification:

- Calculating world coordinates from pixel locations on original image without image rectification. This approach uses transformation output for example by CalibrateWorldPlane to calculate real world coordinates with ImageObjectsToWorldPlane
- Second scenario is very similar to the first one with the difference of using image rectification. In this case, after performing analysis on an rectified image (i.e. image remapped by RectifyImage), the locations can be transformed to a common coordinate system given by the world plane by using the *rectified image to world plane* relation. It is given by auxiliary output **outRectifiedTransform** of RectifyImage filter. Notice that the *rectified image to world plane* relation is different than *original image to world plane* relation.
- Last use case is to perform image rectification and rectified image analysis without its features recalculation to real world coordinates.



Example of taking world plane measurements on the rectified image. Left: original image, as captured by a camera, with mild lens distortion. Right: rectified image with annotated length measurement.

Notes:

- *Image to world plane transform* is still a valid mechanism for telecentric cameras. Is such a case, the image would be related to world plane by an affine transform.
- Camera distortion is automatically accounted for in both world coordinate calculations and image rectification.
- The spatial map generated by CreateRectificationMap filters can be thought of as a map performing *image undistortion* followed by a *perspective removal*.

Extraction of Calibration Grids

Both *camera calibration* and *image to world plane transform* calculation use extracted *calibration grids* in the form of array of image points with grid indices, i.e. annotated points.

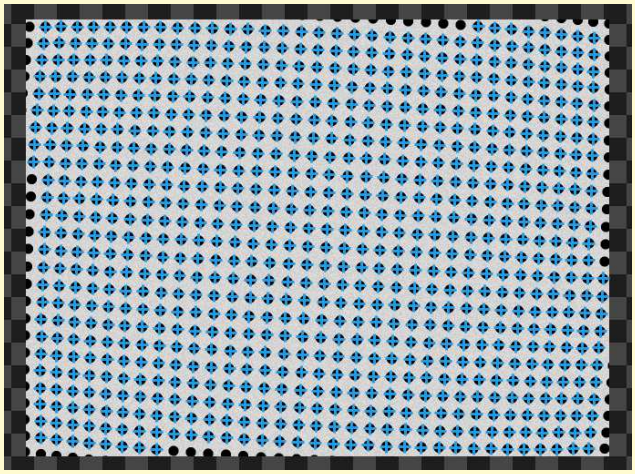
Note that the real-world coordinates of the grids are 2D, because the relative z coordinate of any point on the flat grid is 0.

Aurora Vision Studio provides extraction filters for a few standard grid formats (see: DetectCalibrationGrid_Chessboard and DetectCalibrationGrid_Circles).

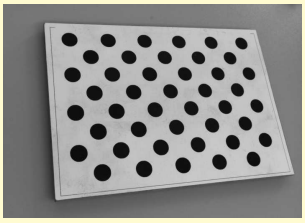
Using custom grids requires a custom solution for extracting the image point array. If the custom grid is a rectangular grid, the AnnotateGridPoints filter may be used to compute annotations for the image points.

Note that the most important factor in achieving high accuracy results is the precision and accuracy of extracted calibration points. The calibration grids should be as flat and stiff as possible (cardboard is not a proper backing material, thick glass is perfect). Take care of proper conditions when taking the calibration images: minimize motion blur by proper camera and grid mounts, prevent reflections from the calibration surface (ideally use diffusion lighting). When using a custom calibration grid, make sure that the points extractor can achieve subpixel precision. Verify that measurements of the real-world grid coordinates are accurate. Also, when using a chessboard calibration grid, make sure that the whole calibration grid is visible in the image. Otherwise, it will not be detected because the detection algorithm requires a few pixels wide quiet zone around the chessboard. Pay attention to the number of columns and rows, as providing misleading data may make the algorithm work incorrectly or not work at all.

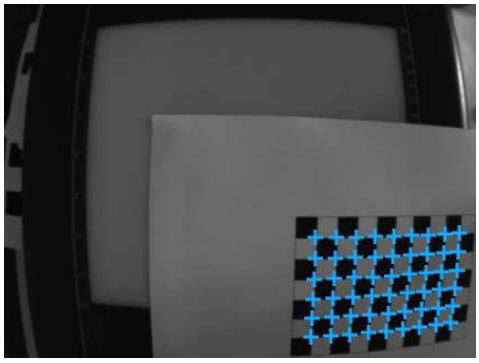
The recommended calibration grid to use in Aurora Vision Studio is a circles grid, see DetectCalibrationGrid_Circles. Optimal circle radius may vary depending on exact conditions, however a good rule of thumb is 10 pixels (20 pixel diameter). Smaller circles tend to introduce positioning jitter. Bigger circles lower the total amount of calibration points and suffer from geometric inaccuracies, especially when lens distortion and/or perspective is noticeable. Note: it is important to use a symmetric board as shown in the image below. Asymmetric boards are currently not supported.



Symmetric circle grid is the recommended one to use in Aurora Vision Studio.



Unsupported asymmetric circle grid.



Detected chessboard grid, with image point array marked.

Please refer to Preparing Rectification Map Transform to get step-by-step instruction on how to perform calibration with the calibration editor (plugin).

Application Guide – Image Stitching

Seamless image stitching in multiple camera setup is, in its essence, an image rectification onto the world plane.

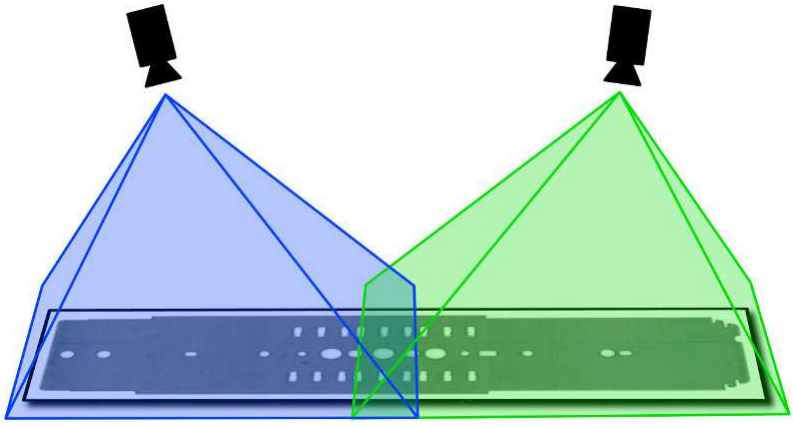
Note that high quality stitching requires a vigilant approach to the calibration process. Each camera introduces both lens distortion as well as perspective distortion, as it is never positioned perfectly perpendicular to the analyzed surface. Other factors that need to be taken into account are the camera-object distance, camera rotation around the optical axis, and image overlap between cameras.

The process consists of two main steps. First, each camera is calibrated to produce a partial, rectified image. Then all partial images are simply merged using the JoinImages filter.

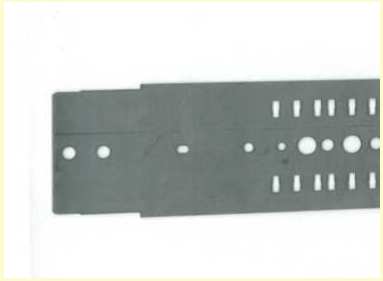
Image stitching procedure can be outlined as follows:

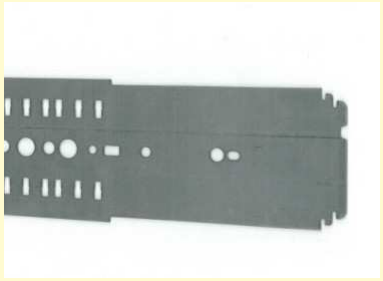
- Cover the inspection area with two or more cameras. Make sure that fields of view of individual cameras overlap a bit.
- Place a calibration grid onto the inspection area. For each camera, capture the image of a part of the calibration grid. The grid defines a world coordinate system used for stitching, and so it should contain some markers from which the coordinates of world plane points will be identifiable for each camera.
- Define the world coordinate extents for which each camera will be responsible. For example, lets define that camera 1 should cover area from 100 to 200 in X, and from -100 to 100 in Y coordinate; camera 2 - from 200 to 300 in X, and from -100 to 100 in Y.
- For each camera, use a wizard associated with the **inRectificationMap** input of RectifyImage filter to setup the image rectification. Use the captured image for camera calibration and world to image transform. Use the defined world coordinate extents to setup the rectification map generation (select "world bounding box" mode of operation). Make sure that the world scale for rectification is set to the same fixed value for all images.
- Use the JoinImages appropriately to merge outputs of RectifyImage filters.

Please refer to the Image stitching tutorial for more.

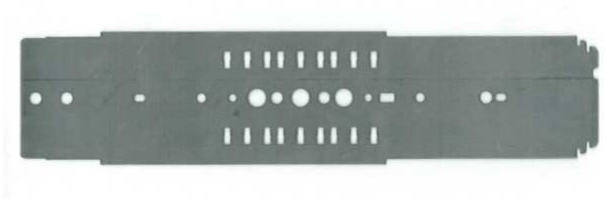


A multi-camera setup for inspection of a flat object.





Input images, as captured by cameras.



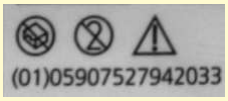
Stitching result.

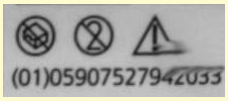
Golden Template

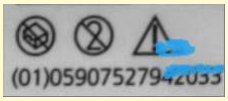
Golden Template technique performs a pixel-to-pixel comparison of two images. This technique is especially useful when the object's surface or object's shape is very complex.

Aurora Vision Studio offers three ways of performing the golden template comparison.

- **Comparison based on pixels intensity** - it can be achieved using the CompareGoldenTemplate_Intensity. In this method two images are compared pixel-by-pixel and the defect is classified based on a difference between pixels intensity. This technique is especially useful in finding defects like smudges, scratches etc.







Golden template	Defected object	Found defects

Example usage of Golden Template technique using the pixels intensity based comparison.

- **Comparison based on objects edges** - this method is very useful when defects may occur on the edge of the object and pixel comparison may fail due to different light reflections or the checking the object surface is not necessary. For matching object's edges use the CompareGoldenTemplate_Edges filter.

Golden template	Defected object	Found defects

Example usage of Golden Template technique using the edges comparison.

- **Second version of the comparison based on objects edges** - this method uses more than one image to create the model for the inspection. Due to that it is not vulnerable to pixel-sized errors and displacements. Advanced tips on how to use its parameters are located here: CompareGoldenTemplate2.

How To Use

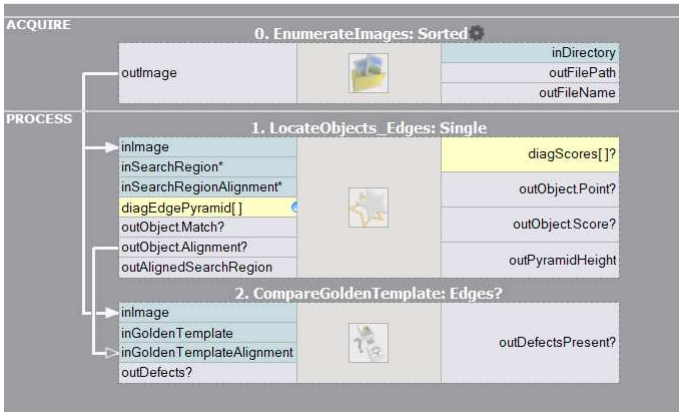
Golden template is a previously prepared image which is used to compare image from the camera. This robust technique allows us to perform quick comparison inspection but some conditions must be met:

- stable light conditions,
- position of the camera and the object must be still,
- precise object positioning

Most applications use the Template Matching technique for finding objects and then matched rectangle is compared. Golden template image and image to compare must have this same dimensions. To get best results filter CropImageToRectangle should be used. Please notice that filter CropImageToRectangle performs cropping using a real values and it has sub-pixel precision.

Example

Example below shows how to use a basic golden template matching.



Example application performs following operations

1. Finding object location using the Template Matching technique.
2. Comparing input image with previously prepared golden template.

Deep Learning

Table of contents:

1. Introduction

Deep Learning is a breakthrough machine learning technique in computer vision. It learns from training images provided by the user and can automatically generate solutions for a wide range of image analysis applications. Its key advantage, however, is that it is able to solve many of the applications which have been too difficult for traditional, rule-based algorithms of the past. Most notably, these include inspections of objects with high variability of shape or appearance, such organic products, highly textured surfaces or natural outdoor scenes. What is more, when using ready-made products, such as our Aurora Vision Deep Learning, the required programming effort is reduced almost to zero. On the other hand, deep learning is shifting the focus to working with data, taking care of high quality image annotations and experimenting with training parameters – these elements actually tend to take most of the application development time these days.

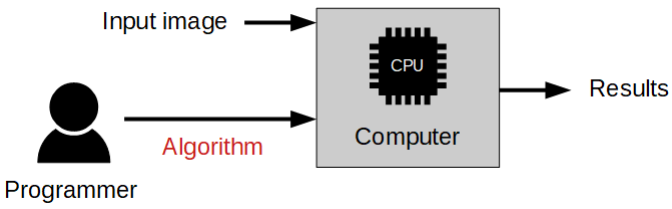
Typical applications are:

- detection of surface and shape defects (e.g. cracks, deformations, discoloration),
- detecting unusual or unexpected samples (e.g. missing, broken or low-quality parts),
- identification of objects or images with respect to predefined classes (i.e. sorting machines),
- location, segmentation and classification of multiple objects within an image (i.e. bin picking),
- product quality analysis (including fruits, plants, wood and other organic products),
- location and classification of key points, characteristic regions and small objects,
- optical character recognition.

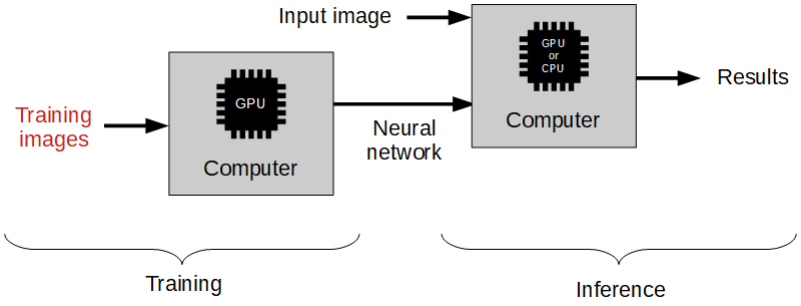
The use of deep learning functionality includes two stages:

1. Training – generating a model based on features learned from training samples,
2. Inference – applying the model on new images in order to perform the actual machine vision task.

The difference to the traditional image analysis approach is presented in the diagrams below:



Traditional approach: The algorithm must be designed by a human specialist.



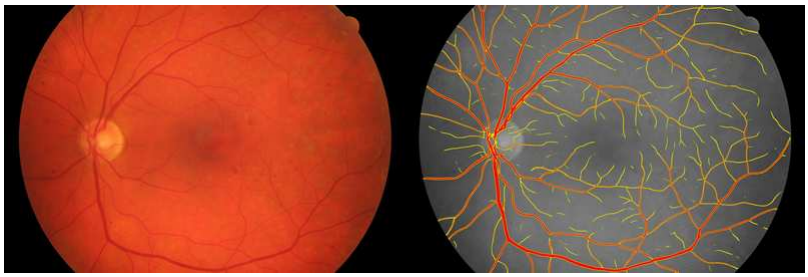
Machine learning approach: We only need to provide a training set of labeled images.

Overview of Deep Learning Tools

1. **Anomaly Detection** – this technique is used to detect anomalous (unusual or unexpected) samples. It only needs a set of fault-free samples to learn the model of normal appearance. Optionally, several faulty samples can be added to better define the threshold of tolerable variations. This tool is useful especially in cases where it is difficult to specify all possible types of defects or where negative samples are simply not available. The output of this tool are: a classification result (normal or faulty), an abnormality score and a (rough) heatmap of anomalies in the image.

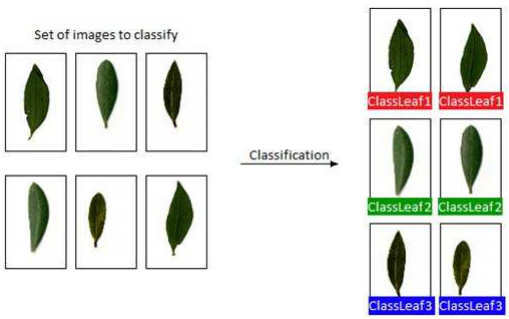
An example of a missing object detection using DL_DetectAnomalies2 tool.
Left: The original image with a missing element. Right: The classification result with a heatmap of anomalies.

2. **Feature Detection (segmentation)** – this technique is used to precisely segment one or more classes of pixel-wise features within an image. The pixels belonging to each class must be marked by the user in the training step. The result of this technique is an array of probability maps for every class.



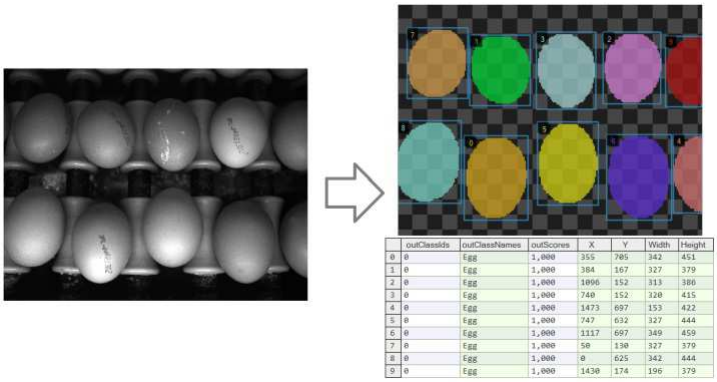
An example of image segmentation using DL_DetectFeatures tool.
Left: The original image of the fundus. Right: The segmentation of blood vessels.

3. **Object Classification** – this technique is used to identify an object in a selected region with one of user-defined classes. First, it is necessary to provide a training set of labeled images. The result of this technique is: the name of detected class and a classification confidence level.



An example of object classification using DL_ClassifyObject tool.

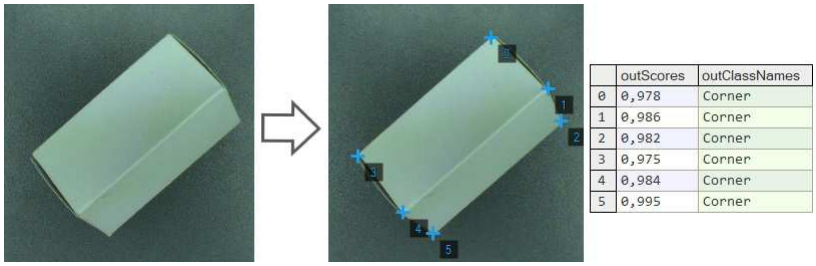
4. **Instance Segmentation** – this technique is used to locate, segment and classify one or multiple objects within an image. The training requires the user to draw regions corresponding to objects in an image and assign them to classes. The result is a list of detected objects – with their bounding boxes, masks (segmented regions), class IDs, names and membership probabilities.



	outClassIds	outClassNames	outScores	X	Y	Width	Height
0	0	Egg	1,000	355	705	342	451
1	0	Egg	1,000	384	167	327	379
2	0	Egg	1,000	1096	152	313	386
3	0	Egg	1,000	740	152	320	415
4	0	Egg	1,000	1473	697	153	422
5	0	Egg	1,000	747	632	327	444
6	0	Egg	1,000	1117	697	349	459
7	0	Egg	1,000	50	130	327	379
8	0	Egg	1,000	0	625	342	444
9	0	Egg	1,000	1430	174	196	379

An example of instance segmentation using DL_SegmentInstances tool. Left: The original image. Right: The resulting list of detected objects.

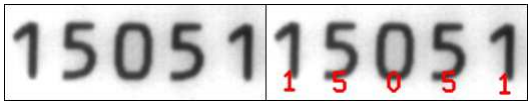
5. **Point Location** – this technique is used to precisely locate and classify key points, characteristic parts and small objects within an image. The training requires the user to mark points of appropriate classes on the training images. The result is a list of predicted point locations with corresponding class predictions and confidence scores.



	outScores	outClassNames
0	0,978	Corner
1	0,986	Corner
2	0,982	Corner
3	0,975	Corner
4	0,984	Corner
5	0,995	Corner

An example of point location using DL_LocatePoints tool. Left: The original image. Right: The resulting list of detected points.

6. **Reading Characters** – this technique is used to locate and recognize characters within an image. The result is a list of found characters.



An example of optical character recognition using DL_ReadCharacters tool. Left: The original image. Right: The image with the recognized characters drawn.

Basic Terminology

You do not need to have the specialistic scientific knowledge to develop your deep learning solutions. However, it is highly recommended to understand the basic terminology and principles behind the process.

Deep neural networks

Aurora Vision provides access to several standardized deep neural networks architectures created, adjusted and tested to solve industrial machine vision tasks. Each of the networks is a set of trainable convolutional filters and neural connections which can model complex transformations of an image with the goal to extract relevant features and use them to solve a particular problem. However, these networks are useless without proper amount of good quality data provided for training process. This documentation presents necessary practical hints on creating an effective deep learning model.

Depth of a neural network

Due to various levels of task complexity and different expected execution times, the users can choose one of five available network depths. The **Network Depth** parameter is an abstract value defining the memory capacity of a neural network (i.e. the number of layers and filters) and the ability to solve more complex problems. The list below gives hints about selecting the proper depth for a task characteristics and conditions.

1. Low depth (value 1-2)
    - A problem is simple to define.
    - A problem could be easily solved by a human inspector.
    - A short time of execution is required.
    - Background and lighting do not change across images.
    - Well-positioned objects and good quality of images.
2. Standard depth (default, value 3)
    - Suitable for a majority of applications without any special conditions.
    - A modern CUDA-enabled GPU is available.
3. High depth (value 4-5)
    - **A big amount of training data is available.**
    - A problem is hard or very complex to define and solve.
    - Complicated irregular patterns across images.
    - Long training and execution times are not a problem.
    - A large amount of GPU RAM (≥4GB) is available.
    - Varying background, lighting and/or positioning of objects.

Tip: Test your solution with a lower depth first, and then increase it if needed.

Note: A higher network depth will lead to a significant increase in memory and computational complexity of training and execution.

Training process

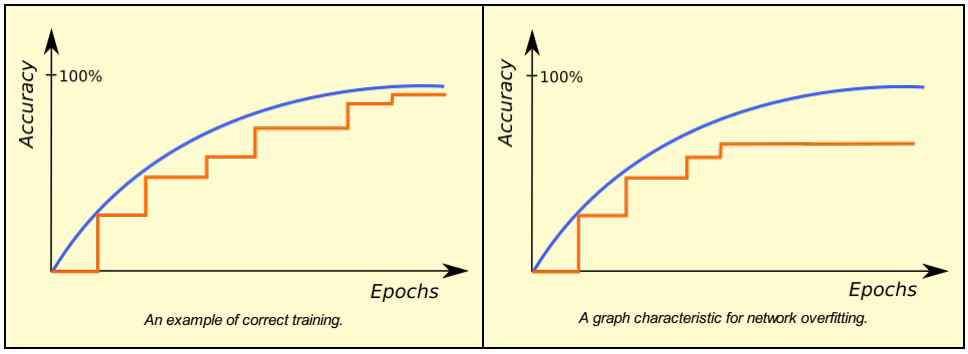
Model training is an iterative process of updating neural network weights based on the training data. One **iteration** involves some number of steps (determined automatically), each step consists of the following operations:

1. selection of a small subset (**batch**) of training samples,
2. calculation of an error measure for these samples,
3. updating the weights to achieve lower error for these samples.

At the end of each iteration, the current model is evaluated on a separate set of validation samples selected before the training process. Validation set is automatically chosen from the training samples. It is used to simulate how neural network would work with real images not used during training. Only the set of network weights corresponding with the best validation score at the end of training is saved as the final solution. Monitoring the training and validation score (blue and orange lines in the figures below) in consecutive iterations gives fundamental information about the

progress:

1. Both training and validation scores are improving – keep training, the model can still improve.
2. Both training and validation scores has stopped improving – keep training for a few iterations more and stop if there is still no change.
3. Training score is improving, but validation score has stopped or is going worse – you can stop training, model has probably started **overfitting** to your training data (remembering exact samples rather than learning rules about features). It may also be caused by too small amount of diverse samples or too low complexity of the problem for a network selected (try lower **Network Depth**).



An example of correct training. *A graph characteristic for network overfitting.*

The above graphs represent training progress in the Deep Learning Editor. The blue line indicates performance on the training samples, and the orange line represents performance on the validation samples. Please note the blue line is plotted more frequently than the orange line as validation performance is verified only at the end of each iteration.

Stopping Conditions

The user can stop the training manually by clicking the **Stop** button. Alternatively, it is also possible to set one or more stopping conditions:

1. **Iteration Count** – training will stop after a fixed number of iterations.
2. **Iterations without Improvement** – training will stop when the best validation score was not improved for a given number of iterations.
3. **Time** – training will stop after a given number of minutes has passed.
4. **Validation Accuracy** or **Validation Error** – training will stop when the validation score reaches a given value.

Preprocessing

To adjust performance to a particular task, the user can apply some additional transformations to the input images before training starts:

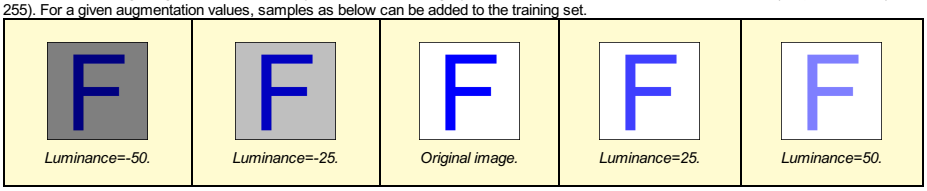
1. **Downsample** – reduction of the image size to accelerate training and execution times, at the expense of lower level of details possible to detect. Increasing this parameter by 1 will result in downsampling by the factor of 2 over both image dimension.
2. **Convert to Grayscale** – while working with problems where color does not matter, you can choose to work with monochrome versions of images.

Augmentation

In case when the number of training images can be too small to represent all possible variations of samples, it is recommended to use data augmentations that add artificially modified samples during training. This option will also help avoiding overfitting.

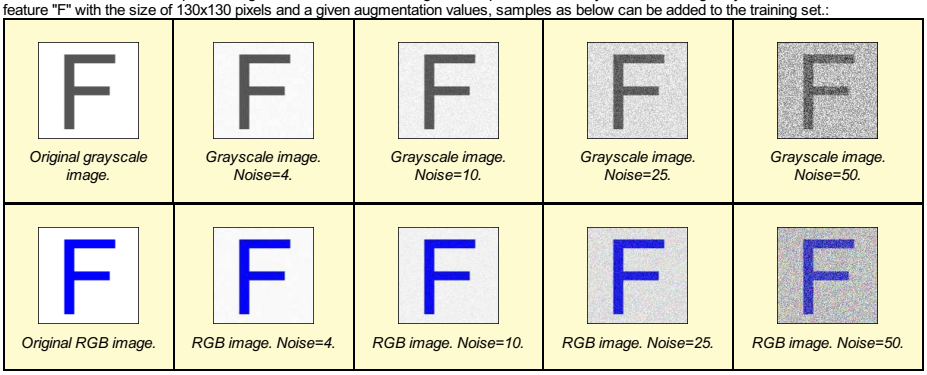
Below is a description of the available augmentations and examples of the corresponding transformations:

1. **Luminance** – change brightness of samples by a random percentage (between -ParameterValue and +ParameterValue) of pixel values (0-255). For a given augmentation values, samples as below can be added to the training set.



Luminance=-50. *Luminance=-25.* *Original image.* *Luminance=25.* *Luminance=50.*

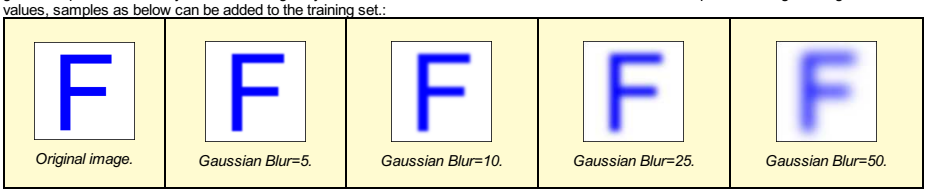
2. **Noise** – modify samples with uniform noise. Value of each channel and pixel is modified separately, by random percentage (between -ParameterValue and +ParameterValue) of pixel values (0-255). Please note that choosing an appropriate augmentation value should depend on the size of the feature in pixels. Larger value will have a much greater impact on small objects than on large objects. For a tile with the feature "F" with the size of 130x130 pixels and a given augmentation values, samples as below can be added to the training set.:



Original grayscale image. *Grayscale image. Noise=4.* *Grayscale image. Noise=10.* *Grayscale image. Noise=25.* *Grayscale image. Noise=50.*

Original RGB image. *RGB image. Noise=4.* *RGB image. Noise=10.* *RGB image. Noise=25.* *RGB image. Noise=50.*

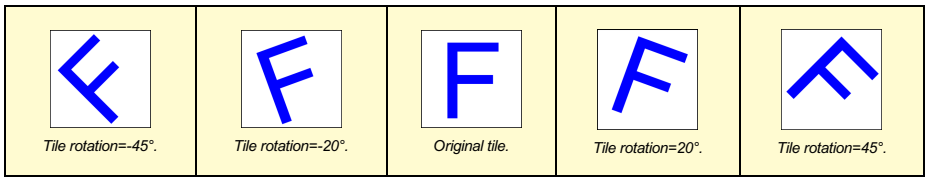
3. **Gaussian Blur** – blur samples with a kernel of a size randomly selected between 0 and the provided maximum kernel size. Please note that choosing an appropriate Gaussian Blur Kernel Size should depend on the size of the feature in pixels. Larger kernel sizes will have a much greater impact on small objects than on large objects. For a tile with the feature "F" with the size of 130x130 pixels and a given augmentation values, samples as below can be added to the training set.:



Original image. *Gaussian Blur=5.* *Gaussian Blur=10.* *Gaussian Blur=25.* *Gaussian Blur=50.*

4. **Rotation** – rotate samples by a random angle between -ParameterValue and +ParameterValue. Measured in degrees.

   In Detect Features, Locate Points and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.



Tile rotation=-45°. | *Tile rotation=-20°.* | *Original tile.* | *Tile rotation=20°.* | *Tile rotation=45°.*

In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.

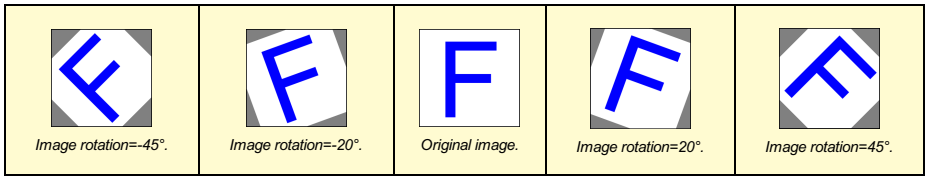
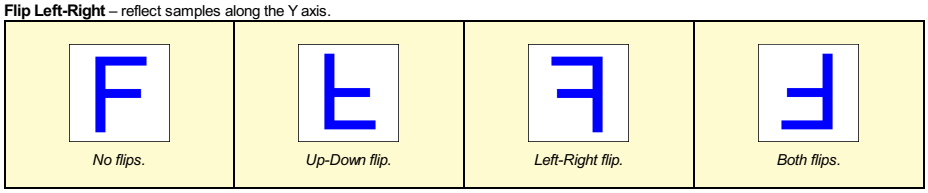


Image rotation=-45°. | *Image rotation=-20°.* | *Original image.* | *Image rotation=20°.* | *Image rotation=45°.*

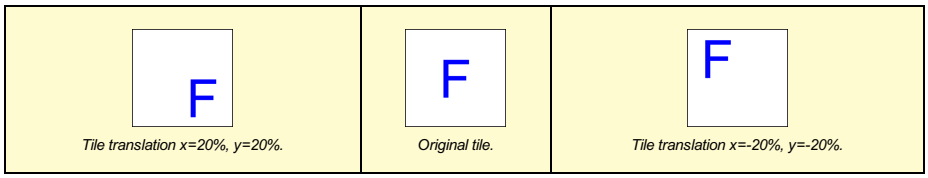
5. **Flip Up-Down** – reflect samples along the X axis.
6. **Flip Left-Right** – reflect samples along the Y axis.



No flips. | *Up-Down flip.* | *Left-Right flip.* | *Both flips.*

7. **Relative Translation** – translate samples by a random shift, defined as a percentage (between -ParameterValue and +ParameterValue) of the tile (in Detect Features, Locate Points and Detect Anomalies) or the image size (in Classify Object and Segment Instances). Works independently in both X and Y dimensions.

   In Detect Features, Locate Points and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.



Tile translation x=20%, y=20%. | *Original tile.* | *Tile translation x=-20%, y=-20%.*

   In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.

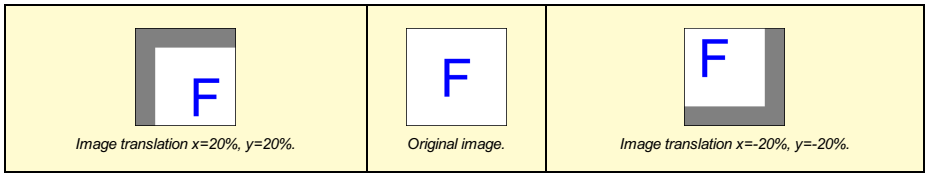
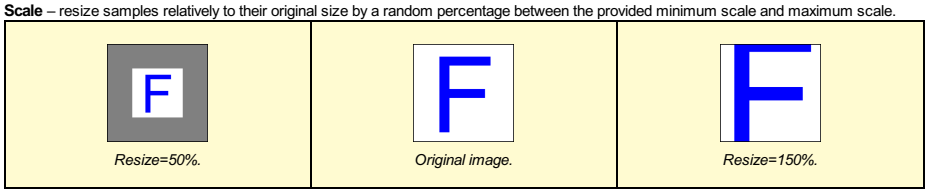


Image translation x=20%, y=20%. | *Original image.* | *Image translation x=-20%, y=-20%.*

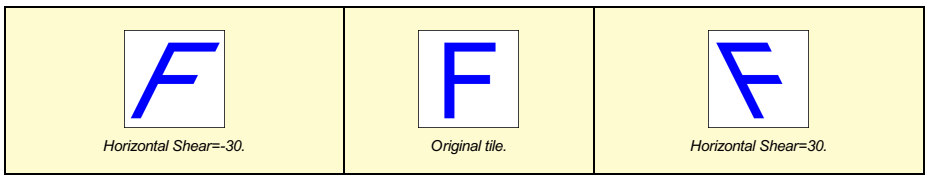
8. **Scale** – resize samples relatively to their original size by a random percentage between the provided minimum scale and maximum scale.



Resize=50%. | *Original image.* | *Resize=150%.*

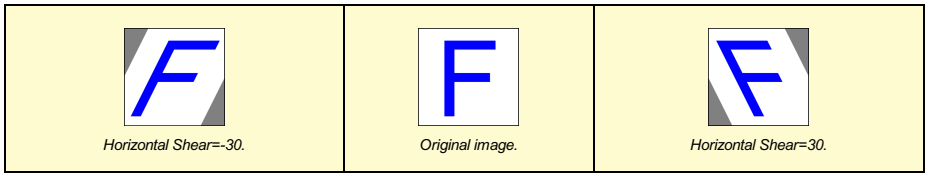
9. **Horizontal Shear** – shear samples horizontally by an angle between -ParameterValue and +ParameterValue. Measured in degrees.

   In Detect Features, Locate Points and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.



Horizontal Shear=-30. | *Original tile.* | *Horizontal Shear=30.*

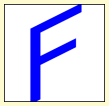
   In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.



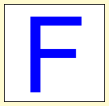
Horizontal Shear=-30. | *Original image.* | *Horizontal Shear=30.*

10. **Vertical Shear** – analogous to Horizontal Shear.

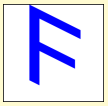
    In Detect Features, Locate Points, and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.



Vertical Shear=-30.



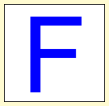
Original tile.



Vertical Shear=30.

In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.

Vertical Shear=-30.



Original image.

Vertical Shear=30.

Warning: the choice of augmentation options depends only on the task we want to solve. Sometimes they might be harmful for quality of a solution. For a simple example, the Rotation should not be enabled if rotations are not expected in a production environment. Enabling augmentations also increases the network training time (but does not affect execution time!)

2. Anomaly Detection

Aurora Vision Deep Learning provides three ways of defect detection:

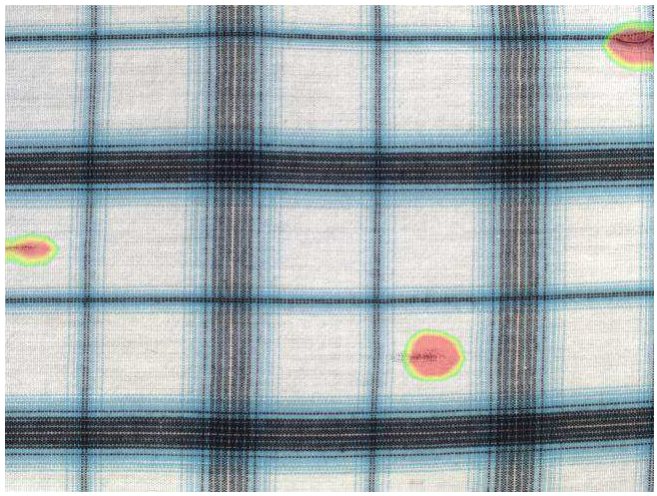
- DL_DetectAnomalies1
- DL_DetectAnomalies2 Single Class
- DL_DetectAnomalies2 Golden Template

The DL_DetectAnomalies1 (reconstructive approach) uses deep neural networks to remove defects from the input image by reconstructing the affected regions. It is used to analyze images in **fragments** of size determined by the Feature Size parameter.
This approach is based on reconstructing an image without defects and then comparing it with the original one. It filters out all patterns smaller than Feature Size that were not present in the training set.

The DL_DetectAnomalies2 **Single Class** uses a simpler algorithm than Golden Template. It uses less space and the iteration time is shorter. It can be used with less complex objects.

The DL_DetectAnomalies2 **Golden Template** is an appropriate method for positioned objects with complex details. The tool divides the images into regions and creates a separate model for each region. The tool has the **Texture Mode** dedicated for texture defects detection. It can be used for plain surfaces or the ones with a simple pattern.

To sum up, while choosing the tool for anomaly detection, first check the **Golden Template** with the **Texture Mode** on or off, depending on the object's kind. If the model takes too much space or the iteration is too long, please try the **Single Class** tool. If the object is complex and its position is unstable, please check the DL_DetectAnomalies1 approach.



An example of textile defect detection using the DL_DetectAnomalies2.

Parameters

- **Feature Size** is related to DL_DetectAnomalies1 and DL_DetectAnomalies2 **Single Class** approach. It corresponds to the expected defect size and it is the most significant one in terms of both quality and speed of inspection. It it is represented by a green square in the Image window of the Editor. The common denominator of all fragment based approaches is that the **Feature Size should be adjusted so that it contains common defects with some margin.**
  For DL_DetectAnomalies1 large Feature Size will cause small defects to be ignored, however the inference time will be shortened considerably. Heatmap precision will also be lowered. For DL_DetectAnomalies2 **Single Class** large Feature Size increases training as well as inference time and memory requirements. Consider using Downscale parameter instead of increasing the Feature Size.
- **Sampling Density** is related to DL_DetectAnomalies1 and DL_DetectAnomalies2 **Single Class** approach. It controls the spatial resolution of both training and inspection. The higher the density the more precise results but longer computational time. It is recommended to use the Low density only for well positioned and simple objects. The High density is useful when working with complex textures and highly variable objects.
- **Max Translation** is related to DL_DetectAnomalies2 **Golden Template** approach. It is the maximal position change tolerance. If the parameter increases, the working area of a small model enlarges and the number of the created small models decreases.
- **Model Complexity** is related to DL_DetectAnomalies2 **Golden Template** and DL_DetectAnomalies2 **Texture** approach. Greater value may improve model effectiveness, especially for complex objects, at the expense of memory usage and interference time.

Metrics

Measuring accuracy of anomaly detection tools is a challenging task. The most straightforward approach is to calculate the Recall/Precision/F1 measures for the whole images (classified as GOOD or BAD, without looking at the locations of the anomalies). Unfortunately, such an approach is not very reliable due to several reasons, including: (1) when we have a limited number of test images (like 20), the scores will vary a lot (like Δ=5%) when just one case changes; (2) very frequently the tools we test will find random false anomalies, but will not find the right ones - and still will get high scores as the image as a whole is considered correctly classified. Thus, it may be tempting to use annotated anomaly regions and calculate the per-pixel scores. However, this would be too fine-grained. For anomaly detection tasks we do not expect the tools to be necessarily very accurate in terms of the location of defects. Individual pixels do not matter much. Instead, we expect that the anomalies are detected "more or less" at the right locations. As a matter of fact, some tools which are not very accurate in general (especially those based on auto-encoders) will produce relatively accurate outlines for the anomalies they find, while the methods based on one-class classification will usually perform better in general, but the outlines they produce will be blurred, too thin or too thick.

For these reasons, we introduced an intermediate approach to calculation of Recall. Instead of using the per-image or the per-pixel methods, we use a per-region one. Here is how we calculate Recall:

- For each anomaly region we check if there is any single pixel in the heatmap above the threshold. If it is, we increase **TP** (the number of True Positives) by one. Otherwise, we increase **FN** (the number of False Negatives) by one.
- Then we use the formula:

$$Recall = \frac{TP}{TP + FN}$$

The above method works for Recall, but cannot be directly applied to the calculation of Precision. Thus, for Precision we use a per-pixel approach, but it also comes with its own difficulties. First issue is that we often find ourselves having a lot of GOOD samples and a very limited set of BAD testing cases. This means unbalanced testing data, which in turn means that the Precision metric is highly affected with the overwhelming quantity of GOOD samples. The more GOOD samples we have (at the same amount of BAD samples), the lower Precision will be. It may be actually very low, often not reflecting the true performance of the tool. For that reason, we need to incorporate balancing into our metrics.

A second issue with Precision in real-world projects is that False Positives tend to naturally occur within BAD images, outside of the marked anomaly regions. This happens for several reasons, but is repeatable among different projects. Sometimes if there is a defect, it often means that something was broken and other parts of the object may be slightly affected too, sometimes in a visible way, sometimes with a level of ambiguity. And quite often the objects under inspection simply get affected by the process of artificially introducing defects (like someone is touching a piece of fabric and accidentally causes wrinkles that would normally not occur). For this reason, we calculate the per-pixel False Negatives only on GOOD images.

The complete procedure for calculation of Precision is:

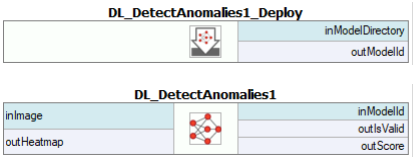
- We calculate the average **pp_TP** (the number of per-pixel True Positives) across all BAD testing samples.
- We calculate the average **pp_FP** (the number of per-pixel False Positives) across all GOOD testing samples.
- Then we use the formula:

$$Precision = \frac{\overline{pp\_TP}}{\overline{pp\_TP} + \overline{pp\_FP}}$$

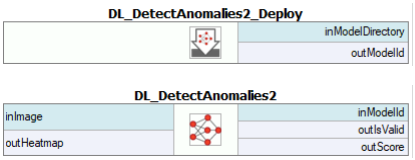
Finally we calculate the F1 score in the standard way, for practical reasons neglecting the fact that the Recall and Precision values that we unify were calculated in different ways. We believe that this metric is best for practical applications.

Model Usage

In Detect Anomalies 1 variant, a model should be loaded with DL_DetectAnomalies1_Deploy prior to executing it with DL_DetectAnomalies1. Alternatively, the model can be loaded directly by DL_DetectAnomalies1 filter, but it will then require time-consuming initialization in the first program iteration.



In Detect Anomalies 2 variant, a model should be loaded with DL_DetectAnomalies2_Deploy prior to executing it with DL_DetectAnomalies2. Alternatively, model can be loaded directly by DL_DetectAnomalies2 filter, but it will then require time-consuming initialization in the first program iteration.



Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

3. Feature Detection (segmentation)

This technique is used to detect pixel-wise regions corresponding to defects or – in a general sense – to any image features. A feature here may be also something like the roads on a satellite image or an object part with a characteristic surface pattern. Sometimes it is also called pixel labeling as it assigns a class label to each pixel, but it does not separate instances of objects.

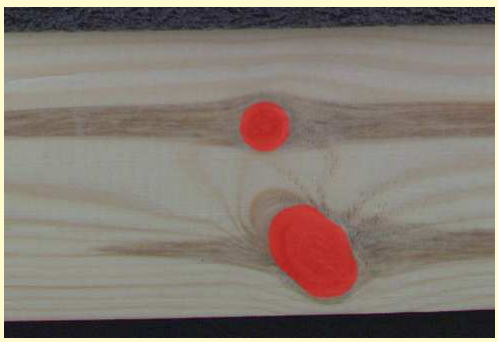
Training Data

Images loaded to the Editor of DL_DetectFeatures can be of different sizes and can have different ROIs defined. However, it is important to ensure that the scale and the characteristics of the features are consistent with that of the production environment.

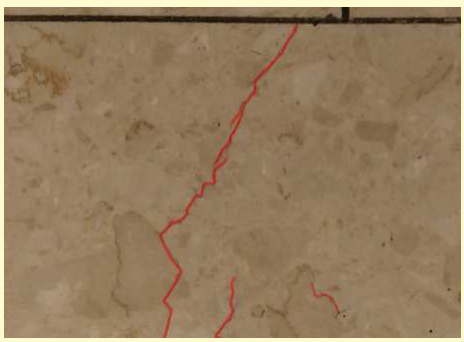
The features can be marked using an intuitive interface in the Editor or can be imported as masks from a file.

Each and every feature should be marked on all training images, or the ROI should be limited to include only marked defects. Incompletely or inconsistently marked features are one of the main reasons of poor accuracy. REMEMBER: If you leave even a single piece of some feature not marked, it will be used as a negative sample and this will highly confuse the training process!

The marking precision should be adjusted to the application requirements. The more precise marking the better accuracy in the production environment. While marking with low precision it is better to mark features with some excess margin.



An example of wood knots marked with low precision.



An example of tile cracks marked with high precision.

Multiple classes of features

It is possible to detect many classes of features separately using one model. For example, road and building like in the image below. Different features may overlap but it is usually not recommended. Also, it is not recommended to define more than a few different classes in a single model. On the other hand, if there are two features that may be mutually confusing (e.g. roads and rivers), it is recommended to have separate classes for them and mark them, even if one of the classes is not really needed in the results. Having the confusing feature clearly marked (and not just left as the background), the neural network will focus better on avoiding misclassification.

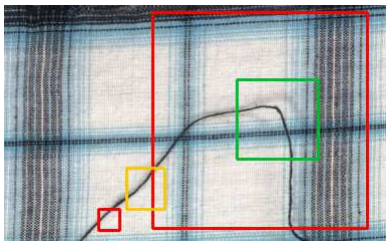
An example of marking two different classes (red roads and yellow buildings) in the one image.

Patch Size

Detect Features is an end-to-end segmentation tool which works best when analysing an image in a medium-sized square window. The size of this window is defined by the Patch Size parameter. It should be not too small, and not too big. Typically much bigger than the size (width or diameter) of the feature itself, but much less than the entire image. In a typical scenario the value of 96 or 128 works quite well.

Performance Tip 1: a larger Patch Size increases the training time and requires more GPU memory and more training samples to operate effectively. When Patch Size exceeds 128 pixels and still looks too small, it is worth considering the **Downsample** option.

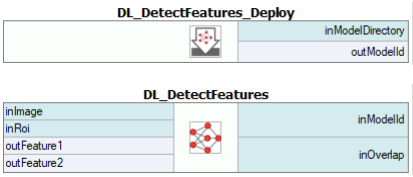
Performance Tip 2: if the execution time is not satisfying you can set the inOverlap filter input to False. It should speed up the inspection by 10-30% at the expense of less precise results.



Examples of Patch Size: too large or too small (red), maybe acceptable (yellow) and good (green). Remember that this is just an example and may vary in other cases.

Model Usage

A model should be loaded with DL_DetectFeatures_Deploy filter before using DL_DetectFeatures filter to perform segmentation of features. Alternatively, the model can be loaded directly by DL_DetectFeatures filter, but it will result in a much longer time of the first iteration.



Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- To shorten feature segmentation process you can disable **inOverlap** option. However, in most cases, it decreases segmentation quality.
- Feature segmentation results are passed in a form of bitmaps to **outHeatmaps** output as an array and **outFeature1**, **outFeature2**, **outFeature3** and **outFeature4** as separate images.

4. Object Classification

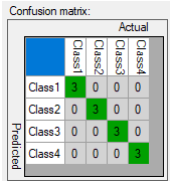
This technique is used to identify the class of an object within an image or within a specified region.

The Principle of Operation

During the training phase, the object classification tool learns representation of user defined classes. The model uses generalized knowledge gained from samples provided for training, and aims to obtain good separation between the classes.

Result of classification after training.

After a training process is completed, the user is presented with a confusion matrix. It indicates how well the model separated the user defined classes. It simplifies identification of model accuracy, especially when a large number of samples has been used.



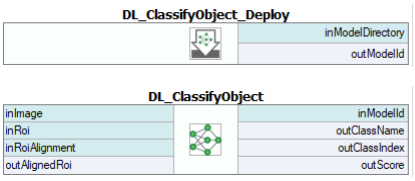
Confusion matrix presents correct (diagonal) and incorrect assignment of samples to the user defined classes.

Training Parameters

In addition to the default training parameters (list of parameters available for all Deep Learning algorithms), the DL_ClassifyObject tool provides a **Detail Level** parameter which enables control over the level of detail needed for a particular classification task. For majority of cases the default value of 1 is appropriate, but if images of different classes are distinguishable only by small features (e.g. granular materials like flour and salt), increasing value of this parameter may improve classification results.

Model Usage

A model should be loaded with DL_ClassifyObject_Deploy filter before using DL_ClassifyObject filter to perform classification. Alternatively, model can be loaded directly by DL_ClassifyObject filter, but it will result in a much longer time of the first iteration.



Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- Classification results are passed to **outClassName** and **outClassIndex** outputs.
- The score value **outScore** indicates the confidence of classification.

5. Instance Segmentation

This technique is used to locate, segment and classify one or multiple objects within an image. The result of this technique are lists with elements describing detected objects – their bounding boxes, masks (segmented regions), class IDs, names and membership probabilities.

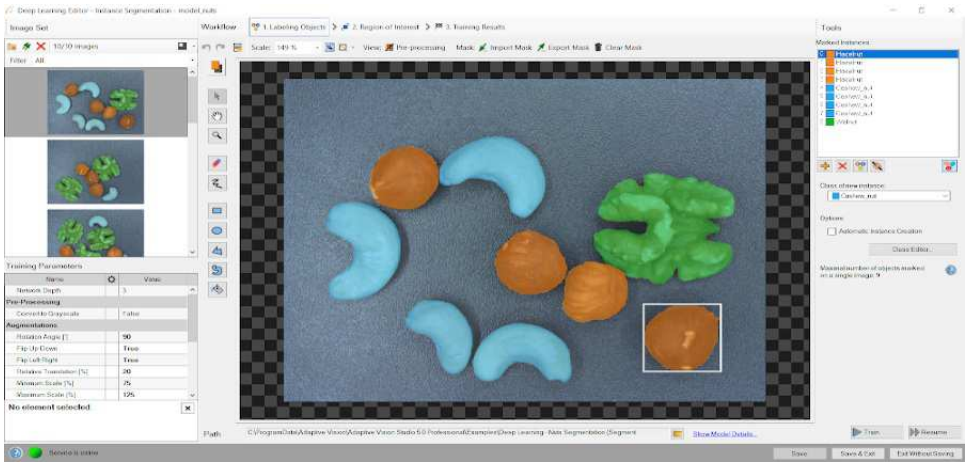
Note that in contrary to feature detection technique, instance segmentation detects individual objects and may be able to separate them even if they touch or overlap. On the other hand, instance segmentation is not an appropriate tool for detecting features like scratches or edges which may possibly have no object-like boundaries.

Original image.

Visualized instance segmentation results.

Training Data

The training phase requires the user to draw regions corresponding to objects on an image and assign them to classes.



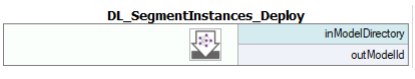
Editor for marking objects.

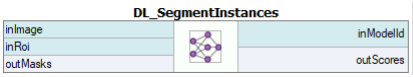
Training Parameters

Instance segmentation training adapts to the data provided by the user and does not require any additional training parameters besides the default ones.

Model Usage

A model should be loaded with DL_SegmentInstances_Deploy filter before using DL_SegmentInstances filter to perform classification. Alternatively, model can be loaded directly by DL_SegmentInstances filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- To set minimum detection score **inMinDetectionScore** parameter can be used.
- Maximum number of detected objects on a single image can be set with **inMaxObjectsCount** parameter. By default it is equal to the maximum number of objects in the training data.
- Results describing detected objects are passed to following outputs:
  - bounding boxes: **outBoundingBoxes**,
  - class IDs: **outClassIds**,
  - class names: **outClassNames**,
  - classification scores: **outScores**,
  - masks: **outMasks**.

6. Point Location

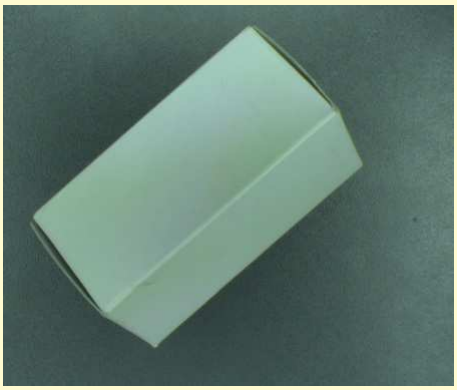
This technique is used to precisely locate and classify key points, characteristic parts and small objects within an image. The result of this technique is a list of predicted point locations with corresponding class predictions and confidence scores.

When to use point location instead of instance segmentation:

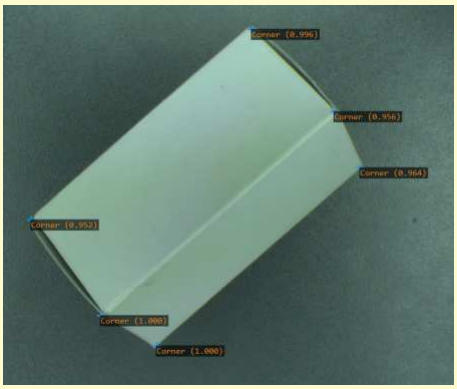
- precise location of key points and distinctive regions with no strict boundaries,
- location and classification of objects (possibly very small) when their segmentation masks and bounding boxes are not needed (e.g. in object counting).

When to use point location instead of feature detection:

- coordinates of key points, centroids of characteristic regions, objects etc. are needed.



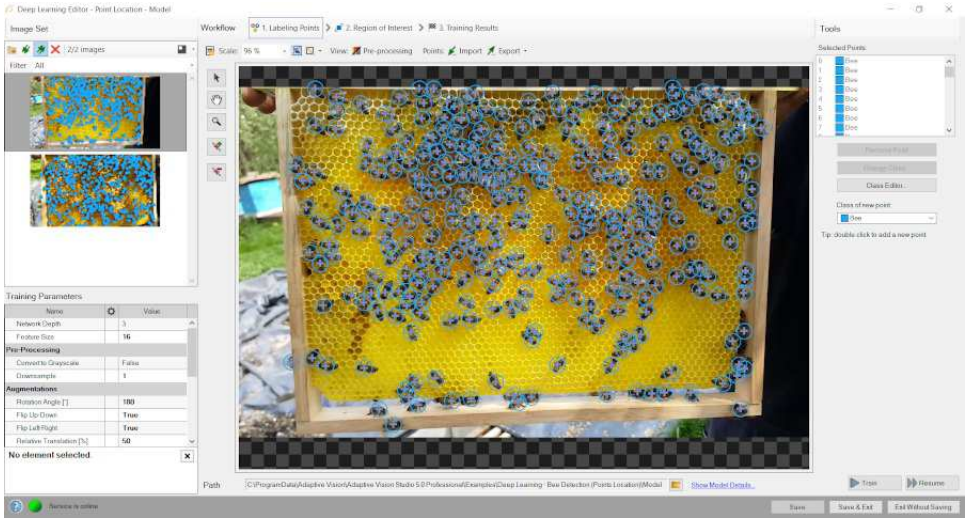
Original image.



Visualized point location results.

Training Data

The training phase requires the user to mark points of appropriate classes on the training images.



Editor for marking points.

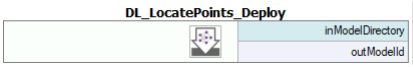
Feature Size

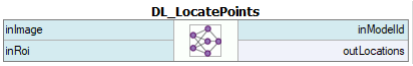
In the case of the Point Location tool, the Feature Size parameter corresponds to the size of an object or characteristic part. If images contain objects of different scales, it is recommended to use a Feature Size slightly larger than the average object size, although it may require experimenting with different values to achieve the best possible results.

Performance tip: a larger feature size increases the training time and needs more memory and training samples to operate effectively. When feature size exceeds 64 pixels and still looks too small, it is worth considering the **Downsample** option.

Model Usage

A model should be loaded with DL_LocatePoints_Deploy filter before using DL_LocatePoints filter to perform point location and classification. Alternatively, model can be loaded directly by DL_LocatePoints filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- To set minimum detection score **inMinDetectionScore** parameter can be used.
- **inMinDistanceRatio** parameter can be used to set minimum distance between two points to be considered as different. The distance is computed as MinDistanceRatio * FeatureSize. If the value is not enabled, the minimum distance is based on the training data.
- To increase detection speed but with potentially slightly worse precision **inOverlap** can be set to False.
- Results describing detected points are passed to following outputs:
  - point coordinates: **outLocations**,
  - class IDs: **outClassIds**,
  - class names: **outClassNames**,
  - classification scores: **outScores**.

7. Locating objects

This technique is used to locate and classify one or multiple objects within an image. The result of this technique is a list of rectangles bounding the predicted objects with corresponding class predictions and confidence scores.

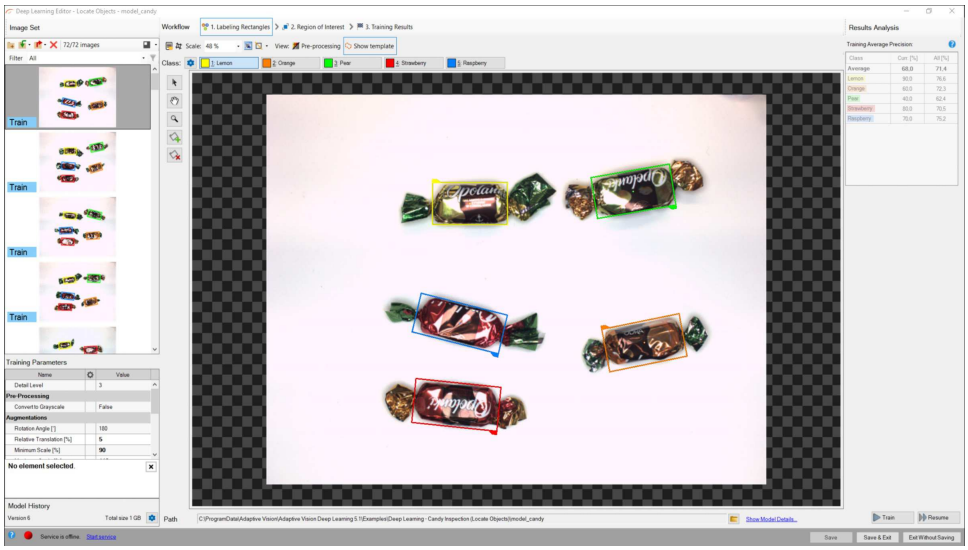
The tool returns the rectangle region containing the predicted objects and showing their approximate location and orientation , but it doesn't return the precise position of the key points of the object or the segmented region. It is an intermediate solution between the Point Location and the Instance Segmentation.

