Zebra
Aurora™ Vision

Aurora Vision Studio 5.7

Programming Reference

Created: 11/10/2026

Product version: 5.7.5.80527

Table of content:

- Data Types
- Structures
- Arrays in Aurora Vision Studio
- Array Synchronization
- Optional Inputs
- Connections
- Conditional Execution
- Types of Filters
- Generic Filters
- Macrofilters
- Formulas
- Testing and Debugging
- Error Handling
- Offline Mode
- Summary: Common Terms that Everyone Should Understand
- Summary: Common Filters that Everyone Should Know

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:

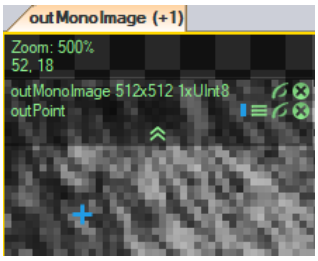
- **Integer** – Integer number in the range of approx. ±2 billions (32 bits).
- **Real** – Floating-point approximation of real numbers (32 bits)
- **Bool** – Logical value – *False* or *True*.
- **String** – Sequence of characters (text) encoded as UTF-16 unicode.

Type Constructs

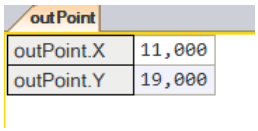
Complex types of data used in Aurora Vision Studio are built with one of the following constructs:

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.

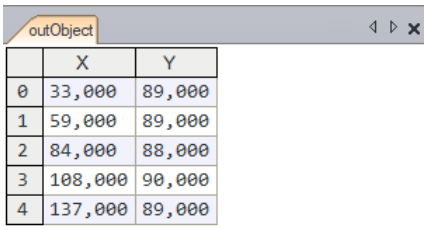
More information about using structures can be found: here.

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.



	X	Y
0	33,000	89,000
1	59,000	89,000
2	84,000	88,000
3	108,000	90,000
4	137,000	89,000

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.

Example filters having arrays on inputs:

- Average – calculates the average of multiple real numbers.
- FitLineToPoints_M – creates a line best matching a set of two-dimensional points.

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.

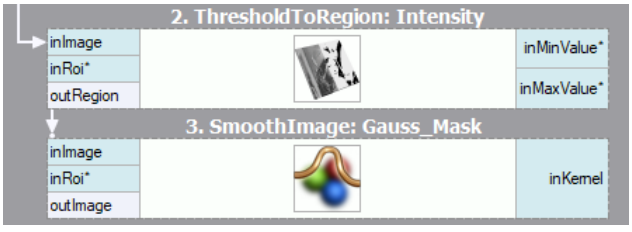
Angles

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++

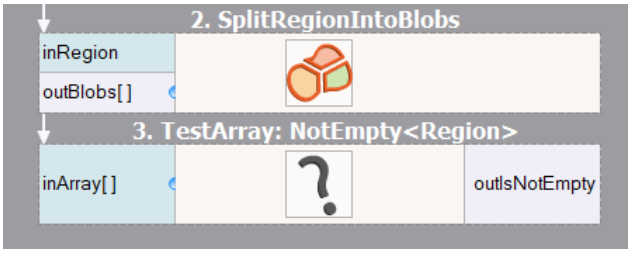
Arrays are very similar to the *std::vector* class template in C++. As one can have *std::vector < std::vector< atl::Point2D > >* in C++, there is also Point2DArrayArray in Aurora Vision Studio.

Generic Filters for Arrays

Arrays can be processed using a wide range of generic filters available in the categories:

- Array Basics
- Array Composition
- Array Set Operators
- Array Statistics
- Array Transforms

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.

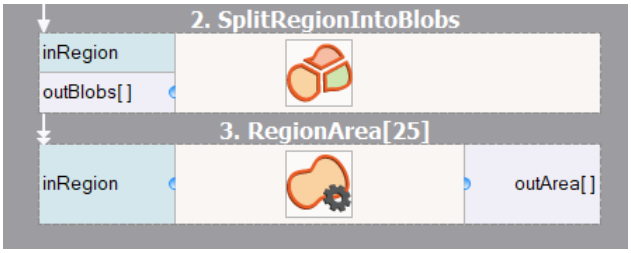
Singleton Connections

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:

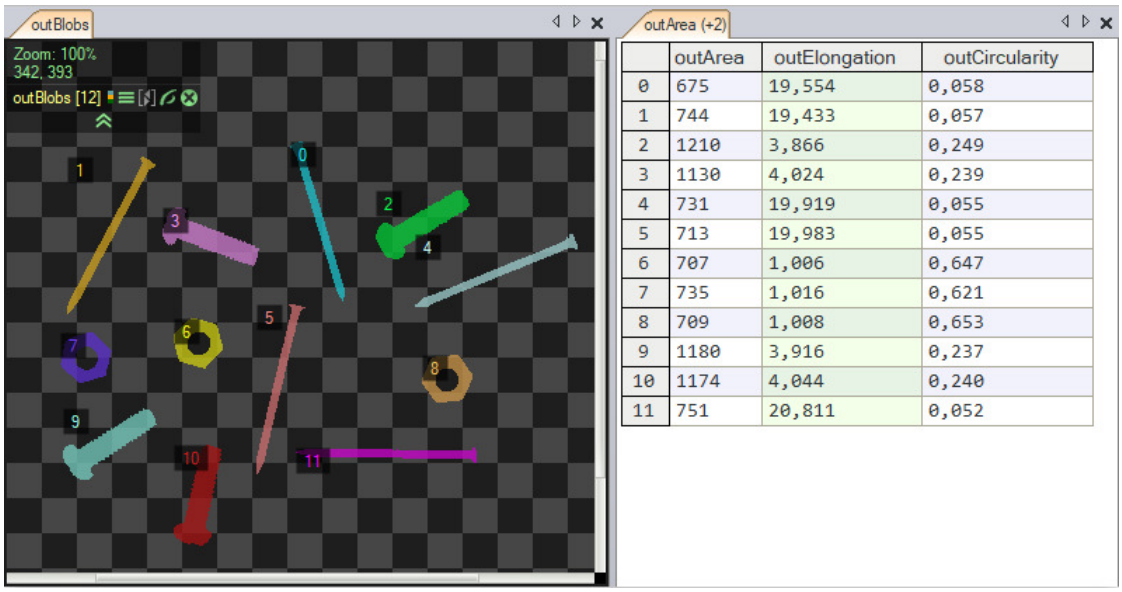
- If the input array connected to a scalar input is empty, then the filter will not be executed at all and empty arrays will be produced on the outputs.
- If there are multiple array connections at a single filter instance, then all of the input arrays must be of the same length. Otherwise a domain error 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.