Original image.

Visualized object location results.

Training Data

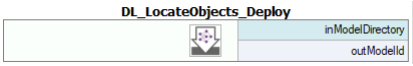
The training phase requires the user to mark rectangles bounding objects of appropriate classes on the training images.

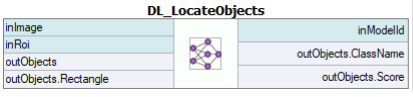


Editor for marking objects.

Model Usage

A model should be loaded with DL_LocateObjects_Deploy filter before using DL_LocateObjects filter to perform object location and classification. Alternatively, model can be loaded directly by DL_LocateObjects filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

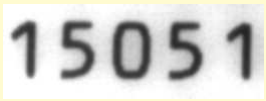
Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- To set minimum detection score **inMinDetectionScore** parameter can be used.
- Results describing detected objects are passed to the object output: **outObjects**.

8. Reading Characters

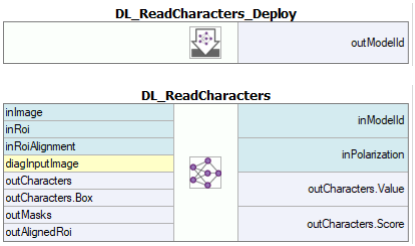
This technique is used to locate and recognize characters within an image. The result is a list of found characters.

This tool uses the pretrained model and cannot be trained.

 *Original image.*	 *Visualized results of the read characters.*

Model Usage

A model should be loaded with DL_ReadCharacters_Deploy filter before using DL_ReadCharacters filter to perform recognition. Alternatively, model can be loaded directly by DL_ReadCharacters filter, but it will result in a much longer time of the first iteration.



Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

- To limit the area of the image analysis and/or to set a text orientation you can use **inRoi** input.
- The average size (in pixels) of characters in the analysed area should be set with **inCharHeight** parameter.
- To improve a performance with a font with exceptionally thin or wide characters you can use **inWidthScale** input. To some extent, it may also help in a case of characters being very close to each other.
- To restrict set of recognized characters use **inCharRange** parameter.

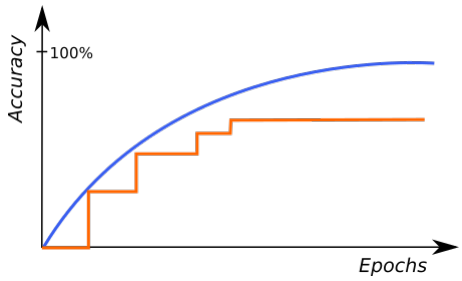
9. Troubleshooting

Below you will find a list of most common problems.

1. Network overfitting

A situation when a network loses its ability to generalize over available problems and focuses only on test data.

Symptoms: during training, the validation graph stops at one level and training graph continues to rise. Defects on training images are marked very precisely, but defects on new images are marked poorly.



A graph characteristic for network overfitting.

Causes:

- The number of test samples is too small.
- Training time is too long.

Possible solutions:

- Provide more real samples of different objects.
- Use more augmentations.
- Reduce Network Depth.

2. Susceptibility to changes in lighting conditions

Symptoms: network is not able to process images properly when even minor changes in lighting occur.

Causes:

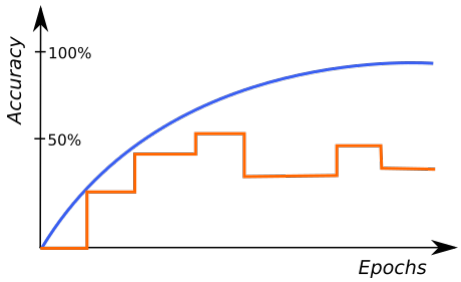
- Samples with variable lighting were not provided.

Solution:

- Provide more samples with variable lighting.
- Enable "Luminance" option for automatic lighting augmentation.

3. No progress in network training

Symptoms — even though the training time is optimal, there is no visible training progress.



Training progress with contradictory samples.

Causes:

- The number of samples is too small or the samples are not variable enough.
- Image contrast is too small.
- The chosen network architecture is too small.
- There is contradiction in defect masks.

Solution:

- Modify lighting to expose defects.
- Remove contradictions in defect masks.

Tip: Remember to mark all defects of a given type on the input images or remove images with unmarked defects. Marking only a part of defects of a given type may negatively influence the network learning process.

4. Training/sample evaluation is very slow

Symptoms — training or sample evaluation takes a lot of time.

Causes:

- Resolution of the provided input images is too high.
- Fragments that cannot possibly contain defects are also analyzed.

Solution:

- Enable "Downsample" option to reduce the image resolution.
- Limit ROI for sample evaluation.
- Use lower Network Depth

See Also

- Deep Learning Service Configuration - installation and configuration of Deep Learning service,
- Creating Deep Learning Model - how to use Deep Learning Editor.

11.

Table of content:

Interfacing Photoneo to Aurora Vision Studio

Purpose and requirement

This document explains how to interface a PhoXi 3D Scanner to Aurora Vision Studio.

A PhoXi 3D Scanner is an advanced sensor created by Photoneo. It is purposed for 3D machine vision and processing point clouds. The PhoXi 3D Scanner has many functionalities including:

- Scanning objects and representing them as a point cloud or intensity images,
- Various representations of scans.

Required equipment:

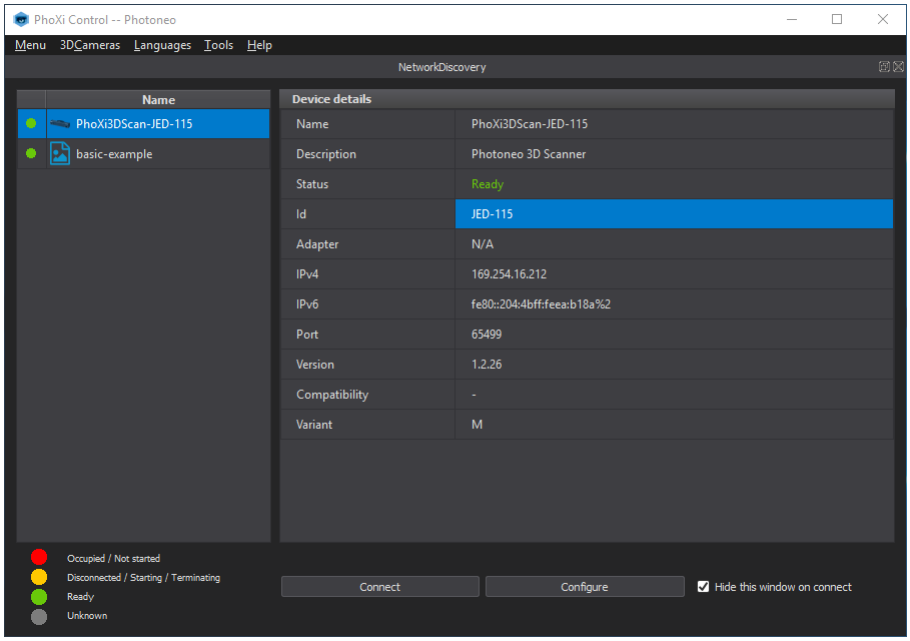
- PhoXi Control v1.2.35 or later
- Aurora Vision Studio 4.11 Professional or later

Getting started with a PhoXi 3D Scanner in Aurora Vision Studio

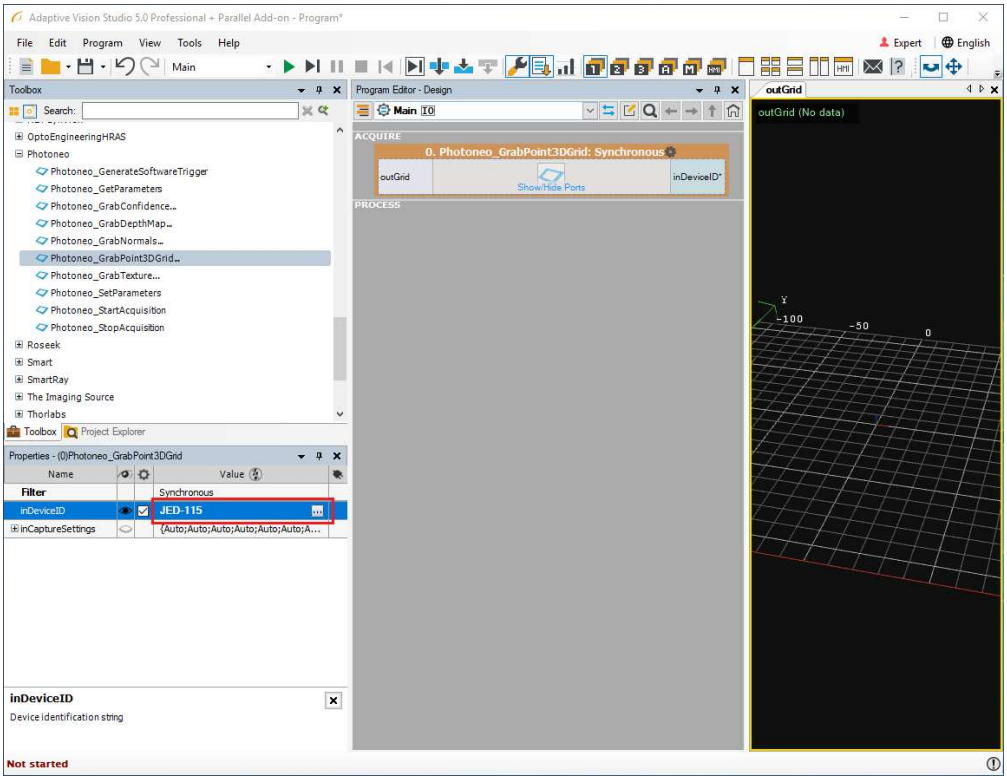
At the beginning download proper version of PhoXi Control from producer's website and install it.

Follow these steps:

1. Connect a PhoXi 3D Scanner to the PC running Aurora Vision Studio Professional.
2. Power up the PhoXi 3D Scanner, connect it to the PC via Ethernet interface.
3. Open PhoXi Control application.
4. Copy ID of the scanner you would like to use in Aurora Vision Studio.



5. To use PhoXi 3D Scanner its Status should be "Ready"
6. Full list of filters intended to work with PhoXi 3D Scanner you can find under this link. In this particular example Photoneo_GrabPoint3DGrid filter will be used.
7. At the beginning create simple program with Photoneo_GrabPoint3DGrid filter to test the connection. Find Photoneo_GrabPoint3DGrid filter in the Filter Catalog or in the Toolbox (category Image Acquisition (Third Party)), then drag it and drop it to the main program.
8. Paste ID (from step 4) into **inDevice** input. If you do not have access to any Photoneo scanner, but you would like to test the connection between PhoXi Control and Aurora Vision Studio, please enter "InstalledExamples-basic-example" into **inDevice** input.

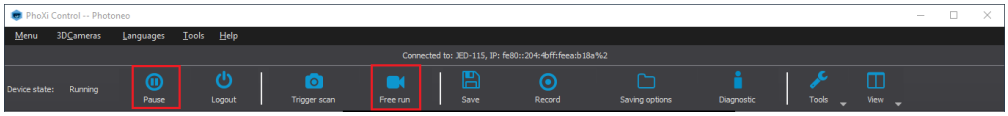


9. Connect **outGrid** output to preview window and run program. If everything is done correctly, you should get Point3DGrid with scanned object.

Troubleshooting

If you had any problems during the connection process, you could find some solutions here.

- Make sure you can grab point cloud in PhoXi Control application. If you cannot, please check connection and change the properties of acquisition.
- Make sure you are using proper version of Aurora Vision Studio and PhoXi Control application (from chapter Purpose and requirement)
- Make sure that device is in running mode. If it is necessary Unpause device in PhoXi Control and use Free run, then you should be able to grab Point3DGrid in Aurora Vision Studio.



- You are not be able to change parameters in PhoXi Control application, while a program in Aurora Vision Studio is running and acquiring data from a scanner. You can stop a program by clicking Stop button or using Shift + F5 key combination.

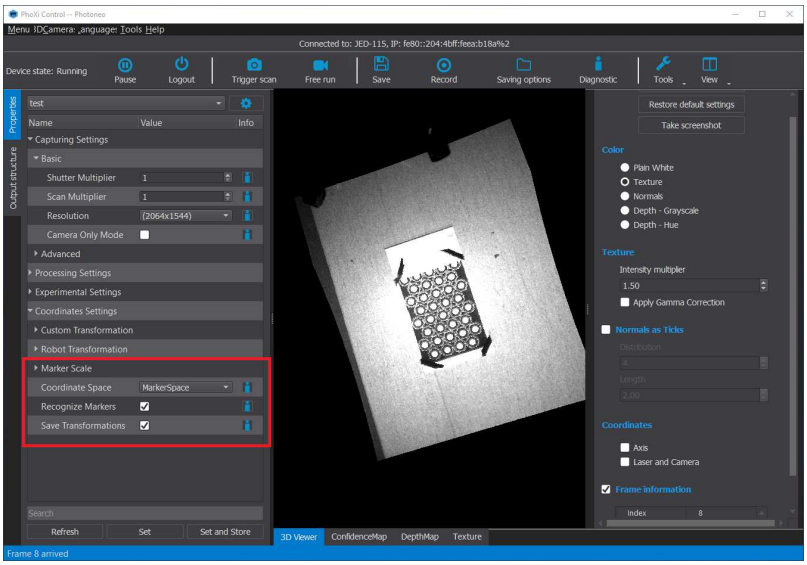
If you encounter a problem that has not been mentioned above, please do not hesitate to contact us, so that we could investigate it and add the description of its solution to this section.

Calibration

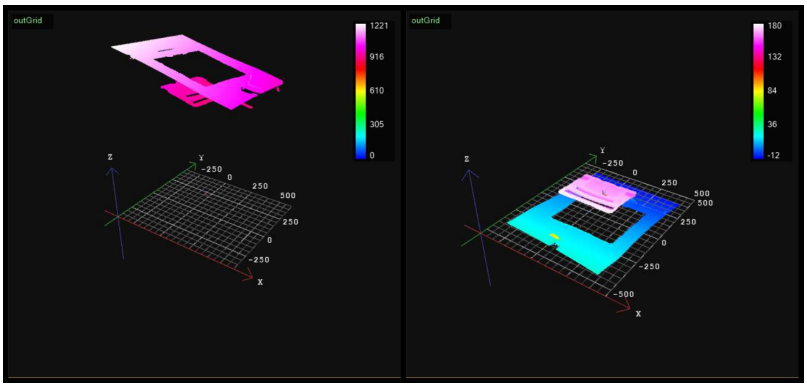
Point's coordinate system is defined by scanner's camera position and its specific angle depending on PhoXi 3D Scanner type. You can read more about Coordinate spaces of Photoneo scanners in this document

You can align coordinate system position by following this instruction. After completing steps outlined by the manufacturer, the coordinate system will be changed also in case of scans acquired with Aurora Vision Studio.

To make an calibration of PhoXi 3D Scanner you have to print calibration board. Then navigate to Properties -> Coordinates Settings -> Marker Scale in PhoXi Control and change parameters to these like on the picture below. Acquire scan of points cloud using Trigger scan and then click Set and Store.

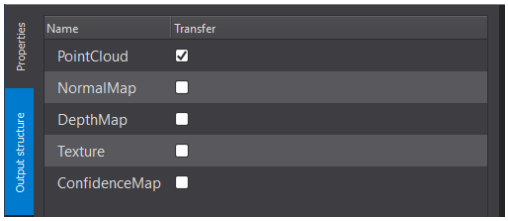


Below picture represents points cloud before and after calibration.



Setting parameters and maps reading

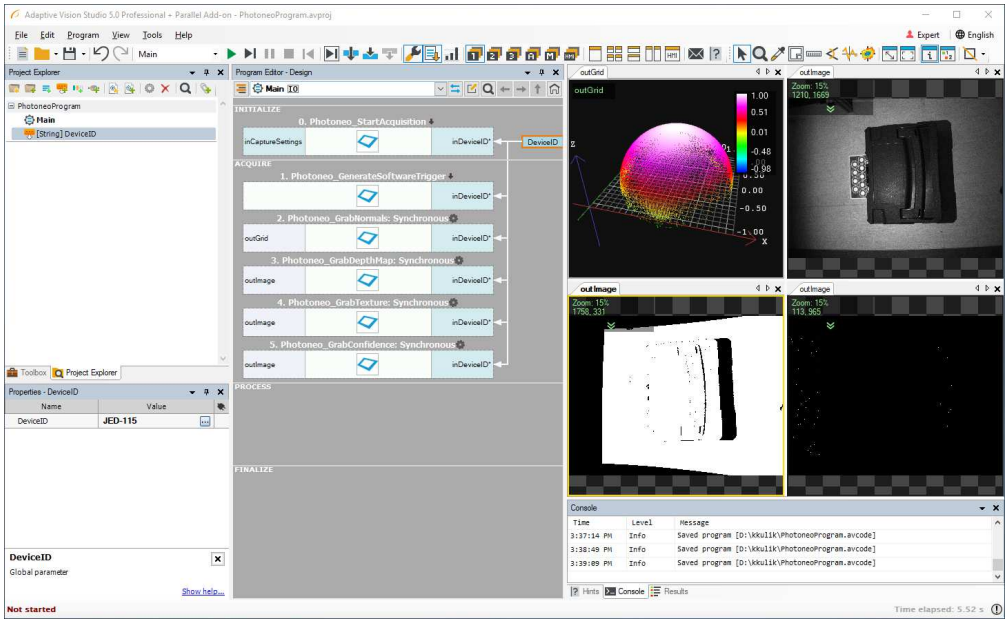
Default parameters of PhoXi 3D Scanner are shown below.



These settings does not allow to send Normal maps, Depth maps, Confidence maps and Textures to Aurora Vision Studio application. You can tick checkboxes in PhoXi Control software or you can set these parameters in Aurora Vision Studio.

To make it possible in Aurora Vision Studio, you need to change the parameters using **Photoneo_SetParameters** filter, which should be executed at the very beginning of program or between **Photoneo_StopAcquisition** and **Photoneo_StartAcquisition** filters.

If parameters, responsible for maps reading, will be set as True, you can use proper filters (**Photoneo_GrabNormals**, **Photoneo_GrabTexture**, **Photoneo_GrabDepthMap**, **Photoneo_GrabConfidence**) to acquire respective maps to Aurora Vision Studio application.



Interfacing Hilscher card (EtherNet/IP) to Aurora Vision Studio

Purpose and requirement

This document explains how to configure EtherNet/IP PLC with Aurora Vision Studio using Hilscher EtherNet/IP card.

Required equipment:

- Supported Hilscher EtherNet/IP card
- Aurora Vision Studio 5.1 or later

Hardware connection

All devices must work in a common LAN network - henceforth called shared network. In most cases they are connected to each other through a network switch.

The devices shall be connected as follows:

- Master device e.g. PLC to the shared network
- Hilscher card to PC
- Hilscher card to the shared network
- PC's network card to shared network

Before proceeding to the further point, make sure that all devices are powered up.

Configuring Hilscher Devices

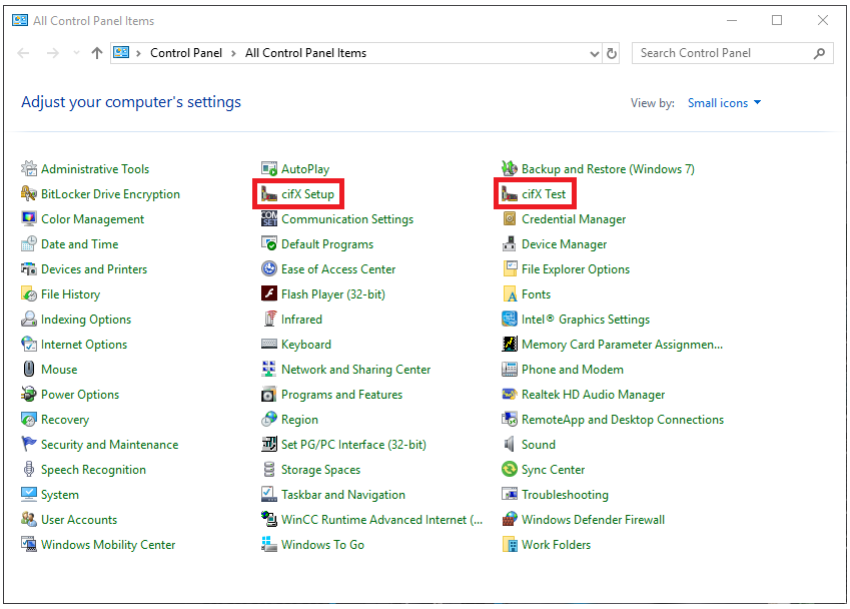
Be sure to have the newest version of the following drivers:

- List of the EtherNet/IP drivers with firmware: download link (for CIFX 50E-RE download V2.x version)
- cifX Driver Setup software: download link
- SYCON.net to configure Hilscher card and generate configuration files: download link

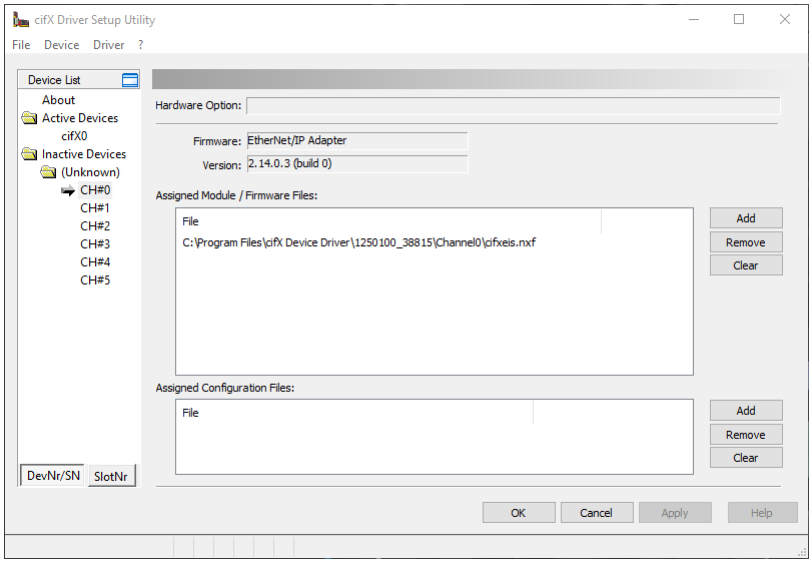
Make sure you have the recommended driver installed, the newest SYCON.net and firmware prepared.

1) Configuration using cifX Driver Setup Software

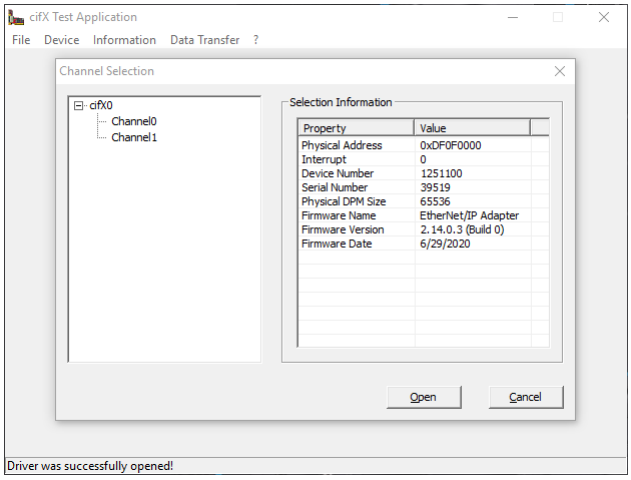
The easiest way to install the firmware to the cifX Driver is using the cifX Driver Setup Software. You can find tools for this software for instance using Control Panel menu as in the picture below.



Using cifX Setup application select the channel you would like to configure (usually channel - CH#0) and remove preexisting firmware by clicking "Clear". Then click "Add" to add a new firmware file and navigate to the downloaded Ethernet/IP Adapter firmware (path \COMSOL-EIS V2.15.0.1\Firmware\cifX). Then click "Apply" and finish configuration with "OK".

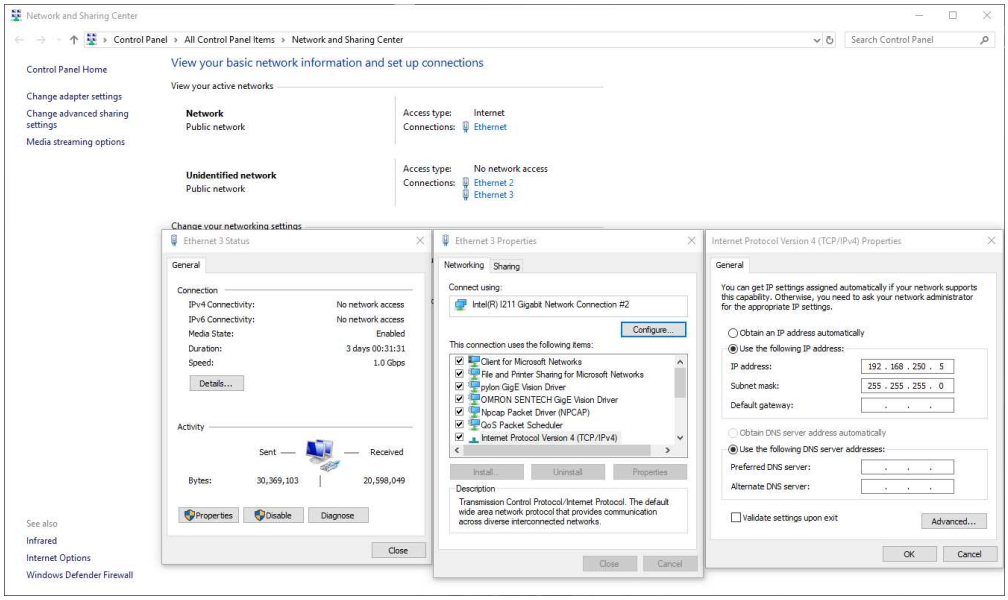


To be sure that firmware is installed properly you can open cifX Test software and then navigate to Device -> Open to achieve the result like below.



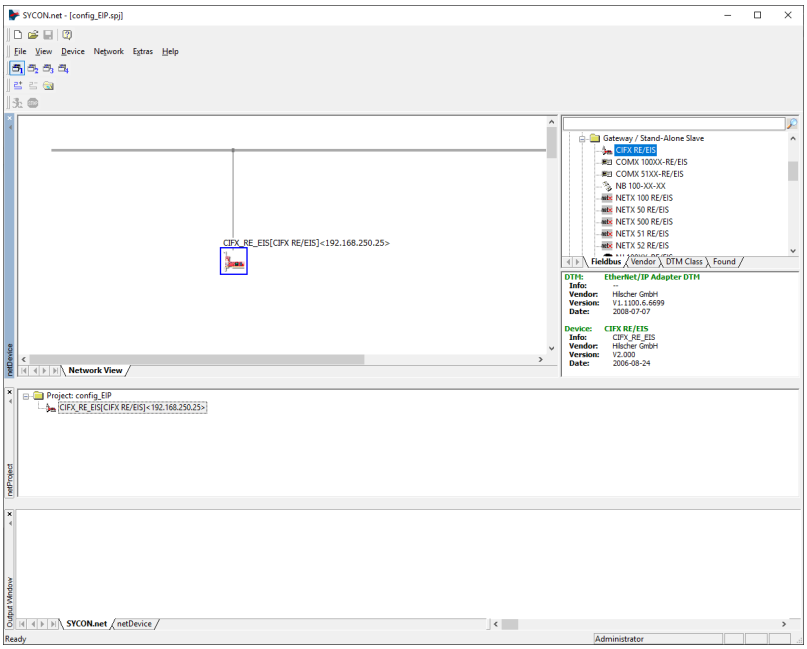
2) Configuration of the network card

It might be necessary to change the IP of the network card from dynamic to static. In this tutorial the following settings are used.

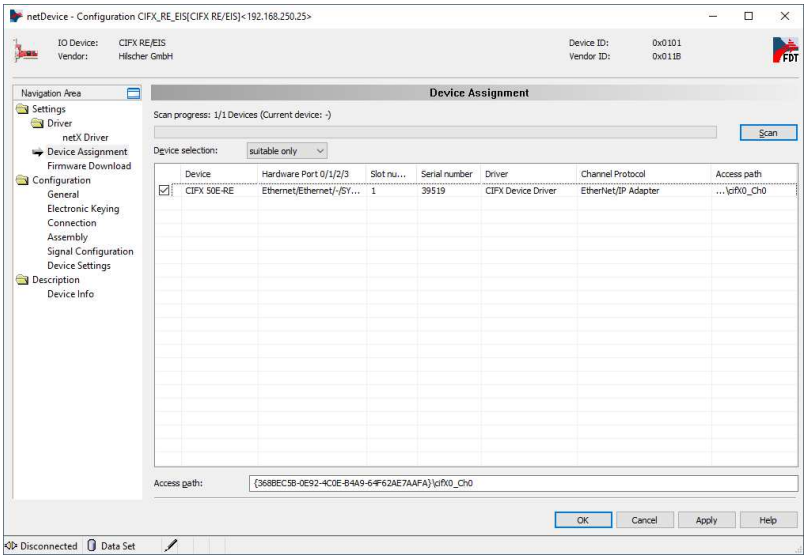


3) Configuration using SYCON.net

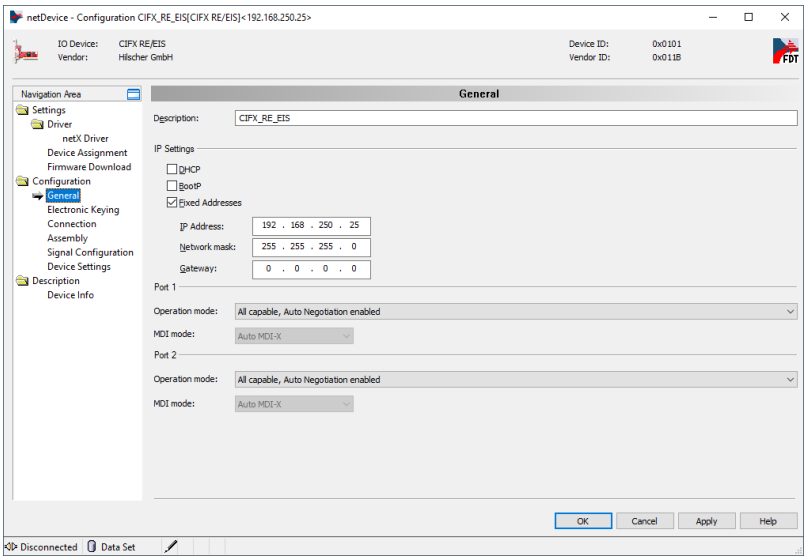
Configuration files are generated in SYCON.net software. Firstly, pick a proper device and drag it to the gray bus on the "Network View". This device you can find navigating to "EtherNet/Ip -> Gateway / Stand - Alone Slave -> CIFX RE/EIS".



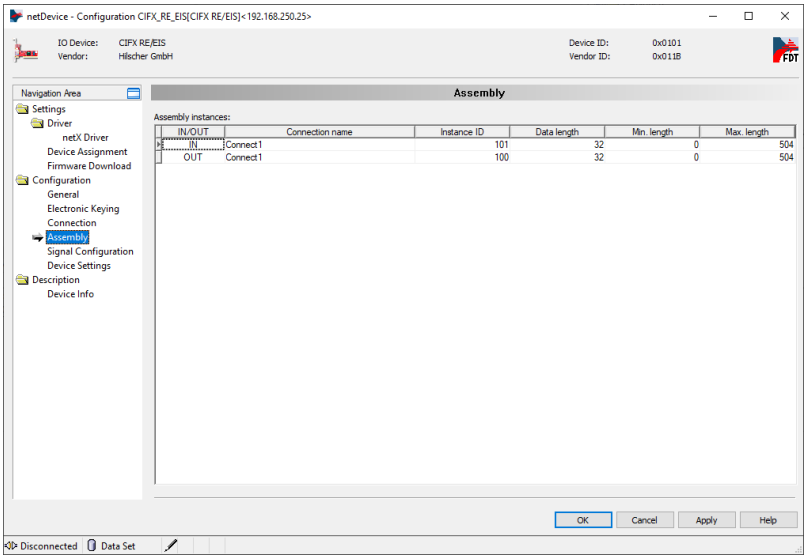
To open the configuration card double click on dropped icon and in the popped-up menu navigate to "Settings/Device Assignment". Then scan for devices by clicking "Scan", mark found one it with a tick like on the image below and then click "Apply".



In "Configuration/General" you can change the IP of the device from DHCP to Static like on the below screen.

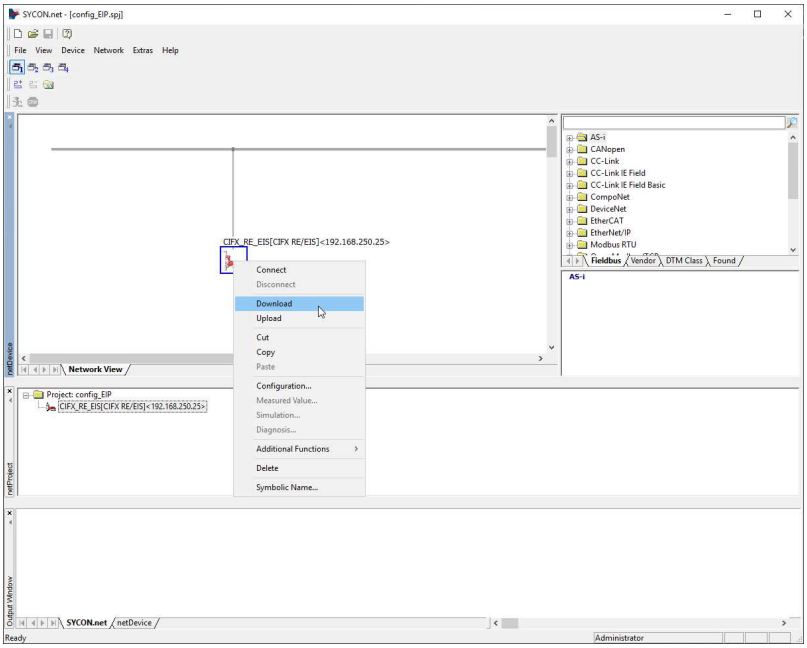


In "Configuration/Assembly" you can change the configuration of memory blocks for IN/OUT operations. By default, the Data length is set to 32 bytes. All subsequent PLC data blocks mentioned in this document are configured for 32 bytes. In case you need memory blocks of a different size, you can change the Data length from 0 up to 504 bytes.

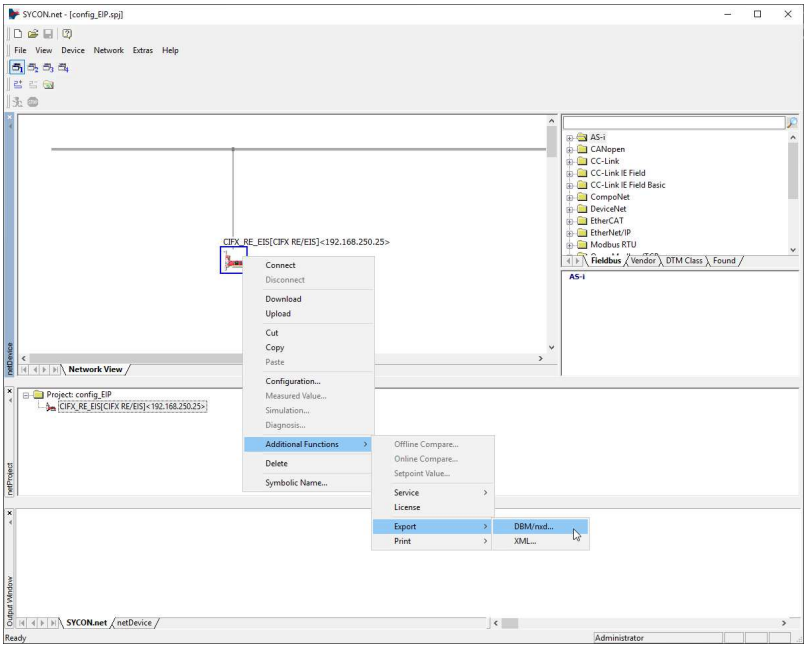


Confirm all changes by clicking "Apply" and then click "OK".

Then you have to download the configuration to the device like on the screen below.



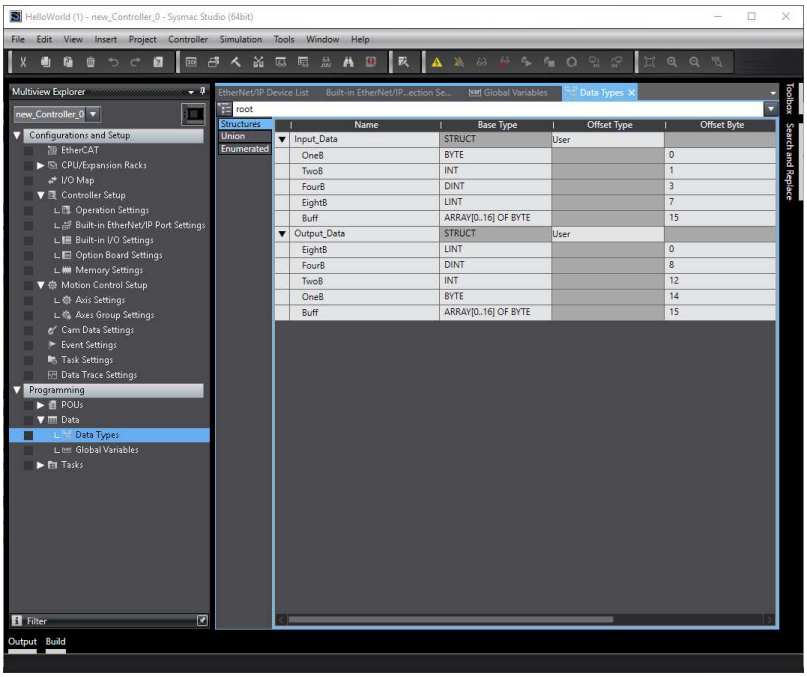
The final step is to generate configuration files for Aurora Vision Studio. You can do this by right-clicking on the device icon, then navigating to "Additional Functions -> Export -> DBM/nxd..", entering your configuration name and clicking "Save". You can now close SYCON.net for this example, **remember to save your project before.**



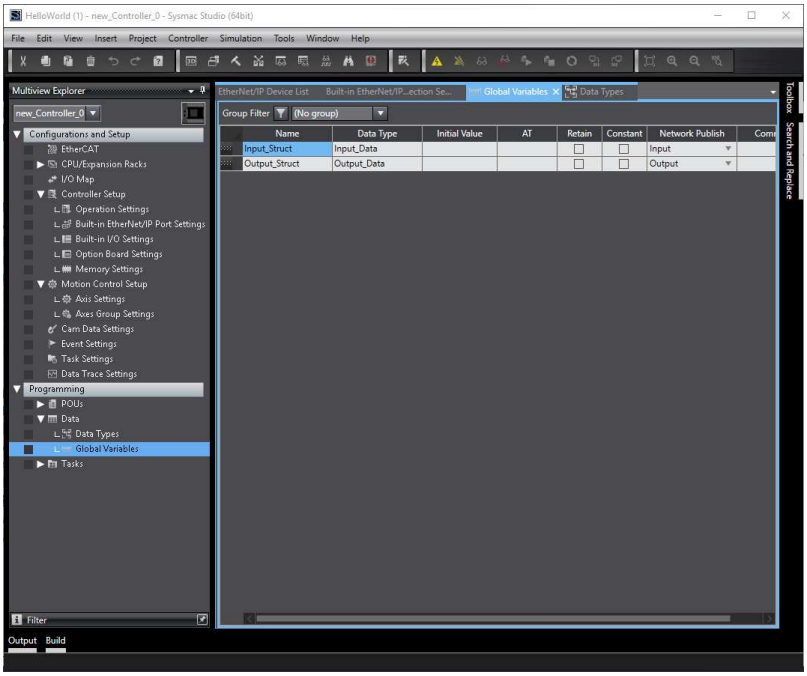
4) Example configuration of EtherNet/IP PLC

Below you will find an example configuration process for EtherNet/IP PLC Omron NXJP2-9024DT in Sysmac Studio software.

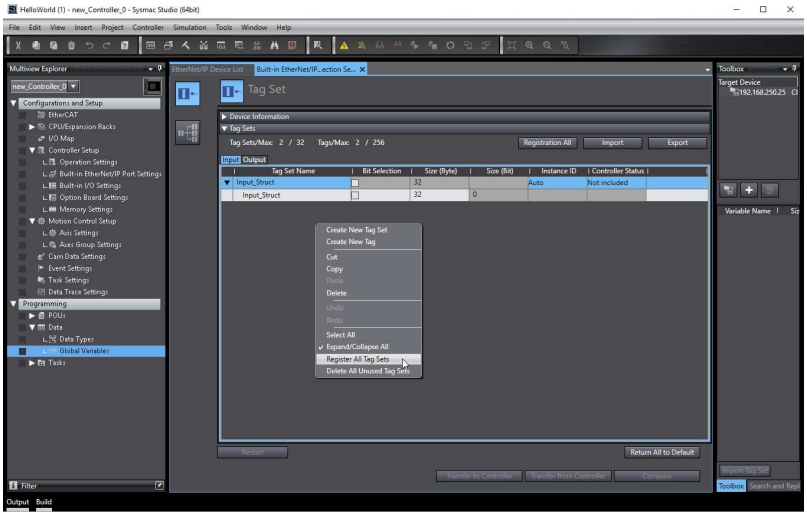
First of all, there were defined two structures Data Types: **Input_Data** and **Output_Data** for further tests of Aurora Vision Studio macrofilters. Information about offsets will be further used in Aurora Vision Studio.



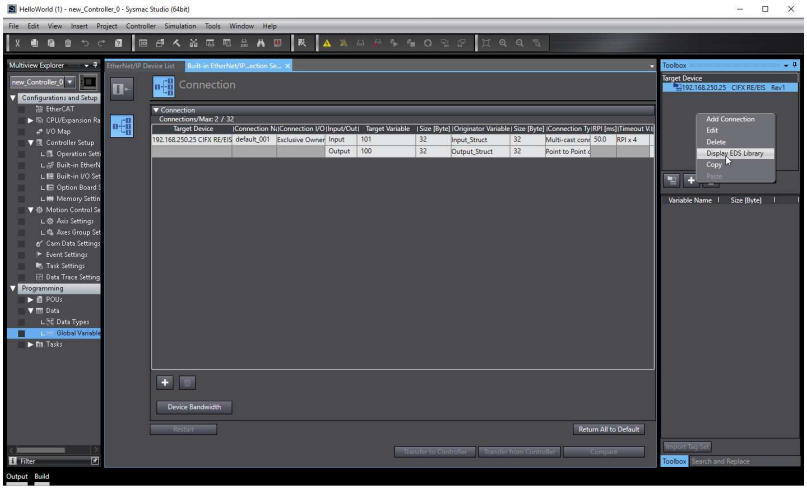
There were created two global variables **Input_Struct** and **Output_Struct** using previously created structures to handle communication with Hilscher card.



Then navigate to (Tools -> EtherNet/IP Connection Setting) and "Register All Tag Sets".

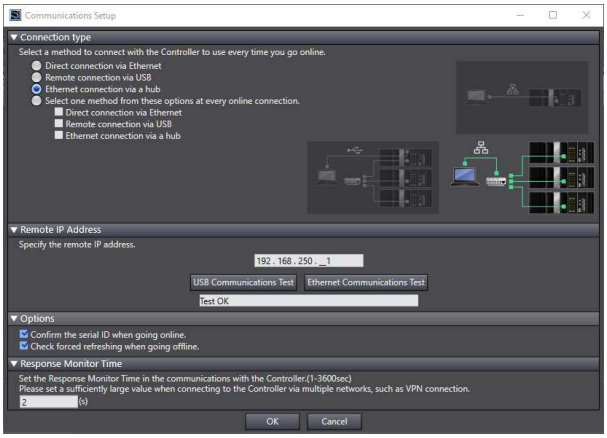


Add support for Hilscher card using EDS Library configuration (path \COMSOL-EIS V2.15.0.1\EDS). Then add "CIFX RE/EIS" and configure Input and Output like below. **It is necessary to match the configuration from SYCON.net.**

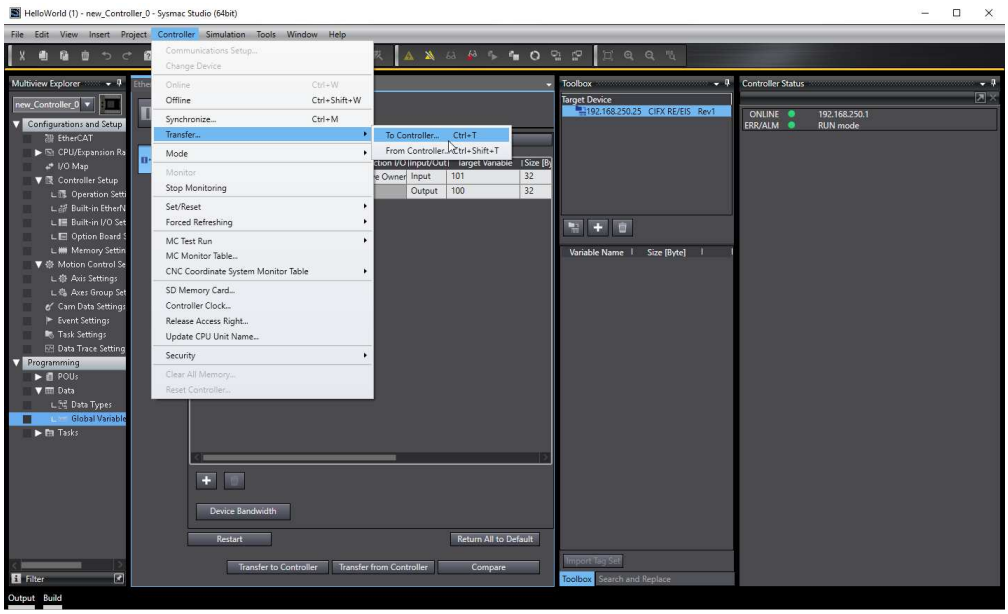


.

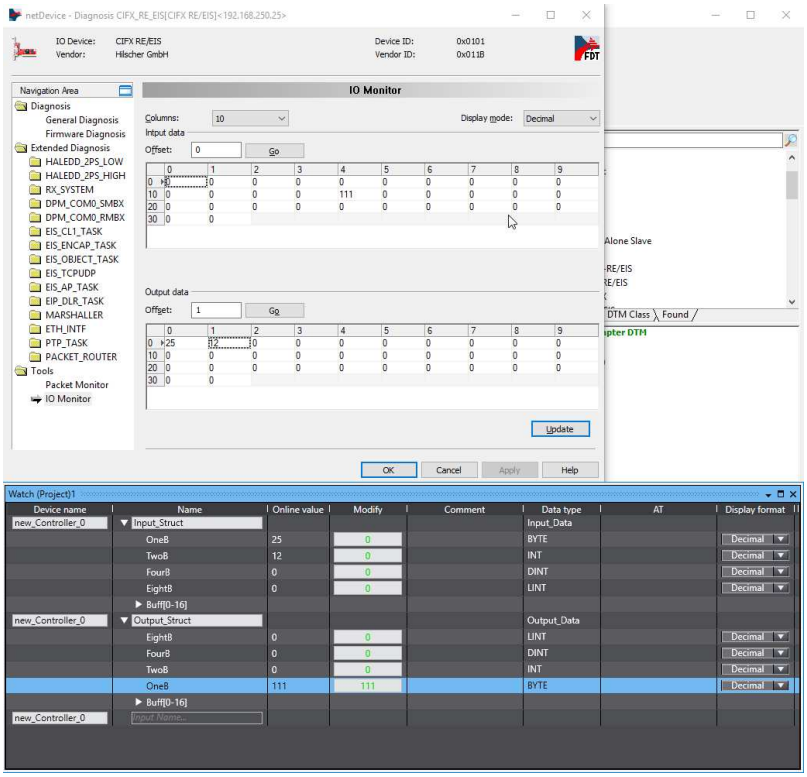
For this moment it is essential to connect with a PLC device, to do this in Sysmac Studio you need to navigate to "Controller -> Communication Setup...". There might be an error in Ethernet Communication Test if your network card is configured as DHCP (**Configuration of the network card** section in this tutorial). If communication with PLC is established you will receive "Test OK".



If communication with PLC is established you can download the configuration to the device.

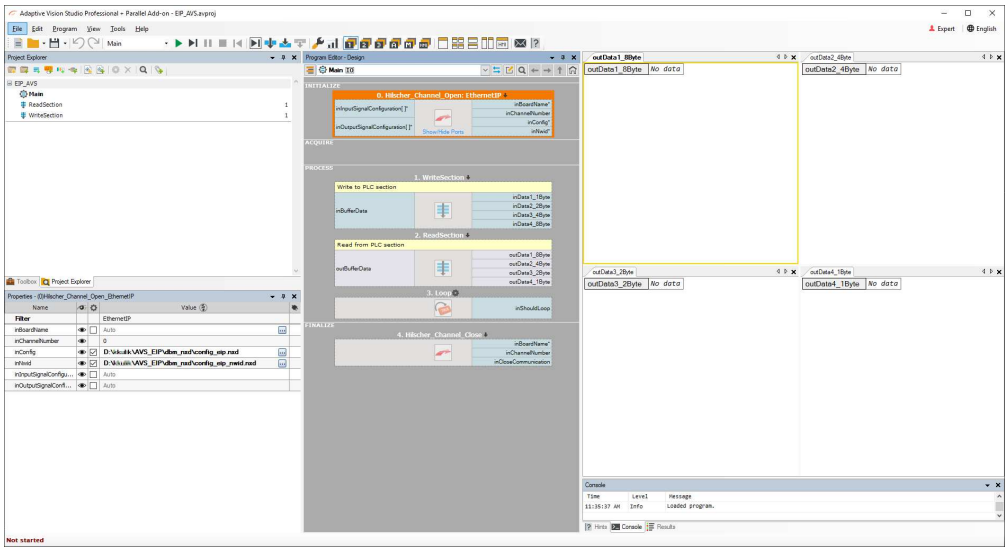


Below you can find a test of input and output memory using a watchtable in Sysmac Studio and IO Monitor in SYCON.net. To turn on IO Monitor you have to right-click on the icon of the card in SYCON.net project and click "Connect". Then double click this icon and navigate to "Tools -> IO Monitor". By clicking "Update" you can send a changed frame to the PLC.

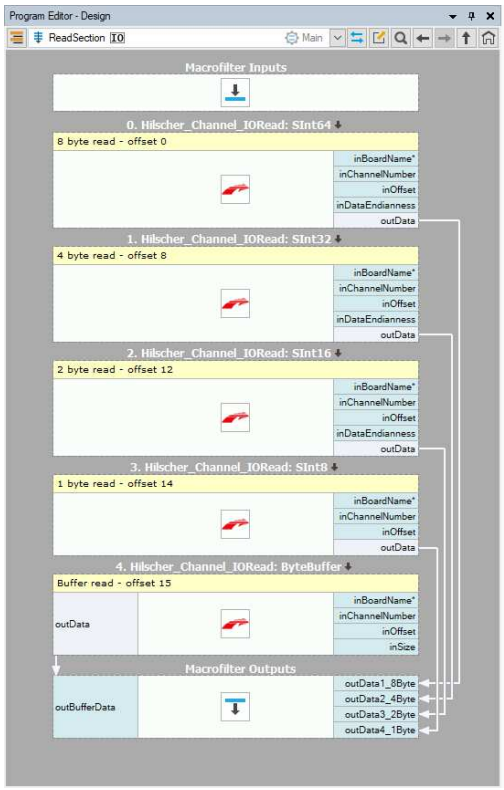


5) Example configuration in Aurora Vision Studio

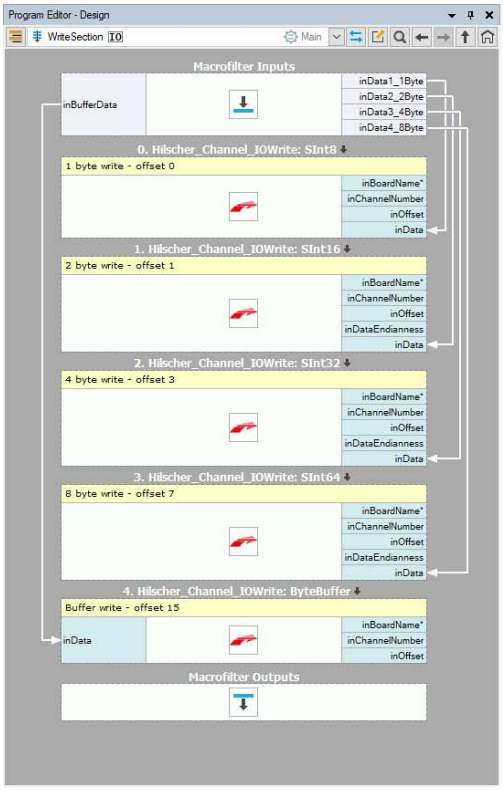
To use EtherNetIP filters in Aurora Vision Studio first you need to attach configuration files from SYCON.net to Hilscher_Channel_Open_EthernetIP filter in INITIALIZE section. The configuration files generated in the previous step are now required in inConfig (`xxx.nxd`) and inNwid (`xxx_nwid.nxd`) properties of that filter. Below you can find two Step macrofilters responsible for writing data to PLC and receiving data from it. To have cyclic communication you have to place Loop macrofilter at the end of PROCESS section. In FINALIZE section place Hilscher_Channel_Close macrofilter.



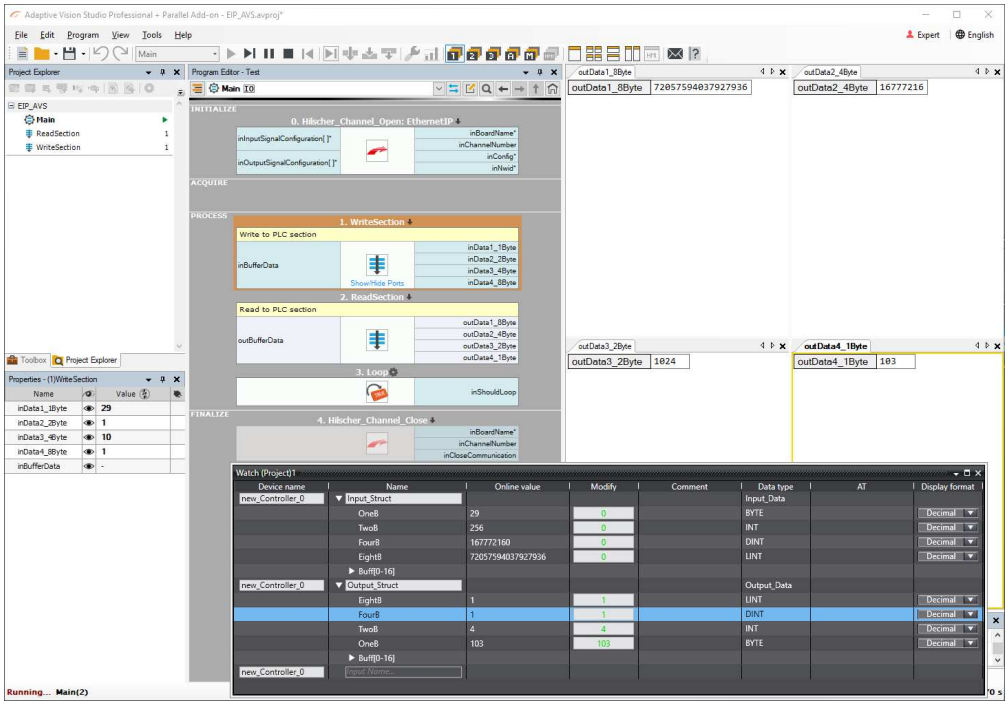
In step macrofilter ReadSection you can find for example Hilscher_Channel_IORead_SInt8 filter which writes 8 bytes of signed data to the predefined memory area. Using different offsets for each macrofilter enables access to different variables created in PLC (**Example configuration of Ethernet/IP PLC** section of this tutorial).



WriteSection step macrofilter has Hilscher_Channel_IOWrite filters (for instance Hilscher_Channel_IOWrite_SInt8) filters with adequate offsets and data types used to match PLC data variables configuration.



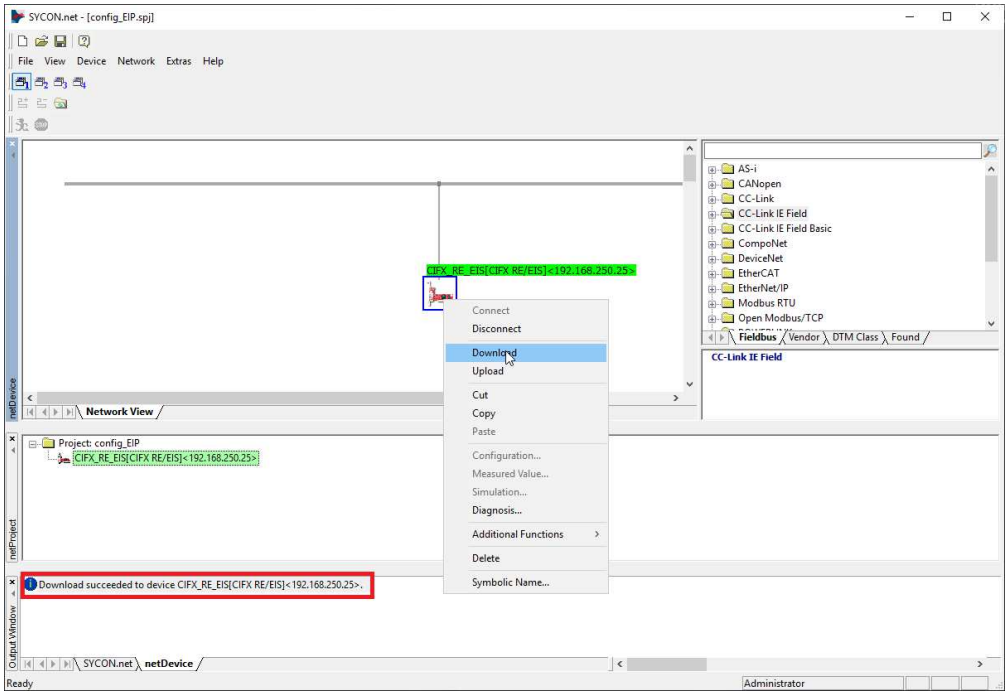
Below is presented reading and writing to PLC with Aurora Vision Studio and Sysmac Studio watchtable. Decimal values of variables are dependent on the data type used.



Troubleshooting

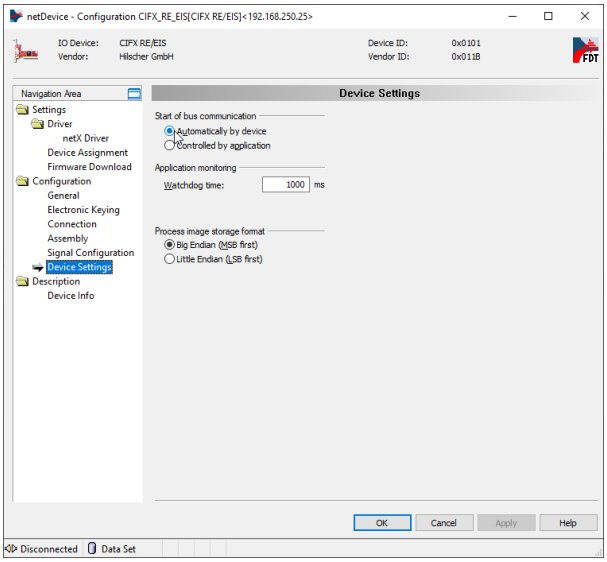
1. Set static IP to Hilscher card. Go to the section **Configuration using SYCON.net** of this tutorial.

2. Make sure that the current program setting is loaded to the Hilscher card. If not please use SYCON.net application to connect and download settings to the devices as shown in the picture below.

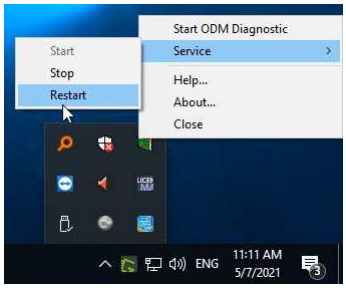


3. Check in your device in Device Assignment. This step was described in **Configuration using SYCON.net** section of this tutorial.

4. If your master device has a problem with connection with the Hilscher card use the following settings in the SYCON.net application. In order to do this right-click on the card icon and select Configuration...



5. After changes it may be necessary to restart ODMV3 service.



6. If none of the above advice has helped, please restart your computer.

Interfacing Hilscher card (EtherCAT) to Aurora Vision Studio

Purpose and requirement

This document explains how to configure EtherCAT PLC with Aurora Vision Studio using Hilscher EtherCAT card.

Required equipment:

- Supported Hilscher EtherCAT card
- Aurora Vision Studio 5.1 or later

Hardware connection

PLC and PC must work in a common LAN network - henceforth called shared network. In most cases they are connected to each other through a network switch. In case of EtherCAT interface Hilscher communication card must be configured as slave device. In this tutorial Port 1 of Hilscher Card has to be connected to the EtherCAT Port 2 of the PLC.

The devices shall be connected as follows:

- Master device e.g. PLC to the shared network
- Hilscher card to PC
- Hilscher card to the EtherCAT port of PLC
- PC's network card to shared network

Before proceeding to the further point, make sure that all devices are powered up.

Configuring Hilscher Devices

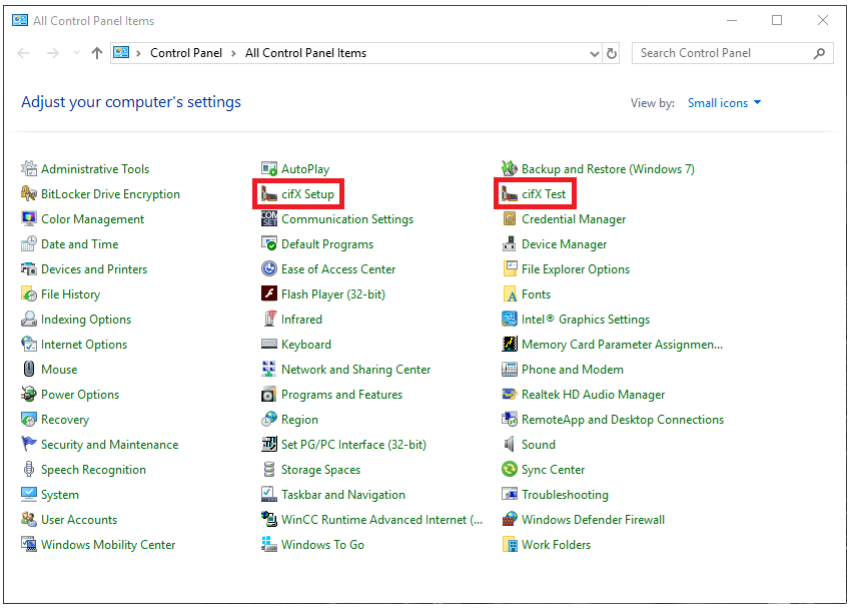
Be sure to have the newest version of the following drivers:

- List of the EtherCAT drivers with firmware: download link
- cifX Driver Setup software: download link
- SYCON.net to configure Hilscher card and generate configuration files: download link

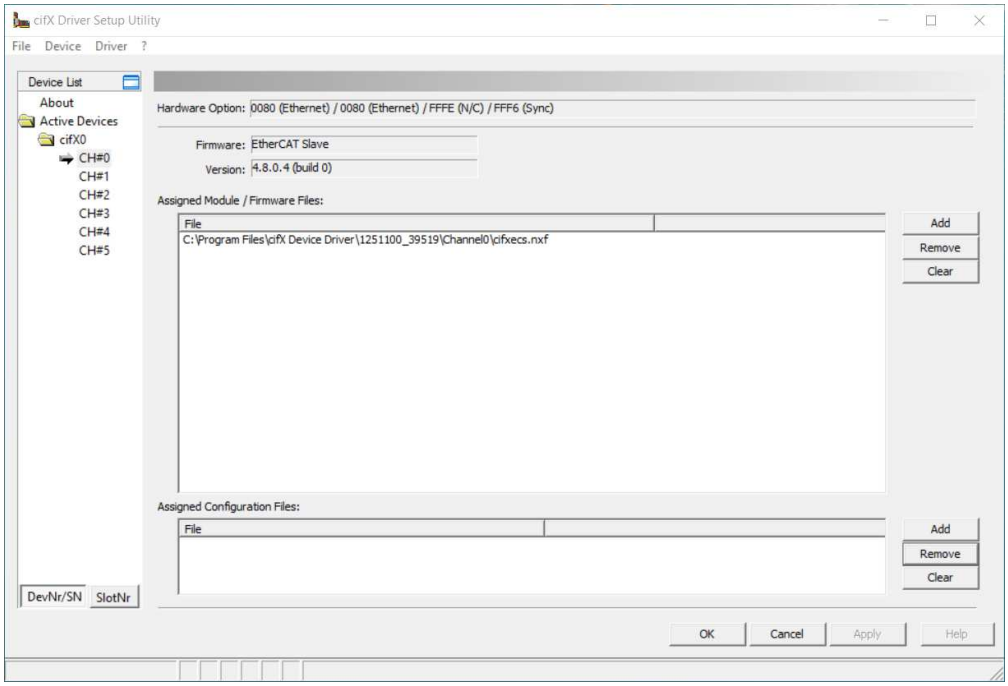
Make sure you have the recommended driver installed, the newest SYCON.net and firmware prepared.

1) Configuration using cifX Driver Setup Software

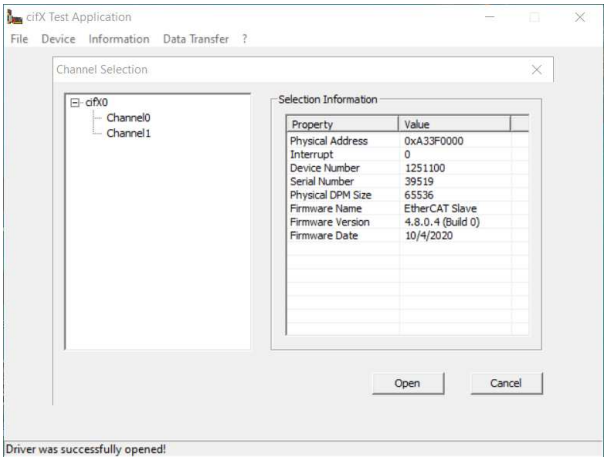
The easiest way to install the firmware to the cifX Driver is using the cifX Driver Setup Software. You can find tools for this software for instance using Control Panel menu as in the picture below.



Using cifX Setup application select the channel you would like to configure (usually channel - CH#0) and remove preexisting firmware by clicking "Clear". Then click "Add" to add a new firmware file and navigate to the downloaded EtherCAT Slave firmware (path \COMSOL-ECS V4.8.0.4\Firmware\cifX). Then click "Apply" and finish configuration with "OK".

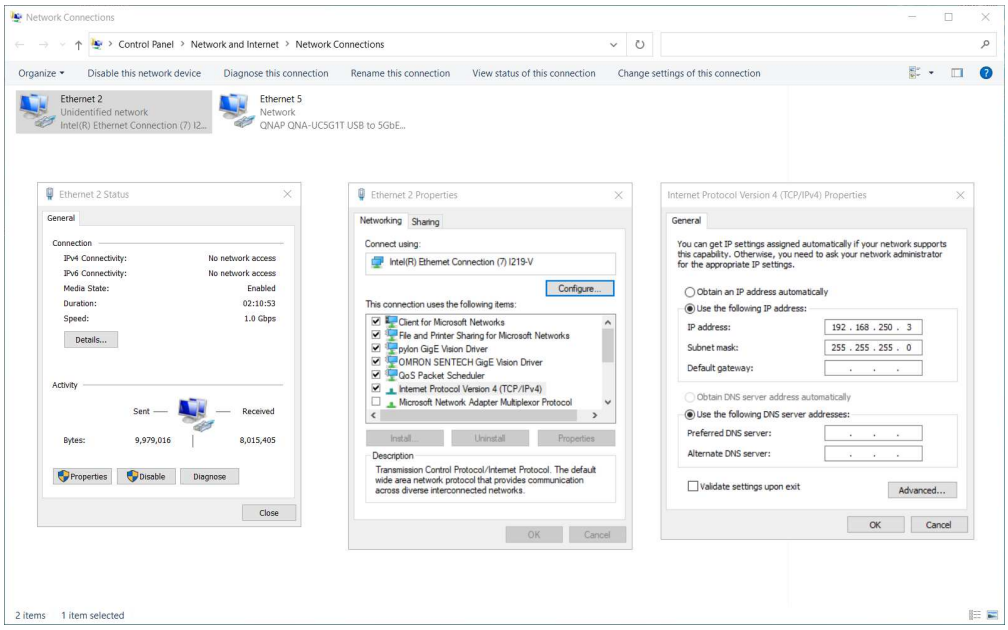


To be sure that firmware is installed properly you can open cifX Test software and then navigate to Device -> Open to achieve the result like below.



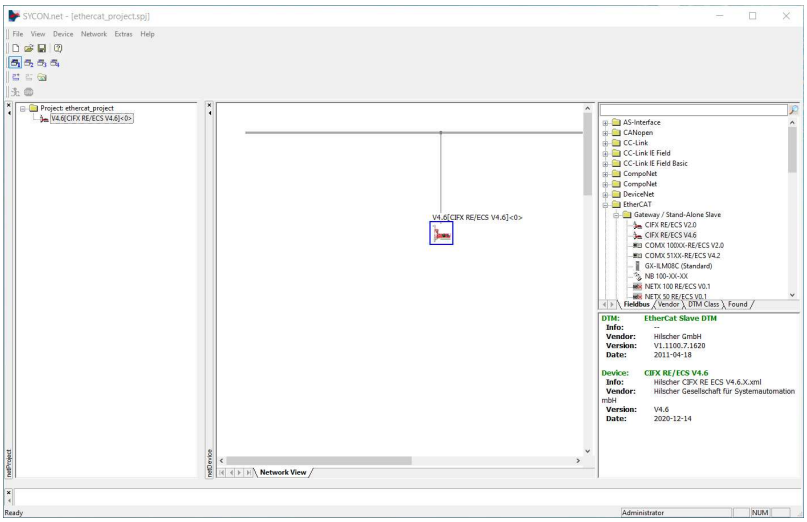
2) Configuration of the network card

It might be necessary to change the IP of the network card from dynamic to static. In this tutorial the following settings are used.

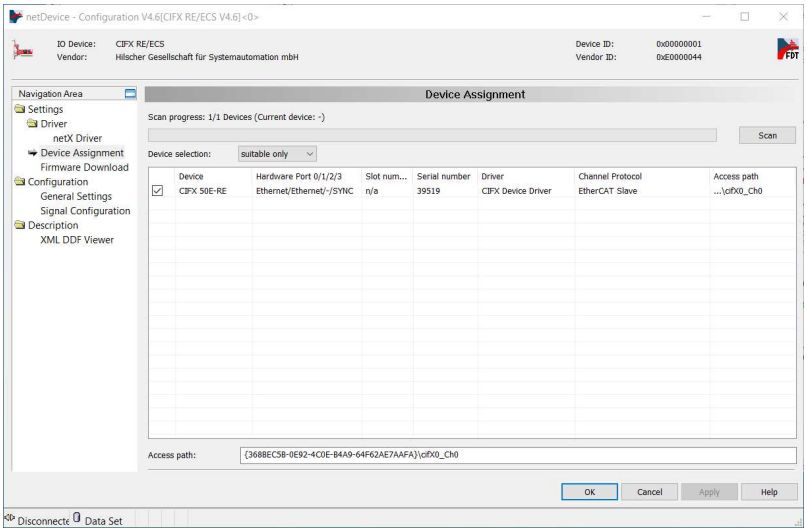


3) Configuration using SYCON.net

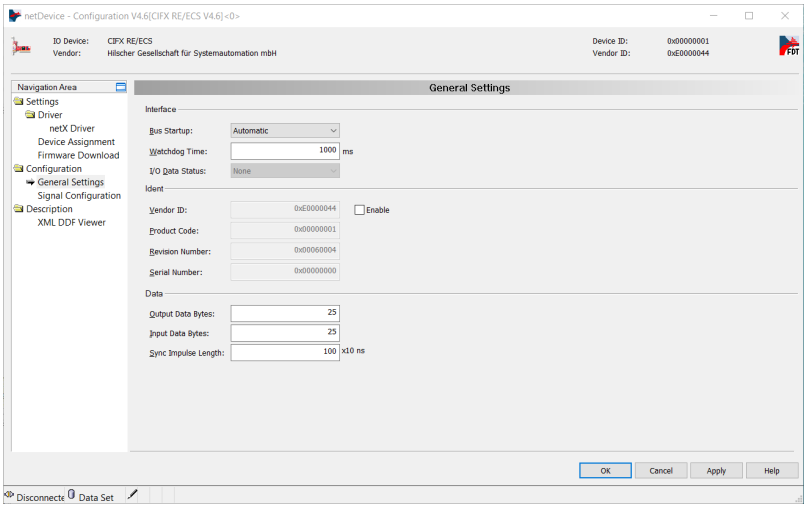
Configuration files are generated in SYCON.net software. Firstly, pick a proper device and drag it to the gray bus on the "Network View". This device you can find navigating to "EtherCAT -> Gateway / Stand - Alone Slave -> CIFX RE/ECS V4.6".



To open the configuration card double click on dropped icon and in the popped-up menu navigate to "Settings/Device Assignment". Then scan for devices by clicking "Scan", mark found one it with a tick like on the image below and then click "Apply".

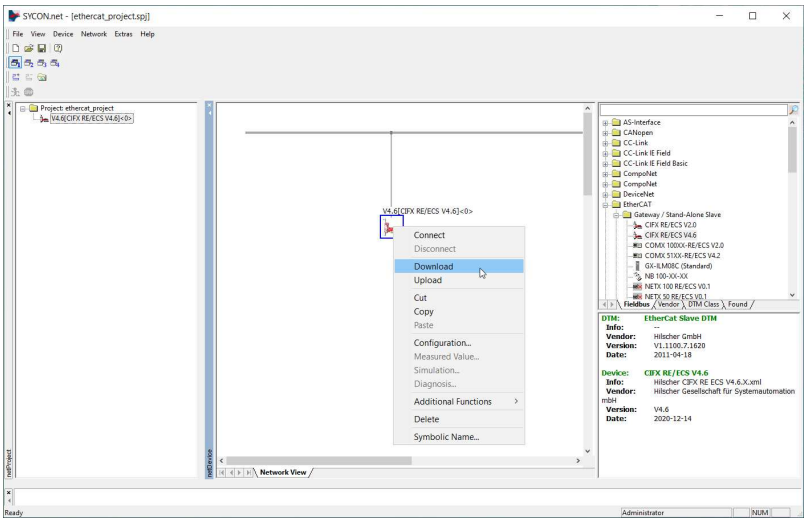


In "Configuration/General Settings" you can change the configuration of memory blocks for IN/OUT operations. By default, the Data length is set to 200 bytes. All subsequent PLC data blocks mentioned in this document are configured for 25 bytes. In case you need memory blocks of a different size, you can change the Data length from 0 up to 256 bytes.

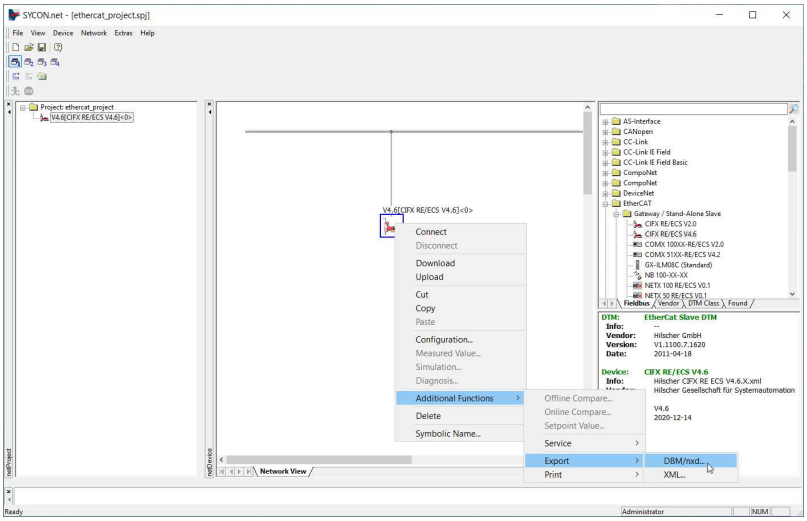


Confirm all changes by clicking "Apply" and then click "OK".

Then you have to download the configuration to the device like on the screen below.



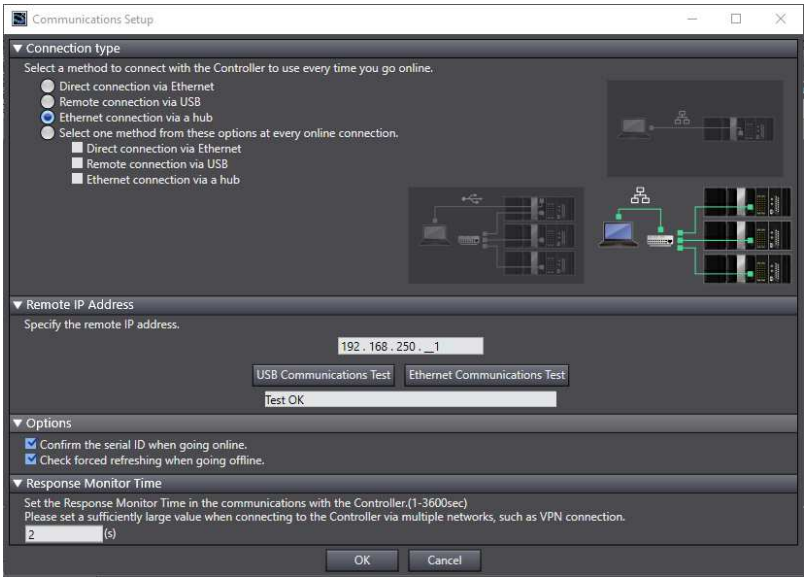
The final step is to generate configuration files for Aurora Vision Studio. You can do this by right-clicking on the device icon, then navigating to "Additional Functions -> Export -> DBM/nxd..", entering your configuration name and clicking "Save". You can now close SYCON.net for this example, **remember to save your project before.**



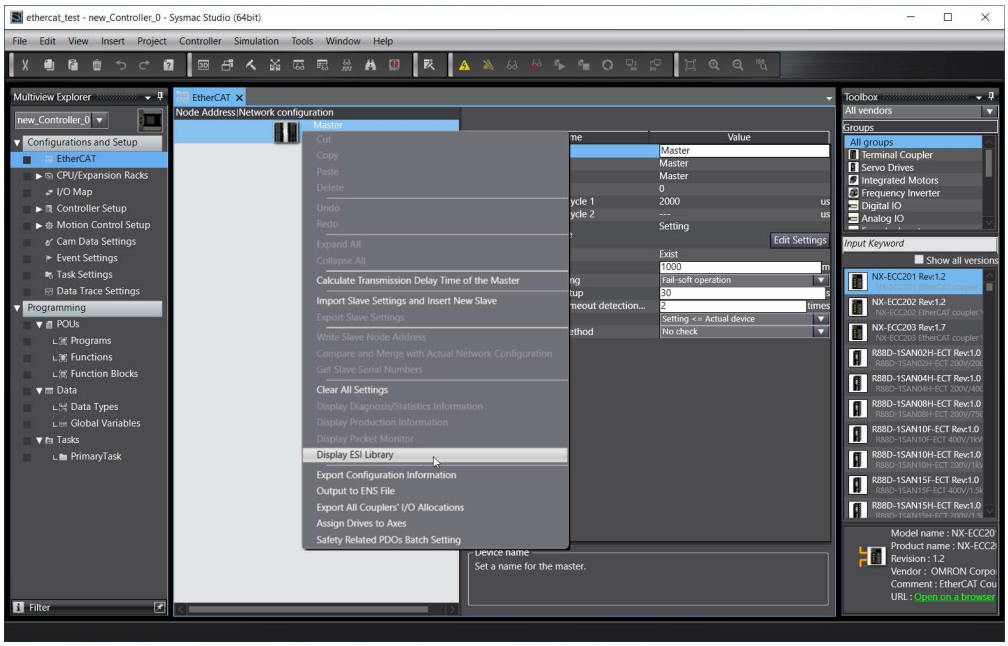
4) Example configuration of EtherNet/IP PLC

Below you will find an example configuration process for EtherCAT PLC Omron NXJP2-9024DT in Sysmac Studio software.

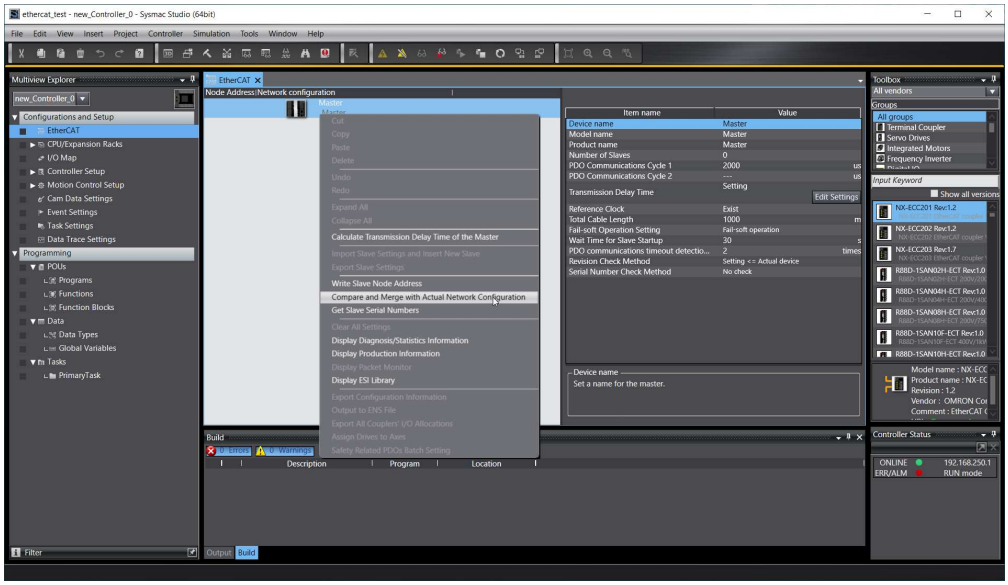
For this moment it is essential to connect with a PLC device, to do this in Sysmac Studio you need to navigate to "Controller -> Communication Setup...". There might be an error in Ethernet Communication Test if your network card is configured as DHCP (**Configuration of the network card** section in this tutorial). If communication with PLC is established you will receive "Test OK".



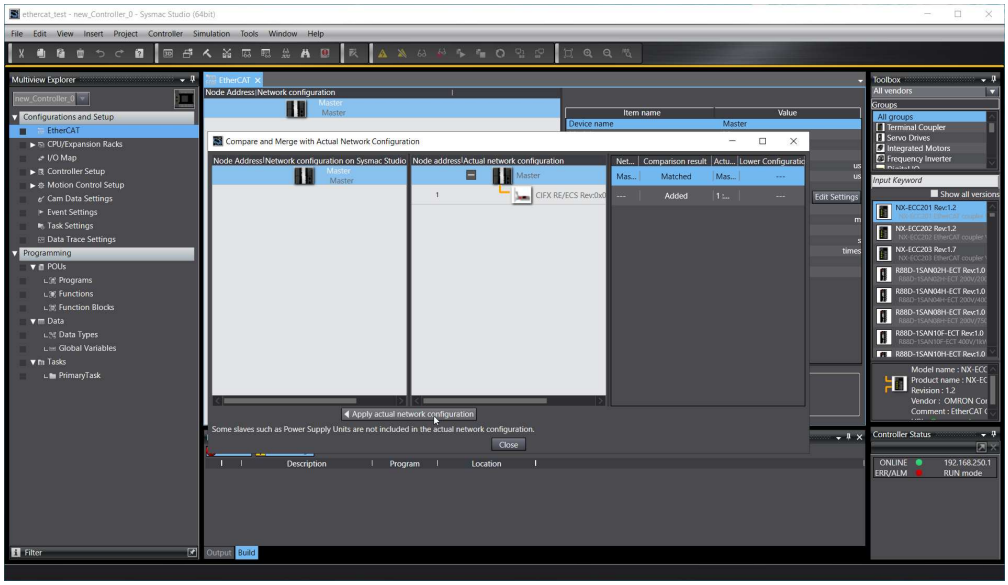
Add support for Hilscher card using ESI Library configuration (path \COMSOL-ECS V4.8.0.4\EDS). To do this in Sysmac Studio you need to navigate to EtherCAT tab, right-click the Master icon and choose "Display ESI Library".



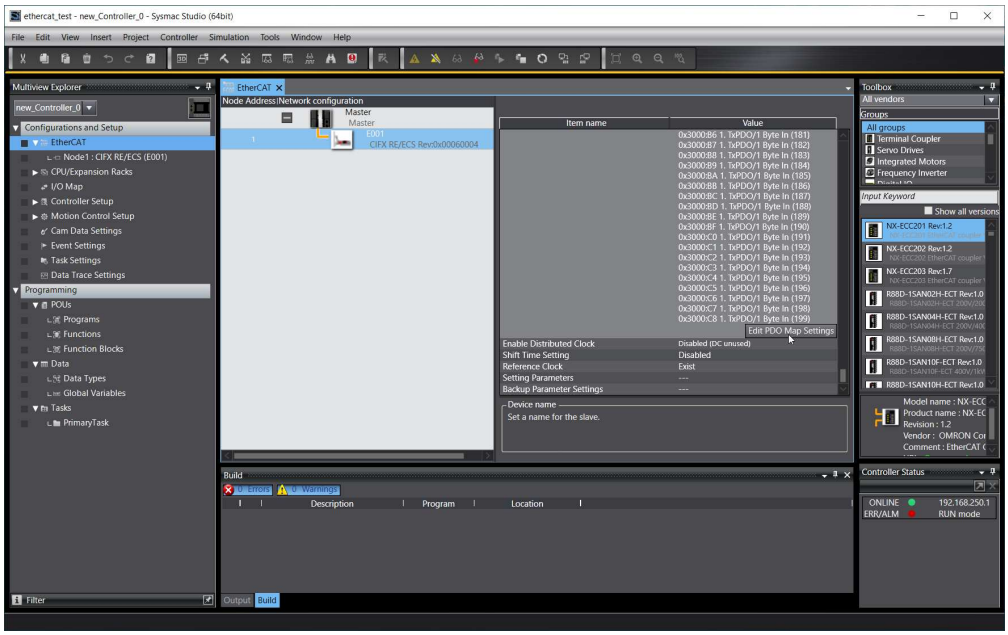
If communication with PLC is established and Hilscher Card is connected to the EtherCAT port of PLC it is possible to load EtherCAT configuration to the PLC. To accomplish that right-click Master icon and choose "Compare and Merge with Actual Network Configuration".



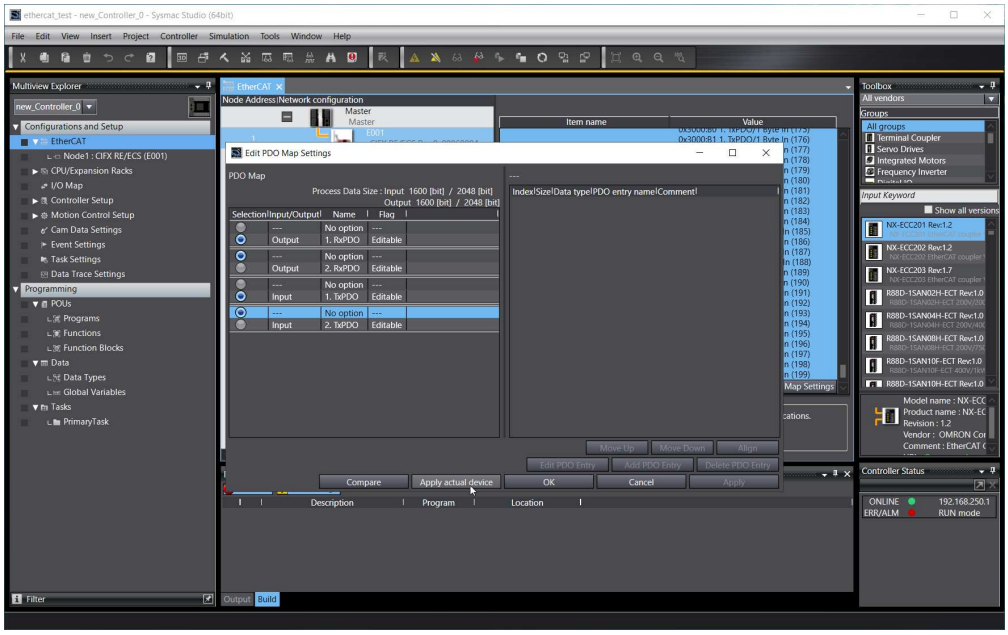
If everything is connected correctly Compare and Merge with Actual Network Configuration window should pop-up. To match configuration in software to configuration in hardware click "Apply actual network configuration".



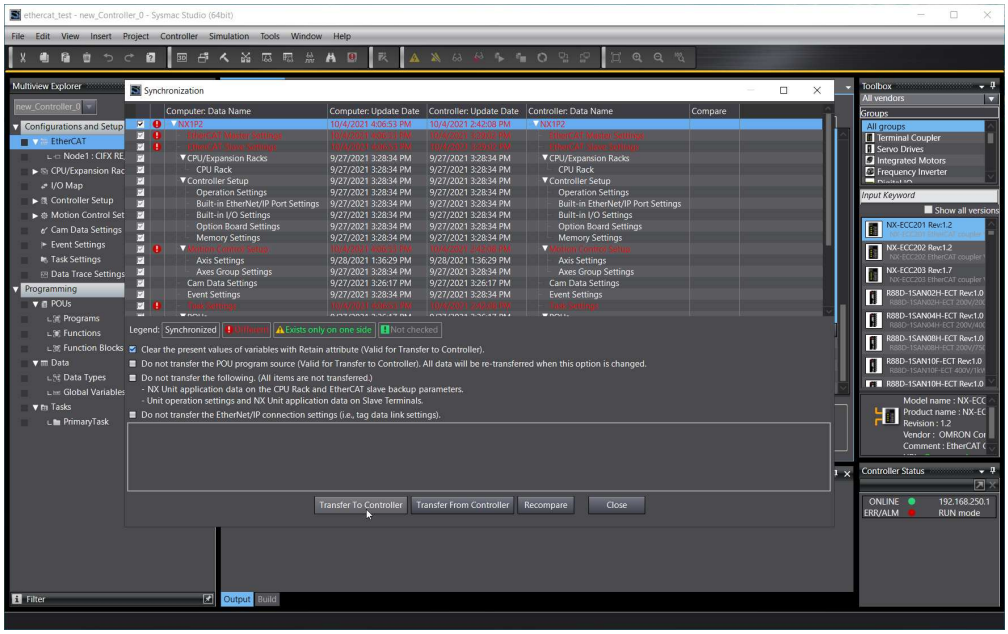
Choose previously added slave device and navigate to the "Edit PDO Map Settings" button and click it. This step is essential to match the configuration from SYCON.net.



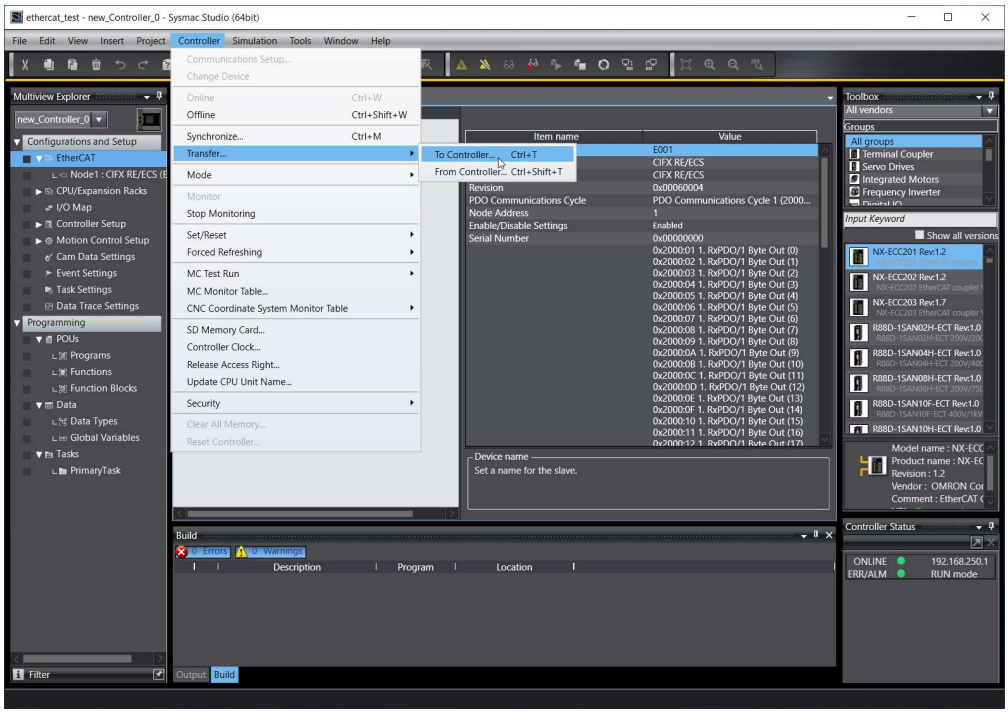
In Edit PDO Map Settings window click "Apply actual device". Then click "Apply" and "OK".



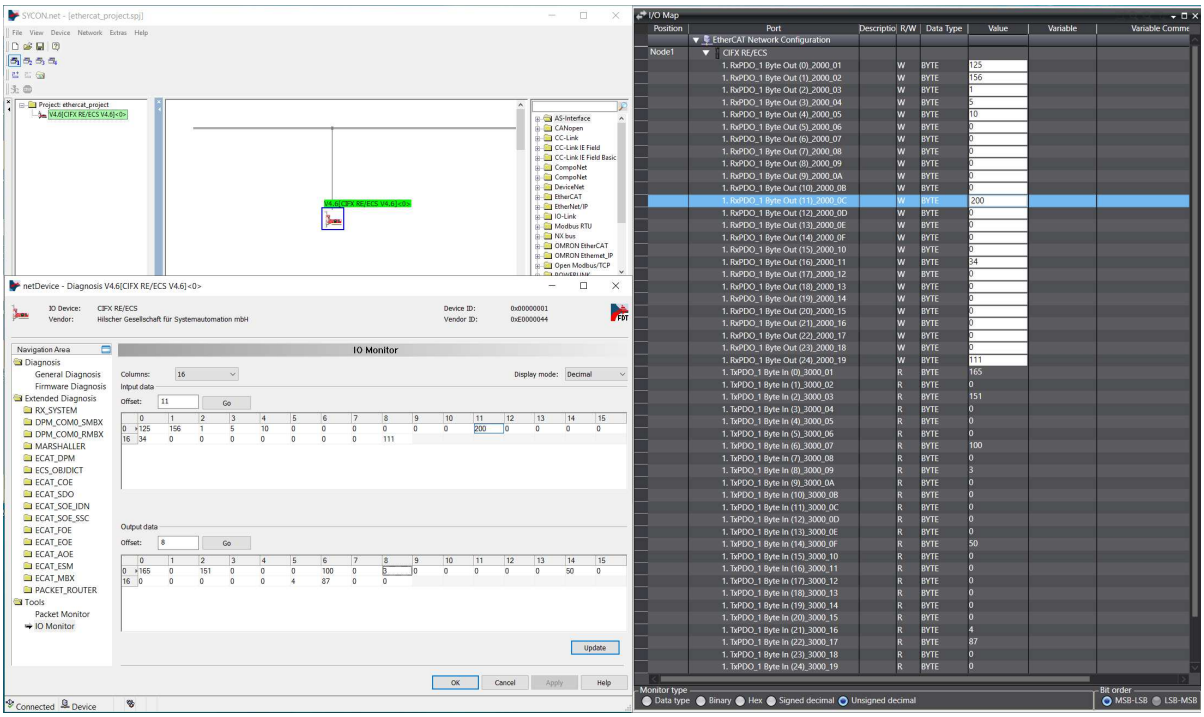
In the online mode of the PLC controller navigate to "Controller -> Synchronize...". Mark with ticks all the changes and click "Transfer To Controller".



In the online mode of the PLC controller navigate to "Controller -> Transfer... -> Transfer To Controller" and download changes.

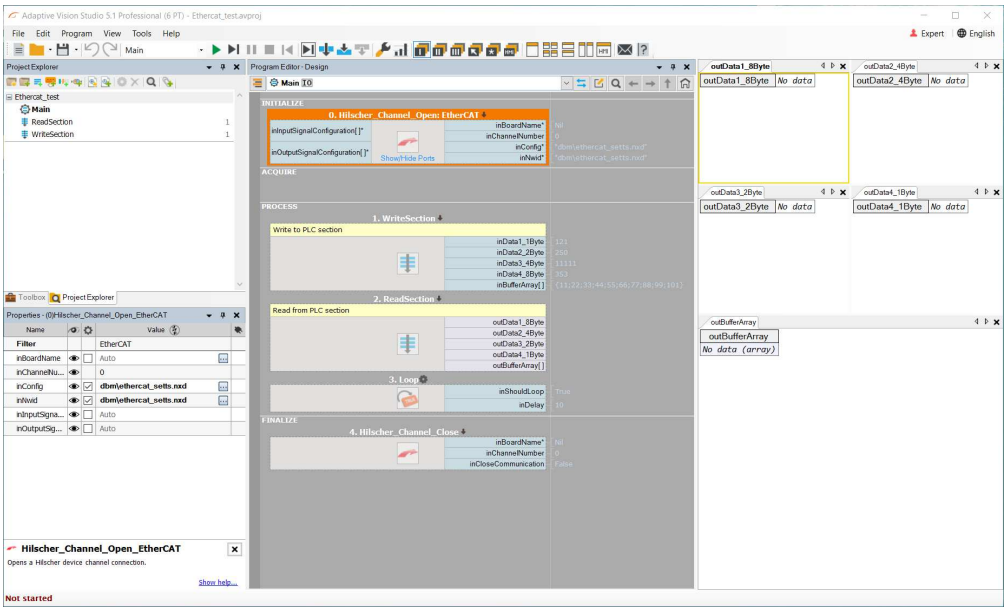


Below you can find a test of input and output memory using a watchtable in Sysmac Studio and IO Monitor in SYCON.net. To turn on IO Monitor you have to right-click on the icon of the card in SYCON.net project and click "Connect". Then double click this icon and navigate to "Tools -> IO Monitor". By clicking "Update" you can send a changed frame to the PLC.

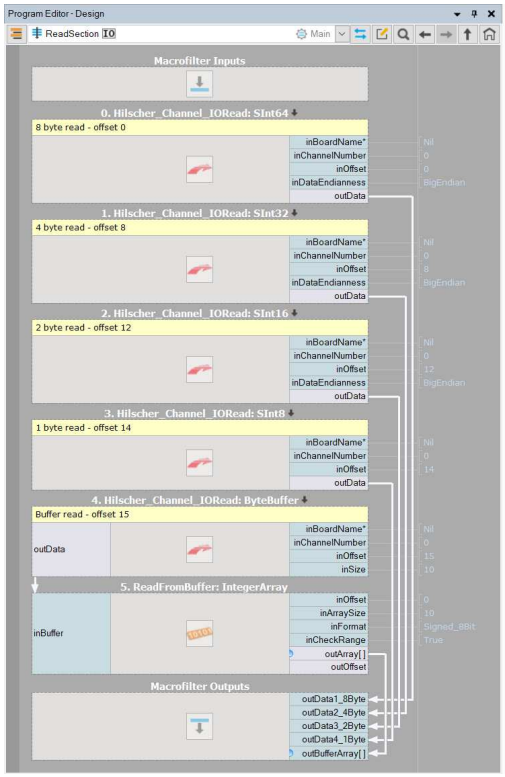


5) Example configuration in Aurora Vision Studio

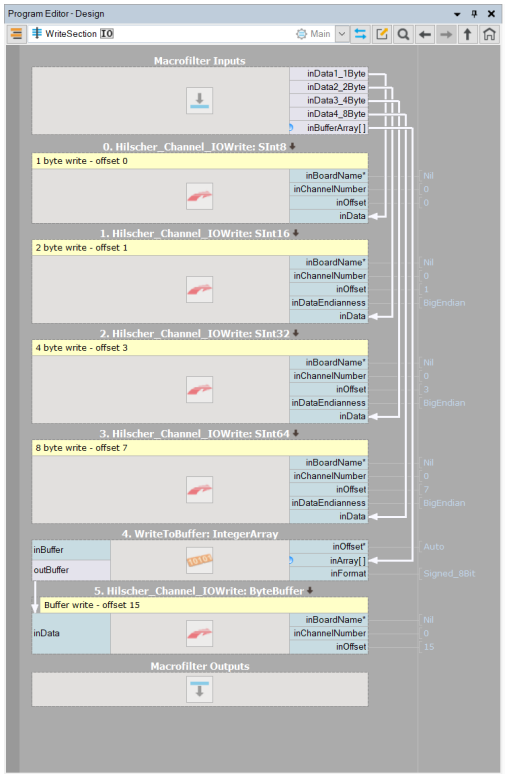
To use EtherCAT filters in Aurora Vision Studio first you need to attach configuration files from SYCON.net to Hilscher_Channel_Open_EtherCAT filter in INITIALIZE section. The configuration files generated in the previous step are now required in inConfig (`xxx.nxd`) and inNwid (`xxx_nwid.nxd`) properties of that filter. Below you can find two Step macrofilters responsible for writing data to PLC and receiving data from it. To have cyclic communication you have to place Loop macrofilter at the end of PROCESS section. In FINALIZE section place Hilscher_Channel_Close macrofilter.



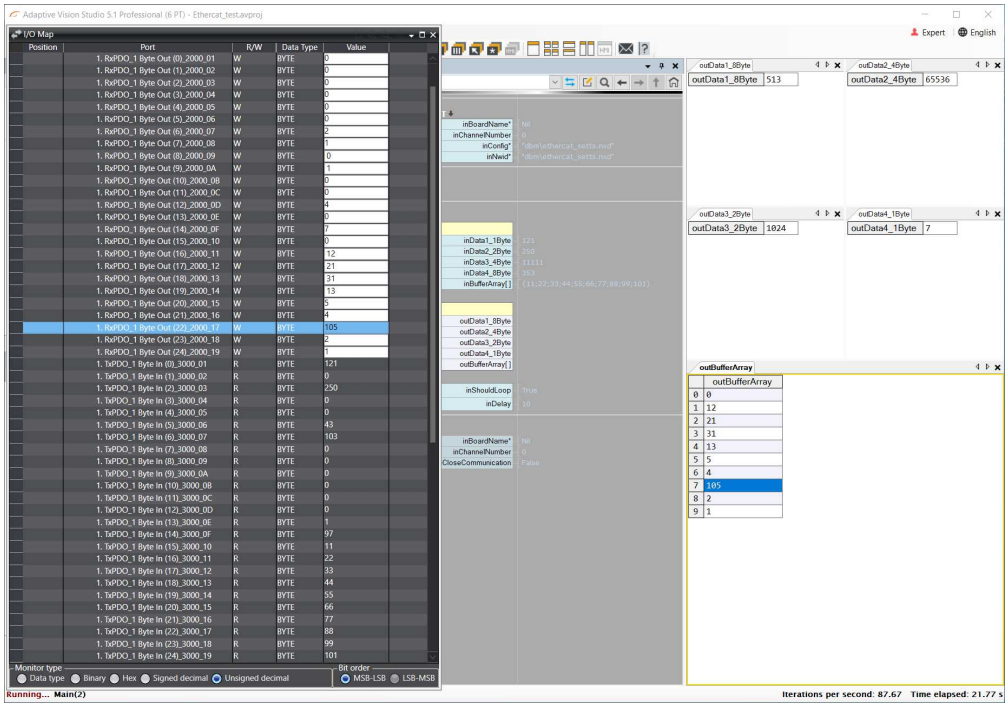
In step macrofilter ReadSection you can find for example Hilscher_Channel_IORead_SInt8 filter which writes 8 bytes of signed data to the predefined memory area. Using different offsets for each macrofilter enables access to different variables created in PLC (**Example configuration of EtherCAT PLC** section of this tutorial).



WriteSection step macrofilter has Hilscher_Channel_IOWrite filters (for instance Hilscher_Channel_IOWrite_SInt8) filters with adequate offsets and data types used to match PLC data variables configuration.

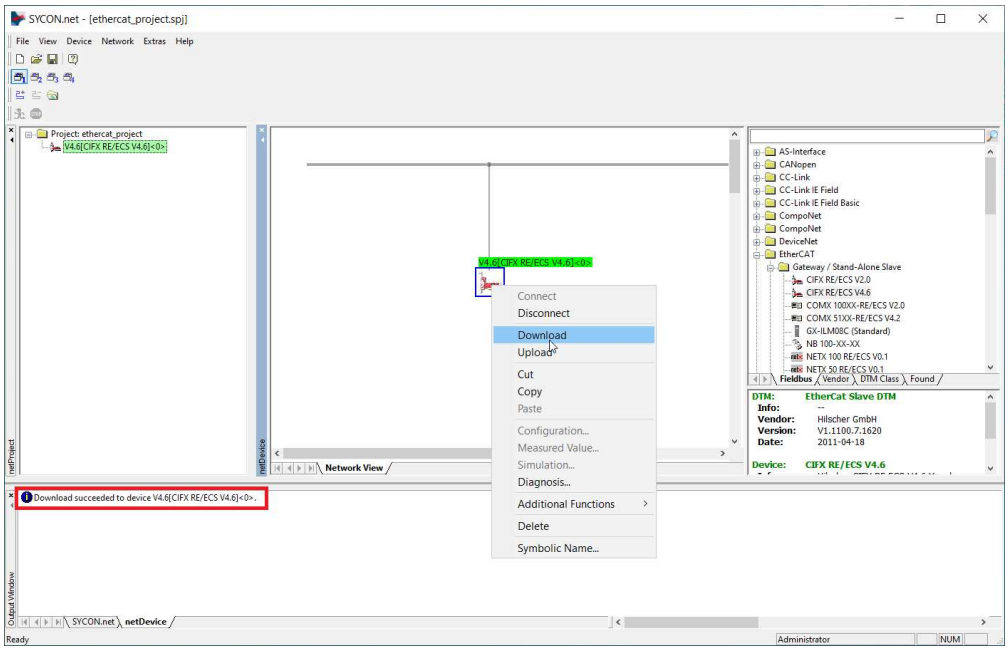


Below is presented reading and writing to PLC with Aurora Vision Studio and Sysmac Studio I/O Map. Decimal values of variables are dependent on the data type used.



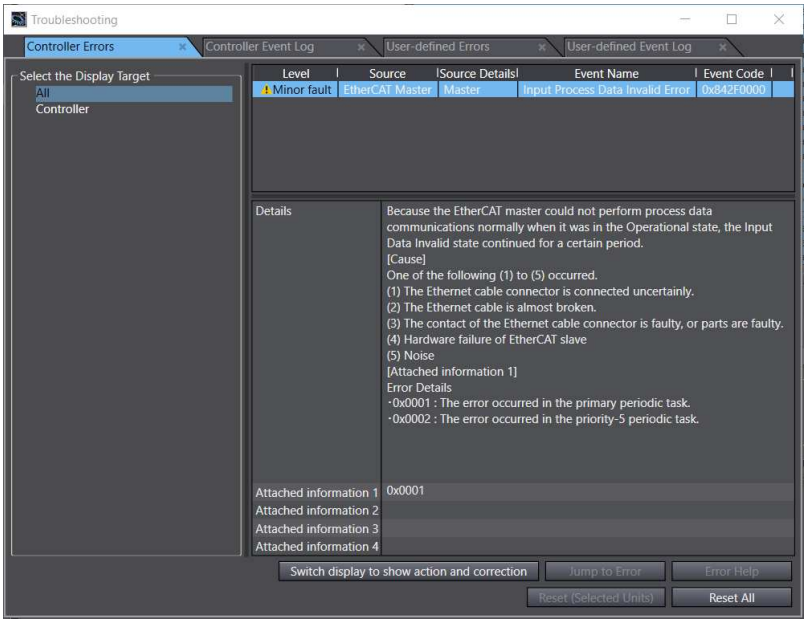
Troubleshooting

1. Make sure that the current program setting is loaded to the Hilscher card. If not please use SYCON.net application to connect and download settings to the devices as shown in the picture below.

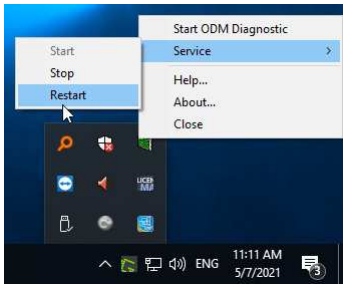


2. Check in your device in Device Assignment. This step was described in **Configuration using SYCON.net** section of this tutorial.

3. Using Sysmac Studio you can encounter Minor fault Controller Error like below which typically does not affect data exchange. Replacing Ethernet cables or hardware could possibly make this warning disappear.



4. After changes it may be necessary to restart ODMV3 service.



5. If none of the above advice has helped, please restart your computer.

Using Modbus TCP Communication

Purpose and equipment

This document describes how to make a practical example of establishing Modbus TCP communication between Aurora Vision Studio and a PLC device or a Modbus TCP simulator. Aurora Vision Studio 5.2 and the Simatic S7- 1200 CPU 1212C AC/DC/RLY controller with TIA Portal V15 were used to accomplish this task. For more theoretical background refer to the Working with Modbus TCP Devices entry.

Required equipment:

- Modbus TCP simulator, e.g. EasyModbusTCP
- SIMATIC S7-1200 or other controller from the SIMATIC S7 family
- TIA Portal V15
- Aurora Vision Studio 5.0 Professional or later (version 5.2 was used)

Overview and first steps

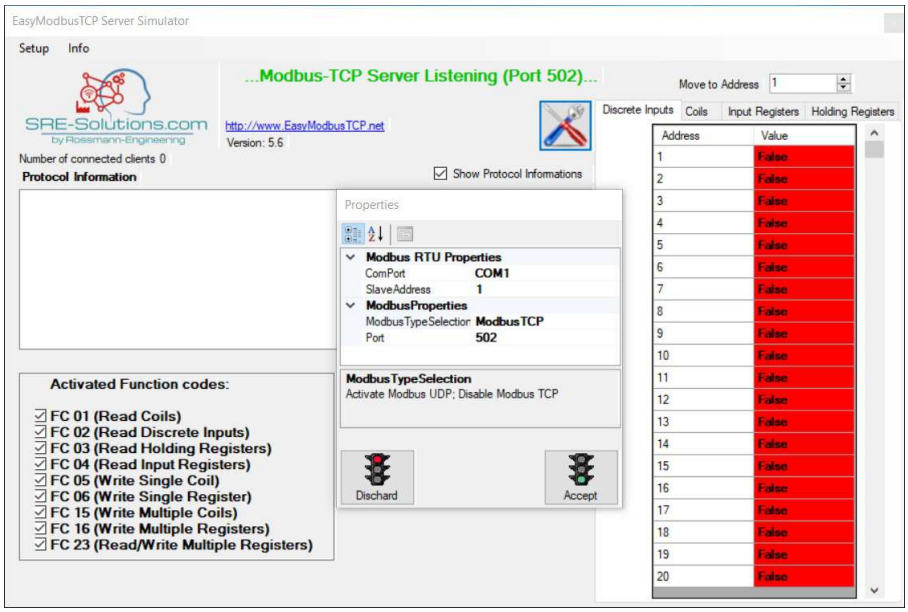
In this example, you will create two versions of the application, basic and advanced:

- the ***basic*** version of the application does not require a PLC; a Modbus TCP simulator can be used instead. This program enables exchanging basic data types between the Aurora Vision Studio client and the Modbus TCP server (PLC or a simulator);
- the ***advanced*** version of the application requires additional logic to be implemented in the PLC program. It simulates a dynamic mathematical model which represents a heater: a user can change parameters of the model parameters so the object can react slower or faster. A user can manually modify the model input parameters which represent the voltage fed to the actuator device. The PLC sends feedback about the temperature level which is then shown on the HMI;

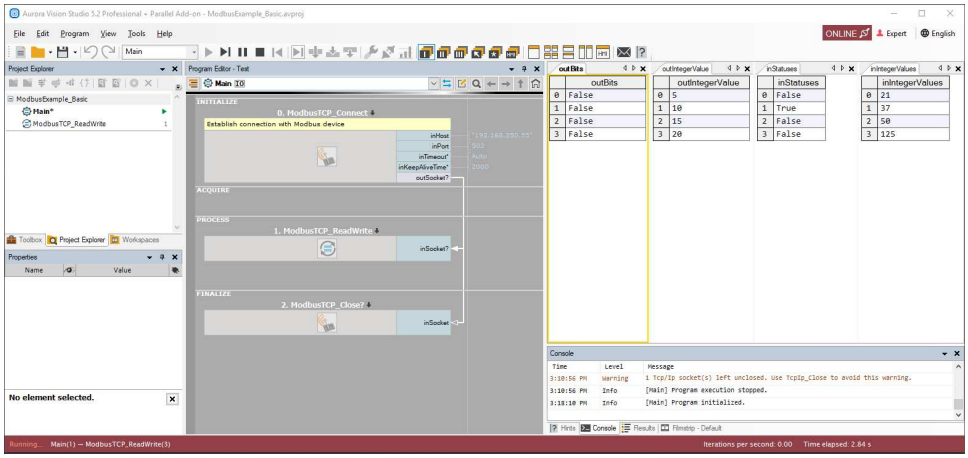
First, you need to establish a hardware connection between the PLC or the PLC simulator and the PC. The network configuration will be described later in this entry.

Basic program with Modbus TCP simulator

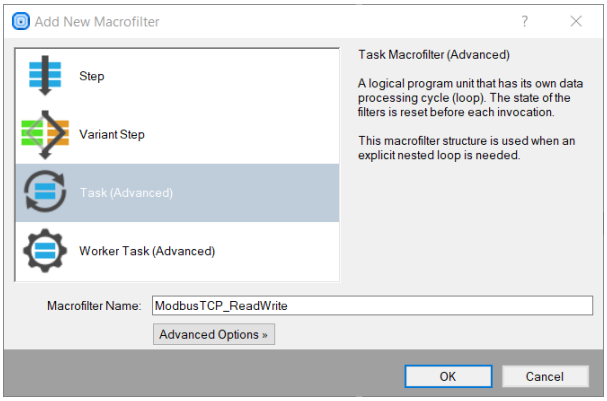
Open the EasyModbusTCP simulator and press the Setup button in the lower left corner of the application window. A *Properties* window will pop up: follow the image below to set up the configuration:



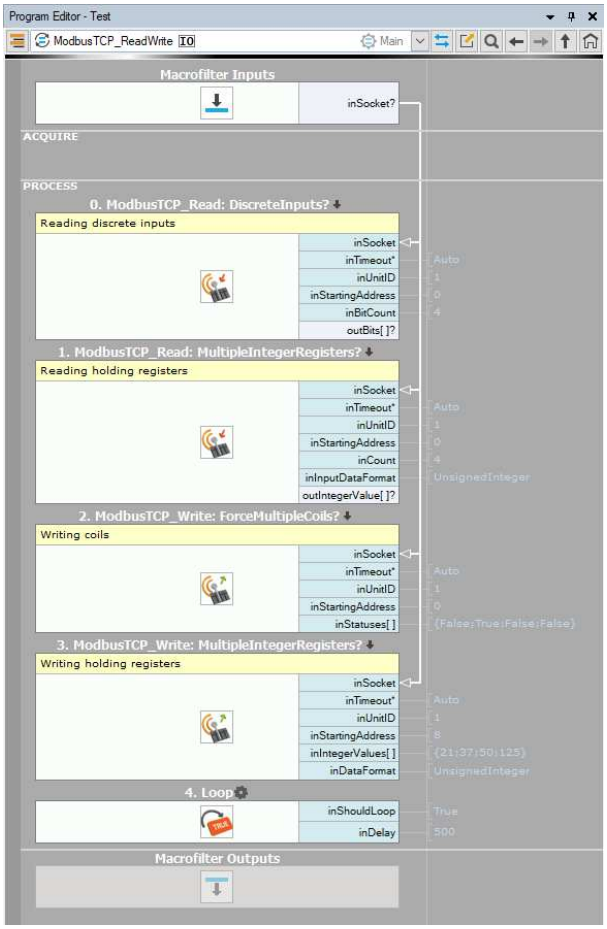
Then open a new Aurora Vision Studio project, add the ModbusTCP_Connect filter to the INITIALIZE section and the ModbusTCP_Close filter to the FINALIZE section. Aurora Vision Studio provides a set of ready-to-use filters for communication over the ModbusTCP protocol. A full list of the relevant filters is available here.



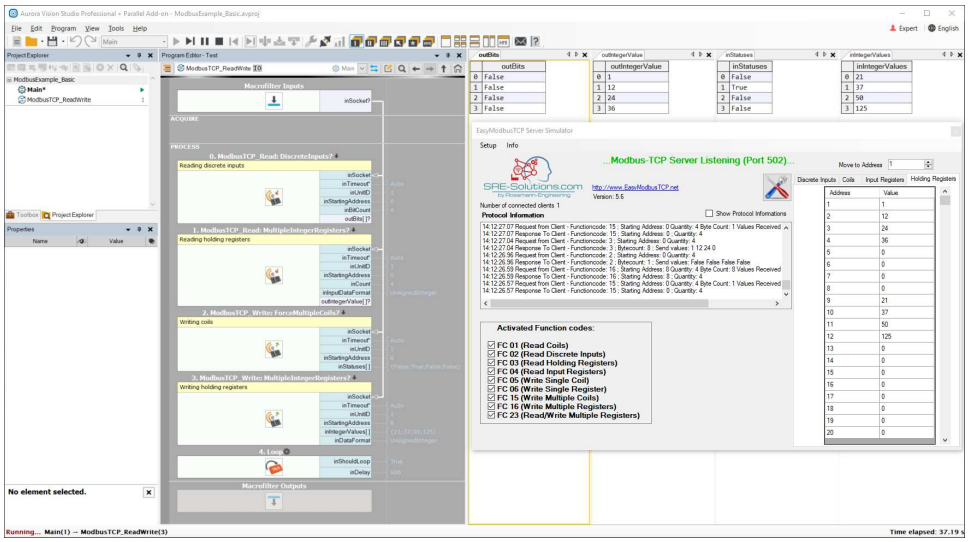
Add a Task Macrofilter to the PROCESS section.



Open the newly created Task Macrofilter and add the ModbusTCP_ReadDiscreteInputs, ModbusTCP_ReadMultipleIntegerRegisters, ModbusTCP_ForceMultipleCoils and ModbusTCP_WriteMultipleIntegerRegisters filters in the arrangement simulating that in the image below:



Below you can see working communication between the ModbusTCP client within the Aurora Vision Studio application and the simulated server.

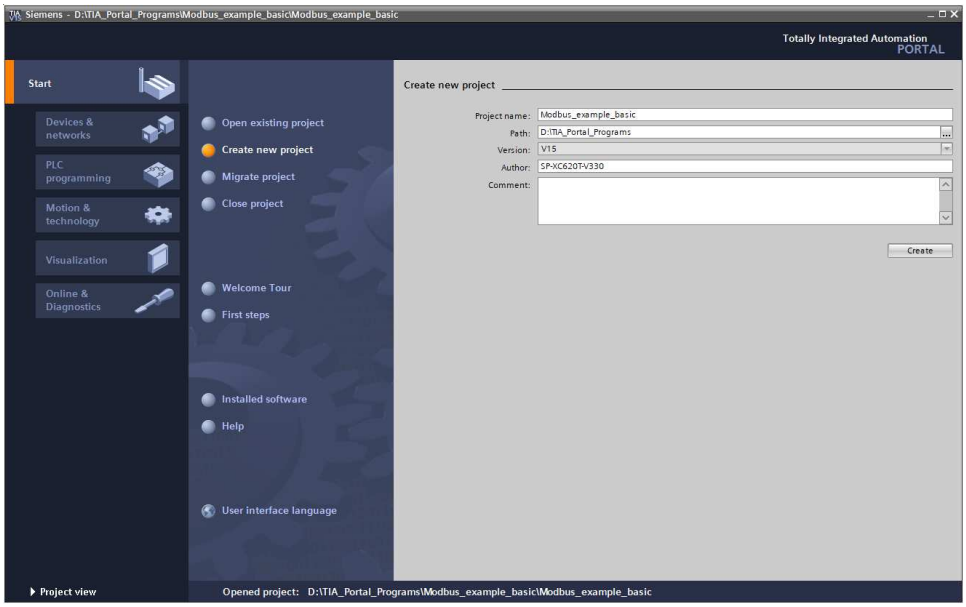


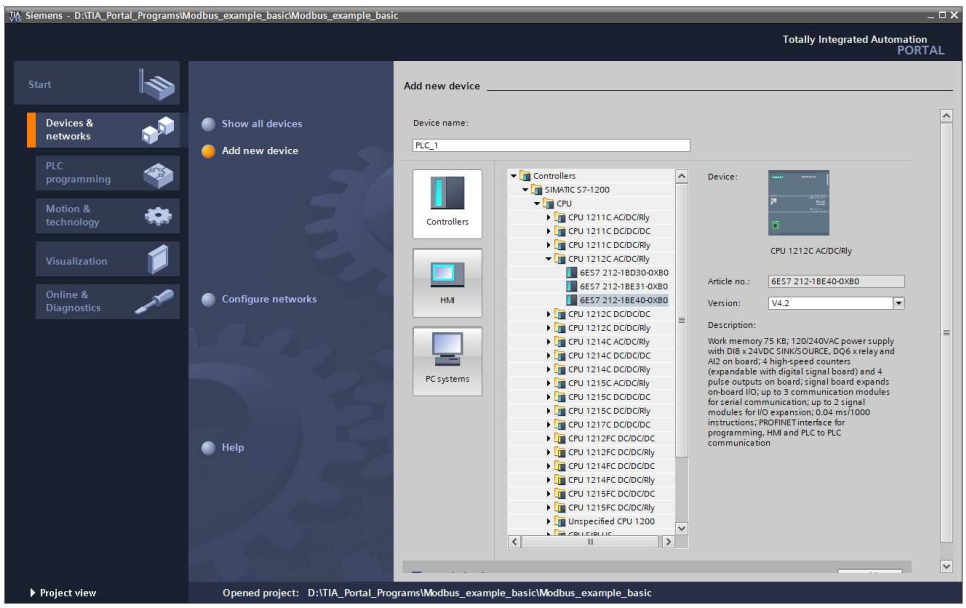
Basic program with a PLC

Network configuration

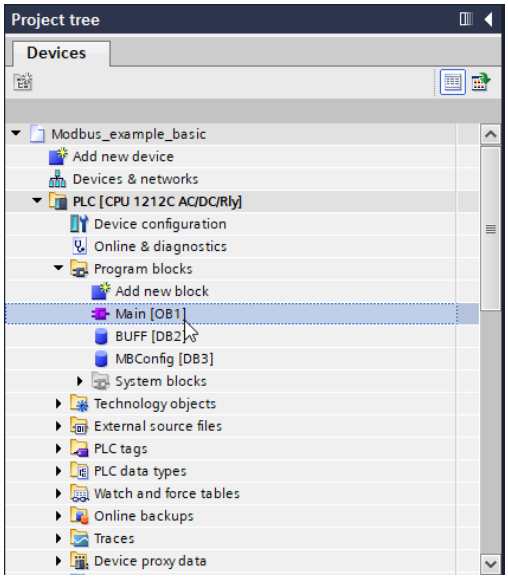
Main network configuration

Create a new project in the TIA Portal environment and add your device by double-clicking on the *Add new device* button. Select your controller model, set its CPU and other available components listed in the drop-down menu. Once you are ready, click the *OK* button.

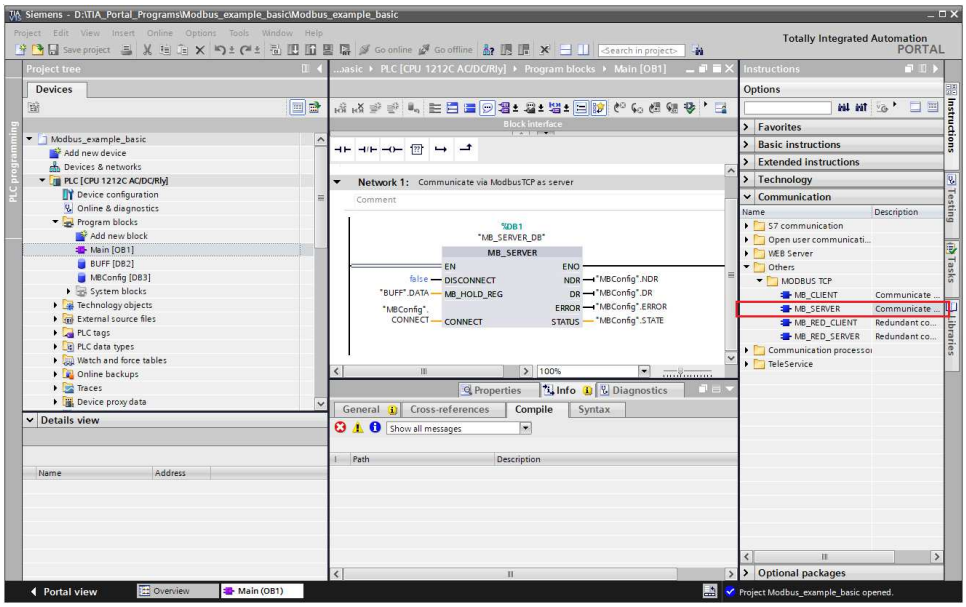




Expand the *Program blocks* list and double-click the *Main* icon. A network view will appear. Expand the Communication tab on the right-hand side (within the Instructions section) and double-click on the MB_SERVER icon available at the *Communication -> Others -> MODBUS TCP* location.

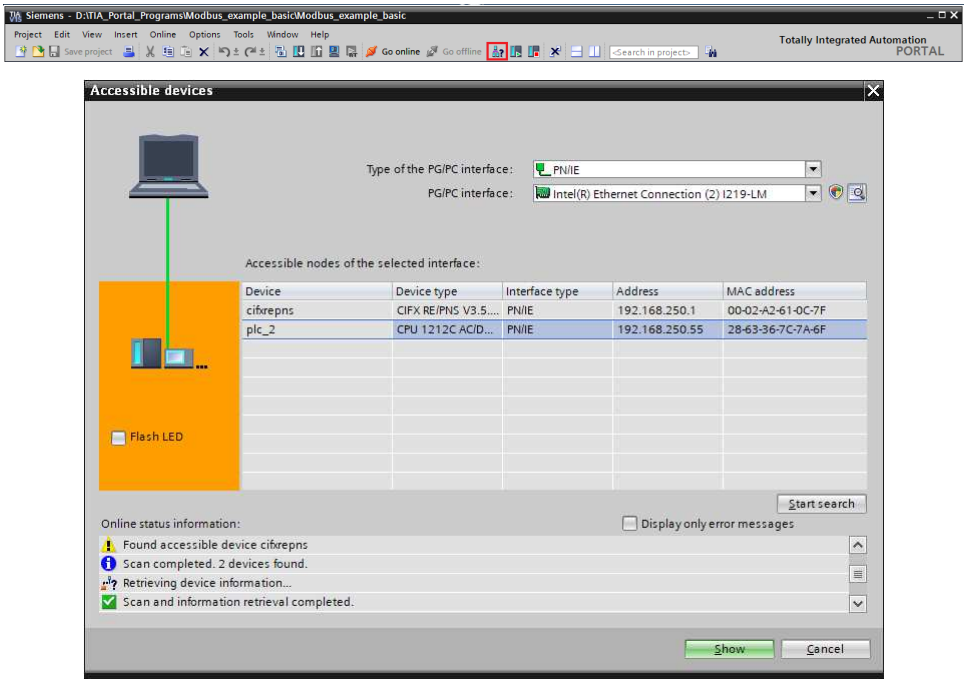


Drag the MB_SERVER block from the *Communication -> Others -> MODBUS TCP* location and drop it on to Network 1 within the application Basic.



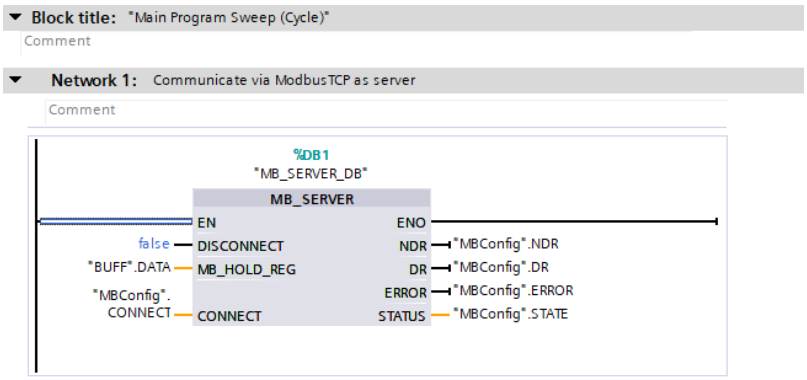
Troubleshooting IP addressing

Select the *Accessible devices* icon from the toolbar to open the configuration window. Select the *PN/IE* type of the *PG/PC* interface and then the Ethernet card connected to the PLC. Click the *Start search* button, choose your PLC from the list of accessible nodes and click the *Show* button.



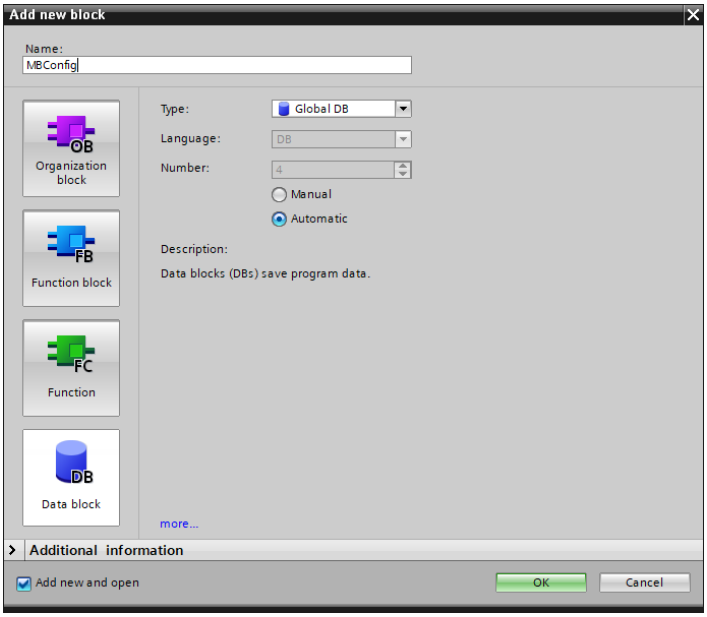
Creating a PLC program

This chapter describes how to create a PLC program to establish communication. The basic application that we start out with has only one function block; it is the same one that was created in the previous section:

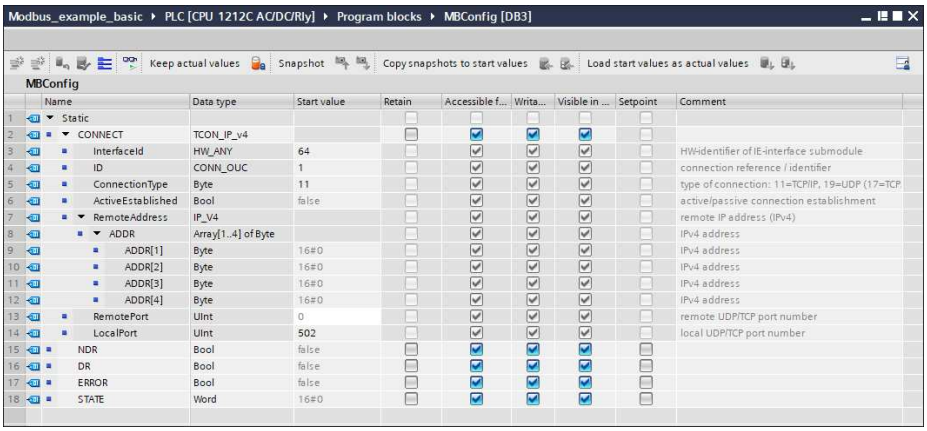


The MB_SERVER instruction is used to establish the Modbus TCP communication. In the image above, you can see some DB blocks connected to several inputs and outputs of the MB_Server block. In order to establish communication between a client and the server, you will need to set them up according to the way presented below:

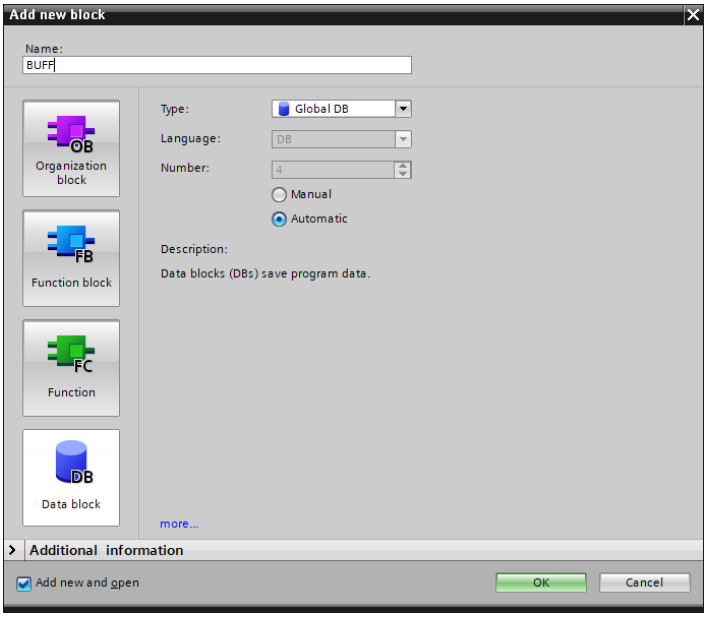
- MBConfig - DB block which defines the connection type, the address of the ModbusTCP server as well as the used Port (by default 502)



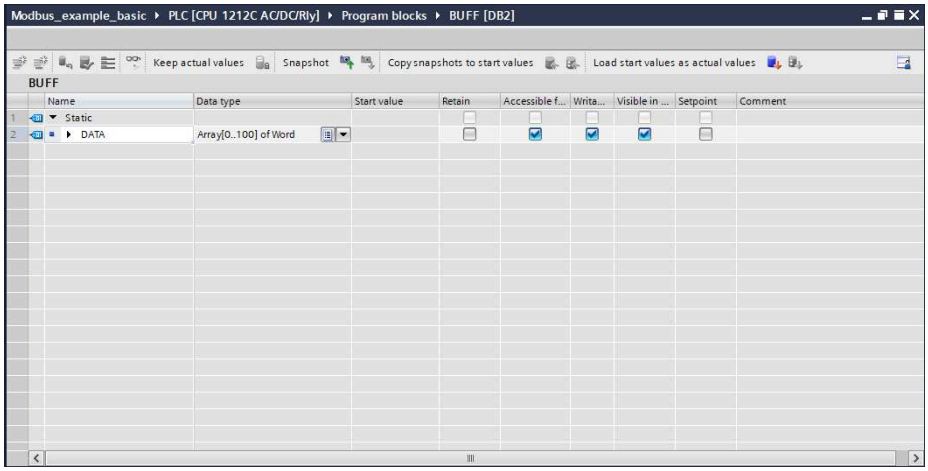
Now, define variables in the Data block. Use the same names and data types as in the image below. If you want to create a new row, just right-click on any row and select *Insert row.* Note that setting the right values of the *Start values* is essential in this step.



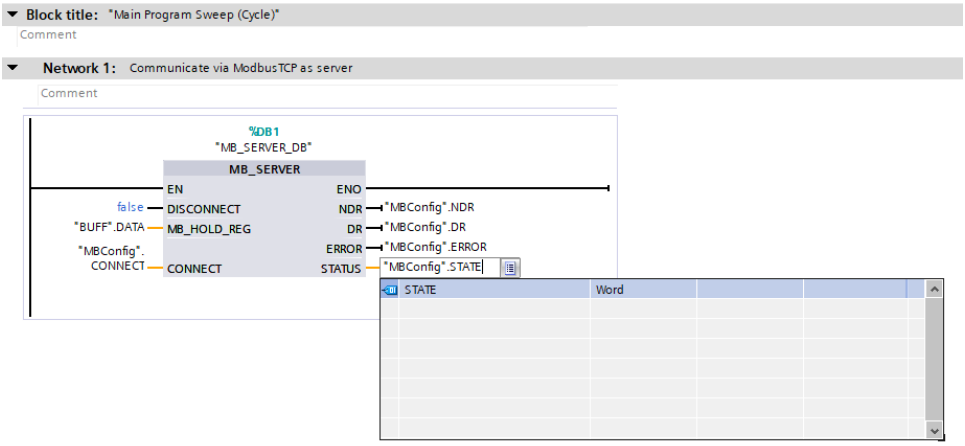
- BUFF - DB block serving as the communication buffer for the data received or sent between the PLC and the Aurora Vision Studio application



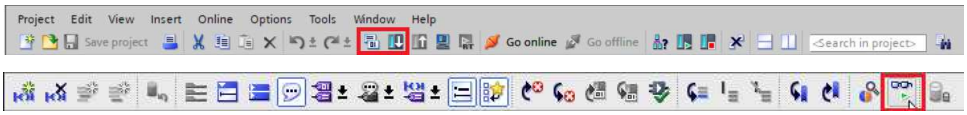
Create a word array in the BUFF data block, it should have the same size as it does in the image below:



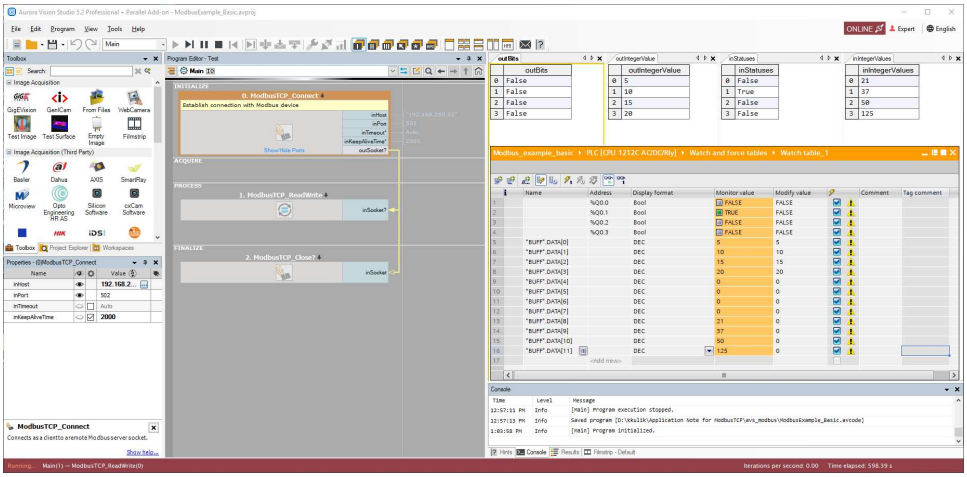
Attach the data block variables to the inputs and outputs of the MB_SERVER block in the way presented below:



If all previous steps have been performed correctly, your PLC program is ready. Compile the application and download it to the device. To see the current states of variables, turn on the Monitoring mode:



Use the application created in Aurora Vision Studio in the section where the Modbus TCP simulator was used. To test discrete inputs, coils and registers in TIA Portal, create a Watch table and a Force table, like in the image below:

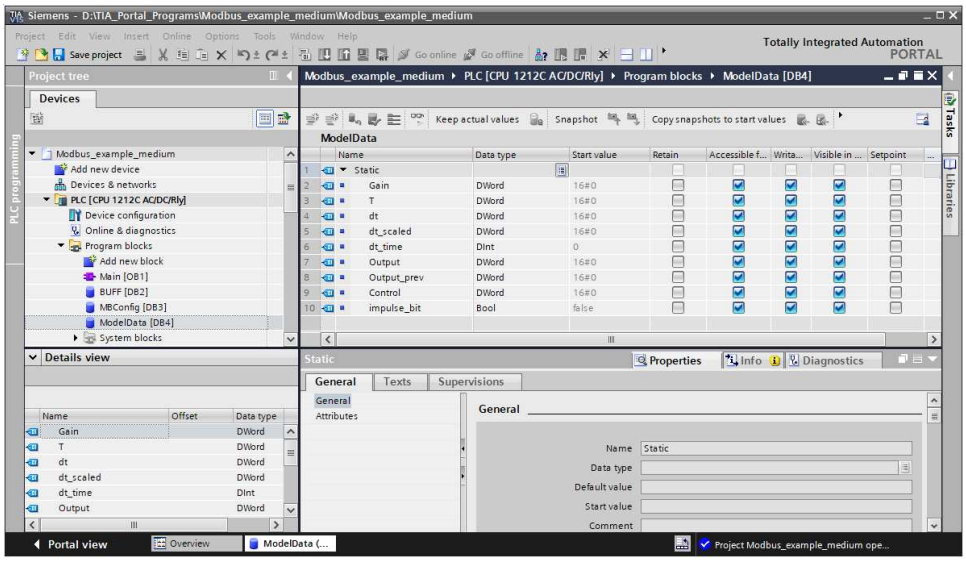


Advanced program with a PLC

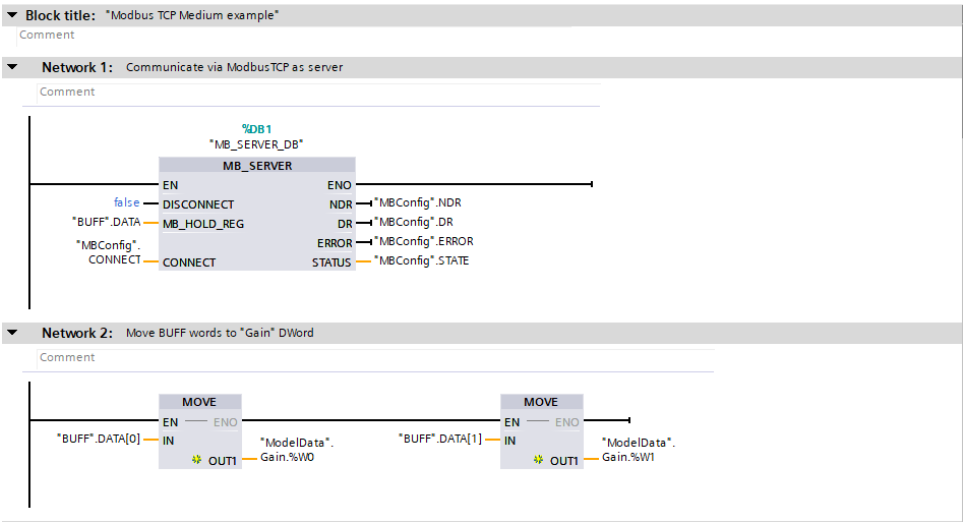
Creating a PLC program

This chapter describes how to modify a basic PLC program to meet requirements of advanced usage as mentioned at the beginning of this article. If you have created a Basic PLC program described in the previous chapters, save this project as a copy with a new name. Otherwise, create BUFF and MBConfig data blocks as described in this section.

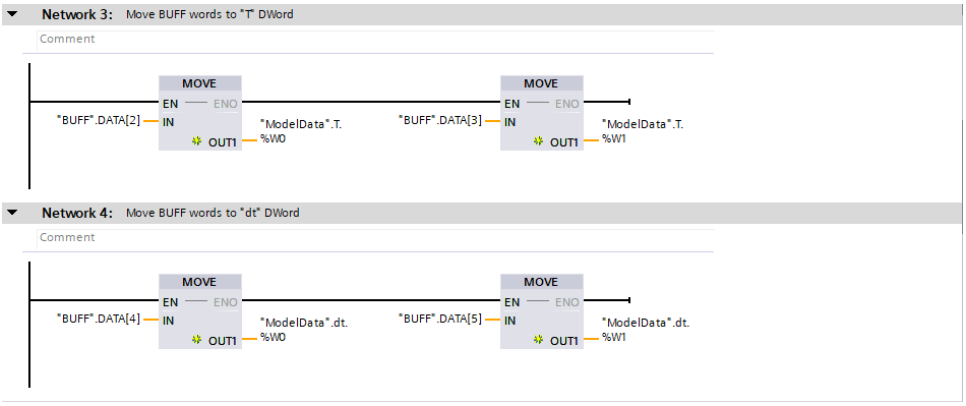
Create a new Data block called ModelData with the variables set like in the image below:



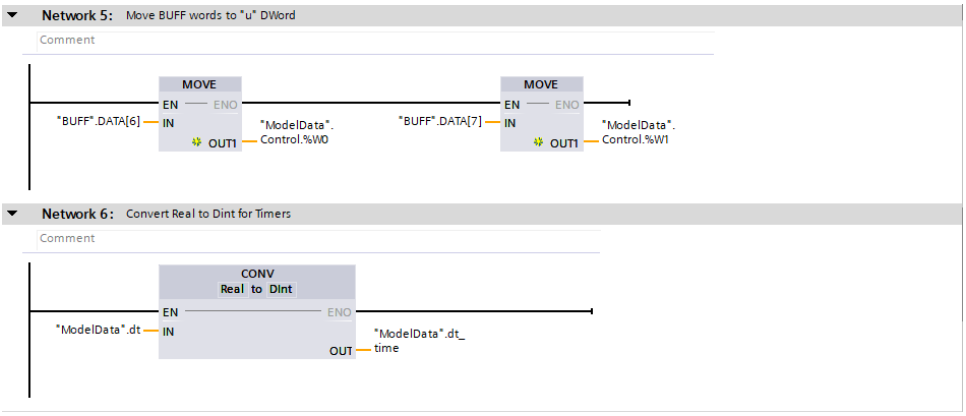
Nothing has changed for Network 1 of the Advanced program; MB_SERVER block is used to establish the Modbus TCP connection. For Network 2 use MOVE blocks to move appropriate BUFF data words to the ModelData.Gain variable.



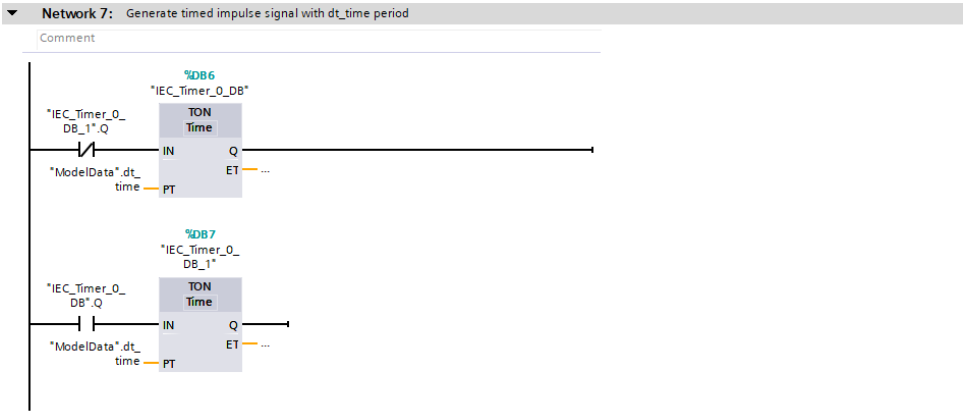
For Networks 3 and 4, use MOVE blocks to move BUFF words to ModelData.T and ModelData.dt.



For Network 5, use MOVE blocks to move BUFF words to ModelData.Control. For Network 6, use a CONV block to convert Real values to Dint values for further use in timers blocks.

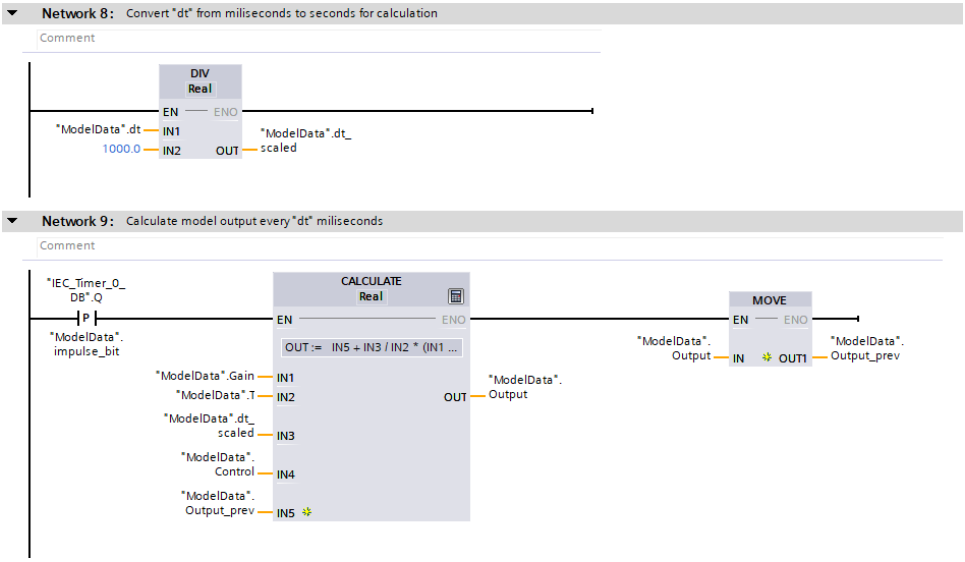


Network 7 uses TON Timer blocks to generate a timed impulse signal for further calculations. Create a new branch in this network, put TON blocks in the network and create Data blocks appropriate for them.

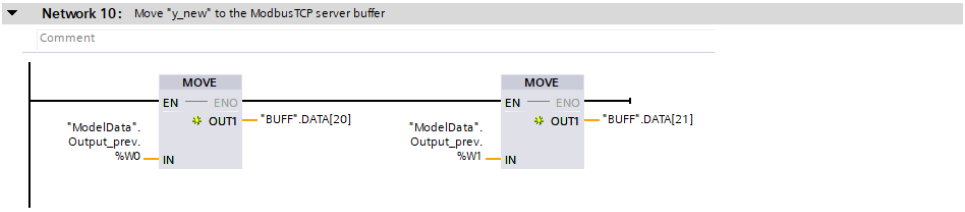


For Network 8, use a DIV block to scale ModelData.dt from milliseconds to seconds for further calculations. For Network 9, use the logic presented in the image below. The formula for the CALCULATE block should be as follows:

OUT:= IN5 + IN3 / IN2 * (IN1 * IN4 - IN5)

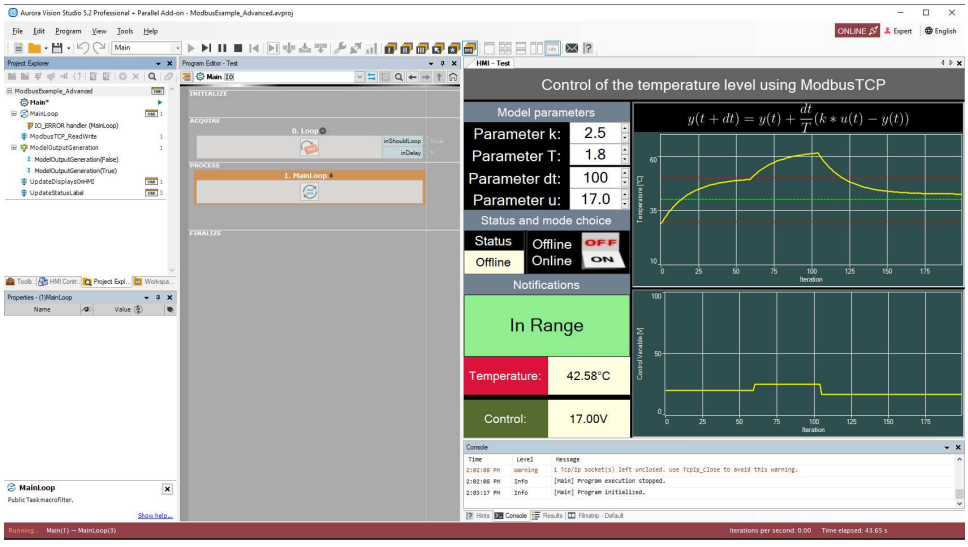


For Network 10, use MOVE blocks to move ModelData.Output_prev words to BUFF. This variable will be read in Aurora Vision Studio application.

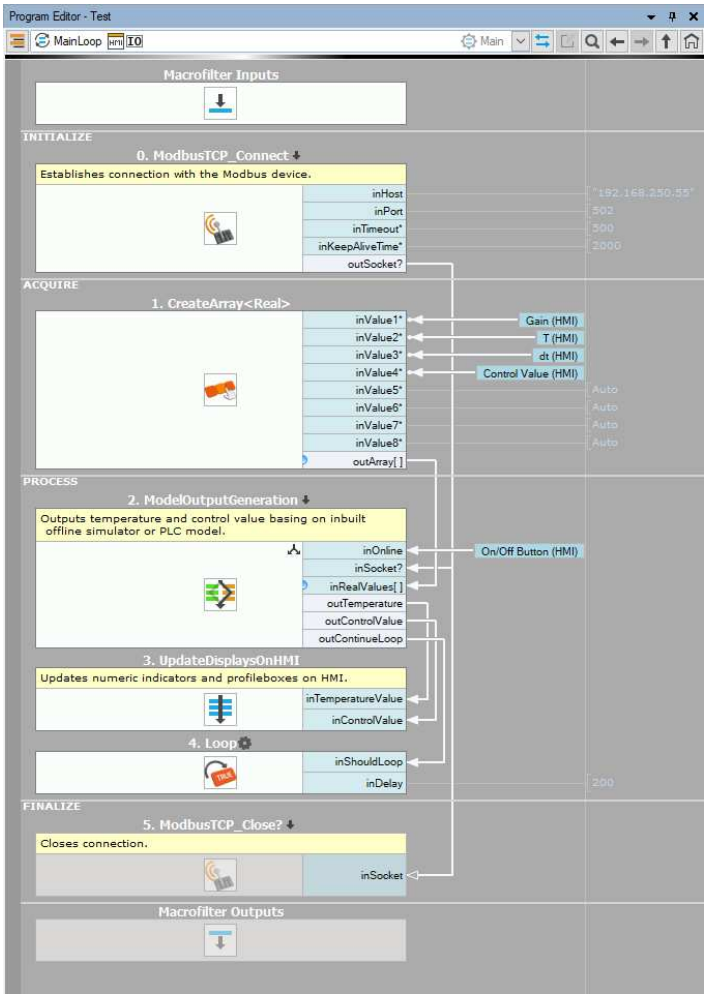


Creating Aurora Vision Studio program

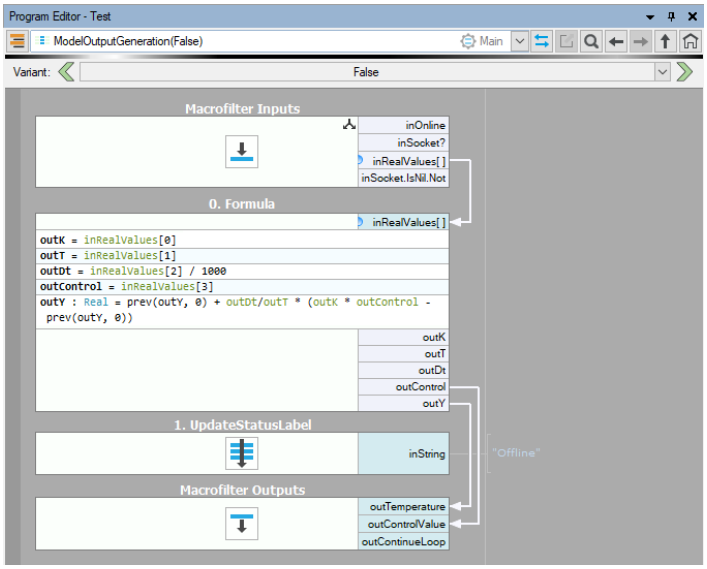
This Application Note does not explain how to create an HMI for an advanced program example. Take a look at the official Aurora Vision Studio example ModbusExample_Advanced.



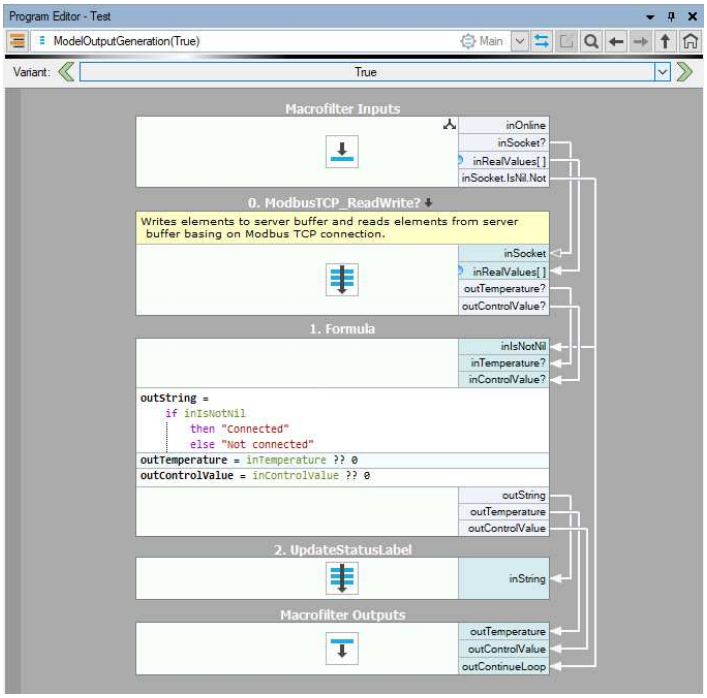
The *MainLoop* macrofilter body which is iterated every 200 ms:



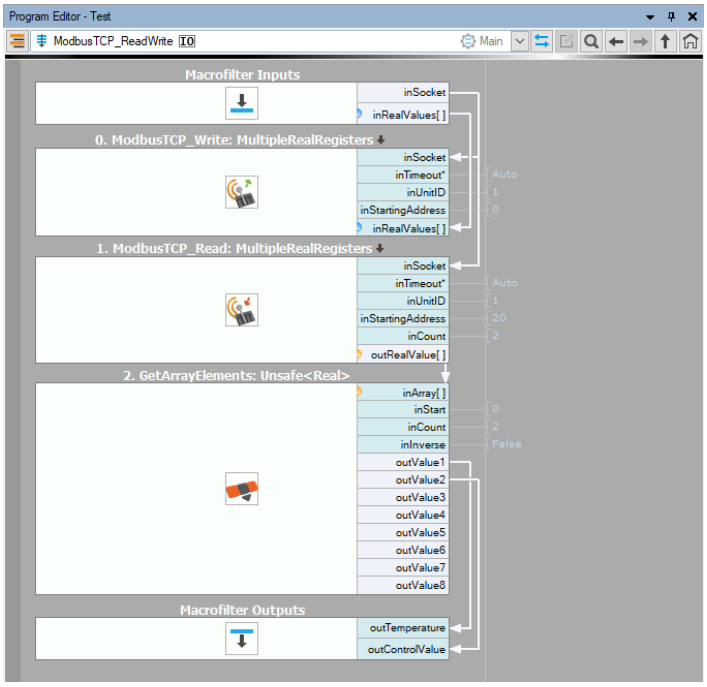
The False branch of the *ModelOutputGeneration* variant macrofilter:



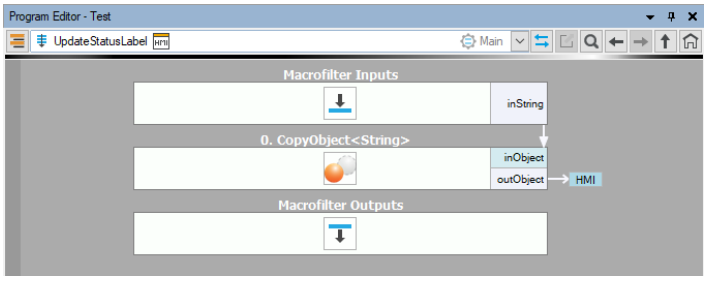
The True branch of the *ModelOutputGeneration* variant macrofilter:



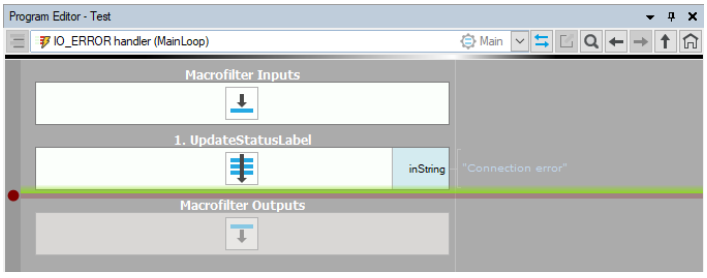
The *ModbusTCP_ReadWrite* macrofilter which is invoked in the online mode:



The *UpdateStatusLabel* macrofilter which is invoked in a few parts of the program; for instance, to notify the user about successful execution or an upcoming error:



An IO_ERROR error handler which is executed when connection in the online mode is interrupted:



Interfacing Wenglor profile sensor to Aurora Vision Studio

Purpose and requirement

This document explains how to interface a Wenglor profile sensor to Aurora Vision Studio.

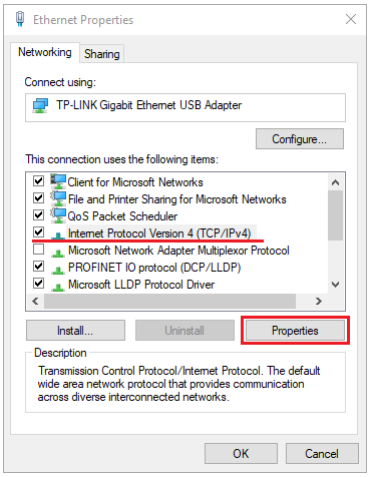
Required equipment:

- Wenglor weCat3D Server 2.0.0 or later. You can download it under **GigE Vision Interface** section (version 2.2.1 used);
- Aurora Vision Studio 5.0 Professional or later (version 5.2 used);
- Wenglor weCat3D sensor (MLSL123 used).

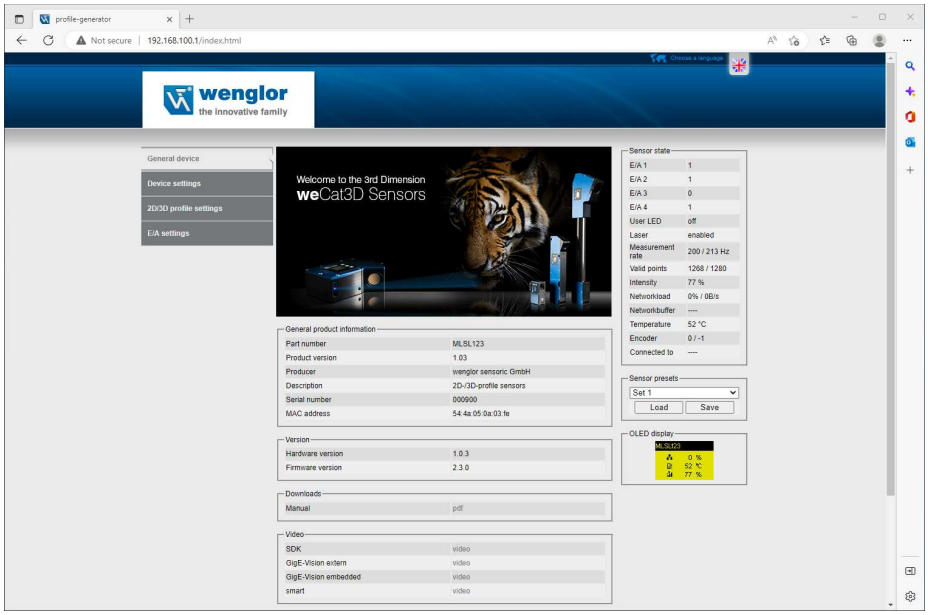
Setting up Wenglor weCat3D sensor

Each sensor is shipped with the default IP address of 192.168.100.1. So before working with the Wenglor you must make sure that the IP address has been set correctly and is unique for each device in the network. You can verify that with the steps described below:

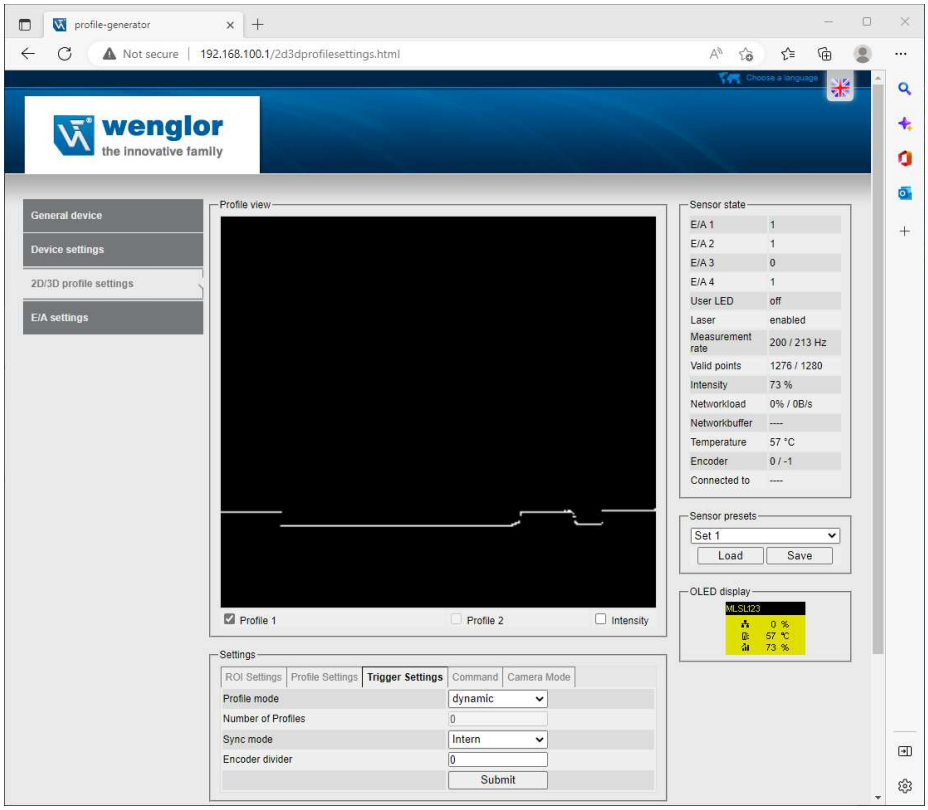
1. Connect the Wenglor to the PC running Aurora Vision Studio Professional.
2. Power up Wenglor, connect it to the PC via Ethernet interface.
3. Open Control Panel on your PC.
4. Find Network and Sharing Center.
5. Choose Change adapter settings.
6. Right-click on unidentified network connection, choose Properties.
7. In Ethernet Properties find Internet Protocol Version 4 (TCP/IPv4) and click Properties.



8. In General tab, choose Use the following IP address and set:
   a. IP address to (for example) 192.168.100.181
   b. Subnet mask to 255.255.255.0
9. Click OK and Close. Now you should be able to connect with the Wenglor scanner. You can verify this using your web browser.
10. In a search tab, type 192.168.100.1 and click Enter.
11. You should see this user interface:

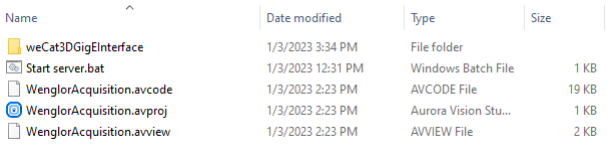


12. The provided browser-based user interface allows you to configure the parameters of the Wenglor sensor. You can navigate to the *2D/3D profile settings tab* to check if profile is acquired with *Intern* Sync mode:

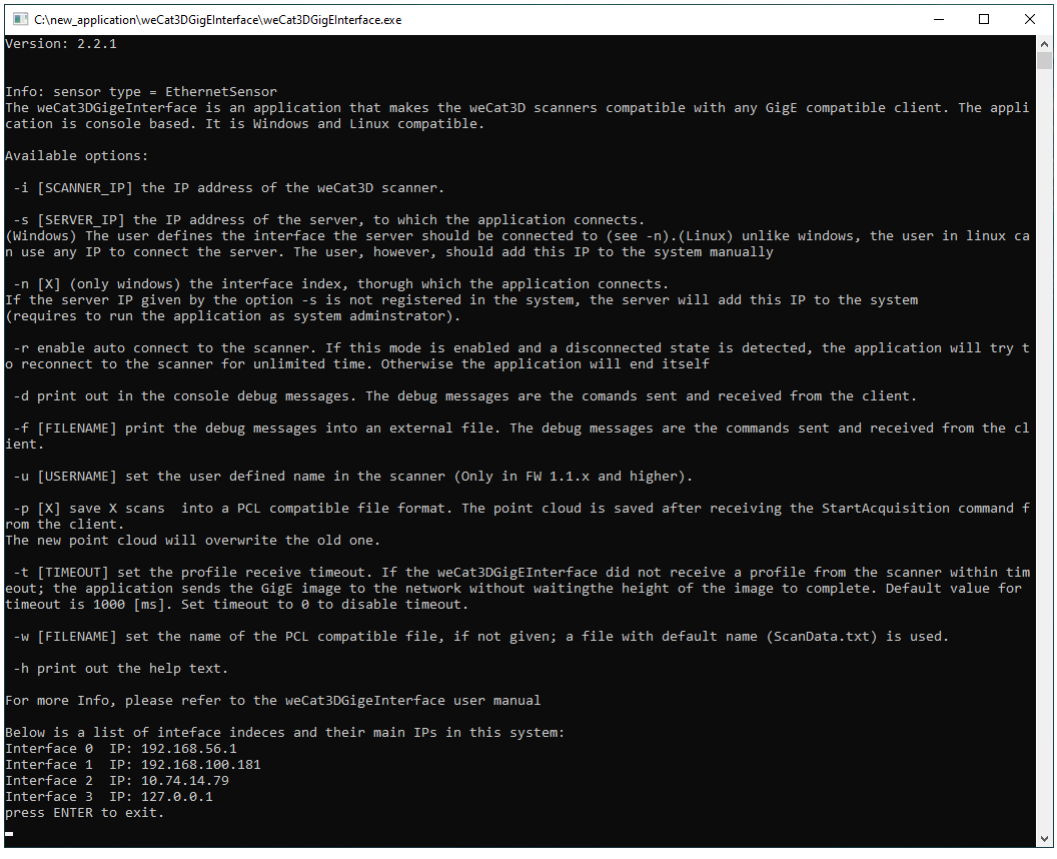


Setting up Wenglor weCat3D server

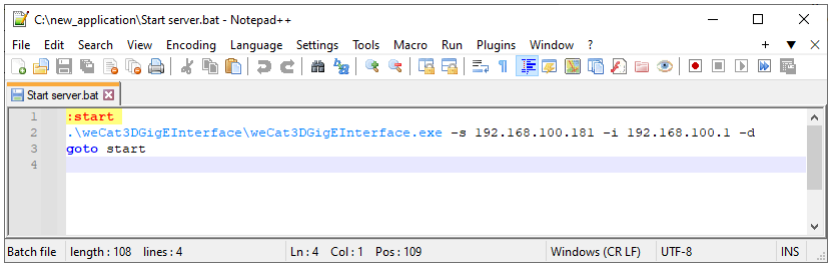
If you download Wenglor server archive, there should be a folder named "weCat3DGigEInterface". Copy it to the folder with Aurora Vision Studio project:



If you run Wenglor server executable, console application will be shown:



This console application should be run with appropriate arguments described in Wenglor server documentation. You can create file with *bat* extension with the following syntax:

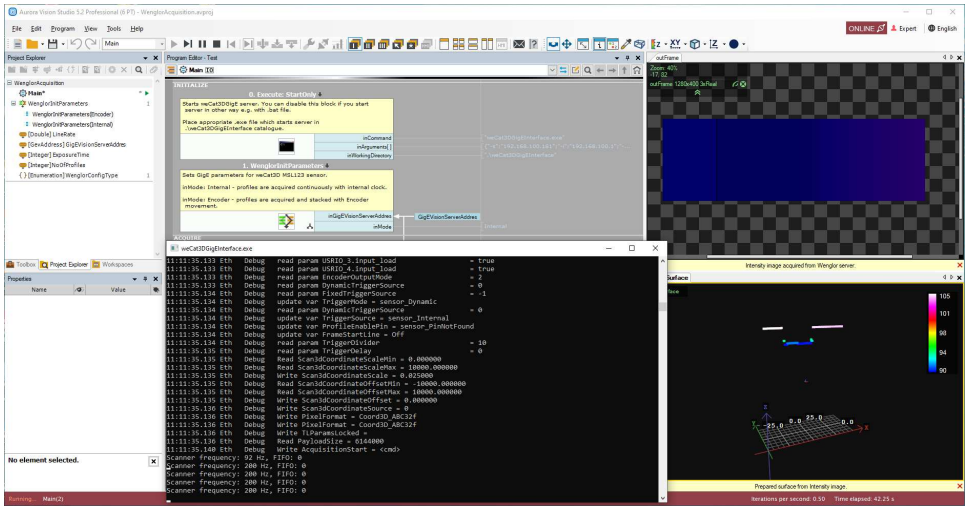


Using *bat* file extension is not obligatory and you can start server with appropriate arguments by *Command prompt*. You can start server with Execute_StartOnly filter - it is described in the next chapter.

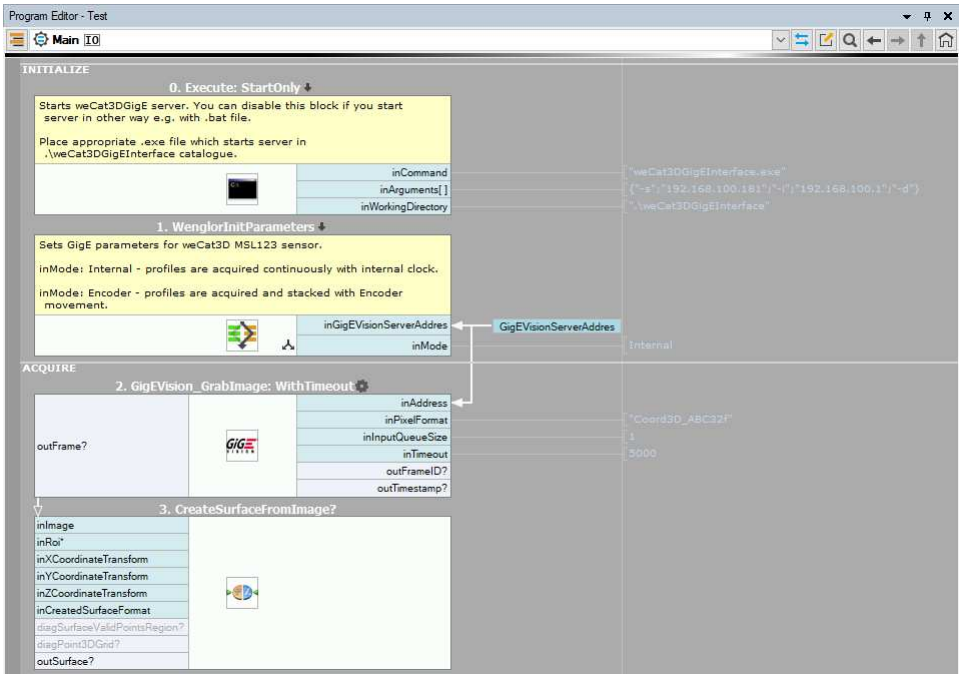
Setting up Wenglor weCat3D scanner with Aurora Vision Studio

In order to combine Wenglor server with Aurora Vision Studio, please refer to **Wenglor - GigE Server 2.0 Connection** official example.

Before you start this application, it is necessary to place "weCat3DGigEInterface" folder inside application folder. If everything is set up correctly, you should be able to acquire profiles with *Internal* mode:

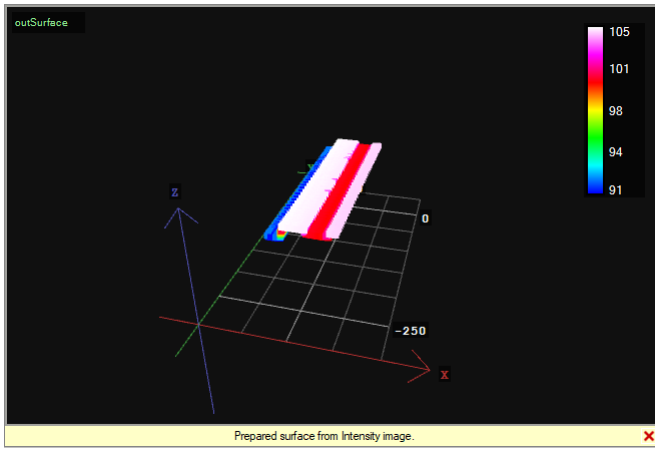


In INITIALIZE section Execute_StartOnly filter starts Wenglor WeCat3D server. Then **WenglorInitParameters** macrofilter sets appropriate parameters with **GigEVision_SetParameter** filters:



For more details about parameters initiation with Wenglor, please refer to the official example inside Aurora Vision Studio and to the manual attached to the downloaded WeCat3D Server.

If you change **inMode** from Internal to Encoder, you will be able to acquire whole surface from profiles acquired with encoder trigger source:



Interfacing Gocator to Aurora Vision Studio

Purpose and requirement

This document explains how to interface a Gocator sensor to Aurora Vision Studio.

A Gocator is an advanced sensor created by LMI Technologies. It is purposed for 3D machine vision and processing point clouds. The Gocator has many functionalities including:

- Scanning objects and representing them as a point cloud or intensity images,
- Various representations of scans,
- Basic tools for 3D processing like: filtering, thresholding etc.,
- Tools for performing measurements.

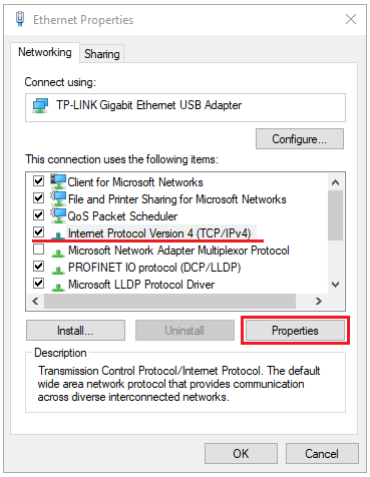
Required equipment:

- Gocator Firmware Release 3.2 or later
- Aurora Vision Studio 4.10 Professional or later

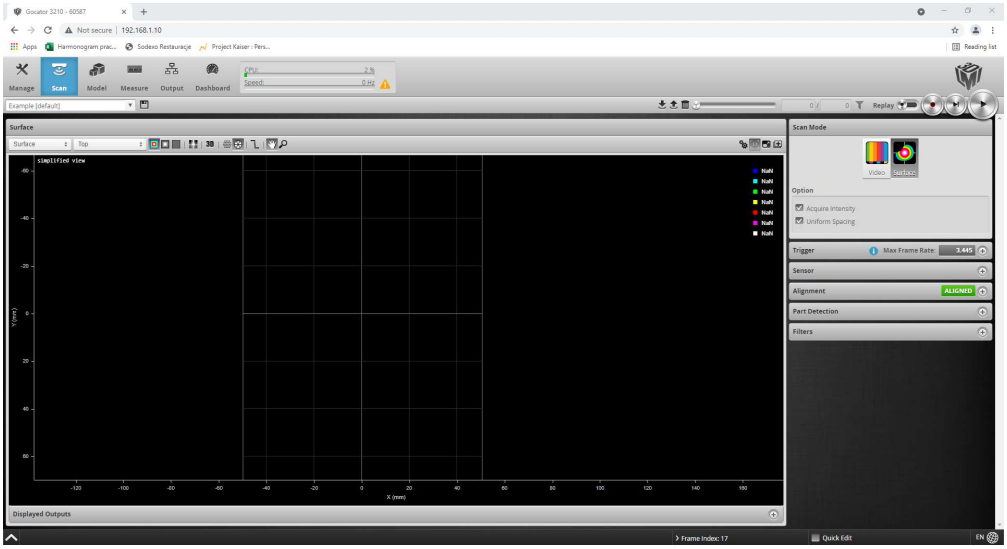
Setting up Aurora Vision Studio with a Gocator for the first time

Each sensor is shipped with a default IP address of 192.168.1.10. So before working with the Gocator you must make sure that the IP address has been set correctly and is unique for each device in the network. You can verify that with the steps described below:

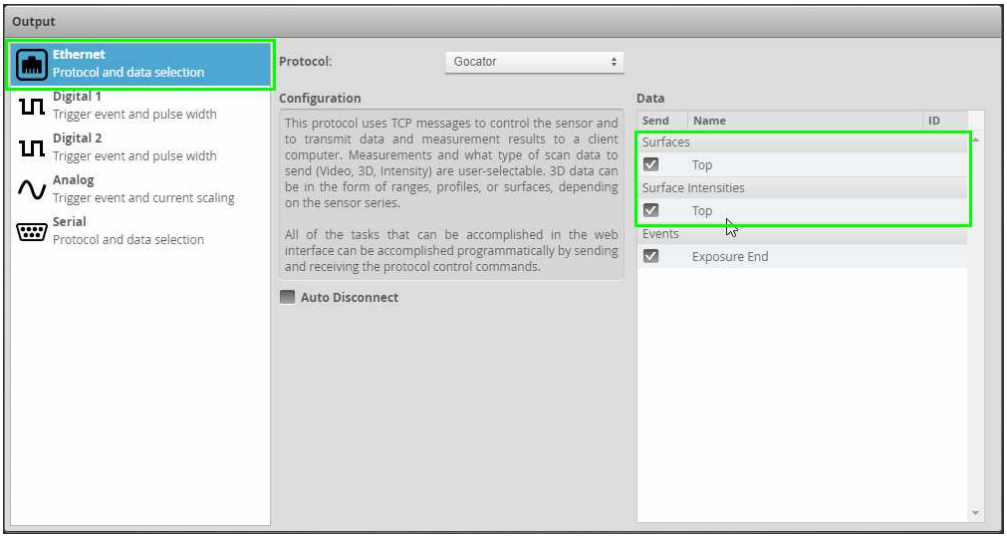
1. Connect a Gocator to the PC running Aurora Vision Studio Professional.
2. Power up the Gocator, connect it to the PC via Ethernet interface.
3. Open Control Panel on your PC.
4. Find Network and Sharing Center.
5. Choose Change adapter settings.
6. Right-click on unidentified network connection, choose Properties.
7. In Ethernet Properties find Internet Protocol Version 4 (TCP/IPv4) and click Properties.



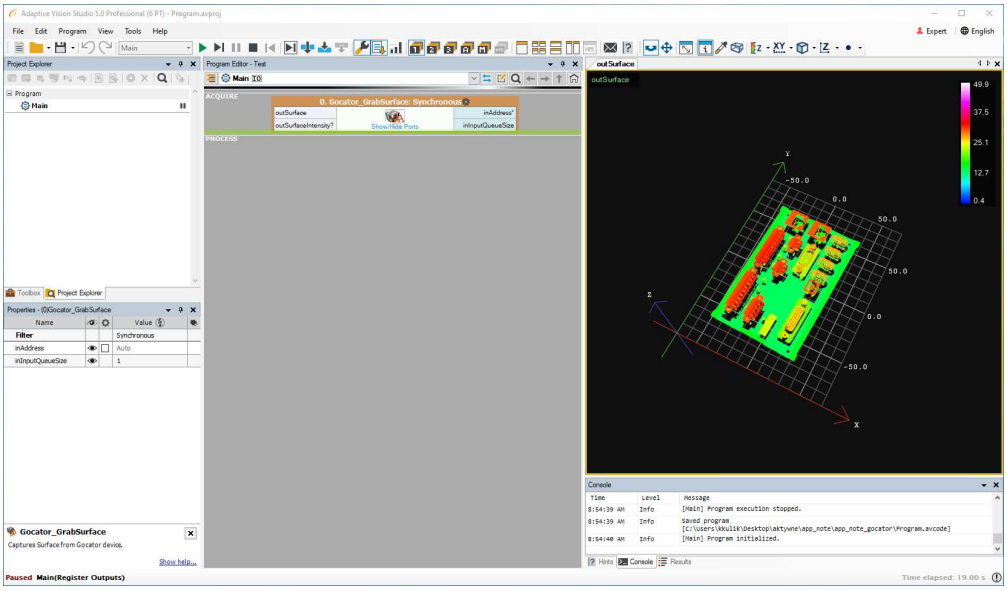
8. In General tab, choose Use the following IP address and set:
    a. IP address to (for example) 192.168.1.5
    b. Subnet mask to 255.255.255.0
9. Click OK and Close. Now you should be able to connect with the Gocator. You can verify this using your web browser.
10. In a search tab, type 192.168.1.10 and click Enter.
11. You should see this user interface:



12. The provided browser-based user interface allows you to configure the parameters of the Gocator sensor. Before working with Gocator in Aurora Vision Studio it is required to set parameters in the browser interface because from the Studio side you are unable to change them. In Aurora Vision you will see data the same as in the browser. To send specific data outputs to Aurora Vision Studio click on the "Output" tab and there select Ethernet output. Then you must choose the data to send. Note that marking all outputs is optional so it is possible to mark only these which are interesting for you. For example, with the setup shown below you will be able to acquire Surface and Surface Intensity in Aurora Vision Studio. For the outputs in the Measurement section there are also specified ID's. The same ID should be later set on the Aurora Vision side.



13. Now go back to Aurora Vision Studio. In Image Acquisition (Third Party) section find LMI and double-click. Choose **Gocator_GrabSurface** to grab a surface or **Gocator_GrabProfile** to grab a profile. Run or iterate the program. It is recommended to use the other option for testing. If everything works fine you will receive the point cloud or the profile of the object from your Gocator Note that this filter has an input inAddress, but it is not necessary to use it if there is only one Gocator connected with your PC. You must set this input if there are more sensors.



14. If no problems occurred as far, it means you have successfully connected Aurora Vision Studio with the Gocator.

Troubleshooting

If you had any problems during the connection process, you could find some solutions here.

- If you cannot connect with a Gocator, please verify your Ethernet connection. Unplug your Internet cable and plug the cable from the Gocator.
- Verify whether you have correctly set IP address.
- If the device cannot connect, use the Security button on your Master device.
- Make sure your browser uses the newest version of Flash.
- In the case that other page is displayed on the 192.168.1.10 address press Ctrl + F5 to force a cache refresh.

If you meet the problem we have not mentioned above, please let us know, so that we could investigate it and add it to this section.

Gocator filters in Aurora Vision Studio

Full list of filters intended to work with Gocator devices you can find under this **link**.

Working with Emulator

Gocator Emulator is a software that allows for running a virtual sensor instead of using the real one. Emulator's interface is identical as the one in a web browser. Thanks to that you can get acquainted with all functionalities that the Gocator provides. To start working download the proper tools package from: http://lmi3d.com/support. Then to run the application first unzip the package then find GoEmulator file and run it. The emulator lets you to change a language and select version. After launching the application start from step 12 according to the walkthrough in chapter 3. After starting GoEmulator, you must choose a scenario to be run. It is possible to display data from scenarios in Aurora Vision Studio. The connection between Aurora Vision Studio and GoEmulator should be set automatically - you do not have to set a specific IP.

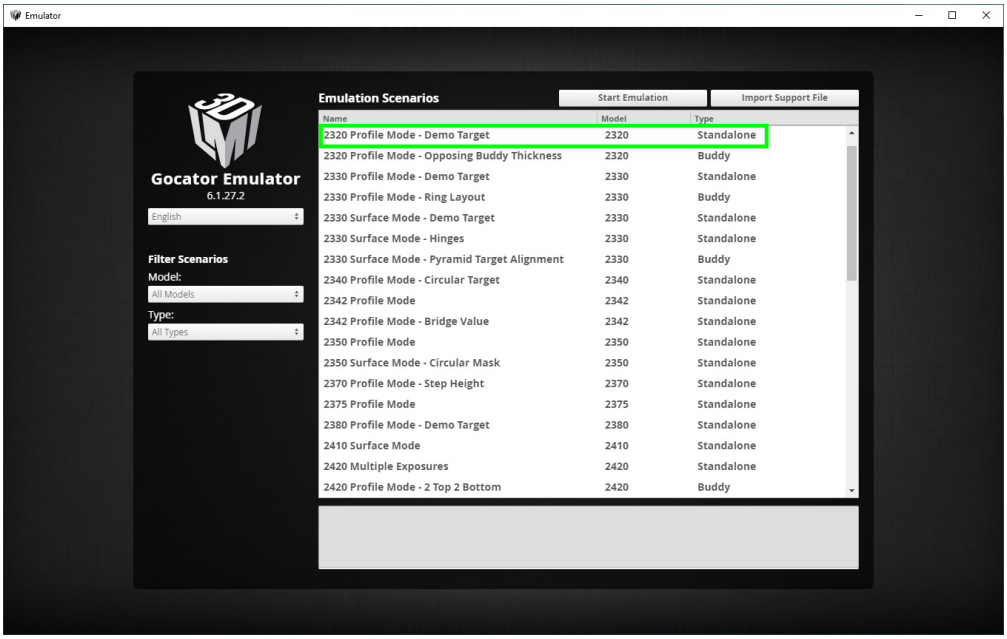
Examples

In this chapter several examples are presented to help you understand how to use functionalities described above and to help you learn how to work with the Gocator's user interface and filters in Aurora Vision Studio.

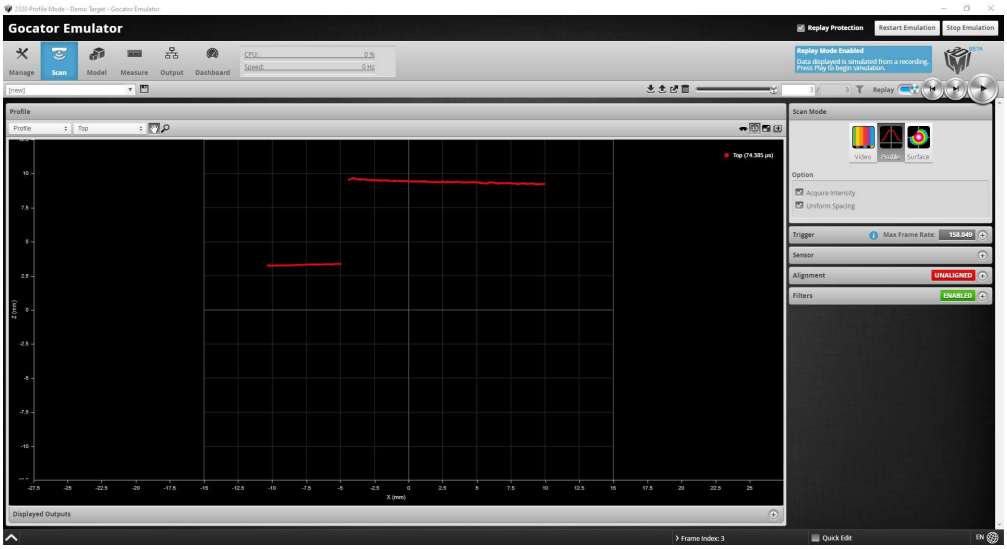
Example 1 - getting profile

To grab profile from a Gocator (in this case Emulator) follow the steps below:

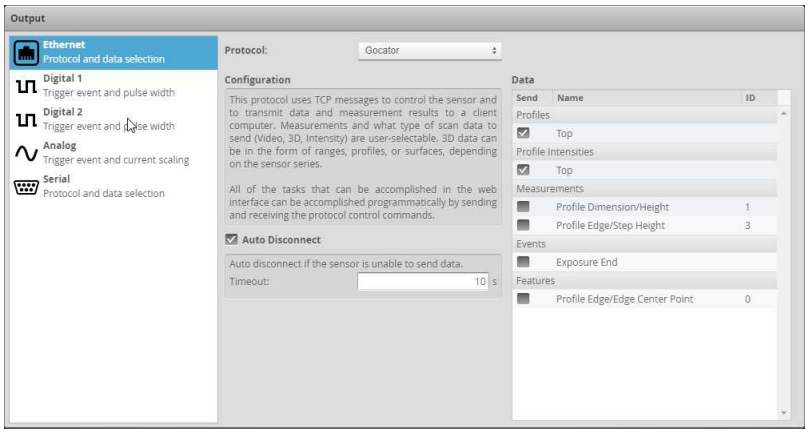
- Choose scenario:



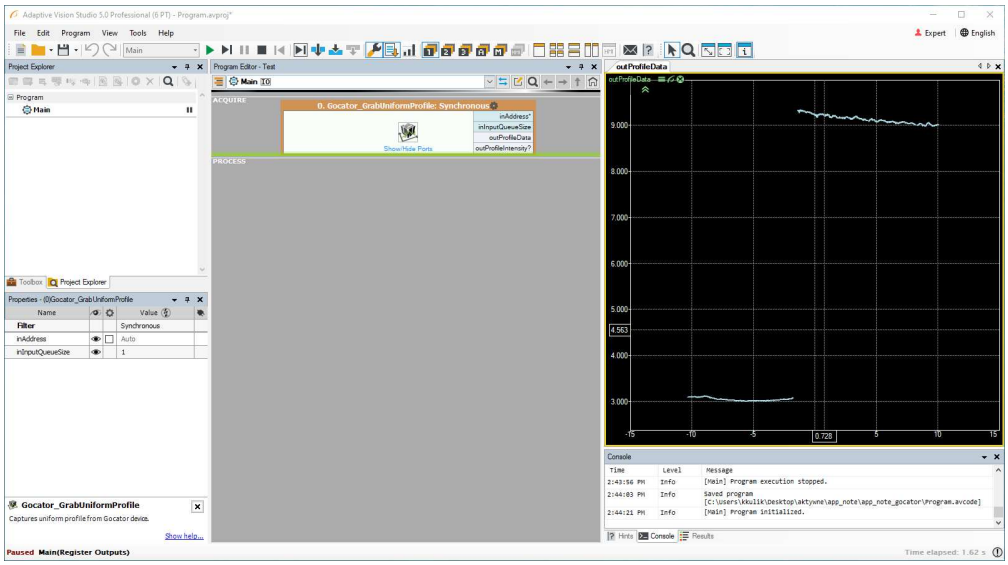
- In the "Scan" tab run the scenario:



- In the "Output" tab make sure that the profile top output is marked:



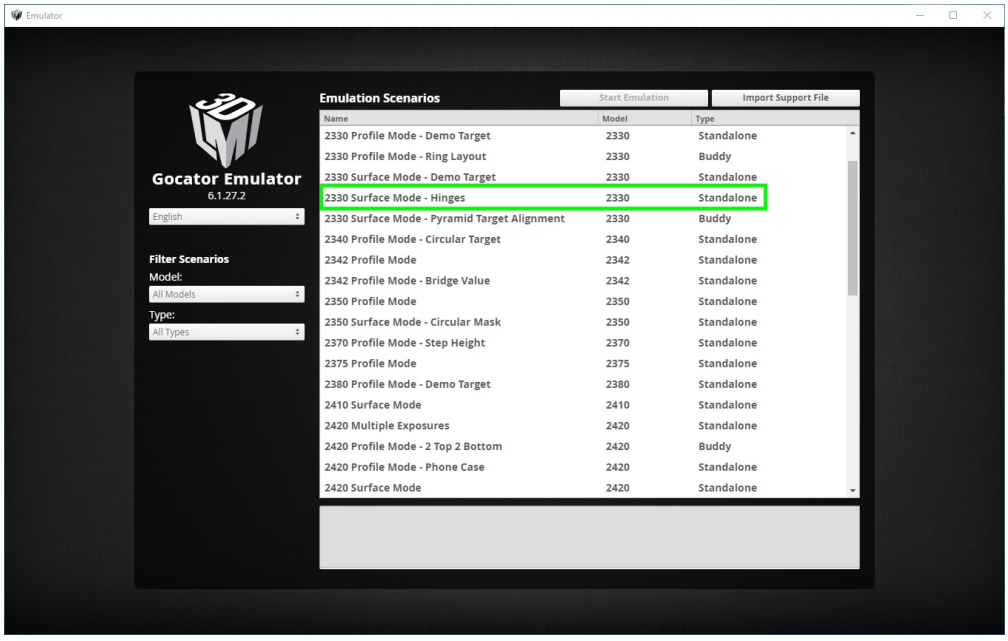
- In Aurora Vision Studio choose the Gocator_GrabUniformProfile filter and add the output **outProfileData** to a preview window:



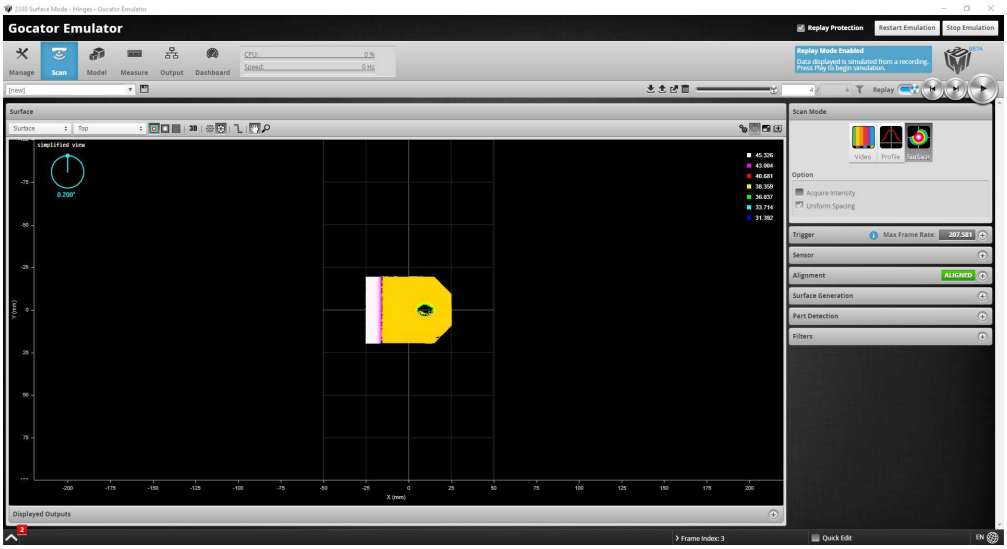
Example 2 - getting surface

To grab surface from a Gocator (in this case Emulator) follow the steps below:

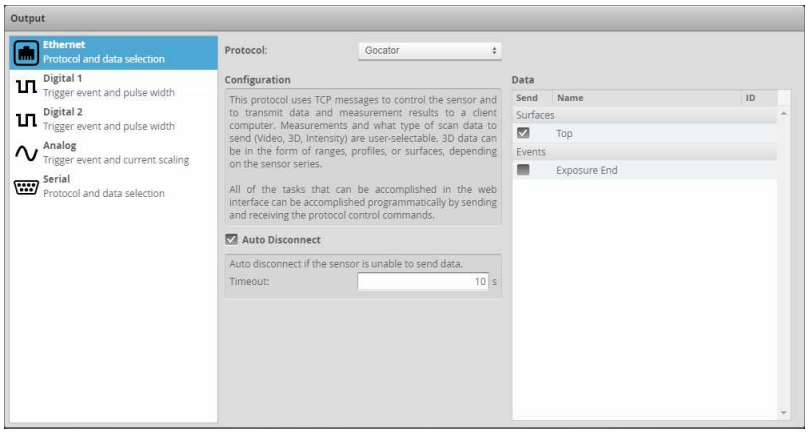
- Choose scenario:



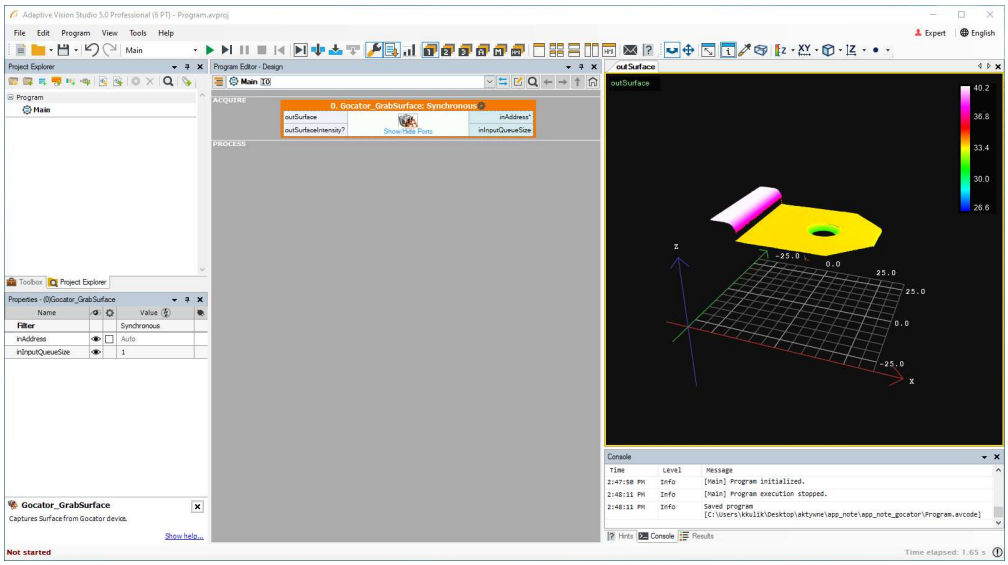
- In the "Scan" tab run the scenario:



- In the "Output" tab make sure that the profile top output is marked:



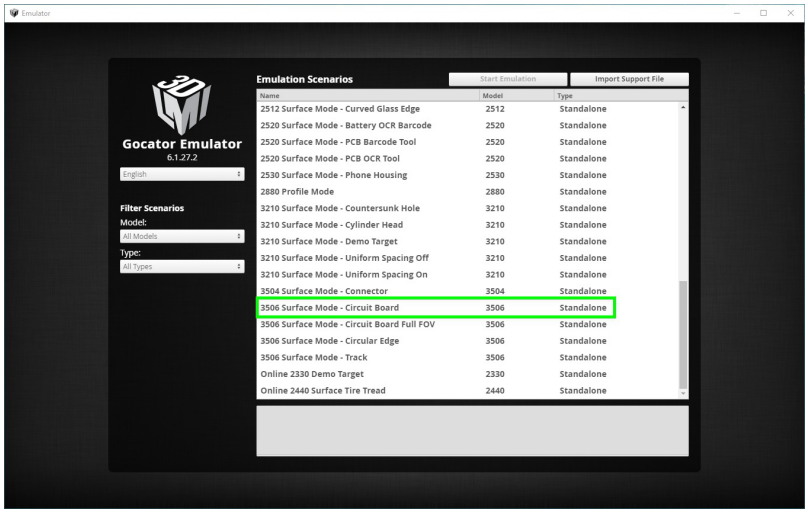
- In Aurora Vision Studio choose the Gocator_GrabSurface filter and add the output **outSurface** to a preview window:



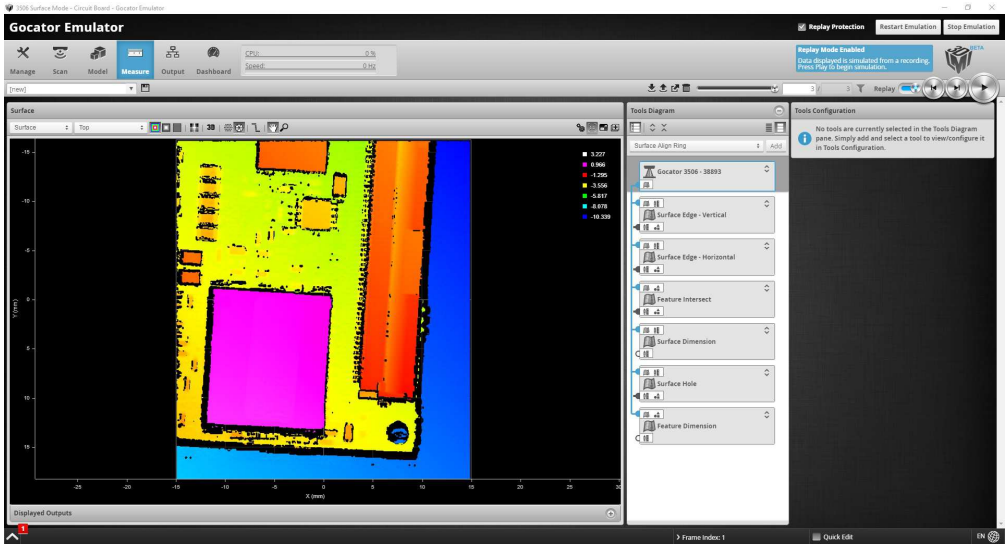
Example 3 - getting measurements

To grab measurement from a Gocator (in this case Emulator) follow the steps below:

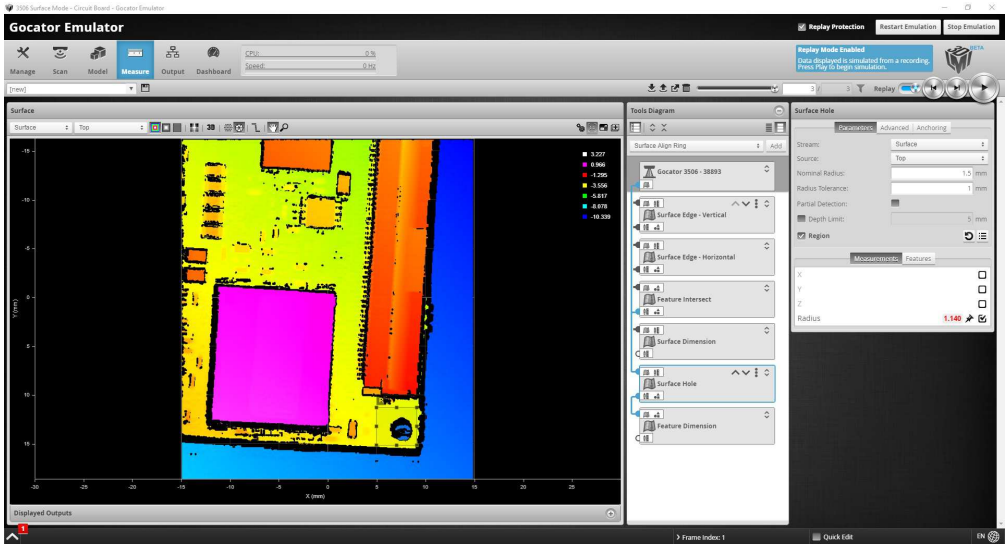
- Choose scenario:



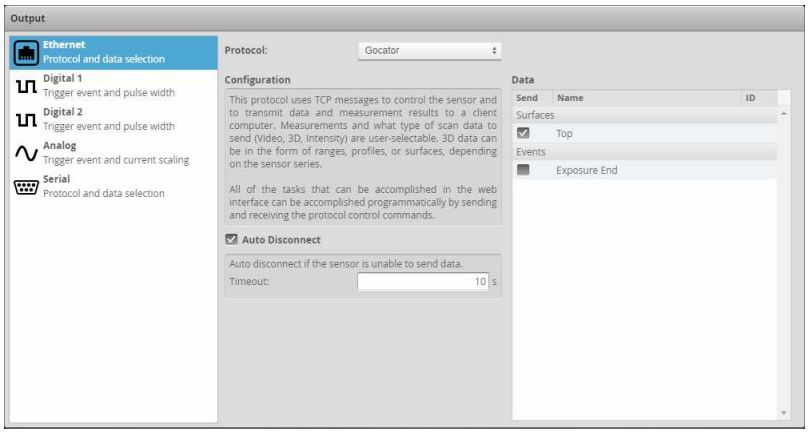
- In the "Scan" tab run the scenario:



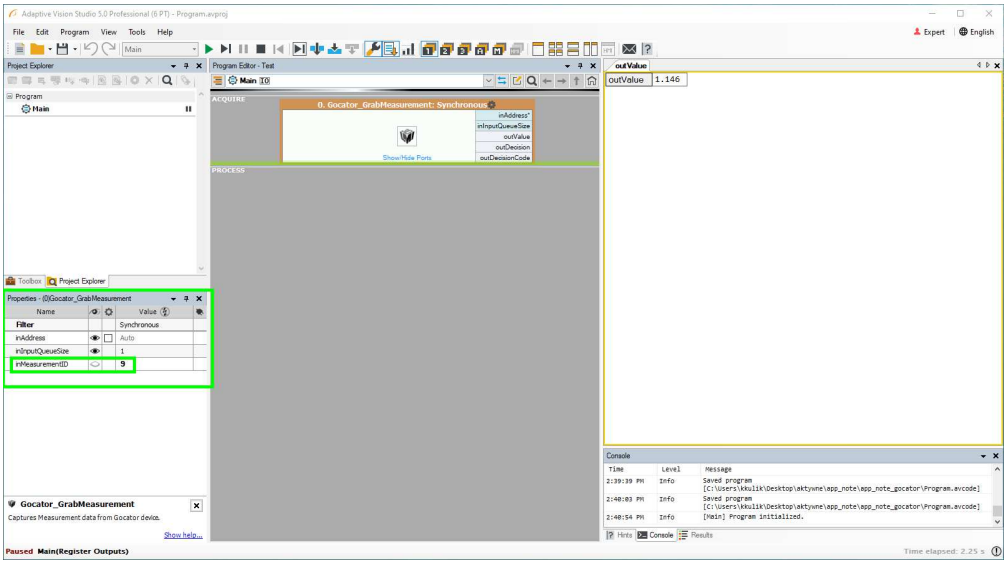
- In the "Measure" tab choose required measurements:



- In the "Output" tab make sure that the output data you would like to send is marked:



- In Aurora Vision Studio choose the Gocator_GrabMeasurement filter and add the output **outValue** to a preview window. Please note that the data output from a previous paragraph has a specific ID, so you must enter this value in Aurora Vision Studio - in Property window there is the parameter named **inMeasurementID** that should be set to 9, because our goal is to display the value of hole's radius:



Interfacing Hilscher card (Profinet) to Aurora Vision Studio

Purpose and equipment

This document explains how to configure Profinet PLC with Aurora Vision Studio using Hilscher Profinet card.

Required equipment:

- Supported Hilscher Profinet PC card
- Aurora Vision Studio 4.12 or later

Hardware connection

All devices must work in a common LAN network - henceforth called shared network. In most cases they are connected to each other through a network switch.

The devices shall be connected as follows:

- Master device e.g. PLC to the shared network
- Hilscher card to PC
- Hilscher card to the shared network
- PC's network card to shared network

Before proceeding to the further point, make sure that all devices are powered up.

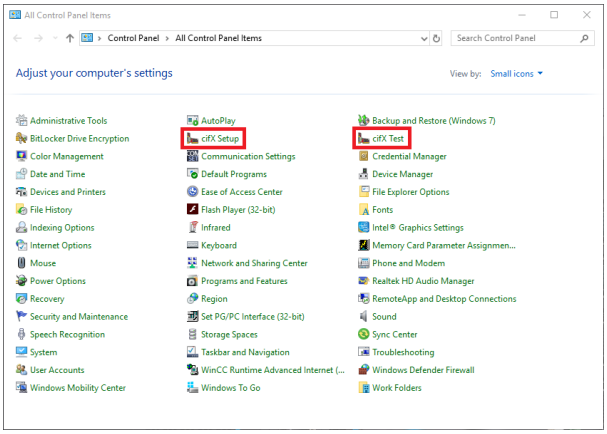
Configuring Hilscher Devices

- Be sure to have the newest version of the following drivers:
  - Windows driver (recommended 1.5.0.0): download link (for CIFX 50E-RE download V2.x version)
  - SYCON.net for configuring slots, generating configuration files: download link
  - For Profinet: Firmware (recommended 3.13 series) download link
- A computer restart is highly recommended after driver installations.

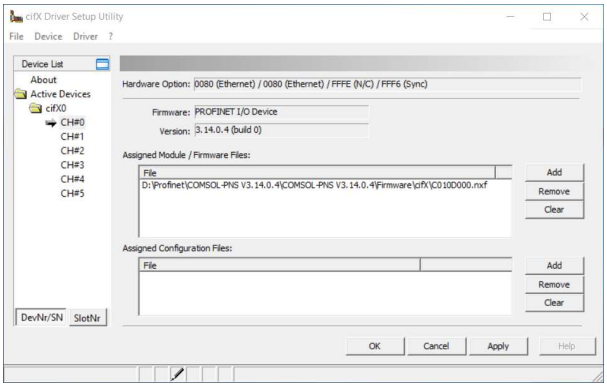
Make sure you have the recommended driver installed, newest SYCON.net, and firmware prepared. Profinet requirements are written down in Hilscher_Channel_Open_Profinet documentation.

1) Configuration using cifX Driver Setup Software

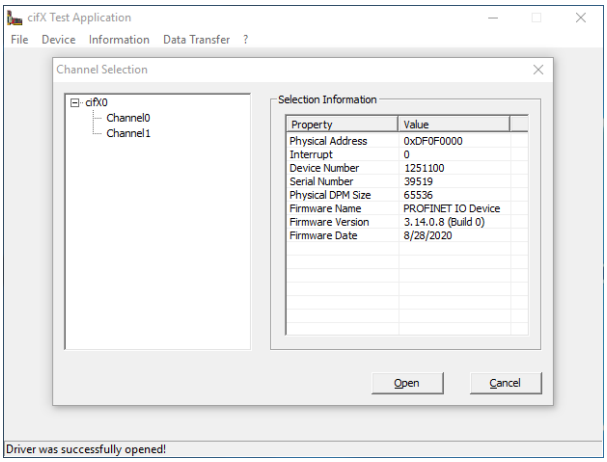
The easiest way to install the firmware to the cifX Driver is using the cifX Driver Setup Software. You can find tools for this software for instance using Control Panel menu as in the picture below.



Using cifX Setup application select the channel you would like to configure (usually channel - CH#0) and remove preexisting firmware by clicking "Clear". Then click "Add" to add a new firmware file and navigate to the downloaded Profinet Adapter firmware (path \COMSOL-PNS V3.14.0.4\Firmware\cifX). Then click "Apply" and finish configuration with "OK".

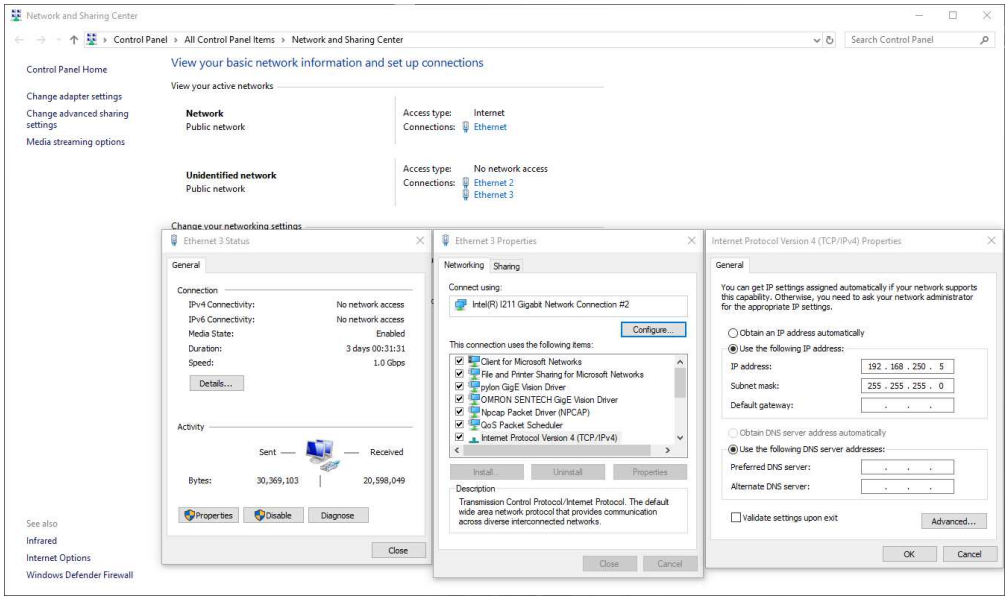


To be sure that firmware is installed properly you can open cifX Test software and then navigate to Device -> Open to achieve the result like below.



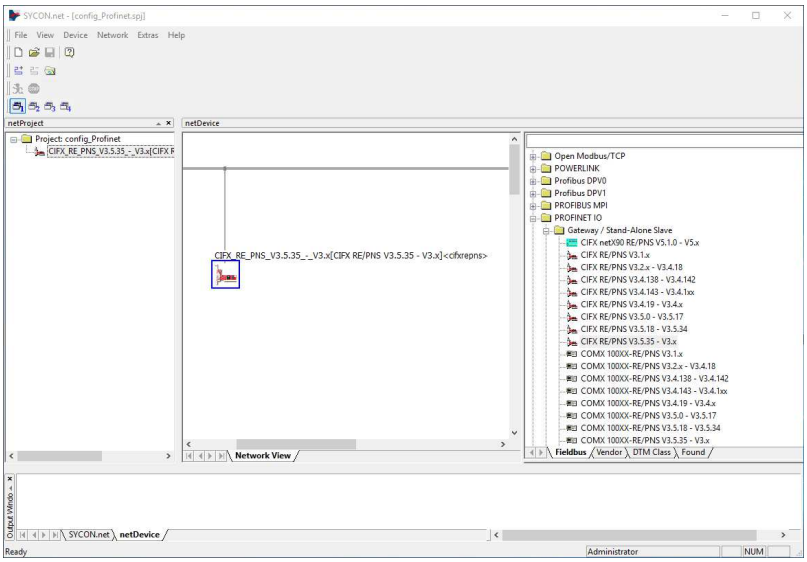
2) Configuration of the network card

It might be necessary to change the IP of the network card from dynamic to static. In this tutorial the following settings are used.

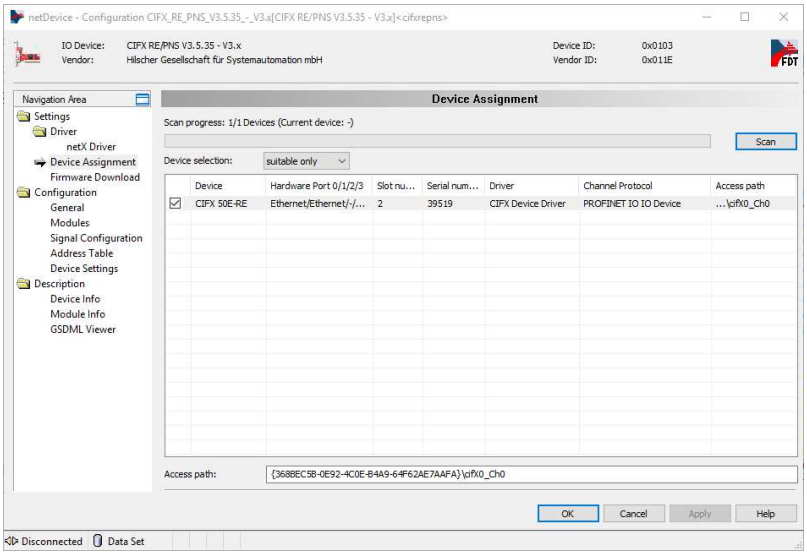


3) Configuration using SYCON.net

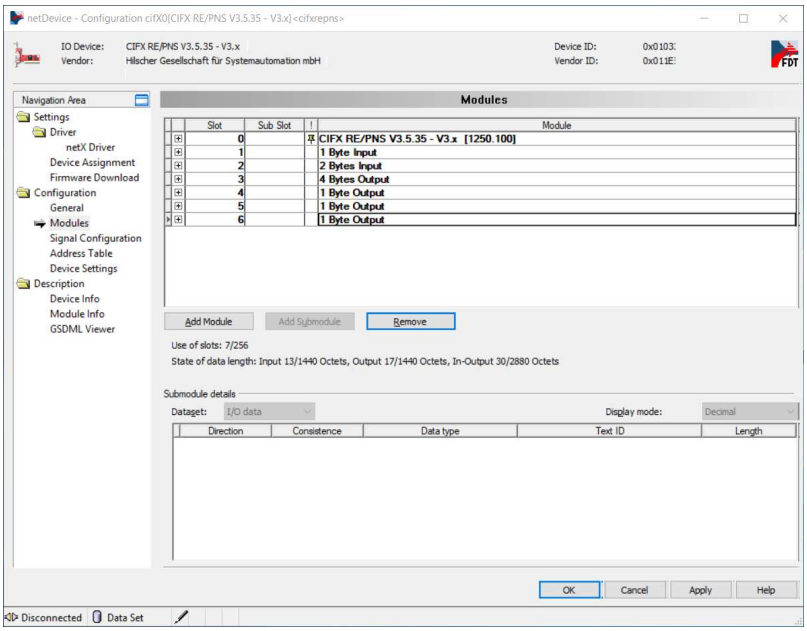
Configuration files are generated in SYCON.net software. Firstly, pick a proper device and drag it to the gray bus on the "Network View". This device you can find navigating to "PROFINET IO -> Gateway / Stand - Alone Slave -> CIFX RE/PNS V3.5.35. - V3.x".



To open the configuration card double click on dropped icon and in the popped-up menu navigate to "Settings/Device Assignment". Then scan for devices by clicking "Scan", mark found one it with a tick like on the image below and then click "Apply".



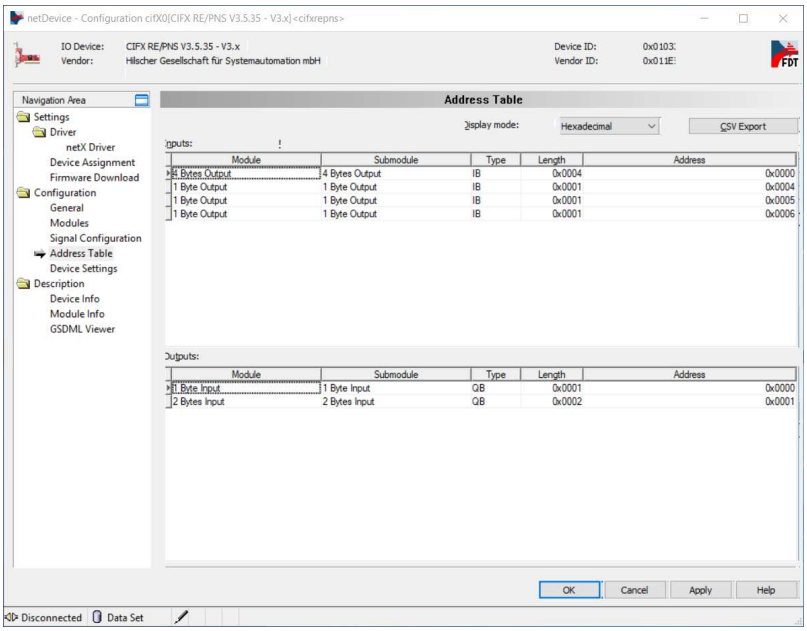
To open card configuration, double click on its icon and select the "Configuration/Modules" category in the Navigation Area on the left side of the dialog. In the main window you can add individual modules. Remember that in Profinet, **Slot configuration must match the configuration from your master device, otherwise the connection will not work**. For boolean indicators we recommend "1 Byte Output" or "1 Byte Input".



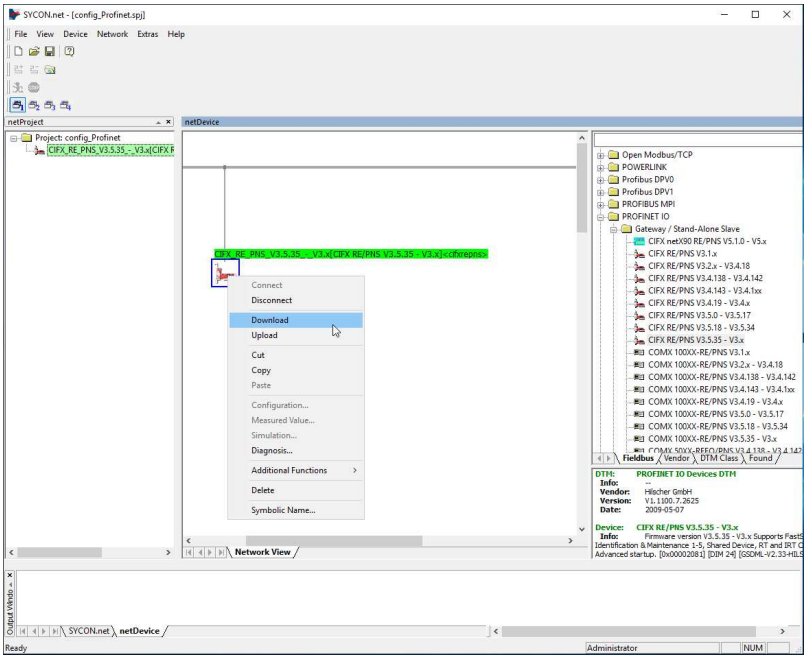
Confirm all changes by clicking "Apply" and then click "OK".

You can see the final address table (addresses on the Hilscher device where input or output data will be available) in the "Configuration/Address table" category. If you plan to use *IORead* and *IOWrite* filters, for example Hilscher_Channel_IORead_SInt8, note the addresses. You can switch the display mode to decimal, as Aurora Vision Studio accepts decimal addressing, not hexadecimal. Aurora Vision Studio implementation of Profinet checks whether the address and data size match. In the sample configuration below, writing a byte to address 0x002 would not work, because that

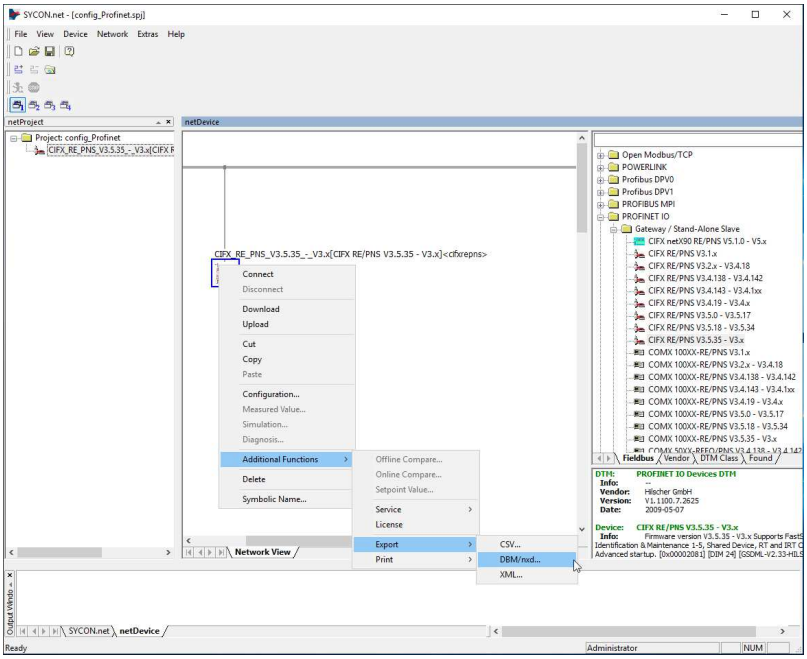
module address starts at 0x001 and spans 2 bytes. Moreover, Aurora Vision Studio prohibits writing with a *SlotWrite* filter to input areas and reading with a *SlotRead* filter from output areas. Click "OK" when you have finished the proper configuration.



Then you have to download the configuration to the device like on the screen below.



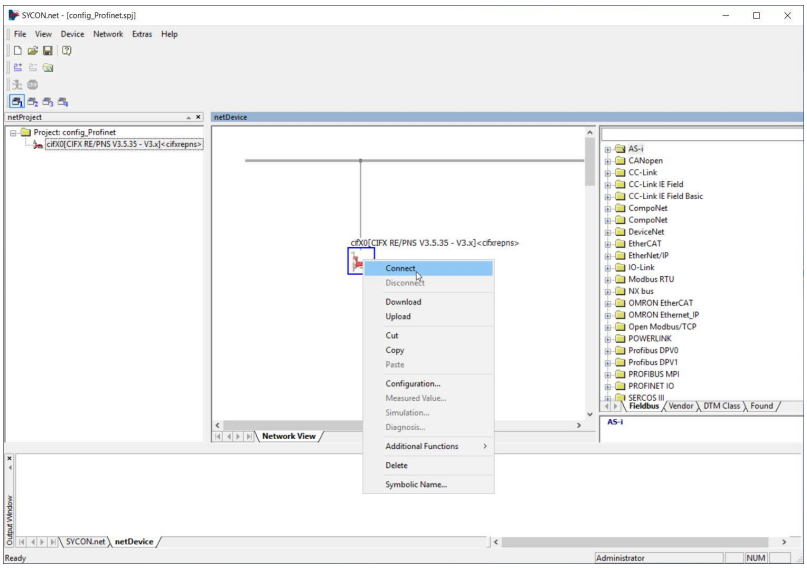
The final step is to generate configuration files for Aurora Vision Studio. You can do this by right-clicking on the device icon, then navigating to "Additional Functions -> Export -> DBM/nxd..", entering your configuration name and clicking "Save". You can now close SYCON.net for this example, **remember to save your project before, so that it is easier to add new slots.**



4) Working with Profinet in Aurora Vision Studio

The full list of filters for communication over Hilscher devices is available under this **link**.

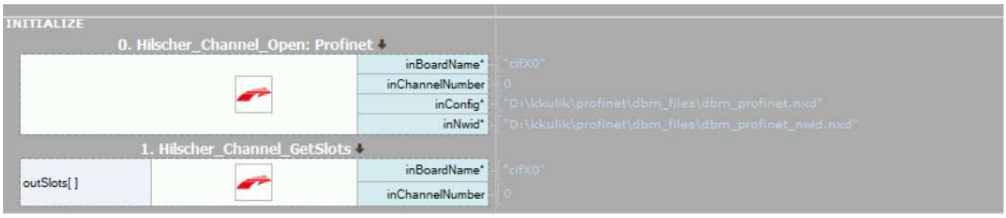
Before proceeding to the further steps, please check Hilscher and AVS connection using Hilscher_Driver_GetBoardInformation filter. If all previous steps are done correctly you will be able to get device board information including BoardName and ID. Please note that in this step the connection with the Hilscher card via Sycon.net is required, so you need to run the software, right-click on the card icon and click *Connect*.



As described in previous chapter the Sycon.net application is not required for a proper operation. On the contrary, it is highly recommended to keep it closed and use the generated configuration files to guarantee high stability of the connection.

After following the instruction from chapter 3, you have a xxx.nxd and xxx_nwid.nxd files. To send data over Profinet, follow the below steps:

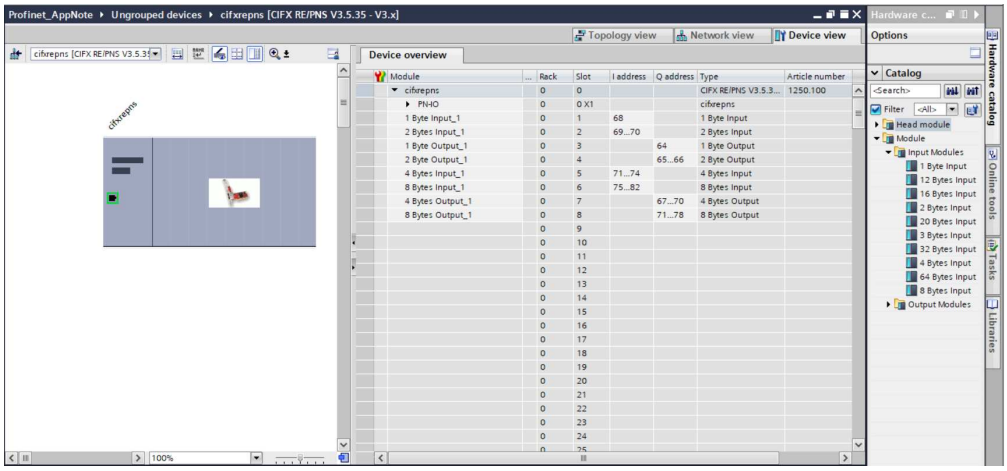
1. Configure connection in AVS with Hilscher_Channel_Open_Profinet filter. Drag it from Program I/O category in the Toolbox and drop it in the Program Editor.
2. In **inBoardName** input enter value acquired from Hilscher_Driver_GetBoardInformation filter. In **inConfig** and **inNwid** inputs enter paths to generated files from Sycon.net.

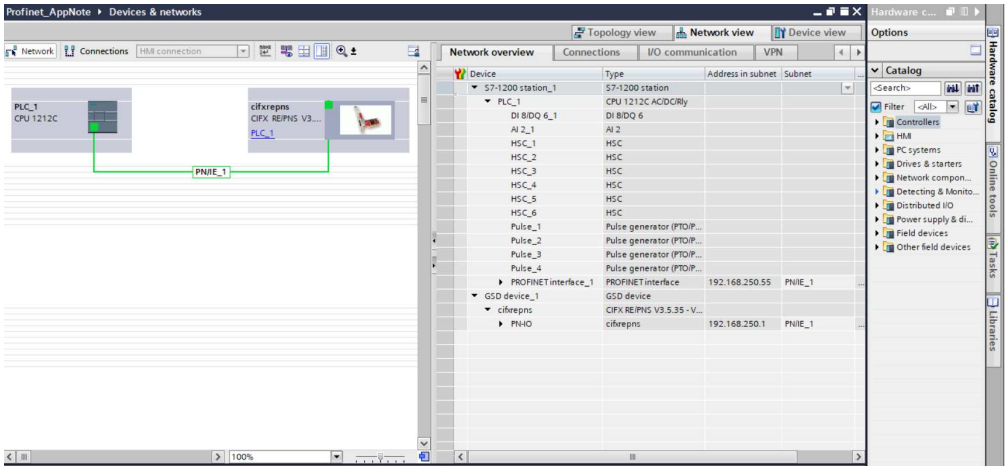


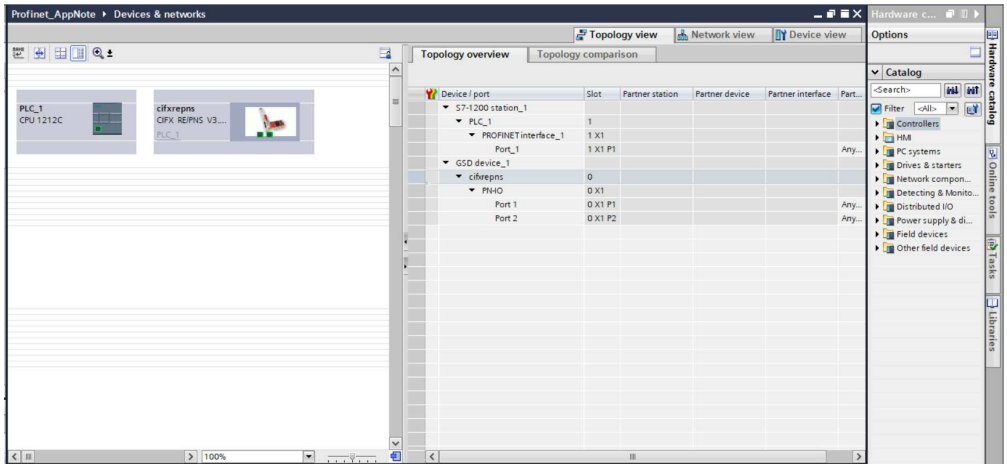
3. To prevent from I/O errors make sure that you open connection in the initialization step with Hilscher_Channel_Open_Profinet and close it at the end of the program using Hilscher_Channel_Close filter.
4. For IO, we recommend *SlotRead* and *SlotWrite* filters, as they are more convenient. For example, the Hilscher_Channel_SlotWrite_SInt8 filter writes 8 bytes of signed data to the selected slot. Slot numbers match those in the "Configuration/Modules" category of card configuration in SYCON.net program.

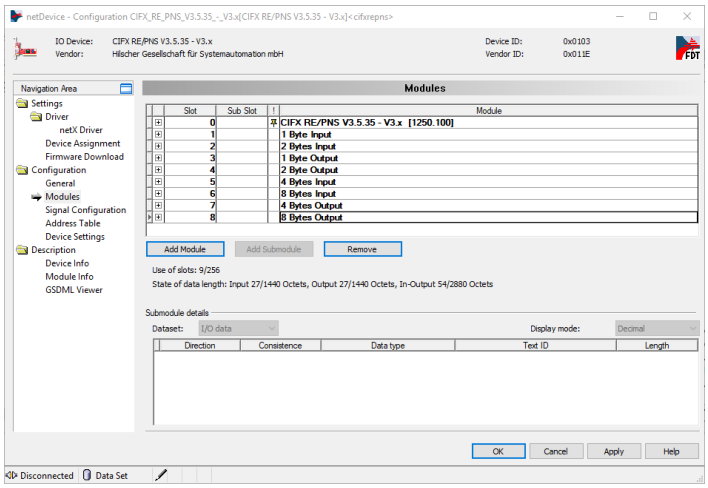
5) Example Configuration

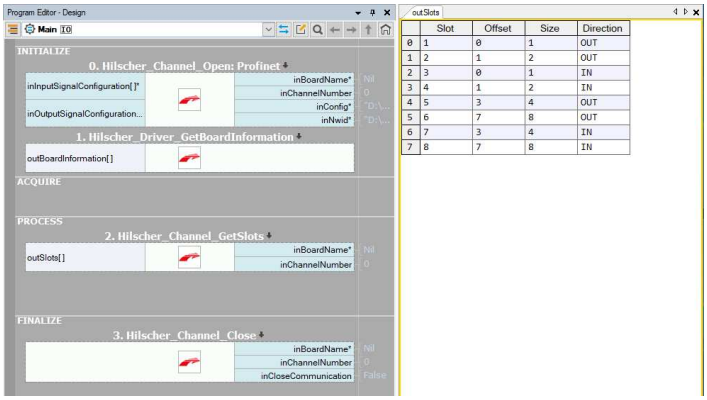
Below you will find an sample configuration of Siemens TIA Portal, Hilscher Sycon and Aurora Vision Studio.


Profinet_AppNote ▸ Ungrouped devices ▸ cifxrepns [CIFX RE/PNS V3.5.35 - V3.x]
Topology view
Network view
Device view
Device overview
Module
Rack
Slot
I address
Q address
Type
Article number
cifxrepns
PN-IO
1 Byte Input_1
2 Bytes Input_1
1 Byte Output_1
2 Byte Output_1
4 Bytes Input_1
8 Bytes Input_1
4 Bytes Output_1
8 Bytes Output_1
Hardware c...
Options
Catalog
Hardware catalog
Online tools
Tasks
Libraries


Profinet_AppNote ▸ Devices & networks
Topology view
Network view
Device view
Network
Connections
HMI connection
Network overview
Connections
I/O communication
VPN
PLC_1
CPU 1212C
cifxrepns
CIFX RE/PNS V3....
PN/IE_1
Device
Type
Address in subnet
Subnet
S7-1200 station_1
PLC_1
DI 8/DQ 6_1
AI 2_1
HSC_1
HSC_2
HSC_3
HSC_4
HSC_5
HSC_6
Pulse_1
Pulse_2
Pulse_3
Pulse_4
PROFINET interface_1
GSD device_1
192.168.250.55
192.168.250.1
Hardware c...
Options
Catalog
Hardware catalog
Online tools
Tasks
Libraries


Profinet_AppNote ▸ Devices & networks
Topology view
Network view
Device view
Topology overview
Topology comparison
PLC_1
CPU 1212C
cifxrepns
CIFX RE/PNS V3....
Device / port
Slot
Partner station
Partner device
Partner interface
S7-1200 station_1
PLC_1
PROFINET interface_1
Port_1
GSD device_1
cifxrepns
PN-IO
Port 1
Port 2
Hardware c...
Options
Catalog
Hardware catalog
Online tools
Tasks
Libraries


netDevice - Configuration CIFX_RE_PNS_V3.5.35_-_V3.x[CIFX RE/PNS V3.5.35 - V3.x]<cifxrepns>
IO Device:
CIFX RE/PNS V3.5.35 - V3.x
Vendor:
Hilscher Gesellschaft für Systemautomation mbH
Device ID:
0x0103
Vendor ID:
0x011E
Navigation Area
Settings
Driver
netX Driver
Device Assignment
Firmware Download
Configuration
General
Modules
Signal Configuration
Address Table
Device Settings
Description
Device Info
Module Info
GSDML Viewer
Modules
Slot
Sub Slot
Module
0 CIFX RE/PNS V3.5.35 - V3.x [1250.100]
1 1 Byte Input
2 2 Bytes Input
3 1 Byte Output
4 2 Byte Output
5 4 Bytes Input
6 8 Bytes Input
7 4 Bytes Output
8 8 Bytes Output
Add Module
Add Submodule
Remove
Use of slots: 9/256
State of data length: Input 27/1440 Octets, Output 27/1440 Octets, In-Output 54/2880 Octets
Submodule details
Dataset:
I/O data
Display mode:
Decimal
Direction
Consistence
Data type
Text ID
Length
OK
Cancel
Apply
Help
Disconnected
Data Set



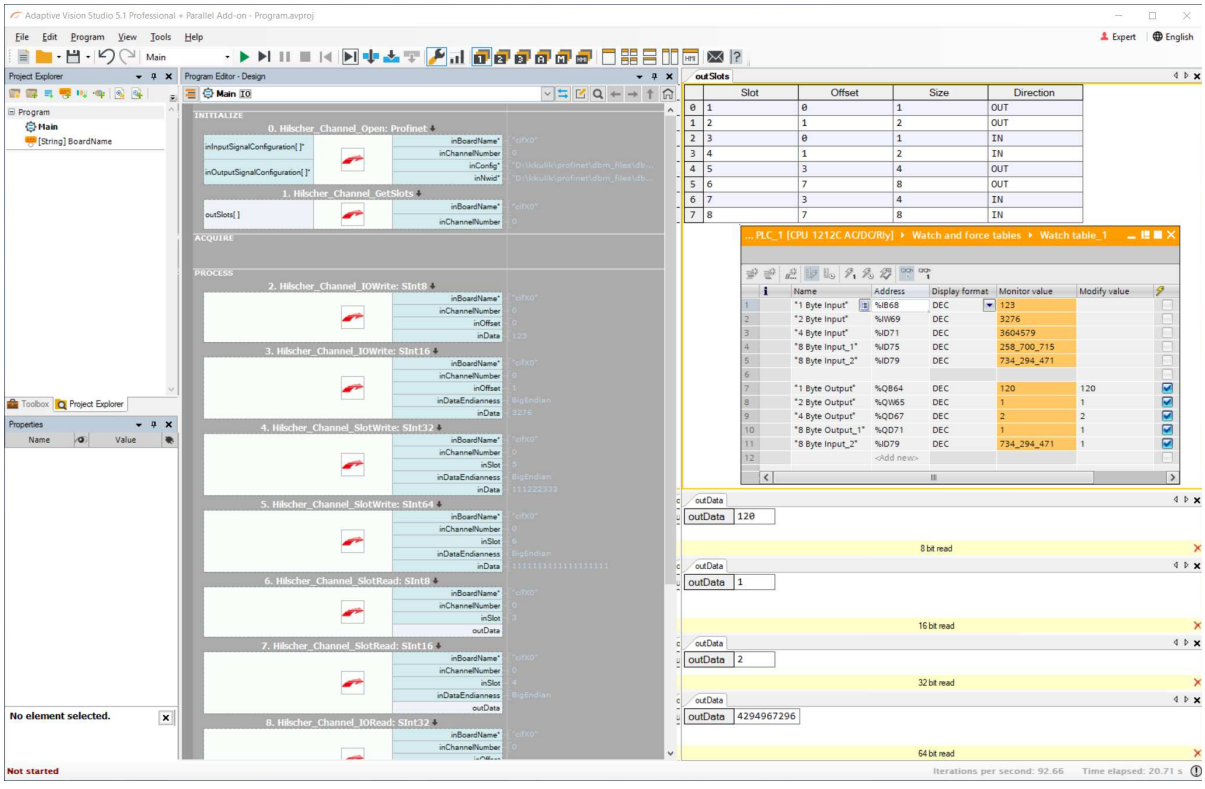
	Slot	Offset	Size	Direction
0	1	0	1	OUT
1	2	1	2	OUT
2	3	0	1	IN
3	4	1	2	IN
4	5	3	4	OUT
5	6	7	8	OUT
6	7	3	4	IN
7	8	7	8	IN

As you can see on the screenshots above the slots are common for each software. The only difference is that Siemens and Hilscher inputs corresponds to outputs from Aurora Vision Studio and vice-versa - Siemens output is Aurora Vision Studio input. It should be considered in the following manner: data outgoing from PLC (output) is incoming to Aurora Vision Studio (input) while data incoming to PLC (input) is outgoing from Aurora Vision Studio (output).

The summarization of data type and directions you will find in the table below:

TIA Portal		Sycon		Aurora Vision Studio	
Size	Type	Size	Type	Size	Type
1 Byte	Input	1 Byte	Input	SInt8	Output
2 Bytes	Input	2 Bytes	Input	SInt16	Output
4 Bytes	Input	4 Bytes	Input	SInt32	Output
8 Bytes	Input	8 Bytes	Input	SInt64	Output
1 Byte	Output	1 Byte	Output	SInt8	Input
2 Bytes	Output	2 Bytes	Output	SInt16	Input
4 Bytes	Output	4 Bytes	Output	SInt32	Input
8 Bytes	Output	8 Bytes	Output	SInt64	Input

For the slot configuration described in this paragraph the Aurora Vision Studio program will look as follows:



	Slot	Offset	Size	Direction
0	1	0	1	OUT
1	2	1	2	OUT
2	3	0	1	IN
3	4	1	2	IN
4	5	3	4	OUT
5	6	7	8	OUT
6	7	3	4	IN
7	8	7	8	IN

	Name	Address	Display format	Monitor value	Modify value
1	"1 Byte Input"	%IB68	DEC	123	
2	"2 Byte Input"	%IW69	DEC	3276	
3	"4 Byte Input"	%ID71	DEC	3604579	
4	"8 Byte Input_1"	%ID75	DEC	258_700_715	
5	"8 Byte Input_2"	%ID79	DEC	734_294_471	
6					
7	"1 Byte Output"	%QB64	DEC	120	120
8	"2 Byte Output"	%QW65	DEC	1	1
9	"4 Byte Output"	%QD67	DEC	2	2
10	"8 Byte Output_1"	%QD71	DEC	1	1
11	"8 Byte Input_2"	%ID79	DEC	734_294_471	1
12				<Add new>	

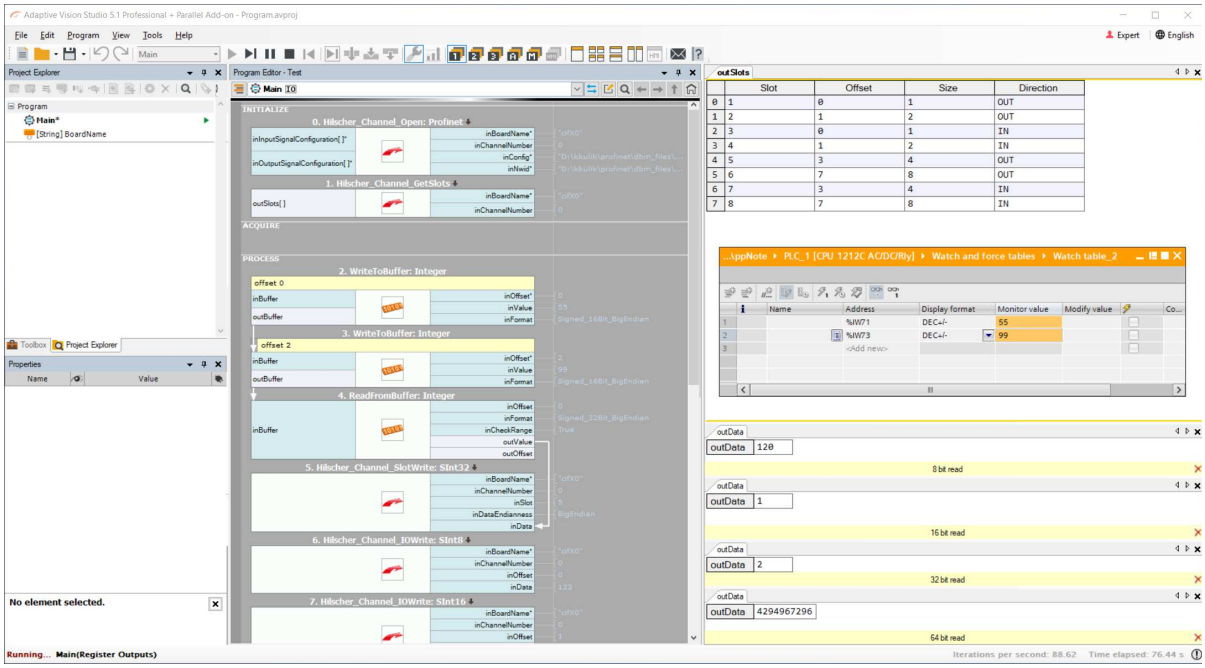
outData	
outData	120
outData	1
outData	2
outData	4294967296

You can use both Hilscher_Channel_IOWrite / Hilscher_Channel_IORead or Hilscher_Channel_SlotWrite / Hilscher_Channel_ SlotRead to send and receive data. The only difference is in addressing the accessible data:

- *IORead* and *IOWrite* filters use the **offset** value
- *SlotRead* and *SlotWrite* filters use the **slot** numbers

Both of the information you will find in output **outSlots** from Hilscher_Channel_GetSlots filter.

Sometimes it is necessary to combine few integer values into one slot of data. In order to accomplish this task you can use binary buffers in Aurora Vision Studio. Let's assume that someone would like to write two 2-bytes integers into one 4 bytes slot (*SInt32* in Aurora Vision Studio). In this situation the program should look as follows:

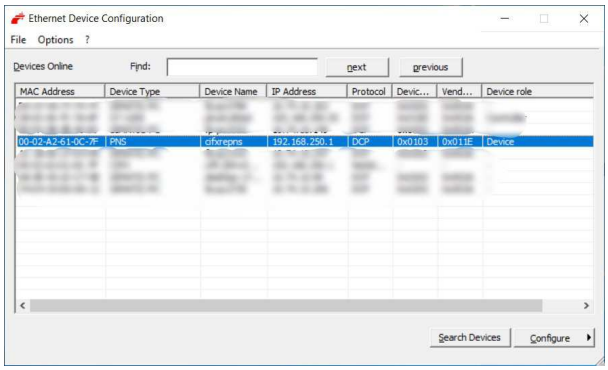


First we use two WriteIntegerToBuffer filters with specified format: **Signed_16Bit_BigEndian** connected in cascade to write two different values into one buffer (in this case 55 and 99). Also we specify the offset where the second value should be written. Later we extract the combined value as one Integer with filter ReadIntegerFromBuffer. For 8 bytes integers should be used ReadLongFromBuffer. Here we also specify the format of data (**Signed_32Bit_BigEndian**). Later we send the obtained number with Hilscher_Channel_SlotWrite_SInt32. Between *ReadFromBuffer* and *SlotWrite* the data is meaningless but later in TIA Portal when we access it as two separate 2-bytes values we get the send values.

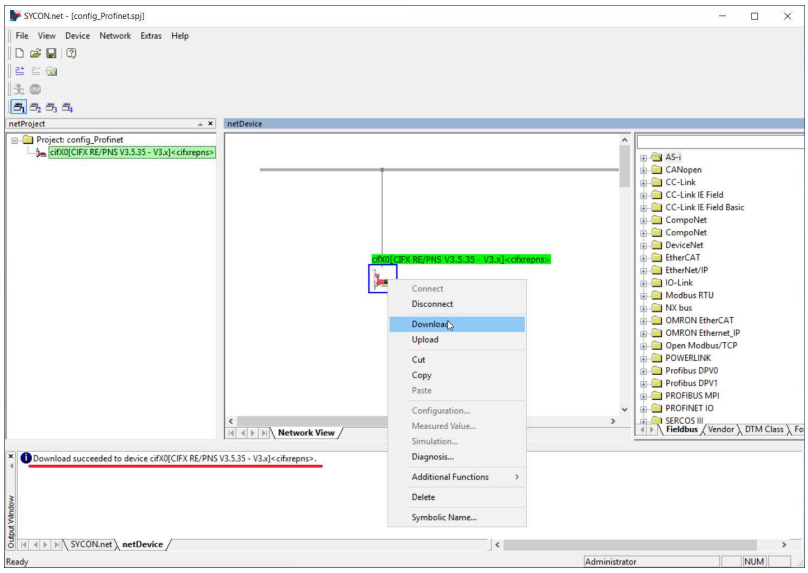
Troubleshooting

If it is still not possible to use AVS with Hilscher card first, make sure that you done correctly all previous steps and then use following advices.

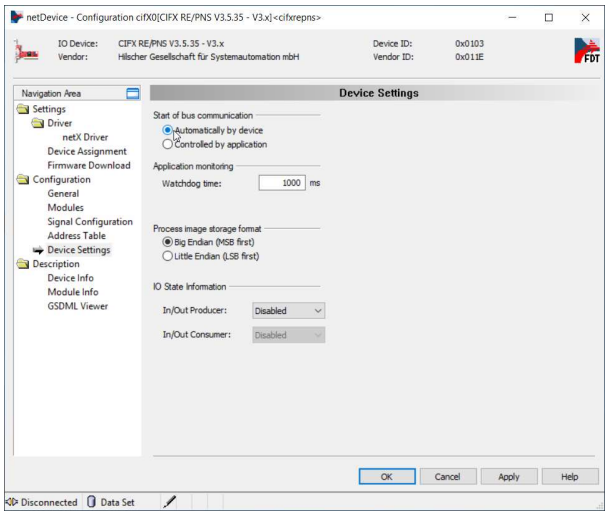
1. Use Ethernet Device Configuration software to set static IP address to Hilscher card.



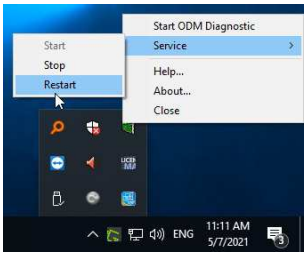
2. Make sure that current program setting is loaded to Hilscher card. If not please use SYCON.net application to connect and download settings to the devices as shown at the picture bellow.



3. If your master device has problem with connection with Hilscher card use following settings using Sycon.net application. In order to do this right-click on the card icon and select *Configuration...*



4. After changes it may be necessary to restart ODMV3 service.



5. If none of the above advice has helped, please restart your computer.

Interfacing Profinet gateway to Aurora Vision Studio

Purpose and equipment

This document explains how to configure Profinet PLC with Aurora Vision Studio using Modbus-TCP - PROFINET gateway.

Required equipment:

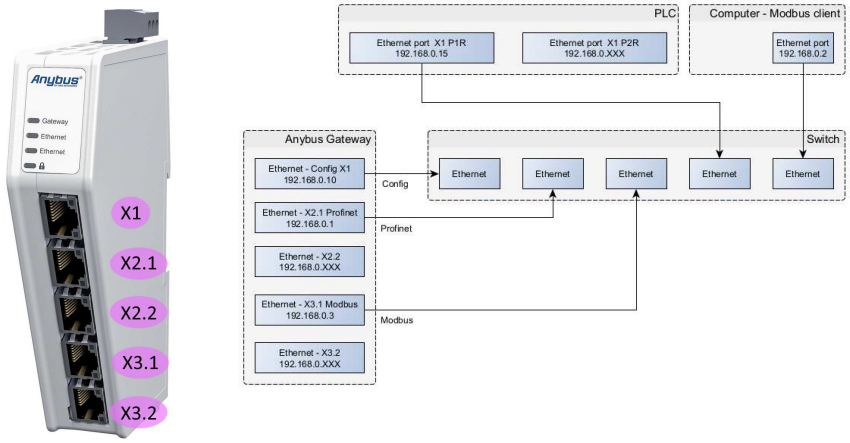
- Supported Modbus-TCP - PROFINET gateway (used Anybus ABC4090);
- Siemens S7-1200 PLC;
- Ethernet switch with at least 5 In/Outs might be useful - simplifies configuration;
- TIA Portal (version v15 used);
- Aurora Vision Studio 5.0 or later (version 5.2 used).

Application notes demonstrates configuration steps for specific gateway model, but it is not obligatory to use this one. If you have different gateway than the one described in Application Notes, please refer to manuals provided by its manufacturer.

Hardware connection

In application notes there is configured one network which uses Ethernet switch. Devices connected to this network have addresses 192.168.0.XXX, where XXX stands for number 1-254 unique for every device/port connected.

If you use Anybus ABC4090 gateway, it has 5 configurable In/Outs. Please find below description of these ports and connecting scheme:



- X1 - Configuration port. By default this address is set to 192.168.0.10 and this address is used in this application notes. If you want to change it, use HMS IPconfig tool in the way described in ABC4090 configuration manual. Connected to switch.
- X2.1 - configurable port from network 1. Configured as Profinet. Connected to switch.
- X2.2 - configurable port from network 1. Configured as Profinet.
- X3.1 - configurable port from network 2. Configured as Modbus-TCP. Connected to switch.
- X3.2 - configurable port from network 2. Configured as Modbus-TCP.

Before proceeding to the further point, make sure that all devices are powered up.

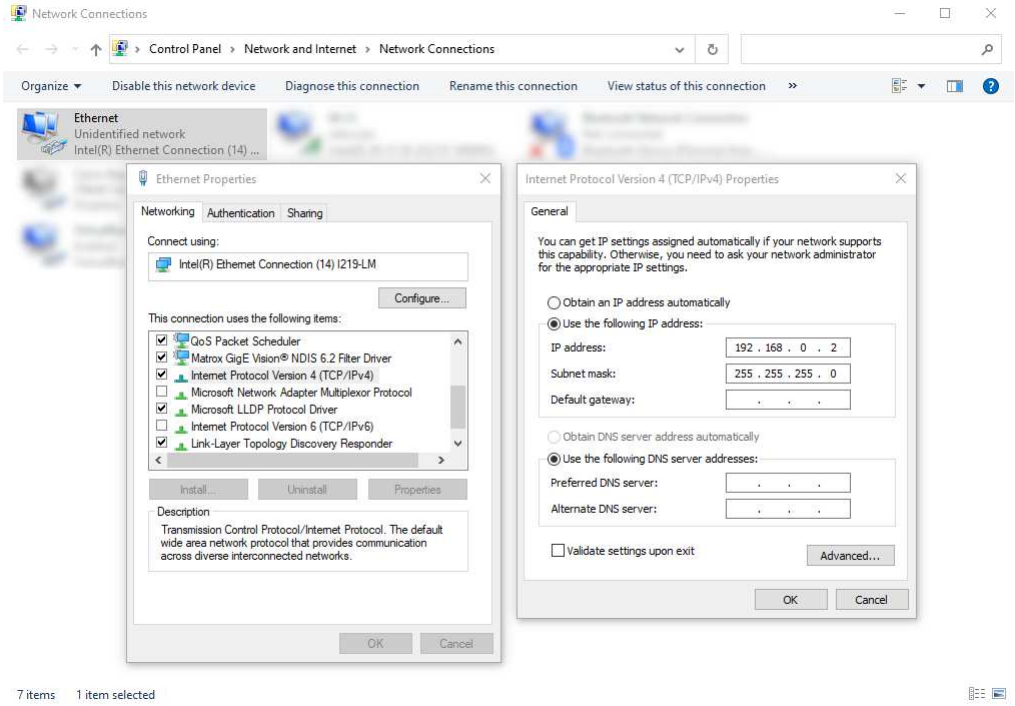
Configuring Anybus gateway in web configurator

1. By default Anybus ABC4090 is configured as EtherNet/IP - PROFINET gateway. To have Modbus-TCP - PROFINET gateway there is need to upload new firmware to the device.

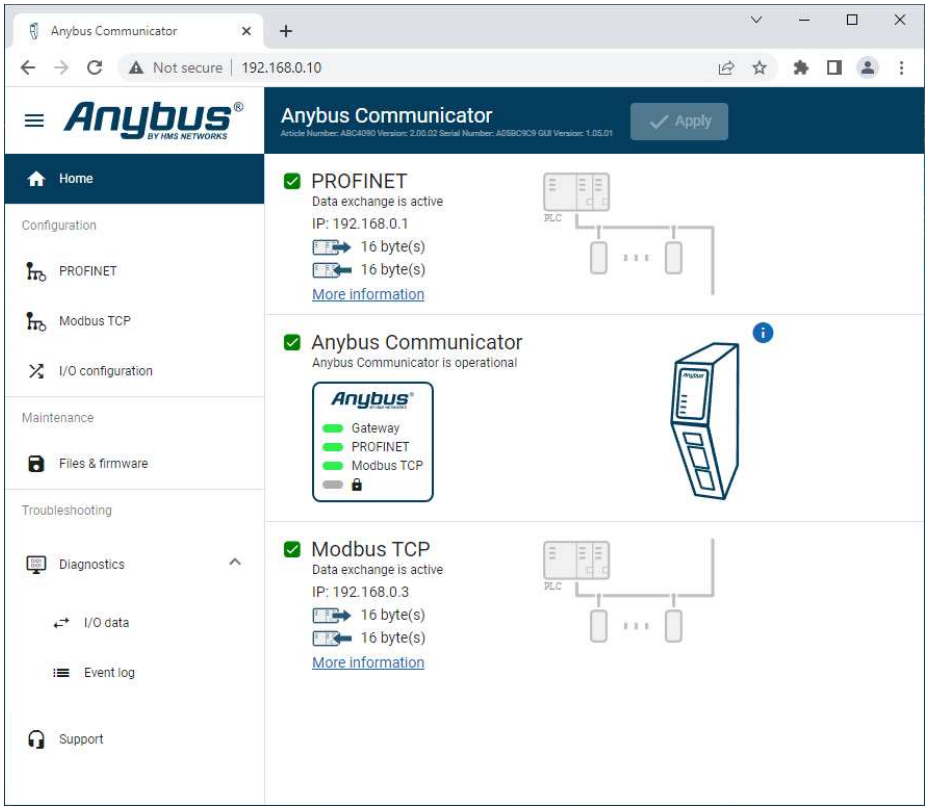
   Please download appropriate drivers before further reading from **firmware download** section:

File	Version	Size	Download
FIRMWARE			
Firmware Update EtherCAT - EtherNet/IP	1.00.01	3.88 MB	Download
Firmware Update EtherCAT - EtherNet/IP	1.01.01	3.90 MB	Download
Firmware Update EtherCAT - Modbus-TCP	1.00.01	3.84 MB	Download
Firmware Update EtherCAT - Modbus-TCP	1.01.01	3.86 MB	Download
Firmware Update EtherCAT - PROFINET	1.00.01	3.96 MB	Download
Firmware Update EtherCAT - PROFINET	1.01.01	4.04 MB	Download
Firmware Update EtherNet/IP - PROFINET	2.00.01	4.02 MB	Download
Firmware Update Modbus-TCP - PROFINET	2.00.02	3.99 MB	Download
Firmware Update Modbus-TCP - EtherNet/IP	2.01.01	3.90 MB	Download

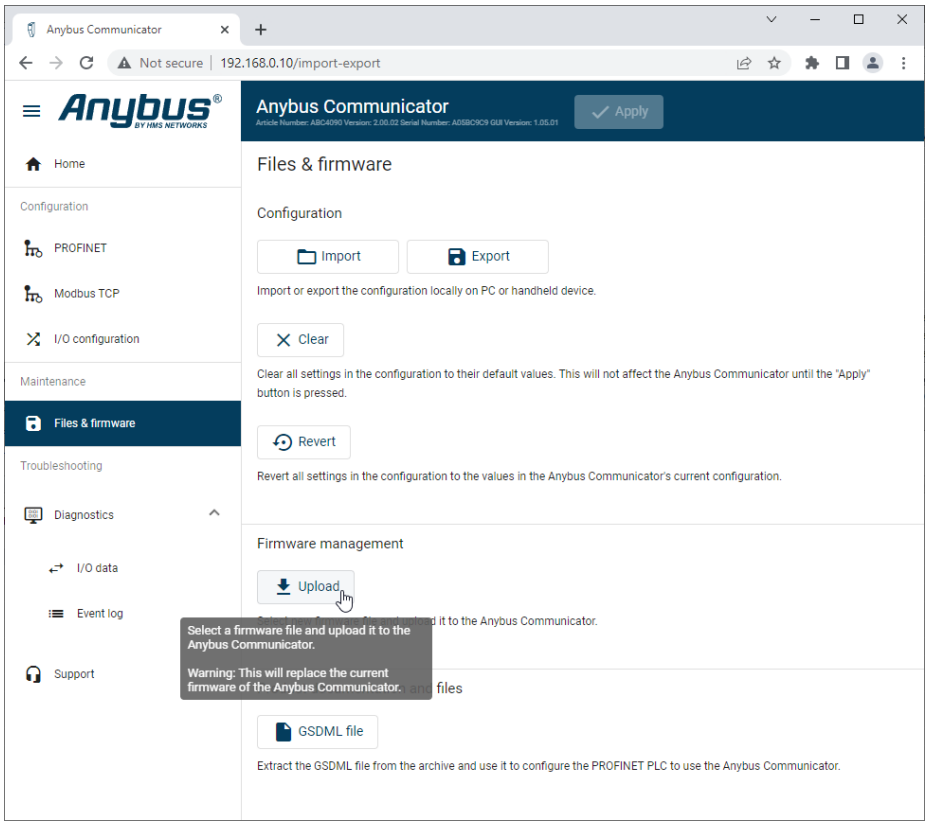
2. Default configuration IP address should be set to 192.168.0.10. PC must work in 192.168.0.0 network, so Ethernet-AnybusConfig IP were changed accordingly:



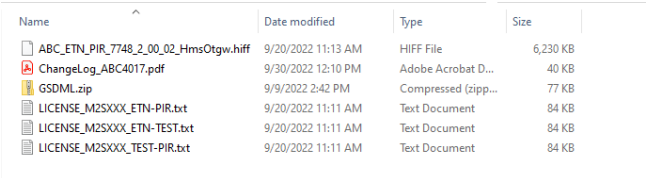
If you change network card's IP as shown above and you will connect it to the CONFIG port of the gateway you should be able to access configuration menu by typing 192.168.0.10 IP in the web browser:



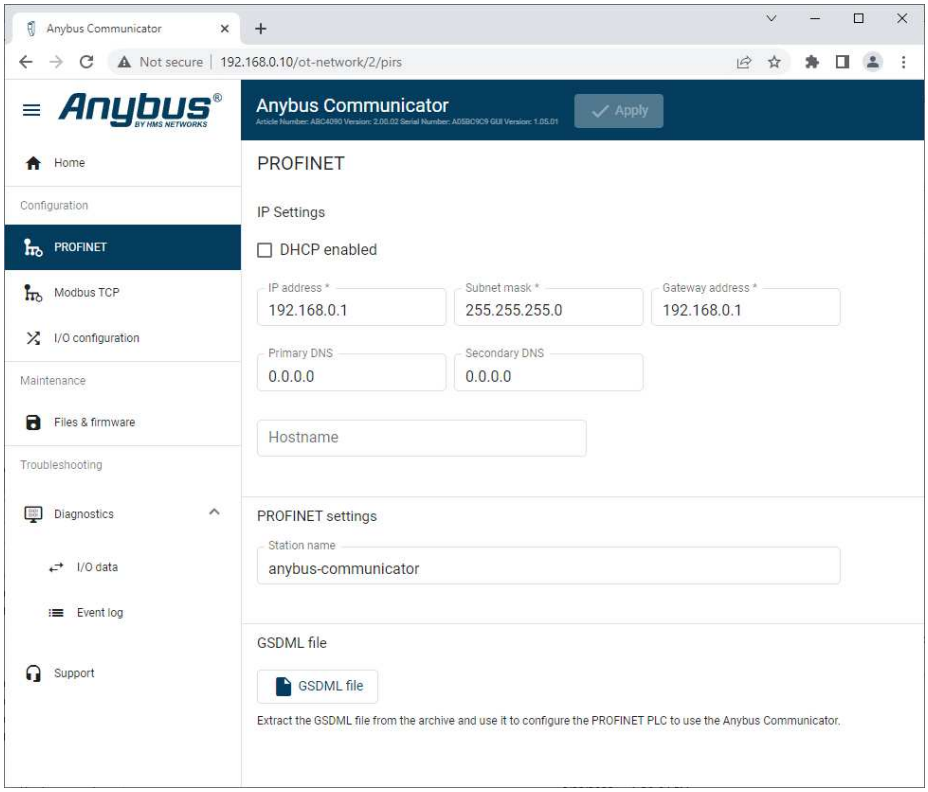
3. In order to change gateway's firmware navigate to **Maintenance -> Files & firmware** tab:



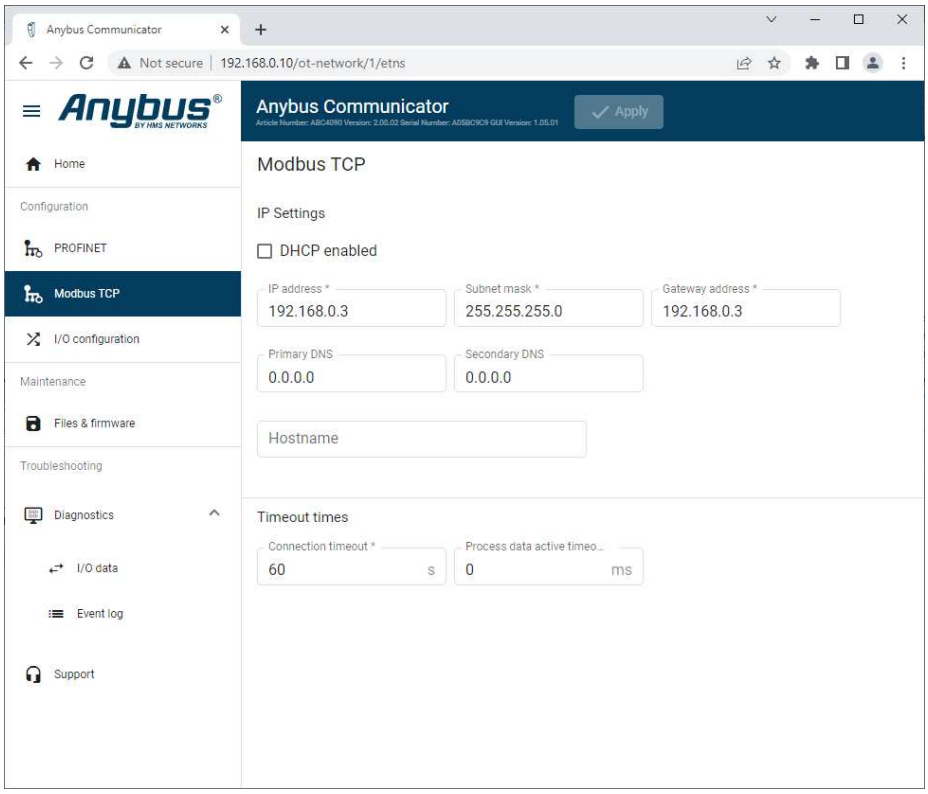
Click **Upload** and navigate to download firmware directory. It consists of GSDML file for later (hardware definition for Profinet PLC) and firmware file:



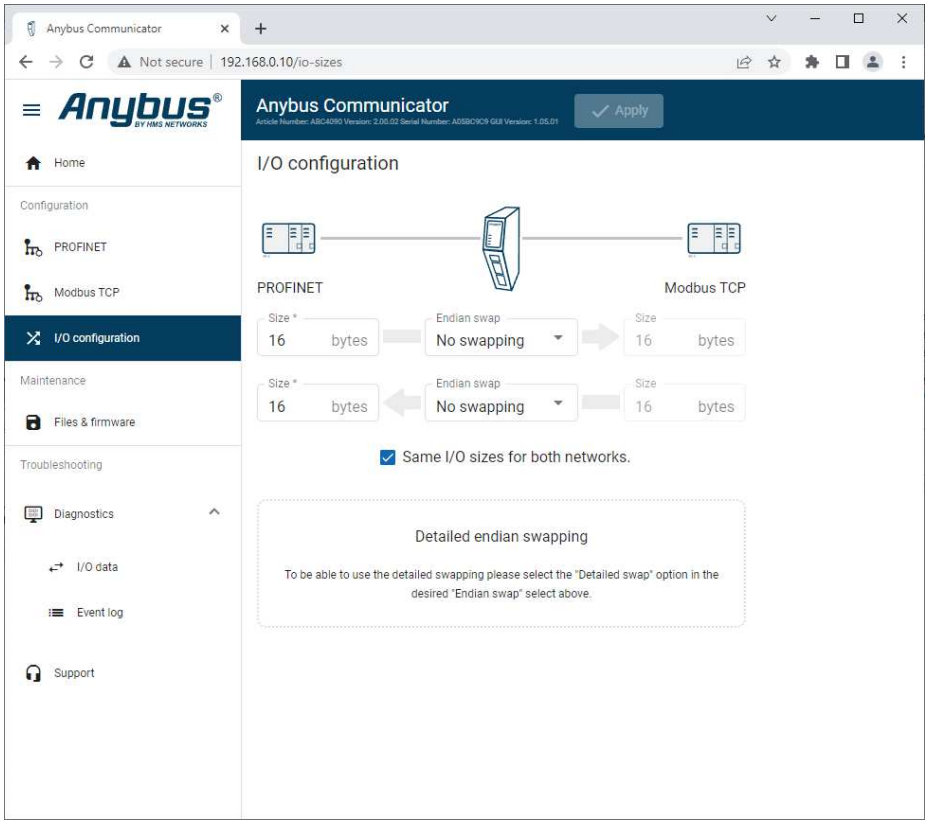
4. Devices linked should have an IP address which belongs to 192.168.0.0 network. Go to **Configuration -> PROFINET** tab, change settings like on the screen attached below and click **Apply**.



5. Devices linked should have an IP address which belongs to 192.168.0.0 network. Go to **Configuration -> Modbus TCP** tab, change settings like on the screen attached below and click **Apply**.



6. Navigate to **I/O configuration** tab, change settings like on the screen attached below and click **Apply**.



If your application requires more or less bytes to be exchanged feel free to change its values. **It is important to have the same configuration in the gateway and in the PLC.**

Configuring Anybus gateway in TIA Portal

This chapter describes how to configure PLC and gateway in TIA Portal software.

Reminding network connections, our PC with TIA Portal has to communicate with PLC controller in order to modify PLC program. Ethernet-Profinet connection is used and devices have an IP from 192.168.250.0 network:

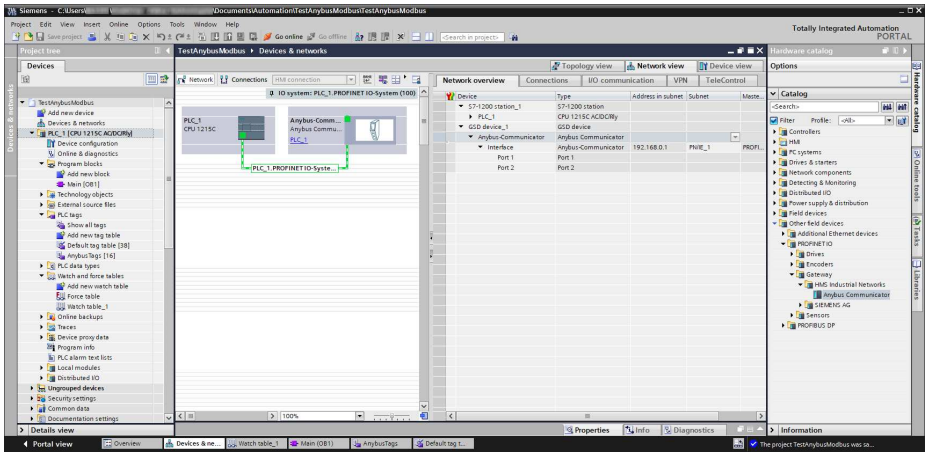
List of IP addresses in network:

- PC - 192.168.0.2;
- PLC - 192.168.0.15;
- Gateway - 192.168.0.1

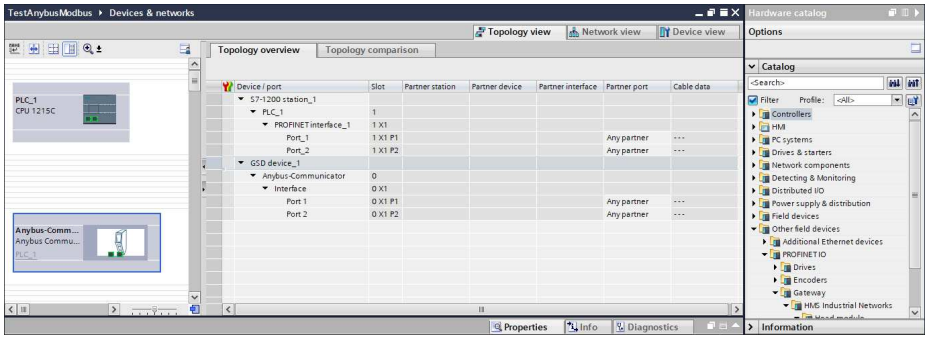
Please find attached below screens of hardware and software configuration from TIA Portal.

1. Create TIA Portal project with your PLC. Add GSD file from downloaded firmware folder in TIA Portal using tab menu on the top: "Options → Manage general station … → Select file → Install".

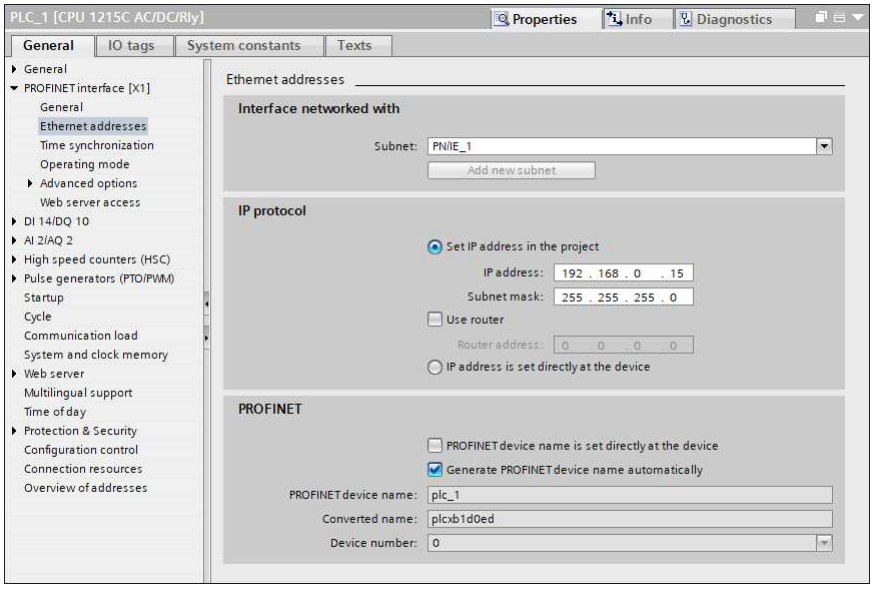
   Once you upload GSD, you can find appropriate device in **Other field devices -> PROFINET IO → Gateway → HMS Industrial Networks**. *Drag & drop* it onto **Network view** and combine it as on the screen attached below:



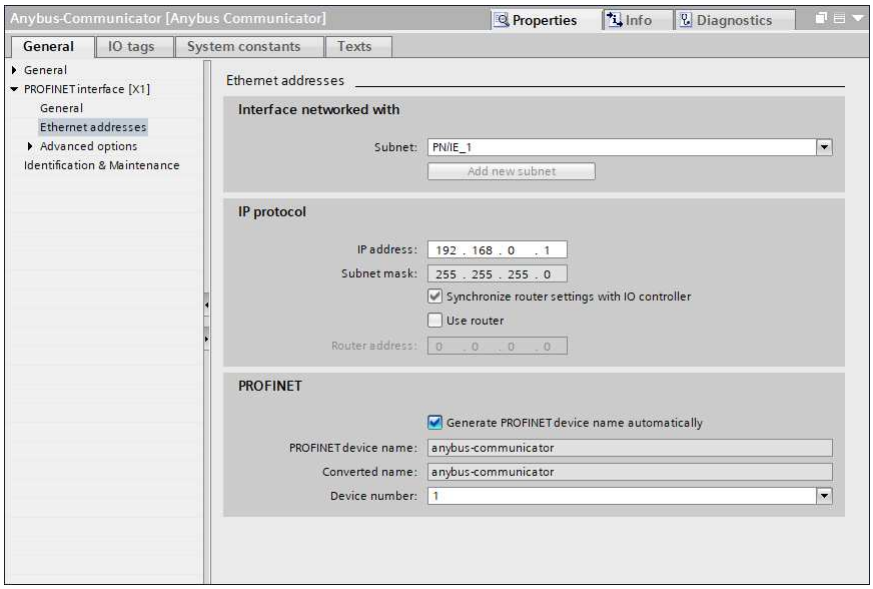
2. **Topology view** is set up this way:



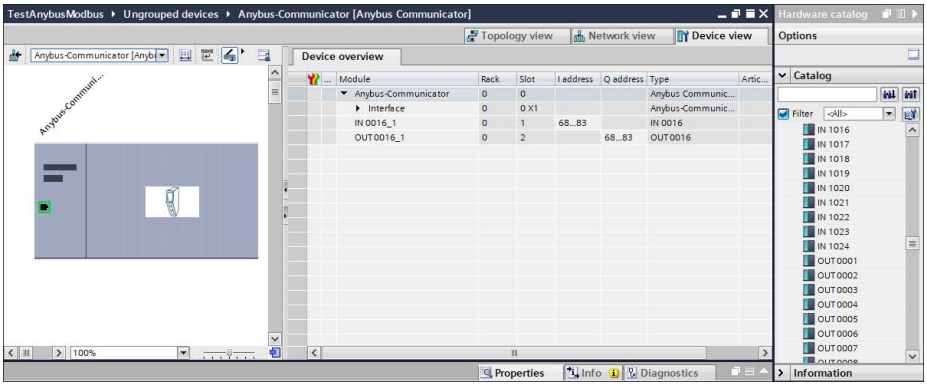
3. Change PLC's IP to **192.168.0.15**:



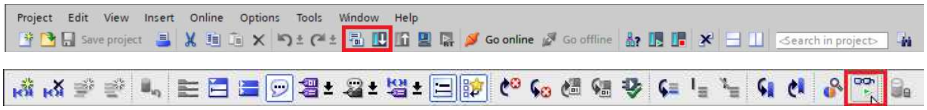
4. Change gateway's IP to **192.168.0.1**:



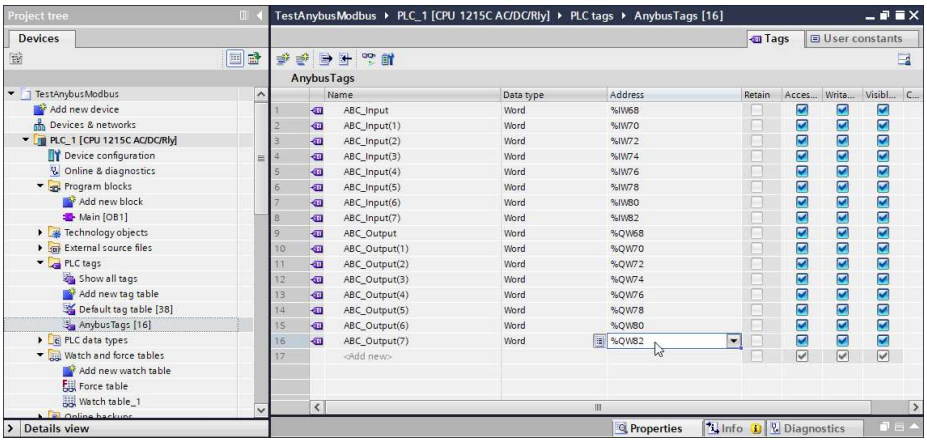
5. Add two modules to the project which match these configured in Anybus web configurator. *Drag & drop* **IN 0016** and **OUT 0016** module onto **Device overview**:



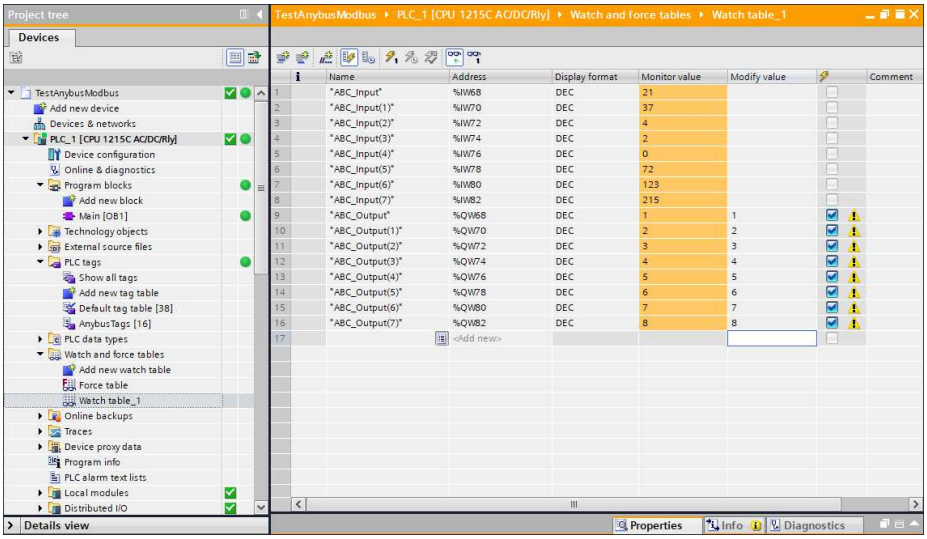
6. If all previous steps have been done correctly, your PLC program is ready. Compile the program and download it to the device. To see current states of variables, turn on Monitoring mode:



7. It is good manner to use gateways In/Outs by PLC tags - using raw addresses you can mistakenly modify/read wrong bytes. Create "AnybusTags" table in **PLC tags** folder:

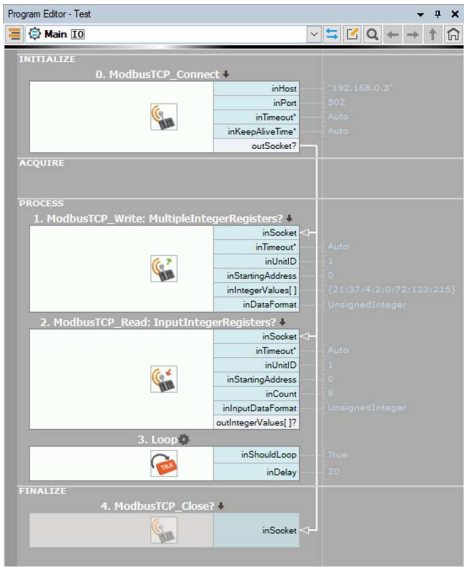


To monitor and easily modify tags please add *Watch table* in **Watch and force tables** folder.:

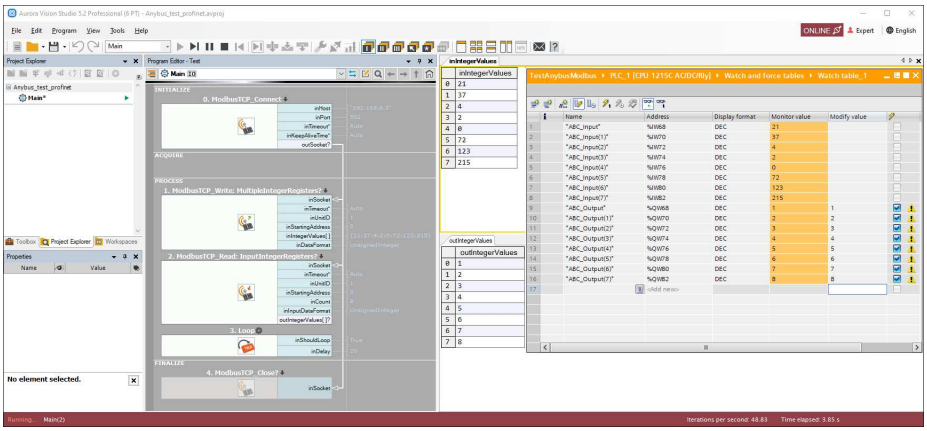


Configuring Aurora Vision Studio application

1. Aurora Vision Studio application should have ModbusTCP_Connect and I/O Read/Write filters as on the screen attached below:



2. Turn on online mode and monitoring mode in TIA Portal. You can create *watch tables* to easily modify data in TIA Portal environment:



Troubleshooting

In case of any problem with gateway configuration, please try manuals and tutorial videos from Anybus support page.

Interacting with GigEVision cameras

Introduction

The following guide provides information on how to utilize a GigE-compliant camera in Aurora Vision Studio.

While the guide focuses on the GigE protocol most of the content can also apply with slight changes to different camera protocols, especially GenICam.

Basic Image Acquisition

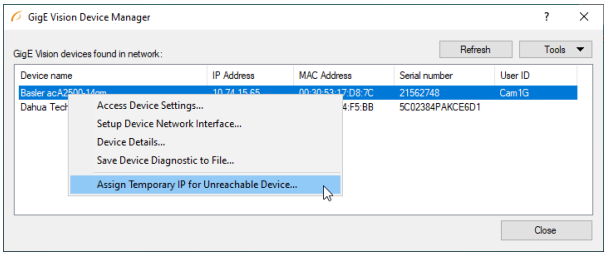
The simplest scenario, in which cameras are utilized in Aurora Vision Studio, is continuous synchronous acquisition.

Continuous means that the camera acquires new images by itself - it does not need to be triggered to do so.

Synchronous means that the program will wait for a new image until it is acquired - the iteration will not be finished without a new image.

The steps to perform such acquisition are as follows:

1. Connect the camera to the computer or network switch. If the camera is to be powered through Power-Over-Ethernet (PoE), make sure that it is connected to a device capable of supply power that way.
2. In a new program in Aurora Vision Studio add the filter **GigEVision_GrabImage**.
3. Open the editor of the inAddress input and set the camera address.
    a. If the camera is displayed with a warning sign, it means it is not in the computer's subnet. This can be solved by assigning a temporary IP address the camera.

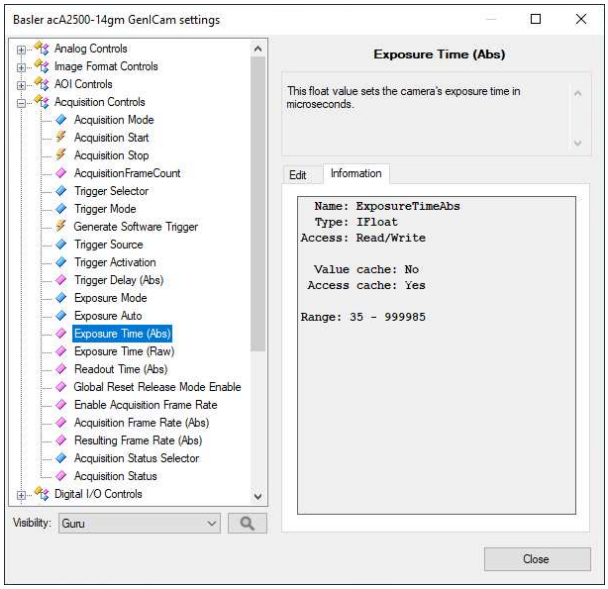


    b. When the camera is in the same subnet, it is possible to change its settings (for example changing its static IP).
    c. If the camera is connected but not detected you may open the window (from point a), manually type the camera MAC address and assign it another IP address. The menu can be opened from the Tools menu.
4. Now you can run the program. Previewing the outImage will show you the view from the camera.

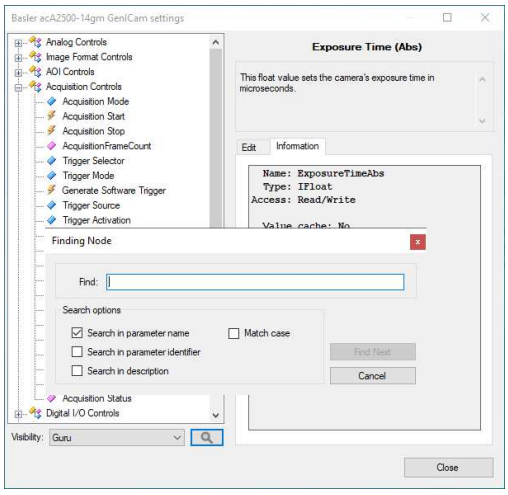
Changing camera parameters

You can change camera parameters programmatically using SetParameters filters. To demonstrate this, we will expand the previous program.

1. Add a GigEVision_SetRealParameter filter and set its **inAddress** to the same address as GigEVision_GrabImage.
2. Specify the parameter name. If you are not sure about the name you can select through the GigEVision device tree. To do that click on the "..." button near the **inParameterName**.
3. Here you can see all available camera parameters with a short description. Select the parameter with name Exposure Time (Abs) (or similar if not present). As you can see it controls the camera exposure time in microseconds and its type is IFloat.



   a. If you cannot find the parameter try using the search function (the magnifying glass icon)



   b. The type of the filter needs to match the type of the parameter. The exception is that IFloat is represented as Real in Aurora Vision Studio.

   c. Some more advanced parameters may not be visible unless Visibility is set to the correct level.

4. Make sure that the parameter ExposureAuto is set to Off. Otherwise, it will not be possible to manually change the exposure value.
5. Before closing the window note the minimum and maximum values of exposure time. After selecting the parameter, set the **inValue** to the lowest acceptable value.
6. Run the program. While running, steadily increase the **inValue** input. You will notice that the camera image gets brighter and brighter.
   a. Try not to go outside the acceptable range. It will result in an error if the input **inVerify** is set to True.
7. It is also possible to check a parameter's value by using GigEVision_GetParameter filters. Try adding one in the Float variant and specifying the same name as in the previous filter.
   a. You many notice that not always the read value is equal to **inValue** of SetParameter. It is because the camera modifies it.

With GigEVision_GetParameter filters it is not only possible to check editable parameters but also to check read-only ones. For example, if the camera was set to automatically adjust exposure, the current exposure time might be useful to know. Another example would be to read parameters holding device specific information, like maximum image size.AvsFilter_cvReinSrc

For some quick testing you may also want to set parameters directly in the GigEVision tree.

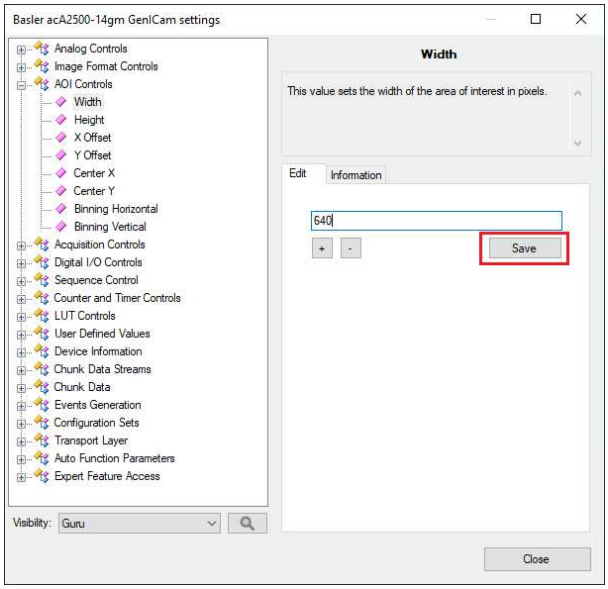
It is important to note that while most parameters are stored in volatile memory and as such will reset after after unplugging the camera from power. There are some parameters that are stored in non-volatile memory, such as Device User ID, but they usually do not affect the acquisition directly.

Because of this it is recommended to design program in such a way that every parameter with value different than default is set programmatically.

Starting and stopping acquisition

It is important to note that not all parameters can be changed while the acquisition is running. Some require stopping the acquisition and restarting it.

1. Run the program and open the device tree (Tools -> Manage GigE Vision Devices...). Find the Width parameter. You will notice that the save button is grayed out due to running acquisition.



2. Add another **GigEVision_SetIntegerParameter** filter and select the Width parameter. You may notice that it is greyed out.
    a. While selecting the parameter check the information tab. There may be additional information about possible value, e.g. increment. Some parameters can only take value that are multiples of certain numbers, like 2 or 4.
3. Enter 240 as inValue and run the program. You will encounter an error saying that the parameter is not writeable.
4. Now add a **GigEVision_StopAcquisition** before the previous Set filter and rerun the program.
5. The parameter is now set without errors and the output image is smaller.

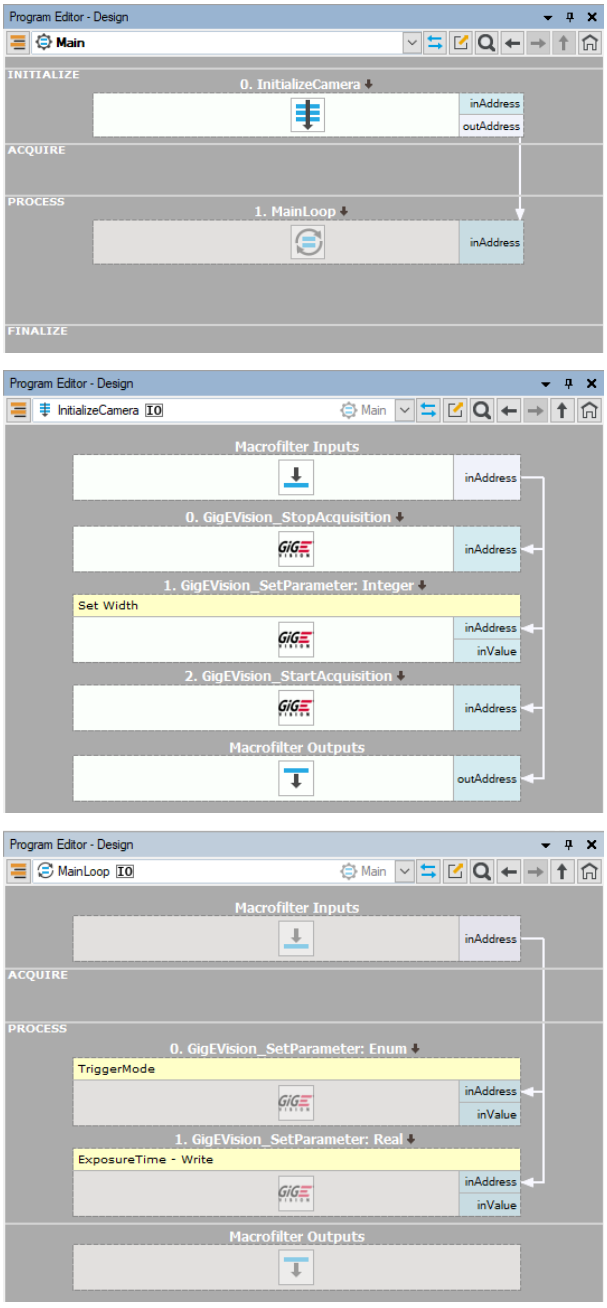
As you can see the Width parameter was not writable while the acquisition was running. Stopping the acquisition before changing the parameter made it writable.

You may also notice that we have a filter to explicitly stop the acquisition but there is not filter to explicitly start it. **GigEVision_GrabImage** will attempt use an ongoing acquisition but if none is present it will start a new one.

While it is acceptable for simple programs, it should be avoided when the status of the acquisition is program-controlled.

Now we will make a proper program out of this.

1. Start with removing the filters added in this chapter.
2. Now select the rest and extract a task macrofilter (called MainLoop) from them. This will be the main acquisition loop.
3. Create a new step macrofilter called InitializeCamera and add it before MainLoop in INITIALIZE section.
4. In InitializeCamera add the three following filters **GigEVision_StopAcquisition**, **GigEVision_SetIntegerParameter** (select Width parameter) and **GigEVision_StartAcquisition**.
    a. Set the width to the maximum possible value.
5. Drag **inAddress** of any of the filter to the top bar to create a new input. Then add an output of the same type and connect it to the input.
6. Go up one level and drag the **outAddress** output to the MainLoop filter creating a new input.
7. Inside that filter connect the newly created input to all the GigE filters
    a. Now every GigE filter in the whole program shares the same camera address.
    b. The final program should look like this:



Now the program is divided into two parts. The first part is executed once, after that the program enters MainLoop which run continuously until the program is stopped.

The parameters that can be changed during acquisition are set in the InitializeCamera filter where the acquisition is guaranteed to be stopped.

All the other parameters are set in MainLoop.

In real applications not all parameters need to be changed during runtime. Even then they probably will only be changed from time to time, not in every iteration.

For instance, specifying if the camera will run continuously in trigger mode will likely be done only once (even though it can be set during acquisition). Exposure time might be changed multiple times.

It is good practice to have all the parameters that are set only once in one macrofilter (like InitializeCamera) regardless of if they required acquisition to be stopped.

The parameters changed from time to time should be set in a variant macrofilter.

The reason of that is time-saving. Setting the parameter to the same value does not take much time (Aurora Vision Studio caches previous values and avoids resending the same ones), but if we have a lot of parameters it adds up. It is also more intuitive if one-off parameters are in a dedicated filter.

It is important to note how the cameras work in Aurora Vision Studio. When the acquisition is started Aurora Vision Studio creates a background thread which buffers incoming frames in memory. This thread persists until the acquisition is stopped or the application exits the task that started the acquisition.

For example, if InitializeCamera was a task and not a step macrofilter the acquisition would stop when exiting it and it would have to be restarted in MainLoop.

Also, instead of passing the camera address as a parameter it is possible to put it into a global parameter.

Changing trigger mode - basic

While the default mode of most cameras is continuous acquisition very often it is more desirable to only acquire new images after a certain signal, i.e. trigger.

Not only you have more control over the acquisition time of each frame, less frequent acquisition requires smaller bandwidth.

To demonstrate triggered acquisition, we will use the current program.

Run the program, ensure that you have displayed the camera image in the preview.

1. Now open the GigeVision device tree and find the parameter Trigger Mode. Set it to On. Also make sure that Trigger Selector is set to Frame Start.
2. You will notice that no new images are grabbed by the camera.
3. Find the Trigger Source parameter and set it to Software.
4. Find the Generate Software Trigger command (commands feature the lightning bolt icon) and execute it.
5. A new image will be grabbed by the camera.

You may also want to have an external trigger (for example, an output from a PLC). Then you must select the appropriate Trigger Source, e.g. Line1.

You may also have to ensure that the signal matches what the camera expects. The Trigger Activation parameter specifies what value will the camera look for.

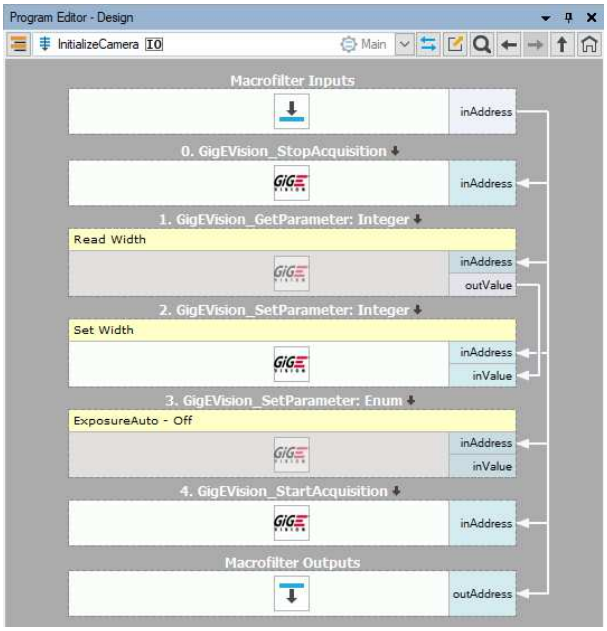
Changing trigger mode - advanced

Very often we want the trigger properties to be controlled directly by the program. For example, the camera may be triggered in one part of the program but also acquire images continuously in another part. Or maybe the camera is to be triggered by different sources depending on the program state.

Trigger parameters can be changed during acquisition. Therefore, it is possible to change them on-the-fly.

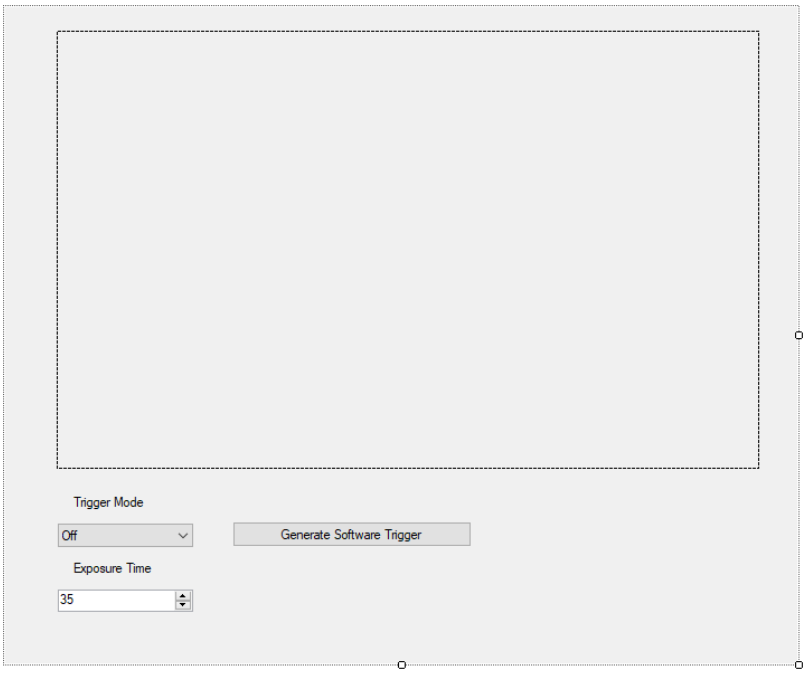
Now we will prepare a more advanced program which that initializes camera and allows the user to change how the camera is triggered and the exposure time from the HMI while the program is running.

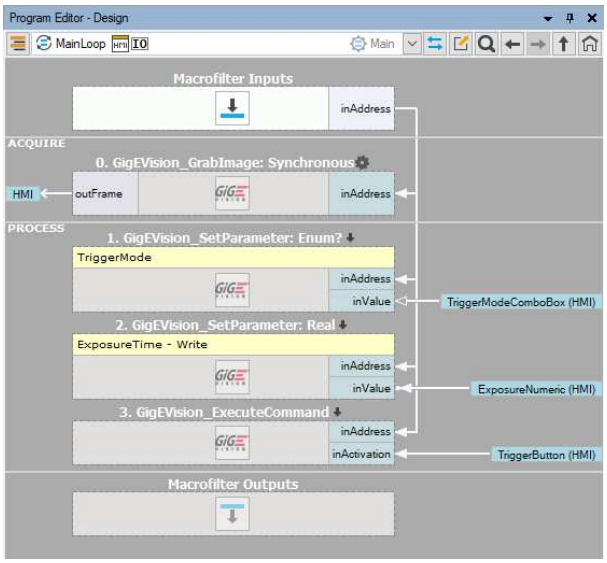
1. To ensure that the camera will be set to use Software as the source of trigger, add a **GigEVision_SetEnumParameter** filter in the InitializeCamera (before **GigEVision_StopAcquisition**). Through the Device Manager choose the TriggerSource parameter and set **inValue** to Software.
2. Now repeat the previous step, changing the parameter to ExposureAuto and set its value to Off.
3. Since different cameras may have different acceptable maximum width you may want to add a **GigEVision_GetIntegerParameter** filter that would read the parameter WidthMax in device information. Connect its output to the filter that changes the Width.



4. Now we will modify the contents of MainLoop worker task. We want to be able to change trigger mode and to execute a software trigger.
    a. Add an instance of **GigEVision_SetEnumParameter**. Connect it to the camera address and select Trigger Mode as its parameter
    b. Add a **GigEVision_ExecuteCommand** filter, connect them to the camera address. Through Device Manager select parameter TriggerSoftware.
    c. Move all those filters to PROCESS section and add **GigEVision_GrabImage** to ACQUIRE section.
5. Let's design an HMI. In this case it will feature:
    a. View2DBox - which will display the camera image;
    b. ComboBox - for selecting trigger mode;
    c. NumericUpDown - to control exposure time;
    d. ImpulseButton - to generate software trigger;
    e. Labels - to label other controls.

6. Now we will configure the controls:
    a. View2DBox - change its InitialSizeMode to FitToWindow and connect to GrabImage's outImage;
    b. ComboBox (for trigger mode)
        I. Connect **outText** to **inValue** of Set Parameter for TriggerMode;
        II. Expand List in Data category in Properties and add the following item: On, Off;
        III. Set Selection to 0
    c. ImpulseButton - connect its **outValue** to ExecuteCommand's **inValue**;
    d. NumericUpDown:
        I. Connect **outValue** to **inValue** of SetParameter for ExposureTime;
        II. Set Minimum and Maximum to the values specified in the Device Tree for that parameter (for the camera used while writing this note the values were 35 and 999985.

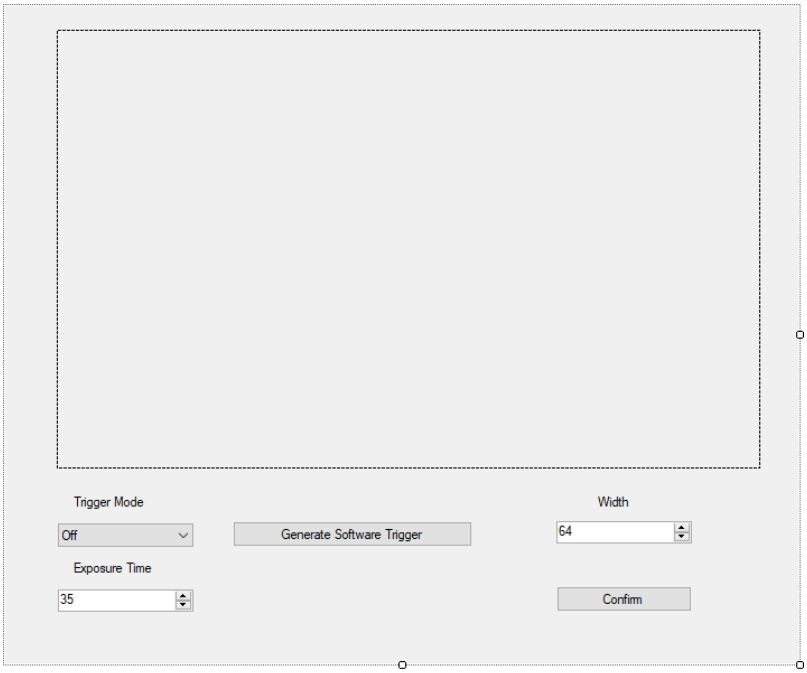




7. The program should display the camera image on the HMI. You should be able to change the exposure time. However, when you change the ComboBox's value to On, the program will freeze. Clicking the trigger button will not do anything.
    a. This is caused by the fact that the GrabImage in its Synchronous variant. The cannot get past GrabImage and execute a trigger, so it waits for an image indefinitely.
    b. To solve that we can change GigEVision_GrabImage variant to GigEVision_GrabImage_WithTimeout. Now the program will only attempt grabbing for a specified amount of time. Let's set **inTimeout** to 100ms and rerun the program.
8. Now the program does not freeze when setting triggered mode on. If no image is grabbed in 100ms GrabImage returns Nil and the program proceeds to the next iteration, where the camera may be triggered.
    a. Generally if the camera is meant to acquire images only from time to time it is a good idea to use WithTimeout variant of GrabImage. It prevents program hang-ups in case of some camera problems and allows to inform the user that the camera may not be working correctly. However, the rest of the program must be designed in a way that handles Nil images properly.

We can now expand the program to enable the user to change other parameters, including those which require acquisition to be off.

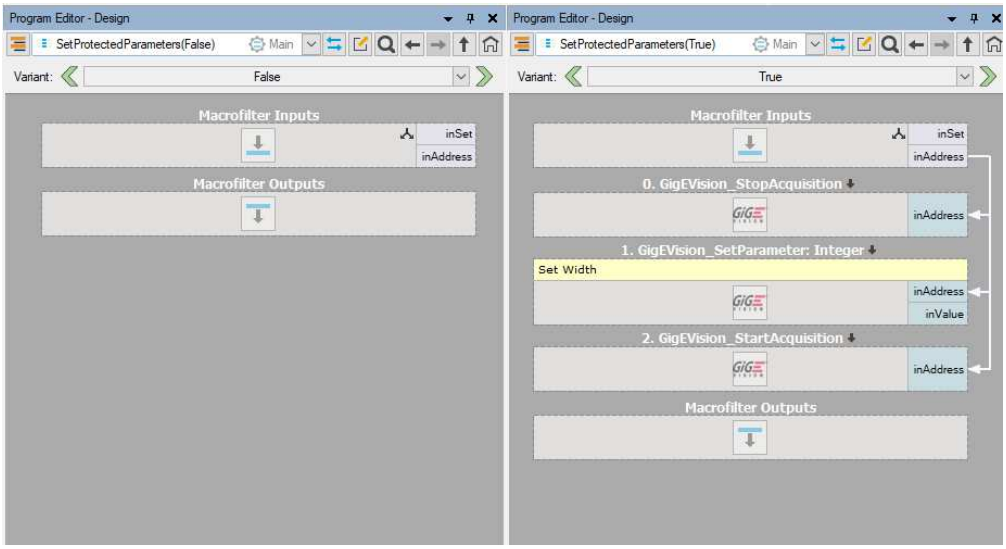
1. Create an empty variant macrofilter inside MainLoop and choose Bool as the fork type. Drag the camera address to the filter to create an input.
2. Enter the filter and choose the variant True. Add a StopAcquisition filter, followed by a SetParameter filter in variant Integer and select Width parameter. After them add a StartAcquisition filter. Connect all filters to the camera address.
3. In the HMI add two new controls:
    a. NumericUpDown - to control width's value; set its Minimum and Maximum to the respective limits of the Width parameter in the camera (for the camera used while writing this note the values were 35 and 999985); set Increment to the respective value as well.
    b. ImpulseButton - to confirm new value.

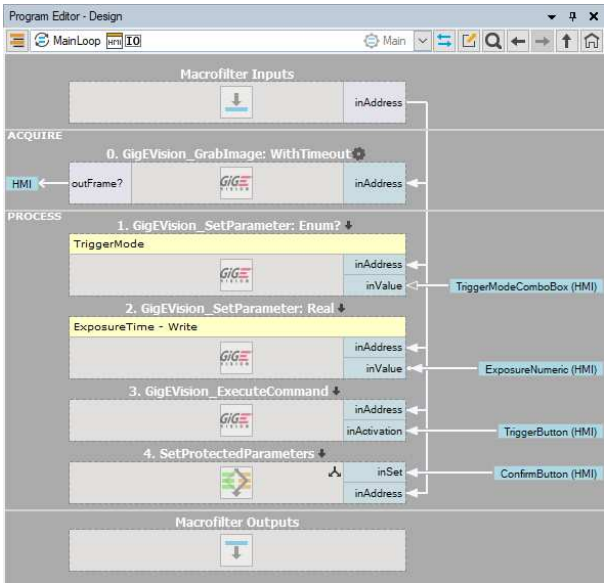


4. Connect NumericUpDown's outValue to the new SetParameter's inValue.
5. Go back to MainLoop and connect the new ImpulseButton's outValue to the condition port of the variant macrofilter.

Now when the user clicks the confirm button the program will enter the variant macro, stop the acquisition, change the width and restart the acquisition. If the user changes the value by using NumericUpDown buttons the value will be properly aligned.

However, if the user enters a value by typing it may be it will not be guaranteed to work with the camera. It is possible to design a macrofilter which will make sure the value accepted by the camera, but it is not the topic of this application note.





Additional remarks

Camera filters cannot be executed in array mode (for example when an array of camera addresses is connected).

Since a camera is an external hardware, they may be causing errors (like connection errors) unrelated to the Aurora Vision application. To properly handle those errors the application should feature error handling.

Using TCP/IP Communication

Purpose and equipment

This document describes how to a make practical example of TCP/IP communication between Aurora Vision Studio and a PLC device. For this case Aurora Vision Studio 4.11 and Simatic S7- 1200 CPU 1212C AC/DC/RLY with TIA Portal V15 have been used. For more theory you can check out TCP/IP Communication article.

Required equipment:

- SIMATIC S7-1200 controller or other from SIMATIC S7 family
- TIA Portal V15
- Aurora Vision Studio 4.11 Professional or later

Overview and first steps

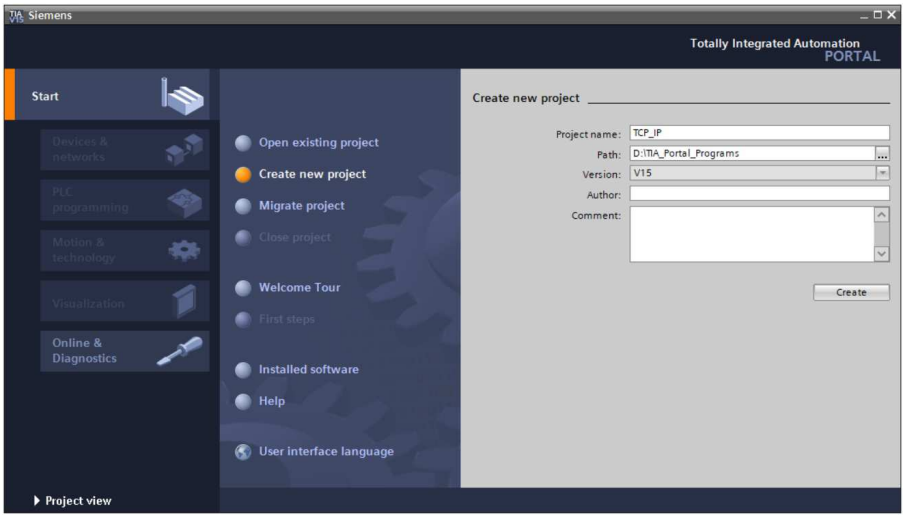
In this example, you will create an application which sends a number from Aurora Vision Studio to a PLC and receives the square of this value calculated by the PLC. The application will be configured as a master and the PLC as a slave.

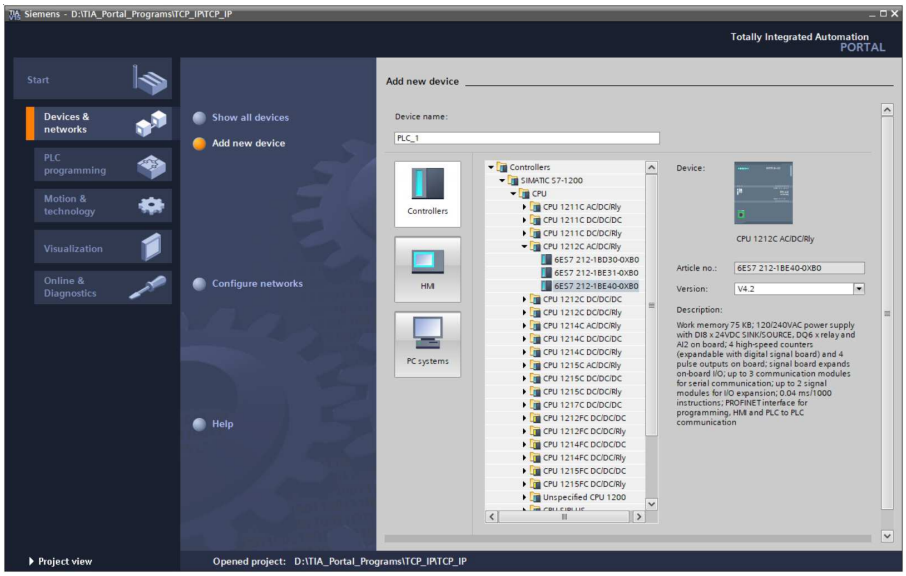
First, you need to make a hardware connection between the PLC and the PC. Network configuration will be described in the next chapter.

Network configuration

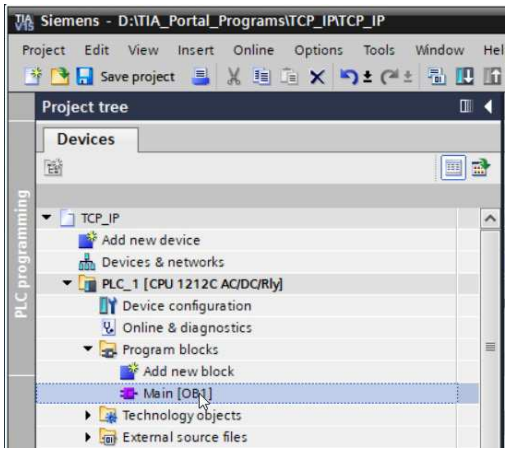
Main network configuration

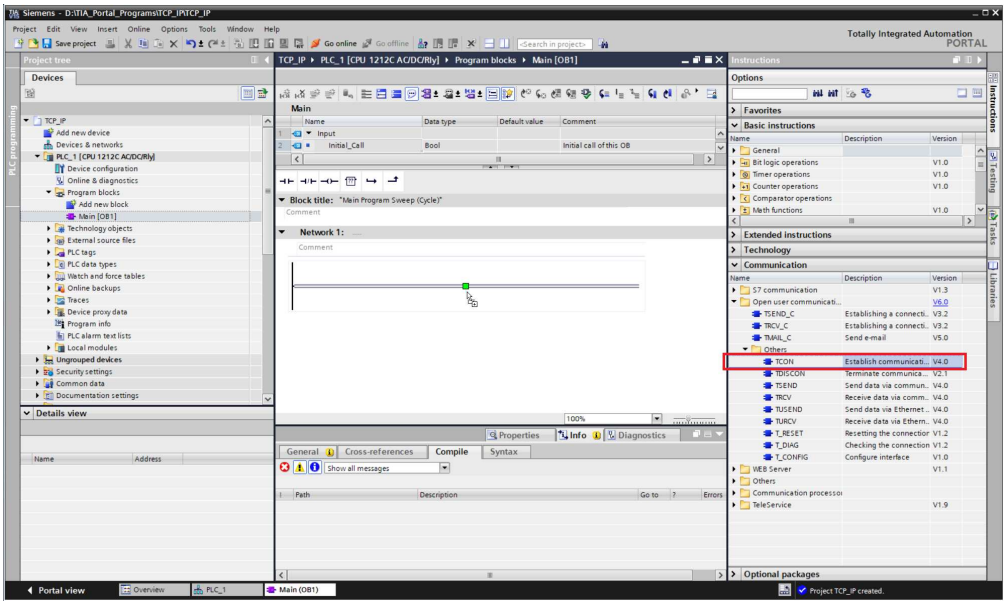
Create a new project in the TIA Portal environment and add your device by double-clicking on *Add new device*. Select your controller model, its CPU etc. in the list of controllers available in a dialog box. Click on *OK* button if you are ready.



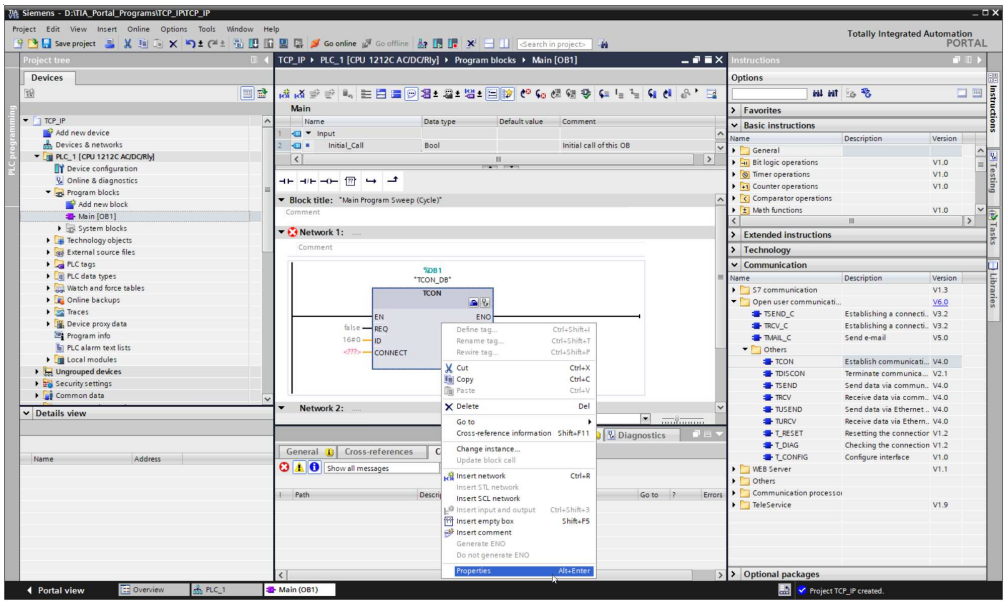


Expand a *Program blocks* list and double-click on the *Main* icon. Now you should see a network view. Find the Communication tab on the right side and double-click on the TCON icon available in *Communication -> Open user communication -> Others*.

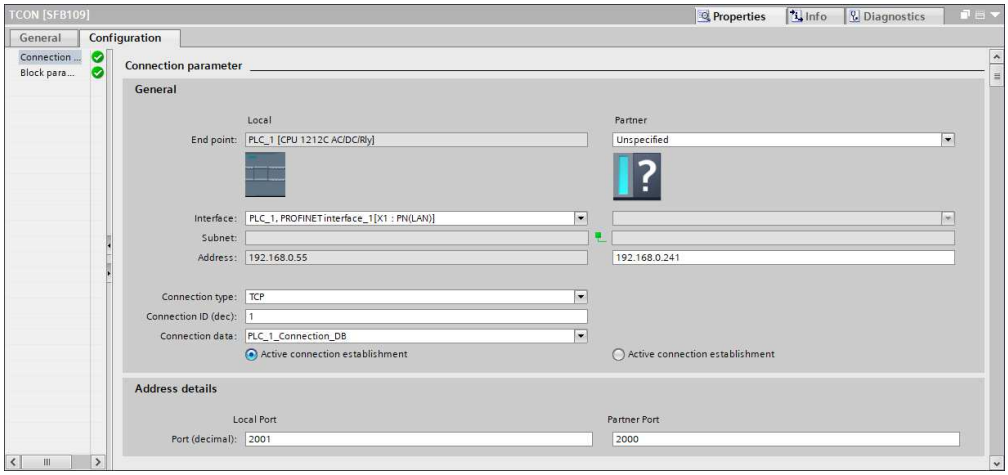




The TCON instruction is used for establishing the TCP communication. It is now visible in the *Network View.* Right-click on it and select *Properties*.



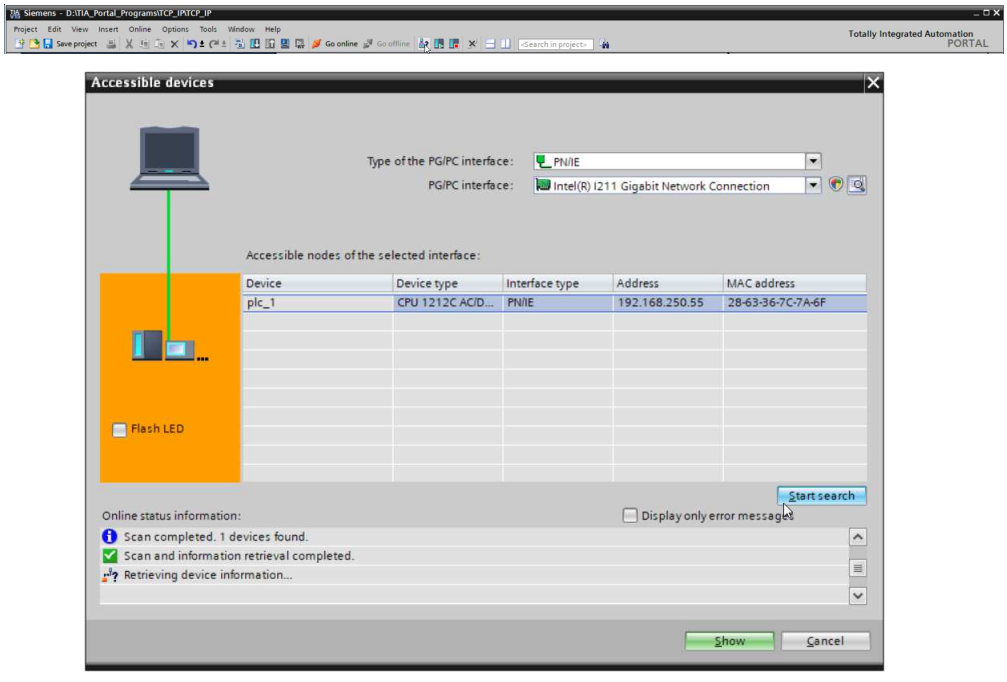
In the *Configuration* tab, you need to set the IP address of your PC. To check the IP address, you can use the Command Prompt and the **ipconfig** command. Other parameters should be set like in the next figure. If you use these settings, the IP of the PLC device should be set automatically. If you cannot establish the connection, please follow steps described in next chapter about troubleshooting.



Troubleshooting IP addressing

If you have not been able to properly set the IP address of the PC, as described in the previous chapter, you should set a static IP following steps described below or otherwise, feel free to skip this step.

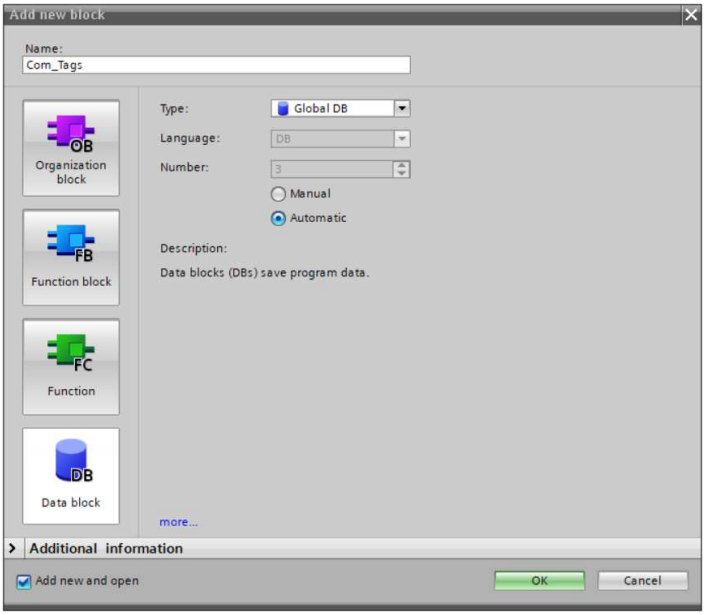
Choose *Accessible devices* from a toolbar to open the configuration window. Select *PN/IE* type of the *PG/PC* interface and select the Ethernet card connected to the PLC. Click the *Start search* button, choose your PLC from the list of accessible nodes and click on the *Show* button. You should get a new static IP which you can use in steps from the previous chapter.



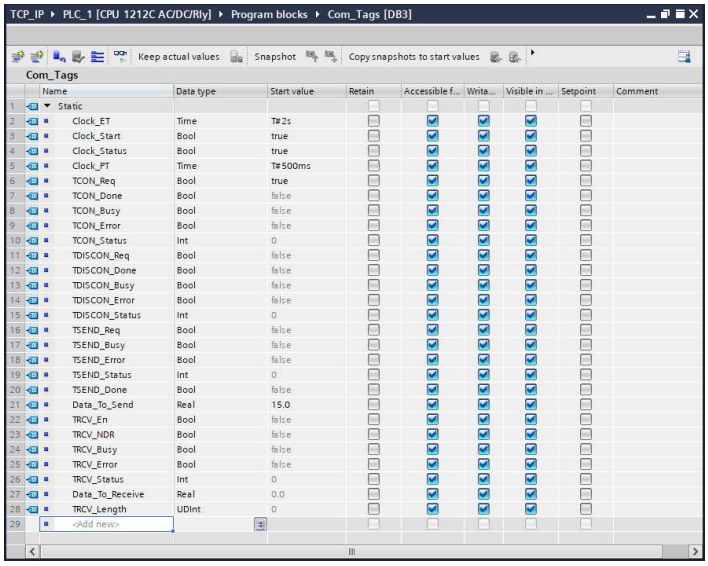
Creating a PLC program

This chapter describes how to create a PLC program establishing TCP/IP communication.

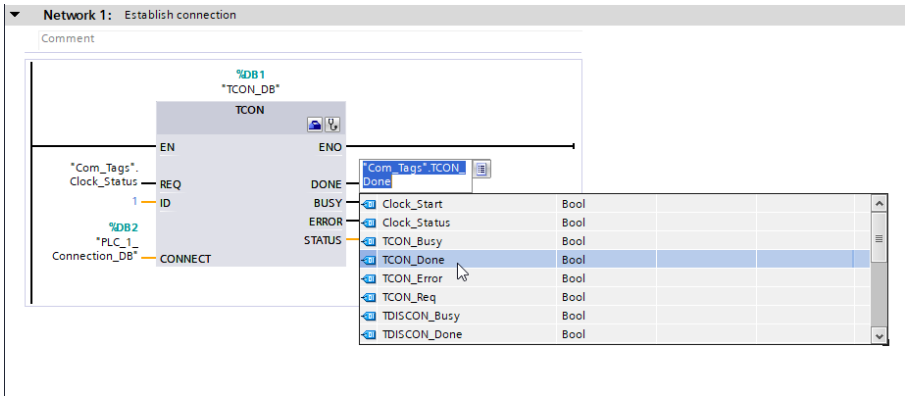
You should already have the TCON function block in the *Network view.* Now it is time to create a *Variable Table* by double-clicking on the *Add new block* in the *Project Tree* like it is shown here. Select the Data block (DB) and name the block as shown in the image below.



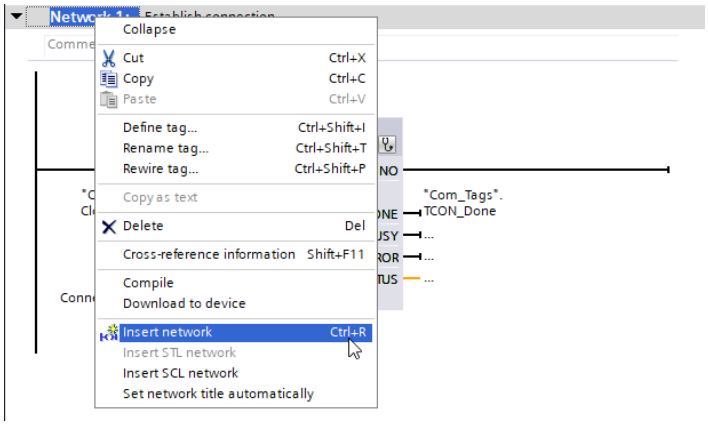
Now define variables in the *Data block*. Use the same names and data types as shown in the image below. If you want to create a new row, just right click on any row and select *Insert row.* Please note that setting the right values of *Start Values* is essential in this step.



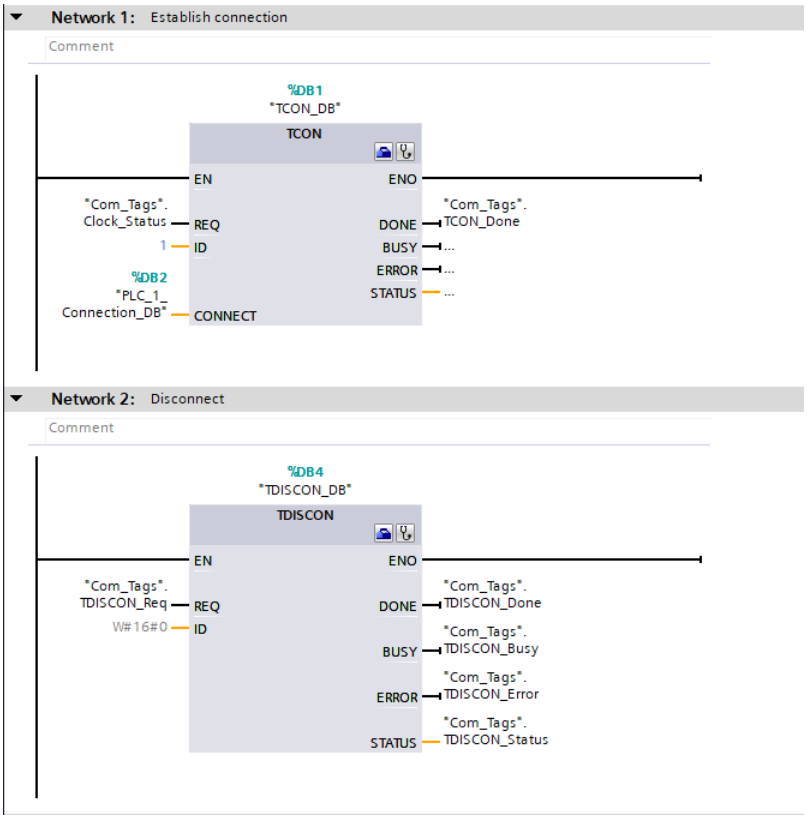
Label all the inputs and outputs of the TCON block in the network view. To label connection drag and drop variables from Data block to the Program block (for example TYCON) or double click on a connection and select displayed icon like in the image shown below.



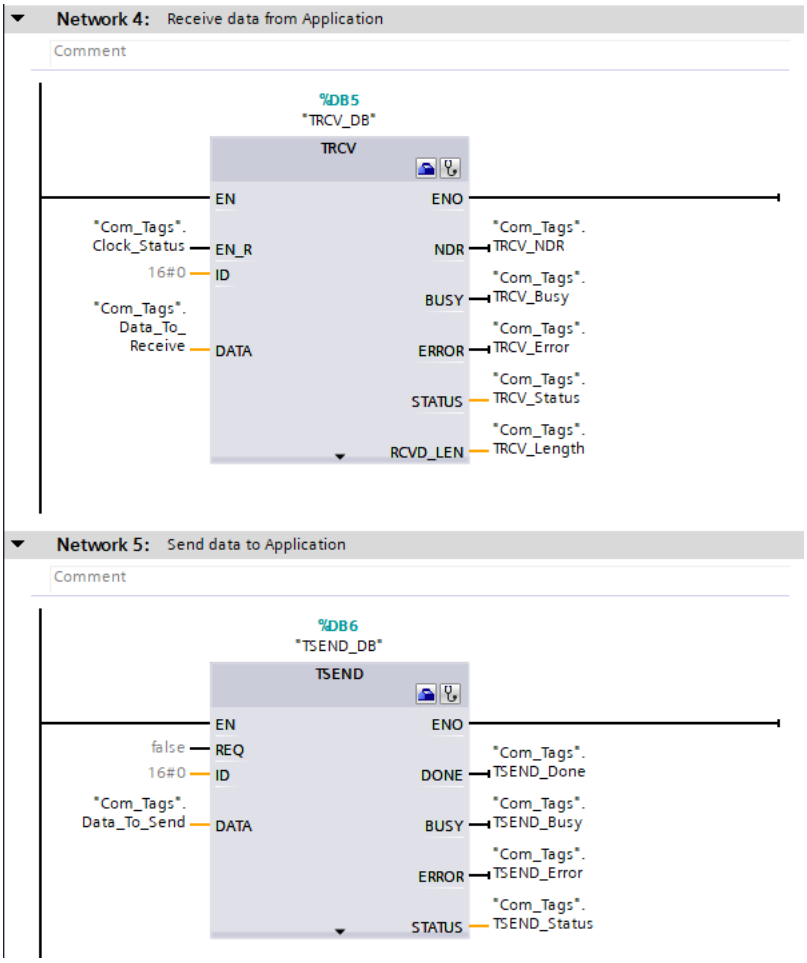
Add 5 networks to the Main program. In order to do that right-click on the existing network and select Insert network as shown in the image below:



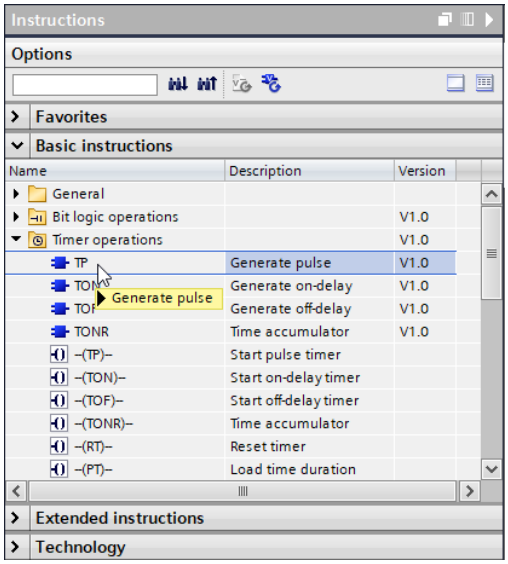
Insert additional communication block, TDISCON, from the Communication tab like on this previous picture. Label all connections as shown in the image below. TCON block will be used to establish TCP/IP connection while TDISCON will be used to close the connection.



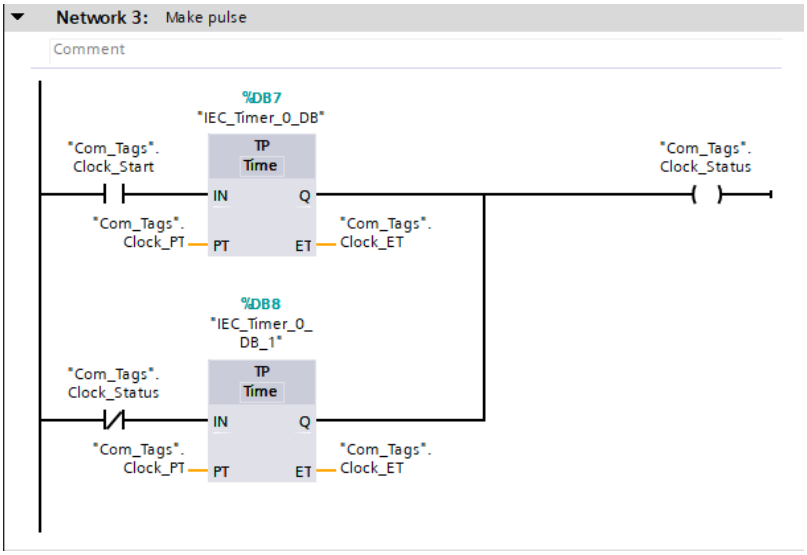
The next step is to allow exchanging data between Aurora Vision Studio and PLC. Use TRCV block for receiving messages and TSEND block for sending messages to Aurora Vision Studio. Label added blocks as shown in the image below:



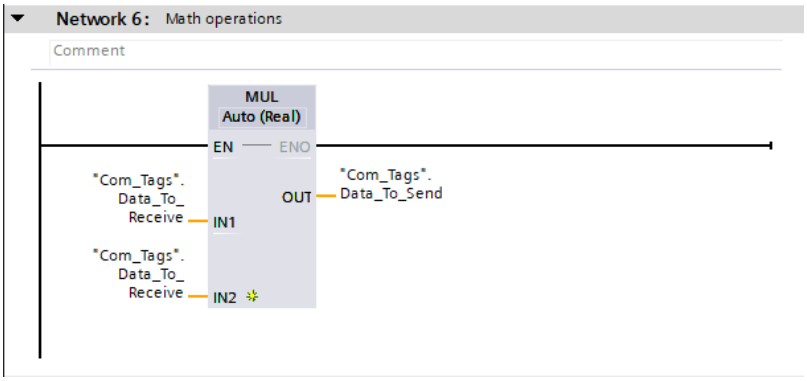
To start connection over TCP/IP using TCON you need to use rising edge signal on REQ input. Same applies to other function blocks (in TRCV a rising edge signal should be set on EN_R input). You can switch these values manually by right-clicking on the connection and selecting *Modify*. In this sample application, an automatic pulse generator will be used in order to avoid switching the values manually. Insert a new network and add a TP block located in the *Basic instructions* tab inside the *Timer operation* folder (picture below).



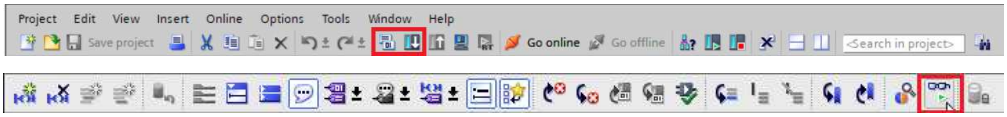
Use configuration from next picture to create a network which will be generating a proper signal.



In the last step please add a sample math function e.g. multiplying. In this example a number received from Aurora Vision Studio will be squared and the result calculated on the PLC's side will be sent back to Aurora Vision Studio.



If all previous steps have been done correctly, your PLC program is ready. Compile the program and download it to the device. To see current states of variables, turn on *Monitoring mode*.



Creating application in Aurora Vision Studio

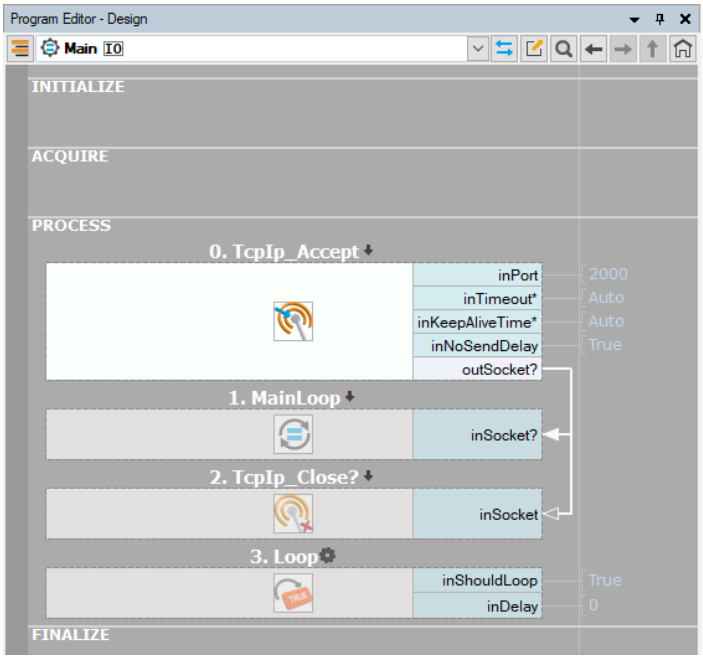
In this chapter, the individual steps how to create an Aurora Vision Studio application to communicate with a PLC over TCP/IP are described. Before proceeding to the next steps, make sure that all previous steps have been done correctly.

Aurora Vision Studio provides a set of ready-to-use filters for communication over the TCP protocol. A full list of the filters is available here. To get more detailed information on how to work with TCP filters, please refer to our official example *IO Simple TCPIP Communication* and the article Using TCP/IP Communication in the Aurora Vision Studio documentation.

First of all, create a new project and add a TcpIp_Accept filter. This filter, as well as other TCP/IP filters, are available in the Program I/O category of a Toolbox. Please note that TcpIp_Accept filter accepts a connection from a remote client, so in this case the Aurora Vision Studio program will be working as a server. If you want the program to work as a Client, the TcpIp_Connect filter should be used.

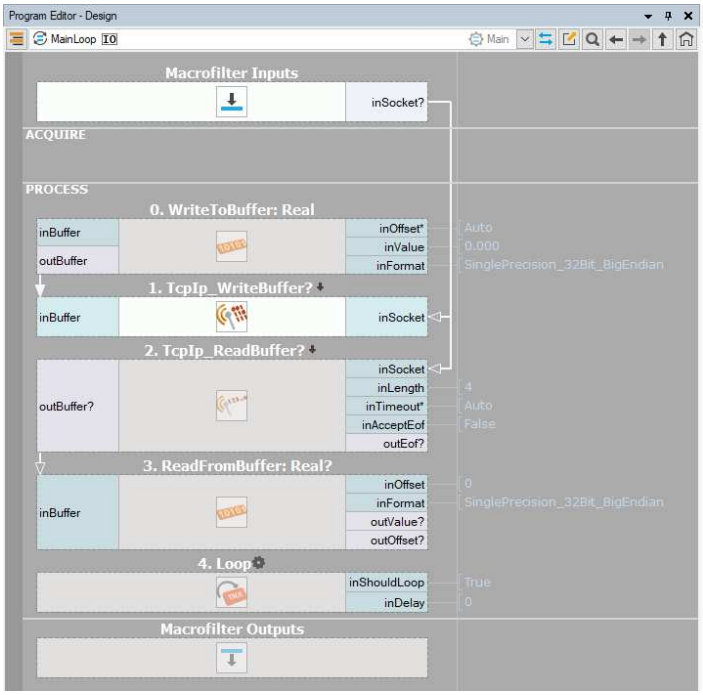
Set inPort value to 2000 to match configuration from PLC. Insert a TcpIp_Close filter below the TcpIp_Accept and connect the **outSocket** output with the **inSocket** input as shown in the image below. If the connection has been configured properly, you should be able to step over all steps in program.

Create a new Task Macrofilter and create an **inSocket** input (you can do this by dragging the outSocket and dropping it on the macrofilter). Connect **outSocket** from TcpIp_Accept filter to the **inSocket** input of *Task Macrofilter*. In this example Aurora Vision Studio will connect over TCP/IP only once and the connection will be held. The data exchange will be executed inside the Task Macrofilter.

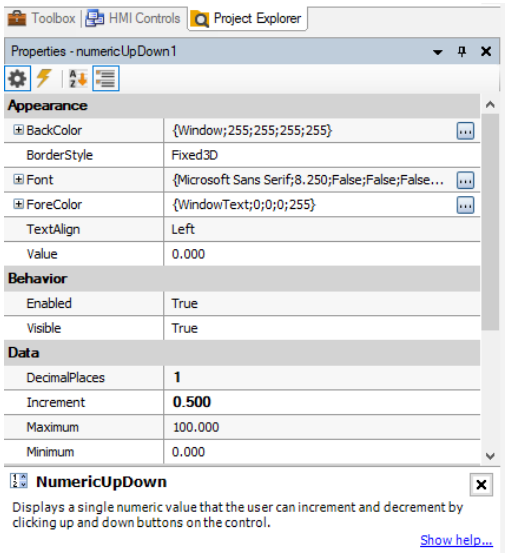


Inside the *MainLoop* macrofilter you need to add the Tcplp_WriteBuffer filter to send messages and Tcplp_ReadBuffer to receive messages over TCP/IP. PLC program works on *Buffer* data type, in our example user will be specifying decimal numbers, therefore conversion from *Buffer* to Real value is needed. To do so, use the WriteRealToBuffer and ReadRealFromBuffer filters which automatically convert decimal value into specified binary representation and write it to/read it from a buffer.

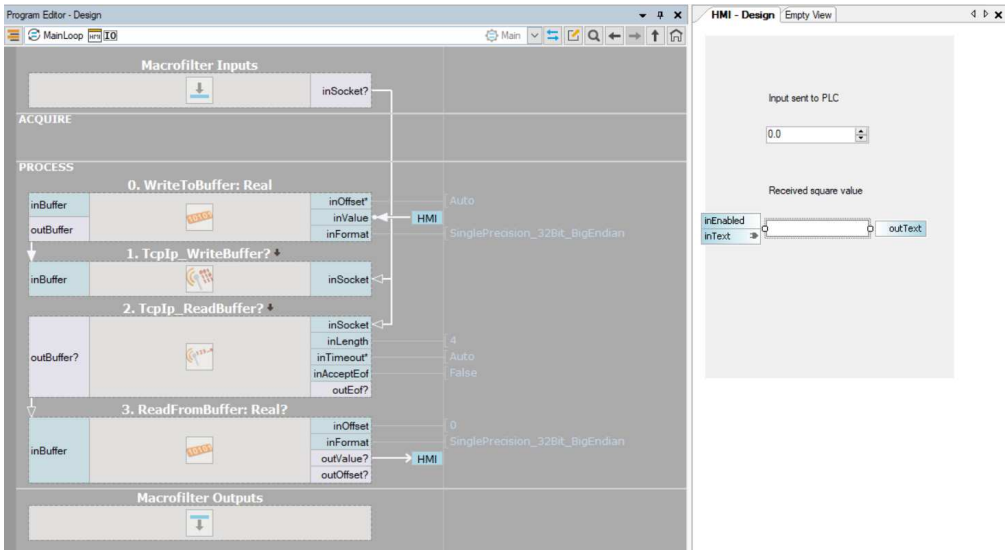
Add the Loop filter at the end of the algorithm inside *MainLoop* macrofilter. Make sure to set proper format - *SinglePrecision_32Bit_BigEndian*, otherwise program will not work properly. The algorithm should look like the one shown in the next picture.



In the next step simple HMI will be created. Add a NumericUpDown and TextBox controls from the *Controls* tab. Connect NumericUpDown **outValue** output to the WriteRealToBuffer **inValue** input (previous picture) and the output **outValue** of ReadRealFromBuffer filter to the **inText** of the TextBox control. Set the properties of the NumericUpDown HMI control as shown in the next picture.

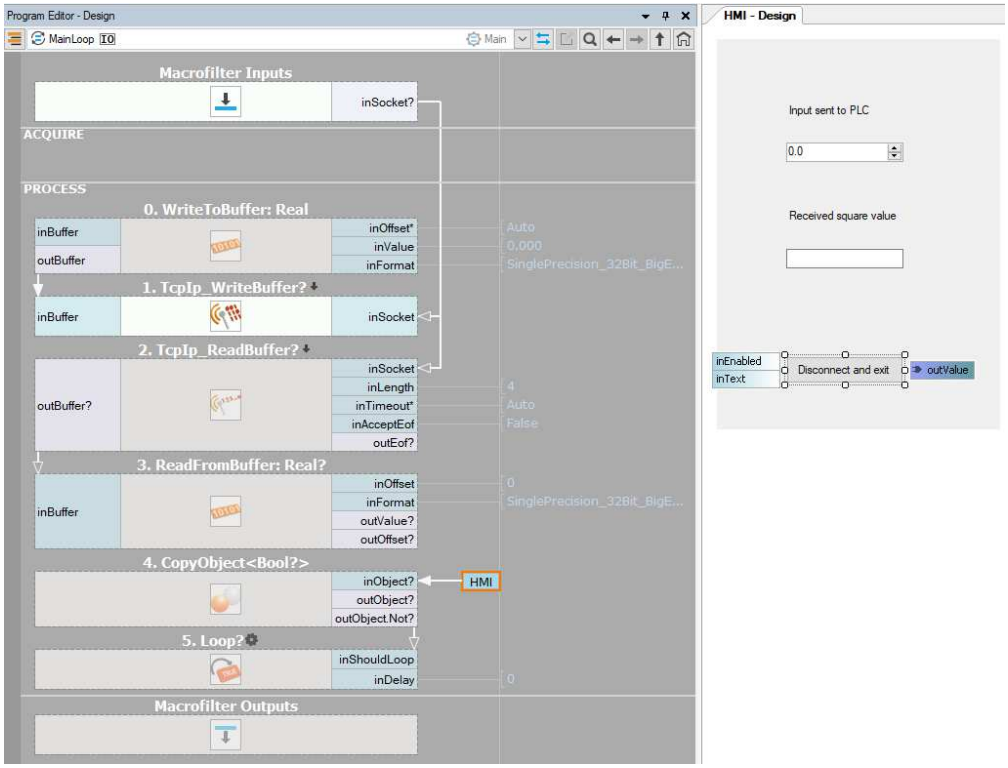


In the last step use Label control to describe the previously added controls. The current algorithm should look like the one shown in the next picture. The program can work in current state; however, it will be improved in next steps.

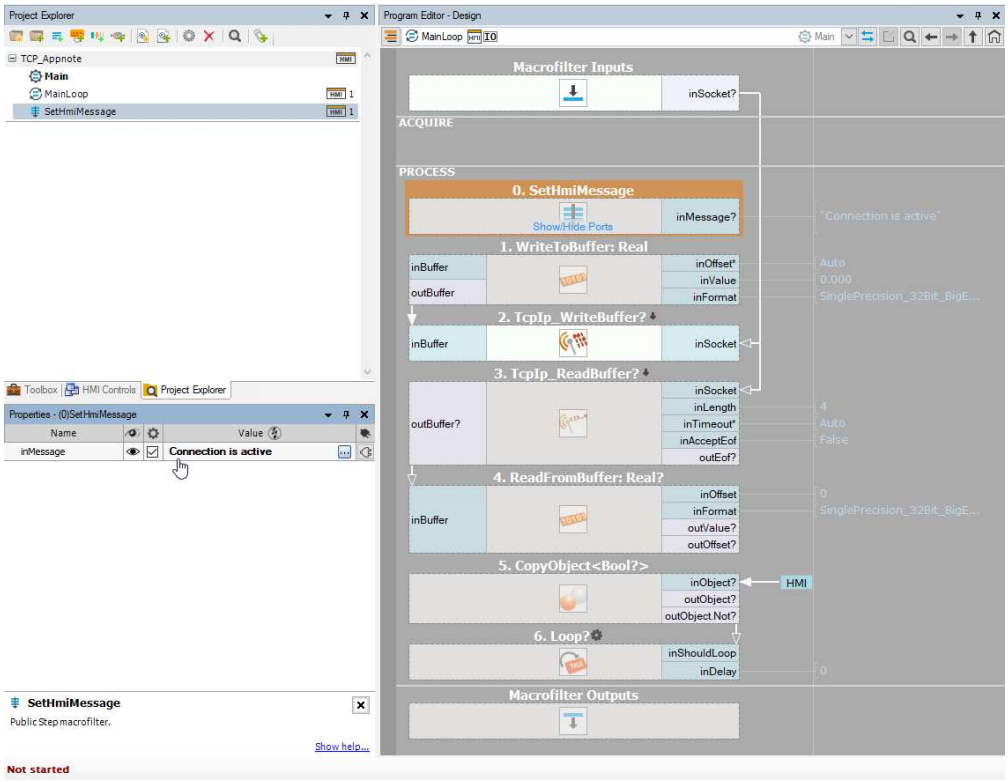


Main Loop is running continuously due to the **inShouldLoop** input of Loop, which is always true in the current structure. In order to change that, add ImpulseButton and name it "Disconnect and exit". Loop filter generates loop when true value is passed to the **inShouldLoop** input. Default state of ImpulseButton **outValue** output is false, so this value should be negated before connecting it to the Loop filter. Insert the CopyObject filter inside *MainLoop* macrofilter and set the Bool? data type. Connect ImpulseButton **outValue** to the **inObject** input of the CopyObject filter. Right click on the **outObject** output, select *Property Output* and *Not* variant. Now connect **outObject.Not** output to the **inShouldLoop** input as shown in the next picture.

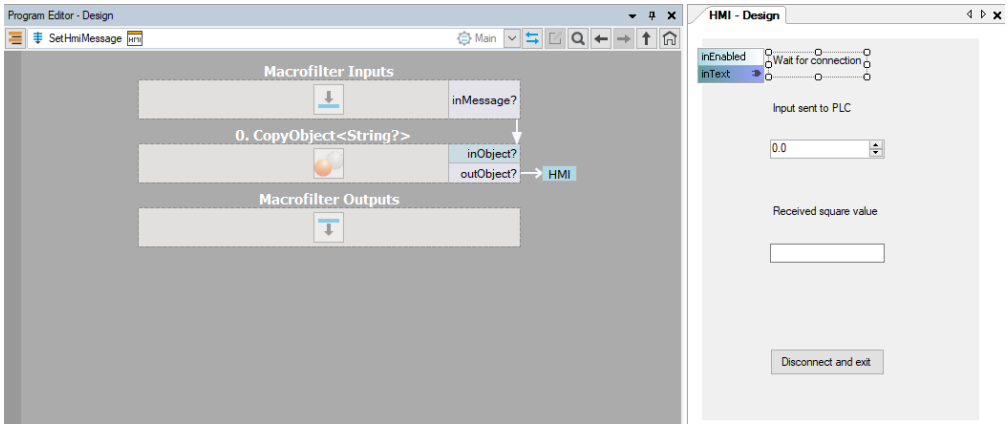
The same *ImpulseButton* control will be used to Close TCP/IP connection in Main task. Use the structure known from description above and Figure 25 to control Loop filter in Main task as shown in the image below:



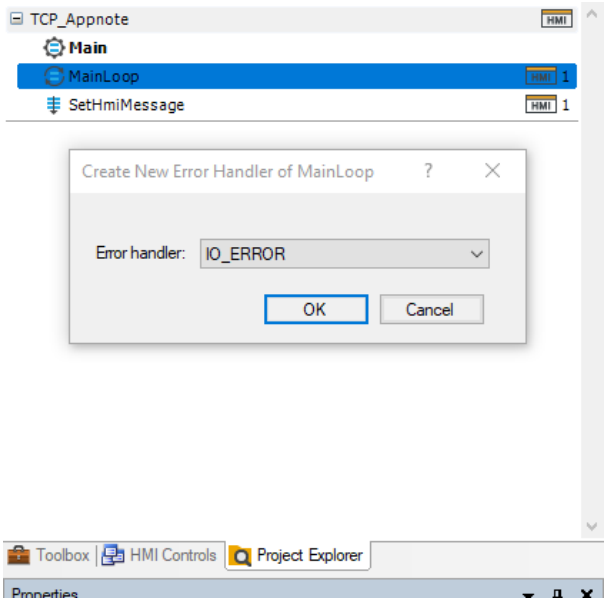
The application is almost ready, but every good and stable application should have Error handling. Next modification will allow you to see the current connection state, for example: wait for connection, connection is active, and lost connection. In order to do that, go inside *MainLoop* macrofilter, right click on the program editor window and insert a step macrofilter. Name new created macrofilter as "SetHmiMessage" and create new String? type input. Name the newly created input as **inMessage**. This macrofilter will be used in a few places is this application, in case you use it inside *MainLoop* macrofilter, insert command "Connection is active" in macrofilter properties as shown in image below:



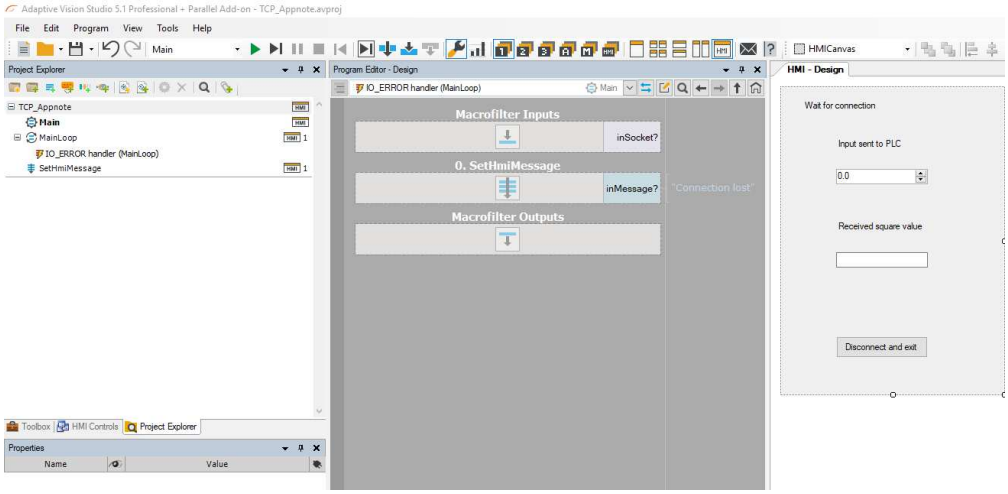
Enter *SetHmiMessage* macrofilter and add the CopyObject filter with String? type. Add a new Label control to the HMI window. As default text set "Wait for connection". Connect the **outObjects** output from CopyObject filter to the **inText** input of Label control. The result should look like in the image below:



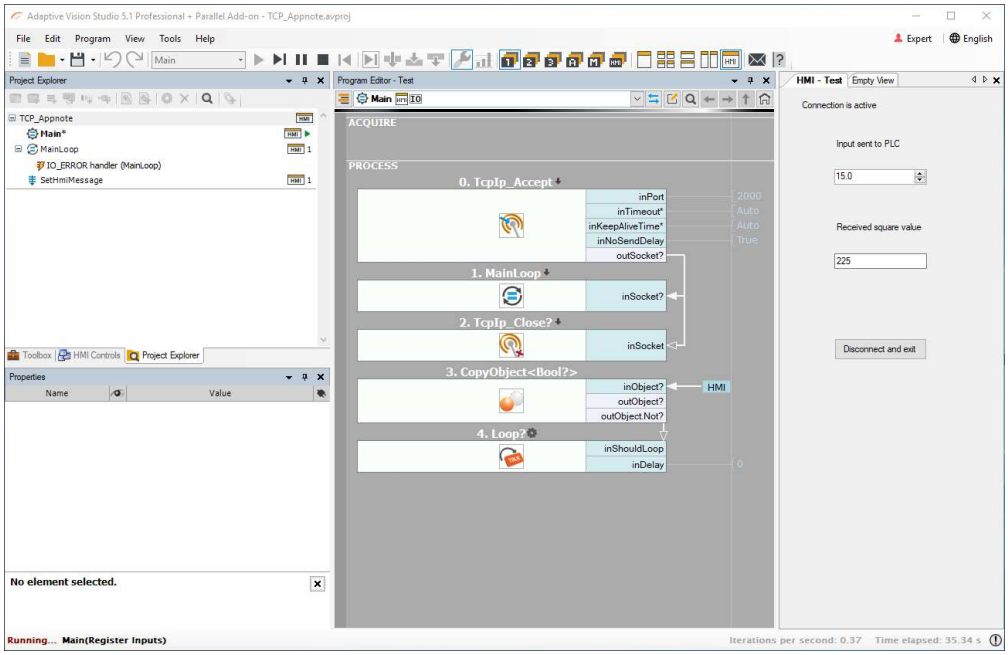
Currently, connection state message has two different variants: Wait for connection and Connection is active. Now, we are going to add message which will be prompted when connection is broken. In order to do that, you need to use Error handlers for *MainLoop* macrofilter. Right click on the MainLoop macrofilter, visible in the Project Explorer window, and select *Add New Error Handler...* Choose IO ERROR from the list, as shown in the next figure.



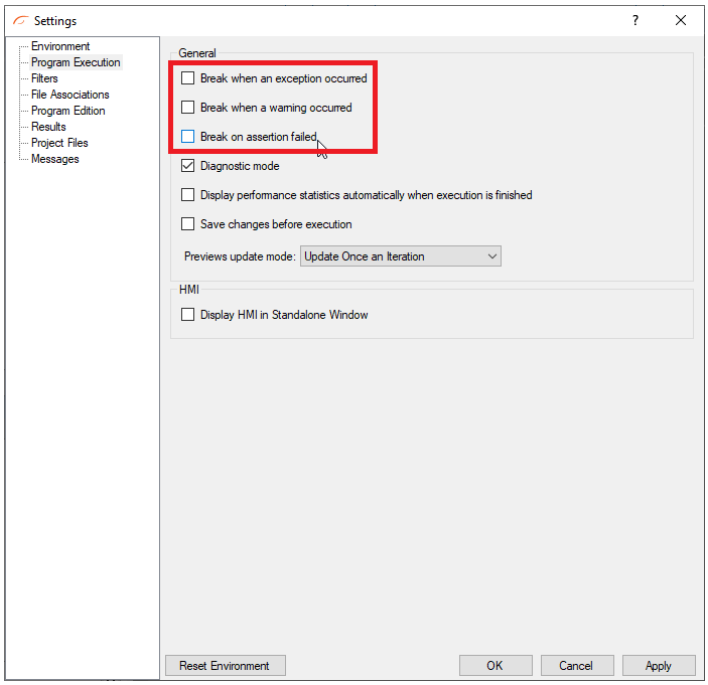
Enter newly created error handler and add *SetHmiMessage* macrofilter. Set **inMessage?** input to "Connection lost" as shown in the figure below:



The result should look like in the following picture. If all previous steps have been done correctly, program should work without any problems.



If you want to test program with occurring errors, you should unlock breaks in settings, as shown in the image below. With these settings, a pop-up window with error messages will not appear.



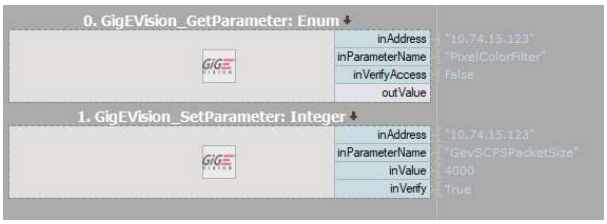
Changing parameters of GigEVision cameras

Introduction

The following guide provides information on how to programmatically choose and connect to a camera and modify its parameters. The modification of the acquisition parameters may involve restarting the acquisition. By implementing an interface to trigger the acquisition by software the program also shows how to execute camera commands. All the functionalities are available to the user through an HMI.

How to interact with camera

In Aurora Vision Studio users can interact with a GigE camera through various filters. Those allow the user to change some of the camera's parameters or access acquisition data. Filters GigEVision_SetParameter and GigEVision_GetParameter can be used to read and write parameter's values. Their common inputs include camera address and name of the parameter. The last common input - **inVerify** is used to determine if the program should check the validity of parameter before write or read.



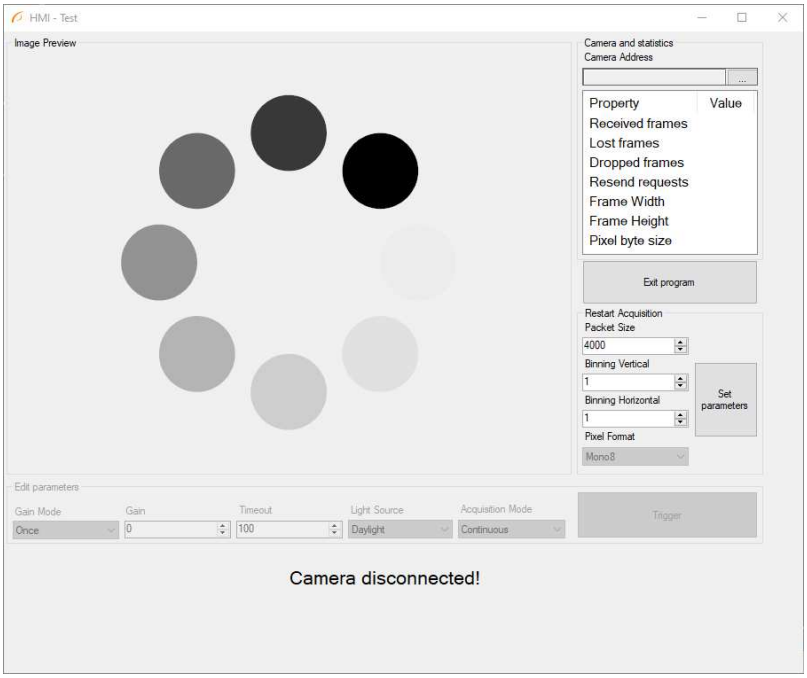
GigEVision_SetParameter has one more input, where the user specifies the value to be written. In GigEVision_GetParameter there is an output with the value of the parameter.

Both filters have variants depending on the type of the parameter being accessed - Real, Integer, Bool, Enum and String.

To see what the possible values for a given parameter are, the user may just click on **inParameterName** which shows a window with available parameters for the currently connected camera.

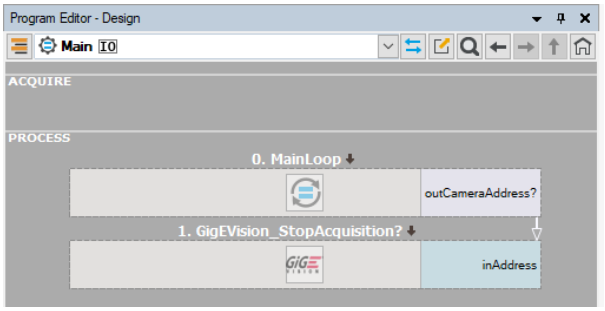
Program overview

The designed program allows the user to select a camera from a list of connected devices. After selecting one, the acquisition starts. The program displays a preview of the camera images. The user may modify selected camera parameters like gain, light source correction, and acquisition mode. If the camera is set to triggered acquisition mode, the user can release a software trigger. While running the program displays information about the current acquisition.



Under the preview there are two groups of controls. The controls on the left allow changing the parameters that require a restart of acquisition. Because of that there is also a button which lets the user restart the acquisition with new parameters.

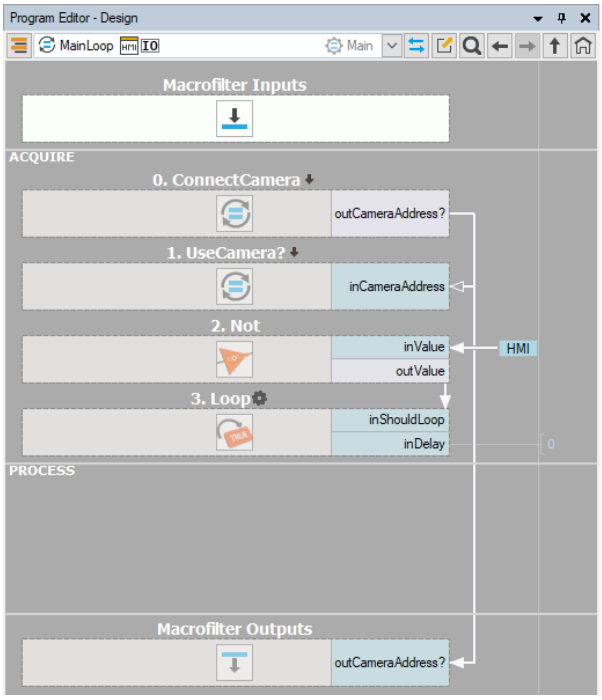
The other group has controls for the parameters that can be changed during the acquisition. The button labeled "Trigger" is used to start the acquisition using with a software trigger. It is disabled when in continuous mode.



The *MainLoop* filter of the program can be divided into 3 parts. The first part is connecting the camera - here it is done by the *ConnectCamera* task macrofilter.

The second and the third part are done by the *UseCamera* task macrofilter. First the program starts the acquisition with set parameters. Then the program continuously acquires images while being able to change some of the camera parameters.

The **Not** and **Loop** filters control whether *MainLoop* will continue. The Not filter is connected to the "Exit program" HMI button. If the button has been pressed the loop will not continue - the program goes back top *Main* and stops the acquisition (if a camera had been connected) which finishes the program.



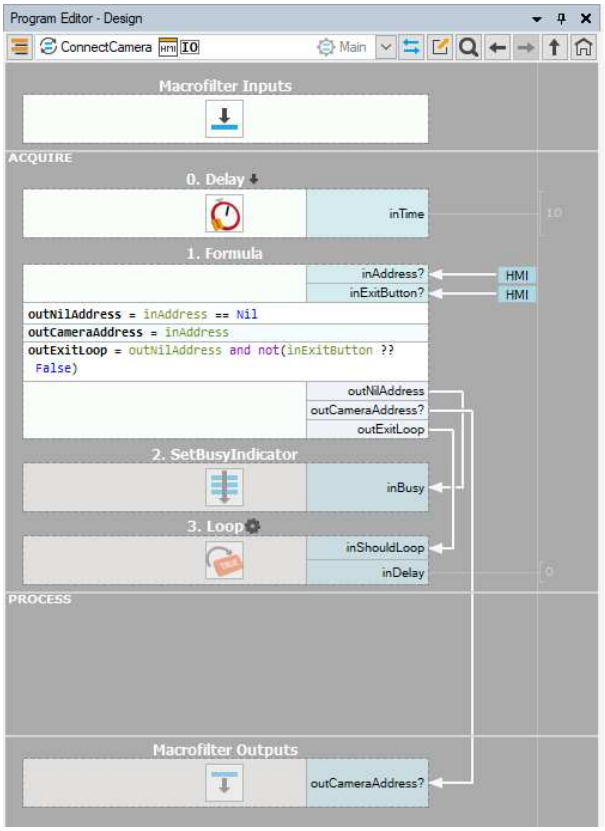
Connecting the camera

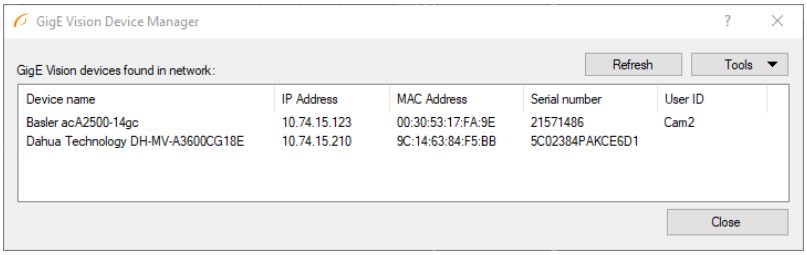
The first part of the program allows the user to connect a camera. The delay in the filter is used to limit the number of iterations per second. The **inAddress** input of the formula is connected to the HMI control labeled *Camera Address*.

After clicking the 3-dot button a window opens letting the user select the camera. The selected camera's address is the sent to the formula block.

The animated waiting indicator is visible only if no camera has been chosen (so the address is Nil). The state of the indicator is controlled by the **outNilAddress** output of the formula.

If the user chooses a camera and its address is no longer Nil this exits the loop of this macrofilter and the camera address is outputted.





Setting the parameters

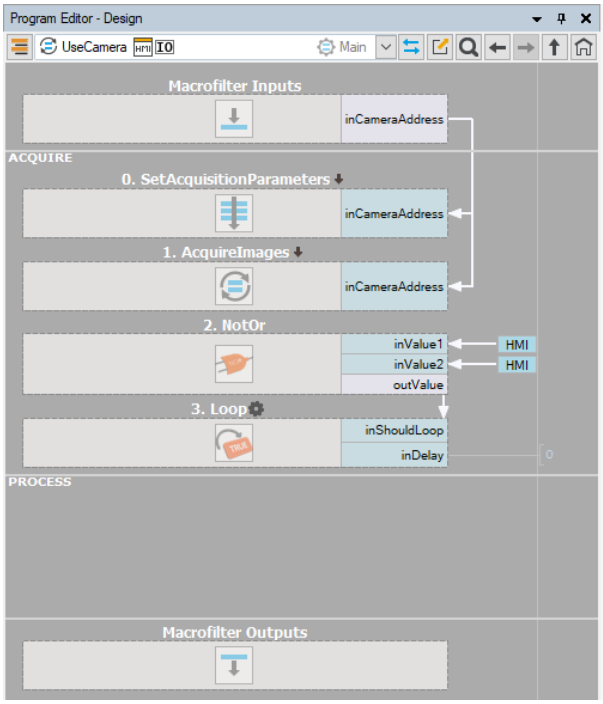
After the camera has been connected the next step is to start the acquisition. However, some acquisition parameters can only be set while not acquiring images. Such parameters include binning, packet size and pixel format. They are set in the *SetAcquisitionParameters* step macrofilter

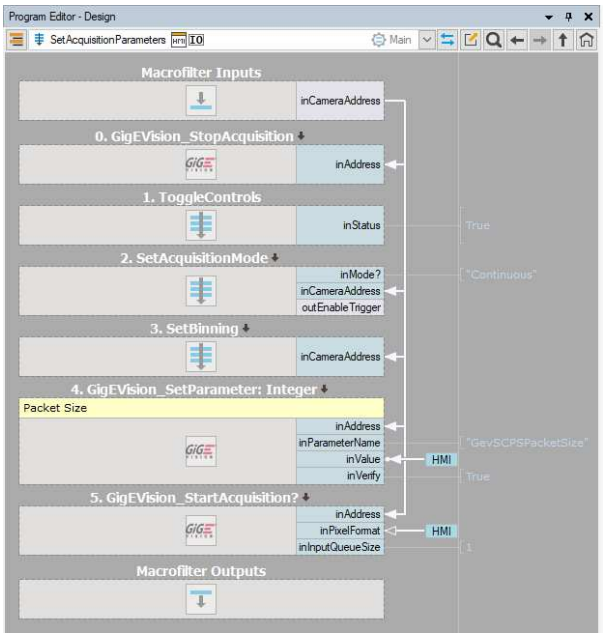
The NotOr filter negates the sum of two boolean values related to HMI controls. The first being **outAddressChanged** of the camera address picker and the second - whether the "Exit program" button has been pressed. If both are false the value of NotOr is true, which enables the loop to continue. If at least one of them is true - the program exits the *UseCamera* loop and goes back to *MainLoop*.

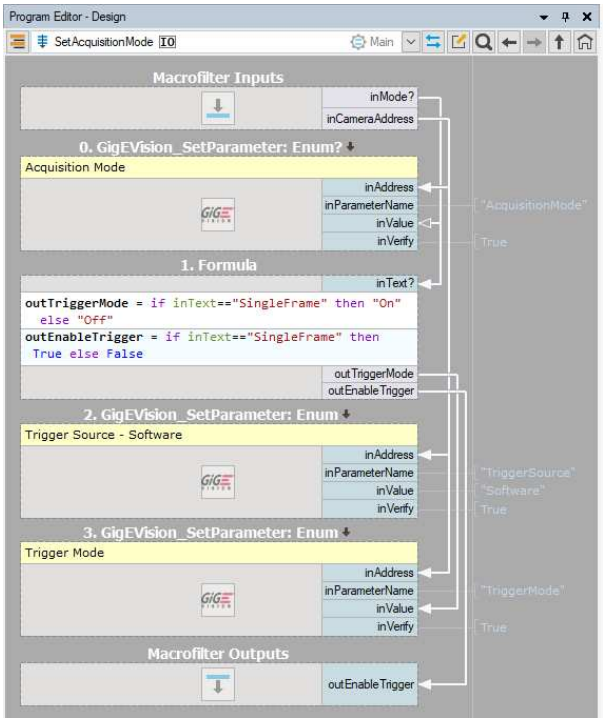
First any undergoing acquisition is stopped to ensure the parameters can be set. After that the mode of acquisition is set to "Continuous". This allows the camera to grab images to show in the preview of the program. The formula sets **outTriggerMode** to "Off" if the **inMode** parameter is anything other than "Single Frame".

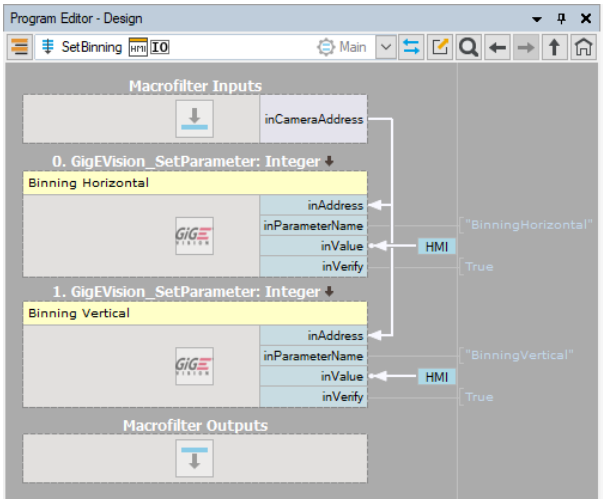
The next steps *SetAcquisitionParameters* set the binning of the camera (in both the horizontal and vertical axes) as well as the packet size. The possible values of the HMI controls used should be limited to be compatible with the camera value range. For example, the packet size here ranges from 220 to 16404 in increments of 4.

Finally, the program starts the acquisition with the pixel format chosen by the user.



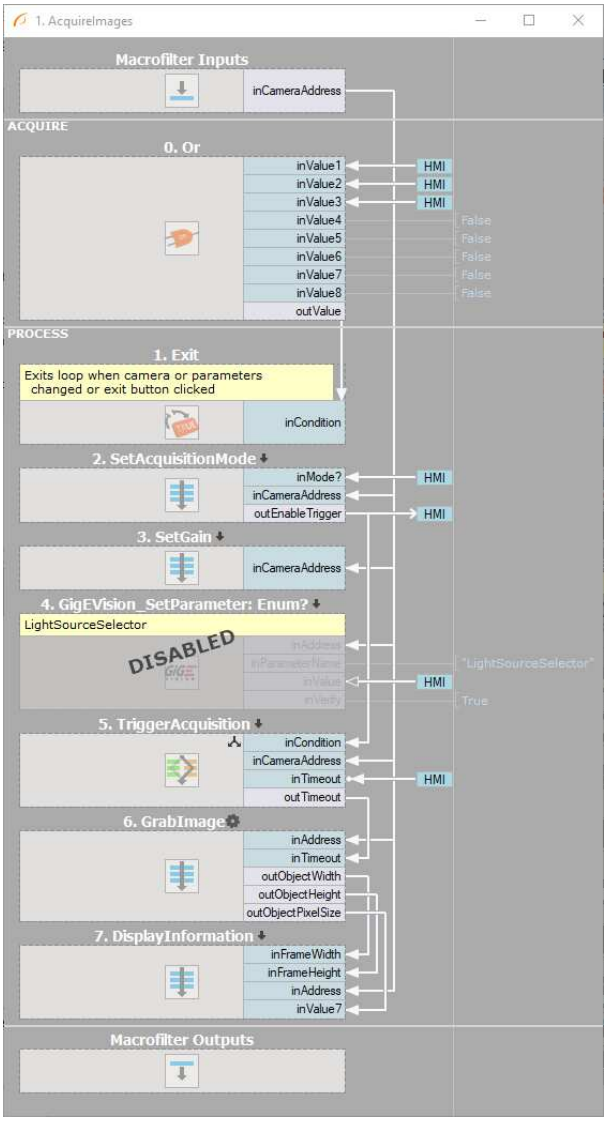






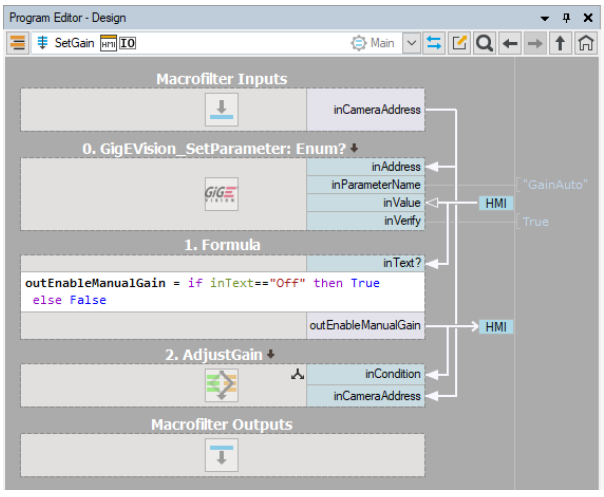
Acquiring images

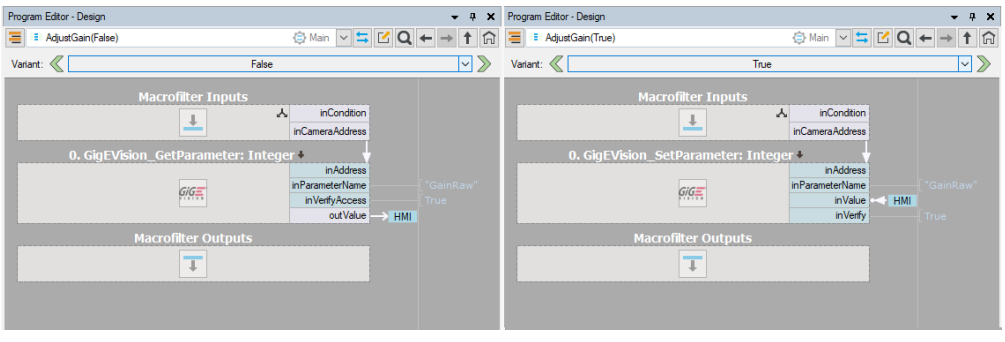
Acquiring images takes place in a loop in the filter *AcquireImages*. It begins with an **Or** filter that's sums the states of three HMI controls: "Exit program" button, **outAddressChanged** of the camera address picker and "Set parameters" button. If any of them is true, the program exits the loop and goes back to *UseCamera*.



After that there is another instance of *SetAcquisitionMode*. This time the **inMode** parameter is connected to the appropriate control allowing the user to switch between continuous and triggered acquisition.

SetGain allows the user to switch between different modes of gain adjustment as well as set its value manually. First the program sets whether the gain will be adjusted manually or not based on the value from the control. If it is, the formula below sets the state for the next filter as well as enables (or disables) the gain value control in the HMI.



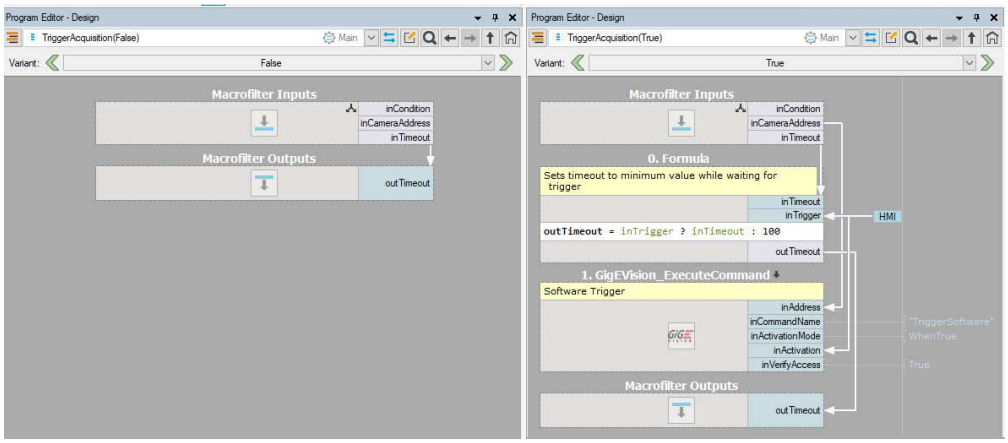


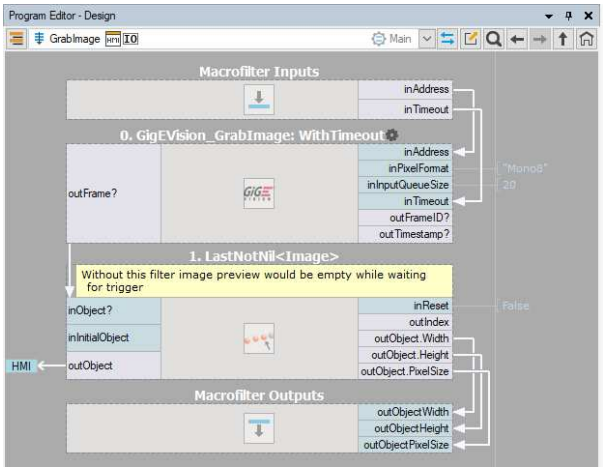
AdjustGain is a variant macrofilter. If gain is to be adjusted manually it sets the parameter to the given value. If gain is controlled automatically, it instead reads the value. This enables the user to see the automatically adjusted gain value in real time.

Next the light source is set. It allows the user to select the preset best suited to the current lighting conditions and by extension - to make colors look more natural.

TriggerAcquisition is another variant macrofilter. The **inTimeout** parameter sets how long should the program wait for an image from the camera. The variant for continuous acquisition is almost empty - it only passes **inTimeout** to **outTimeout**.

If the acquisition is triggered the program first check the state of the "Trigger" button. If it had been pressed the program executes the trigger command and passes **inTimeout** to **outTimeout**. If the button had not been pressed no command is executed and timeout is set to its lowest possible value - 100. Low timeout makes program more responsive when the user does not trigger the acquisition.

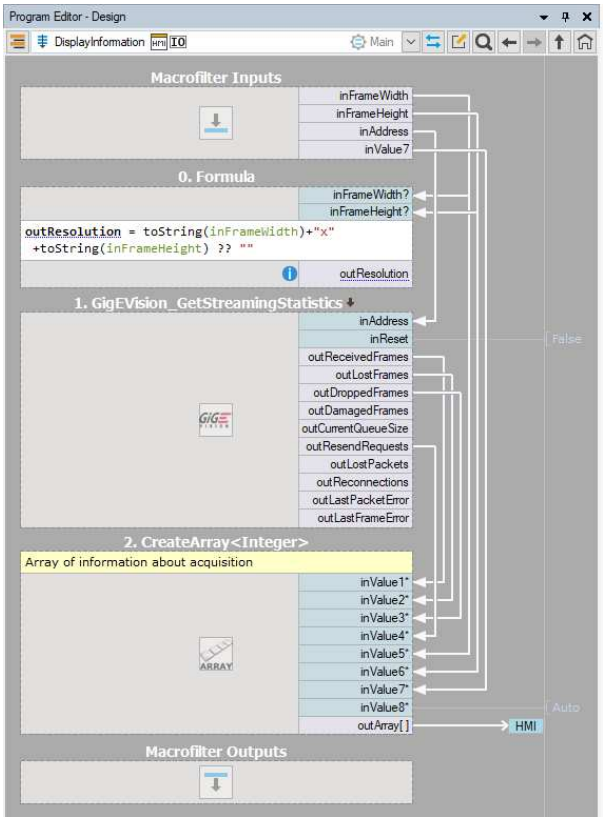




After that, in *GrabImage* macrofilter, the program grabs and displays the image from the camera. Since the **inPixelFormat** has been set when starting the acquisition in the previous part of the program it is not necessary to change the value of this parameter here.

LastNotNil filter keeps the image in the preview in case the user is in triggered acquisition mode. Without it all images from the camera would only be displayed for one iteration of *AcquireImages*.

The last macrofilter in *AcquireImages* gathers data about the current acquisition and displays it on the HMI.



Notes

- The program may not work on every camera. This is dependent on the available parameters as well as their values.
- It is possible to modify more parameters in the program.
- To better understand how some of the parameters work reading the camera's documentation is recommended.
- If functionalities of some of the filters used in this example are unclear, check their documentation.

12. Appendices

Table of content:

Backward Compatibility Policy

Programs created in Aurora Vision Studio are fully backward compatible within the same release number.

Between consecutive minor revisions (3.1, 3.2, ...) the following changes might be anticipated:

- User Filters might have to be rebuilt.
- Data files (e.g. Template Matching models) might have to be recreated.
- Generated C++ code might have to be re-generated.

Quick Start Guide for the Users of LabVIEW

Both Aurora Vision Studio and LabVIEW are based on the idea that the data-flow programming model can be very useful in data-processing applications. This makes the two applications similar on the general level, but still they are very different in the details. Aurora Vision Studio has been primarily designed for machine vision engineers, whereas LabVIEW is advertised as "software ideal for any measurement or control systems". This results in a completely different design of the user interface with Aurora Vision Studio concentrating on more sequential algorithms and Data Previews accessible directly within the Main Window.

If you have used LabVIEW before, below you will find a list of key differences that will allow you to start working with Aurora Vision Studio more quickly by mapping the new concepts to what you already know:

LabVIEW	Aurora Vision Studio	Notes
Nodes	Filter Instances	In both environments these elements have several inputs and outputs, and are the basic data processing elements.
Wires	Connections	Connections in Aurora Vision Studio encompass more program complexity than wires in LabVIEW. Like in LabVIEW there are basic connections and connections with data conversion (LabVIEW: coercion). There are, however, also array connections that transmit data in a loop and conditional connections that can make the target filter not executed at all.
Basic Data Types		Aurora Vision Studio has two numeric data types: Integer and Real, both 32-bit. There are also Booleans (LabVIEW: *Boolean*), Strings, File (LabVIEW: *Path*) and enumerated types.
Arrays and Clusters	Arrays and Structures	Arrays and Structures are more similar to the corresponding elements of the C language. Arrays in Aurora Vision Studio can be multi-dimensional, e.g. one can have arrays of arrays of arrays of integer numbers. Structure types are predefined and their elements can be "bundled" and "unbundled" with appropriate Make and Access filters.
Local Variables	Labels	The programming model of Aurora Vision Studio enforces the use of data flow connections instead of procedural-style variables. Instead of local variables you can use labels, that replace connections visually, but not actually.
Global Variables	Global Parameters	Global Variables in LabVIEW are recommended mainly for passing information between VIs that run simultaneously. In Aurora Vision Studio this is similar – global parameters should be used to communicate between Parallel Tasks and with HMI Events.
Dynamic values / Polymorphic VIs	Generic Filters	Generic Filters of Aurora Vision Studio are more similar to templates of the C++ programming language. The user specifies the actual type explicitly and thus the environment is able to control the types of connections in a more precise way.
Waveform	Profile	A sequence of numeric values that can be depicted with a 2D chart is called Profile.
Virtual Instrument (VI, SubVI)	Macrofilter	A macrofilter is a sequence of other filters hidden beyond an interfaces of several inputs and outputs. It can be used in many places of a program as it was a regular filter. Macrofilters do not have their individual front panels. Instead, the environment of Aurora Vision Studio is designed to allow output data preview and input data control.
Front Panel	HMI	In Aurora Vision Studio, HMI (Human-Machine Interface) is created for the end user of the machine vision system. There is thus single HMI for a project. There are no blocks in the Program Editor that correspond to HMI controls. The connections between the algorithm and the HMI controls are represented with "HMI" labels.
For Loop, While Loop	Array Connections, Task Macrofilter	There are two methods to create loops. The first one is straightforward – when the user connects an output that contains an array to an input that accepts a single element, then an array connection is used. A for-each loop is here created implicitly. The second is more like the structures of LabVIEW, but also more implicit – the entire Task macrofilter works in a loop. Thus, when you need a nested loop you can simply create a new Task macrofilter. These loops are controlled by the filters that are used – more iterations are performed when there are filters signaling ability to generate new data.
Shift Registers	Registers	Registers in Aurora Vision Studio are very similar to Shift Registers. One difference is that the types and initial values of registers have to be set explicitly. Step macrofilters preserve the state of registers between subsequent executions within a single execution of the Task that contains them. There are no Stacked Shift Registers, but you can use the LastTwoObjects / AccumulateElements filters instead.
Case Structures	Variant Macrofilter	While Task Macrofilters can be considered an equivalent of the While Loops of LabVIEW, Variant Macrofilters can be considered an equivalent of the Case Structures. Selector Terminals and Cases are called Forking Ports and Variants respectively.
Sequence Structures	–	All macrofilters in Aurora Vision Studio are executed sequentially, so explicit Sequence Structures are not needed.
Controls Palette	HMI Controls	
Functions Palette	Toolbox	
Formula / Expression Nodes	Formula Blocks	Formula Blocks are used to define values with standard textual expressions. Several inputs and outputs are possible, but loops and other C-like statements are not. This feature is thus something between LabVIEW's Expression and Formula Nodes. If you need C-like statements, just use C++ User Filters that are well integrated with Aurora Vision Studio.
Breakpoints	Iterate Current Macrofilter	As there are no explicit loops other than the loops of Task macrofilters, a macrofilter is actually the most appropriate unit of program debugging. One can use the Iterate Current Macrofilter command to continue the program to the end of an iteration of the selected macrofilter.
Error Handling		In Aurora Vision Studio there are no *error in/out* ports. Instead, errors are handled in separate subprograms called Error Handlers. In some cases when an output of a tool cannot be computed, conditional outputs are used. The special value *Nil* is then signaling a special case. This is for example when you try to find the intersection of two line segments which actually do not intersect.
Call Library Function Node	User Filter	User Filters can be used to execute pieces of code written in Microsoft Visual C++. The process of creating and using User Filters is highly automated.

Quick Start Guide for the C/C++ Programmers

Aurora Vision Studio has been created by developers, who were previously creating machine vision applications in C++. We created this product to make this work much more efficient, whereas another our goal was to retain as much of the capabilities and flexibility as possible. We did not, however, simply create a graphical interface for a low level C++ library. We applied a completely different programming paradigm – the Data Flow model – to find the optimum balance between capabilities and development efficiency. Programming in Aurora Vision Studio is more like designing an electrical circuit – there are no statements and no variables in the same way as they are not present on a PCB board. The most important thing to keep in mind is thus that there is no direct transition from C++ to Aurora Vision Studio. You need to **stop thinking in C++ and start thinking in data flow** to work effectively. Automatic C++ code generation is still possible from the data flow side, but this should be considered a one-way transformation. At the level of a data flow program there are no statements, no *if*s, not *for*s.

So, how should you approach constructing a program, when you are accustomed to such programming constructs as loops, conditions and variables? First of all, you need to look at the task at hand from a higher level perspective. There is usually only a single, simple loop in machine vision applications – from image acquisition to setting digital outputs with the inspection results. It is highly recommended to avoid nested loops and use Array Connections instead, which are data-flow counterparts of *for-each* loops from the low level programming languages. For conditions, there is no *if-then-else* construct anymore. There are Conditional Connections instead (data may flow or not), or – for more complex tasks – Variant Macrofilters. The former can be used to skip a part of a program when some data is not available, the latter allow you to create subprograms that have several alternative paths of execution. Finally, there are no variables, but data is transmitted through (usually unnamed) connections. Moreover, Global Parameters can be used to create named values that need to be used in many different places of a program, and Macrofilter Registers can be applied to program complex behaviors and store information between consecutive iterations of the program loop.

Please note, that even if you are an experienced C++ programmer, your work on machine vision projects will get a huge boost when you switch to Aurora Vision Studio. This is because C++ is designed to be the best general purpose language for crafting complex programs with complicated control flow logic. Aurora Vision Studio on the other hand is designed for one specific field and focuses on what is mostly important for machine vision engineers – the ability to experiment quickly with various combinations of tools and parameters, and to visualize the results instantly, alone or in combination with other data.

Here is a summary:

C++	Aurora Vision Studio	Notes
Conditions (the *if* statement)	Conditional Connections, Variant Macrofilters	Aurora Vision Studio is NOT based on the control flow paradigm. Instead, data flow constructs can be used to obtain very similar behavior. Conditional connections can be used to skip some part of a program when no data is available. Variant Macrofilters are subprograms that can have several alternative paths of execution. See also: Sorting, Classifying and Choosing Objects
Loops (the *for* and *while* statements)	Array Connections, Task Macrofilters	Aurora Vision Studio is NOT based on the control flow paradigm. Instead, data flow constructs can be used to obtain very similar behavior. Array connections correspond to *for-each* style loops, whereas Task Macrofilters can be used to create complex programs with arbitrary nested loops.
Variables	Connections, Global Parameters, Macrofilter Registers	Data flow programming assumes no side effects. Computed data is stored on the filter outputs and transmitted between filters through connections. Global Parameters can be used to define a named value that can be used in many different places of a program, whereas Macrofilter Registers allow to store information between consecutive iterations.
Collections (arrays, std::vector etc.)	Arrays	The Array type is very similar to the std::vector<T> type from C++. This is the only collection type in Aurora Vision Studio.
Templates	Generic Filters	As there can be arrays of many different types (e.g. RegionArray, IntegerArrayArray), we also need to have filters that transform arrays of different types. In C++ we have template metaprogramming and the STL library that is based on it. In Aurora Vision Studio we have generic filters, which are very similar to simplified templates from C++ – the type parameter has to be defined when adding such filter to the program.
Functions, methods	Macrofilters	Macrofilters are subprograms, very similar to functions from C++. One notable difference is that macrofilters can not be recursive. We believe that this makes programs easier to understand and analyze.
GUI Libraries (MFC, Qt, WxWidgets etc.)	HMI Designer	If more complex GUI is needed, the algorithms created in Aurora Vision Studio can be integrated with a GUI written in C++ through C++ Code Generator. See also: Handling HMI Events.
Static, dynamic libraries	Modules	Bigger projects require better organization. As you can create libraries in C++ which can be used in many different programs, you can also create modules (a.k.a. libraries of macrofilters) in Aurora Vision Studio.
Breakpoints	The "Iterate Current Macrofilter" command (Ctrl+F10).	As there are no side effects within macrofilters, there is no need to set breakpoints in arbitrary places. You can, however, run the program to the end of a selected macrofilter – just open this macrofilter in the Program Editor and use the "Iterate Current Macrofilter" command. The program will pause when it reaches the end of the selected macrofilter instance.
Threads		There are no threads in Aurora Vision Studio. Instead, the filters utilize as many processors as possible internally and the HMI (end user interface) is automatically run in parallel and synchronized with the program loop.
Exceptions	Domain Errors, Conditional Outputs	Domain Errors signal unexpected conditions encountered during program execution. They cause the program to stop, so it is crucial to make sure they are not possible. Nevertheless, during development they provide important information about what has to be fixed. If an unexpected condition cannot be easily predicted with careful program construction, then conditional outputs are used and the *Nil* value is returned to signal failure of execution.

Interoperability with C++

Having said, that you can solve even the most challenging machine vision tasks with the data-flow programming model, in real life you most often need to integrate the machine vision solution with a bigger system. This integration most often requires C++ or .NET programming. Aurora Vision Studio comes with several features that make it possible:

- User Filters allow to add your own C++ code in the form of filters of Aurora Vision Studio.
- C++ Code Generator allows to switch from the graphical environment of Aurora Vision Studio to a C++ program based on the AVL.DLL library.
- .NET Macrofilter Interface Generator produces a .NET assembly (a .dll file) with methods corresponding to macrofilters of a program, thus providing a bridge between Aurora Vision Studio and .NET technology

Other remarks:

- The program files of Aurora Vision Studio are based on textual formats. You can use your version control system to store them and monitor their history.

FAQ

Question:

How to mark the end of a loop started with filters such as EnumerateIntegers or Loop?

Answer:

This is by design different than in C++. The loop goes through the entire Task macrofilter, which is a logical part of a program. If you really need a nested loop (which is rare in typical machine vision projects), then create a Task macrofilter for the entire body of the loop. First of all, however, consider array connections. They allow for example to inspect many objects detected on a single image without creating an explicit loop.

Question:

Could you add a simple "if" filter, that takes a boolean value and performs the next filter only if the condition is met?

Answer:

This would be a typical construct in the control-flow based programming languages. We do not what to mix different paradigms, because we must keep our software not too complicated. You can achieve the same thing by using the MakeConditional filter and passing data to the next filter conditionally then. If there is no appropriate data that could be used in that way, then a variant macrofilter might be an another solution.

Question:

How to create a variable?

Answer:

There are no variables in data-flow. This is for the same reason you do not see variables on PCB boards or when you look at a production line in a factory. There is a flow instead and connections transmit data (or objects) from one processing element to another. If you need to store information between consecutive iterations, however, then also stateful filters (e.g. AddIntegers_OfLoop, macrofilter registers or appropriate functions in formula blocks can be used.

Deep Learning Service Installation

Contents

1. Installation guide

To use Deep Learning Filters, Library or Service with Aurora Vision Studio or Aurora Vision Library, a corresponding version of Aurora Vision Deep Learning must be installed (the best idea is to use the newest versions of both from our website). Before installation, please check your hardware configuration.

Deep Learning is available in two versions:

- GPU version (recommended) - version working with CUDA GPU acceleration. Much faster than CPU counterpart.
- CPU version - uses only CPU, GPU is not required and used. Relatively slow, especially during training phase.

Requirements

- Graphics card compatible with CUDA toolkit. List of compatible devices can be found on this website (all CUDA devices with "Compute Capability" greater than or equal 3.5 and less than or equal 8.6). Minimum 2 GB of graphic memory is recommended. Display Driver with at least 461.33 version is required (recommended latest display driver version).
- At least 3.5 GB disk space for program files, SSD recommended.
- At least 8 GB RAM memory.
- 64-bit processor, Intel i5, i7 or better are recommended. AVX support is required.
- Windows 7, 8 or 10.

Known issues

If you are getting Access Denied errors during updating (or uninstalling), close all processes that may use previously installed Deep Learning files, like programs that need Deep Learning Library, Aurora Vision Studio, Aurora Vision Executor and so on.

2. Aurora Vision Deep Learning Library and Filters

Aurora Vision Deep Learning provides Filters for usage in Aurora Vision Studio in 64-bit version. 32-bit version is not supported.

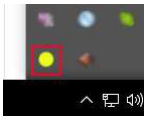
Aurora Vision Deep Learning also provides Library (with Training Api) in 64-bit version (32-bit version is not supported). It is installed in `Library` subdirectory of main installation directory. It contains more subfolders:

- `bin/x64` – a directory containing DLL file (`AVLDL.dll`) for 64-bit applications written in C++. This library is common for all supported versions of Microsoft Visual Studio and for Debug|Release configurations. This directory also contains `AvlDl.Net.dll` file (along with its documentation in `AvlDl.Net.xml` file), which is .Net wrapper for `AVLDL.dll`.
- `include` – a directory containing all header (`.h`) files for Library (`AVLDL.h`) and Training Api (`Api.h`).
- `lib/x64` – a directory containing import library (`AVLDL.lib`) for 64-bit applications. This file needs to be linked into a program that uses Library or Training Api as it serves as intermediary to `AVLDL.dll`.

Installer sets environment variable named `AVLDL_PATH5_3` containing path to `Library` subdirectory. Exemplary use of `AVLDL_PATH5_3` is presented in C++ examples distributed with Aurora Vision Deep Learning.

3. Aurora Vision Deep Learning Service

The Service is installed into `Service` subdirectory in main installation directory. After starting the Service, a new icon should be displayed in system tray.



The Service icon can be displayed in three colors, indicating the Service status:

- Red - Service is starting or an error has occurred;
- Yellow - Service is ready to accept clients;
- Green - client is connected.

Running filters does not require working Service (this is even discouraged). Training always require running Service, regardless of tool.

Please note: to open the **Deep Learning Editor**, place a relevant Deep Learning filter (DL_DetectFeatures, DL_ClassifyObject, DL_LocatePoints, DL_DetectAnomalies1, DL_DetectAnomalies2 or DL_SegmentInstances) in the Program Editor, go to its Properties and click on the icon next to **inModelId.ModelDirectory** parameter.

Alternatively, to open **Deep Learning Editor** from DL_*_Deploy filter (DL_DetectFeatures_Deploy, DL_ClassifyObject_Deploy, DL_LocatePoints_Deploy, DL_DetectAnomalies1_Deploy, DL_DetectAnomalies2_Deploy or DL_SegmentInstances_Deploy), go to its Properties and click on the icon next to **inModelDirectory** parameter.

To change the default graphics card, change its index in the **settings.xml** file in the **UsedGPUIndex** field. The file can be found in: `%APPDATA%\..\Local\Aurora Vision\Aurora Vision Deep Learning 5.3` or click on the **Open Log Directory** button in Deep Learning Service GUI.

The list of available graphics cards and their indexes is displayed in the Deep Learning Service GUI window.

4. Aurora Vision Deep Learning Examples

Examples are installed only if corresponding component has been selected during installation.

Examples for Aurora Vision Studio are installed into `%ProgramData%/Aurora Vision/Aurora Vision Studio 5.3 Professional/Examples` directory. It means that they are located next to other Studio examples, which simplifies opening them from Studio. All Deep Learning Examples are prefixed with "Deep Learning".

Examples using Aurora Vision Deep Learning Library and Training Api are located in the *Public Documents* system folder (e.g. `C:\Users\Public\Documents\Aurora Vision Deep Learning 5.3\Examples` on Windows 10). Shortcut to this directory can be found in the *Start Menu* after the installation.

5. Aurora Vision Deep Learning Standalone Editor

Standalone Deep Learning comes in both runtime and professional version installers. It can be very useful to performing training on client site.

Standalone Editor can be executed using command line with following parameters:

```
DeepLearningEditor.exe path [--alwaysOnTop] [--disableChangingLocation] [--language "language-code"]
```

Where:

- **path** - path to directory with Deep Learning model,
- **--alwaysOnTop** (optional) - setting this flag does not allow Deep Learning Editor to be covered by other windows.
- **--disableChangingLocation** (optional) - user cannot change the model path from the editor.
- **--language** (optional) - language to be used in the editor. Possible values "en","de","ja","pl","zh_hans","zh_hant".

Standalone editor can be started using additional HMI controls or **Execute** filter.

6. Logging

Deep Learning Service and Filters logs some information during execution to several files, located at `%LocalAppData%/Aurora Vision/Aurora Vision Deep Learning 5.3` directory. Total disk space used by these files should not exceed several MB. Files older than a couple of days are automatically deleted. More information are provided in the documentation of DL_ConfigureLogging filter.

If this disk space requirement is unacceptable, Service can be executed in "minimal logging" mode. This can be achieved by running `run_service_minimal_logging.bat` script, located at Service installation folder. Note, that it will not lead to any observable performance improvement.

7. Troubleshooting

Most common problems encountered by our clients can be separated into two groups:

1. **Problems with installed Nvidia drivers** - most problems occurs during loading Deep Learning Filters into Aurora Vision Studio.

   Most common error from the console log:

   ```
   Unable to load filter library "(...)AvlDlFilters.dll". Win32 error: The specified procedure could not be found.
   ```
2. **Resources exhaustion during the training** - training takes to much GPU/System memory that cannot be handled by current system. In such state your computer may lost stability and different problems my occur.

   Most common errors:

   ```
   Out of memory. Try freeing up hard disk space, using less training images, increasing downsample or resizing images to smaller ones.
   ```
   or `Service disconnected.`

Common reported problems and solutions:

1. **Invalid or old version of graphical card drivers** - verify if your GPU card have supported version of drivers. It can be checked in Window's Control Panel.
2. **Corrupted installation of the GPUs drivers** - verify if your GPU drivers are installed properly. In some cases full re-installation of Nvidia drivers may be necessary.

   Please verify if following files are found on your computer `C:\Windows\System32\nvml.dll` and `C:\Windows\System32\nvcuda.dll`.
3. **Deep Learning product version is too old for latest version of the latest Aurora Vision Studio** - Update Deep Learning to the latest version.
4. **Changes in environment PATH variable may affects how Deep Learning filters works** - Remove all paths from PATH variable which may point to Nvidia CUDA runtime DLL. Please verify if command `where cudnn_ops_infer64_8.dll` returns no results.
5. **GPU card doesn't meet minimum software requirements** - in some cases older GPU cards may encounter runtime problems during training or inference.

References

See also:

- Machine Vision Guide: Deep Learning - Deep Learning technique overview,
- Creating Deep Learning Model - how to use Deep Learning Editor.

Zebra
Aurora™ Vision

This article is valid for version 5.3.4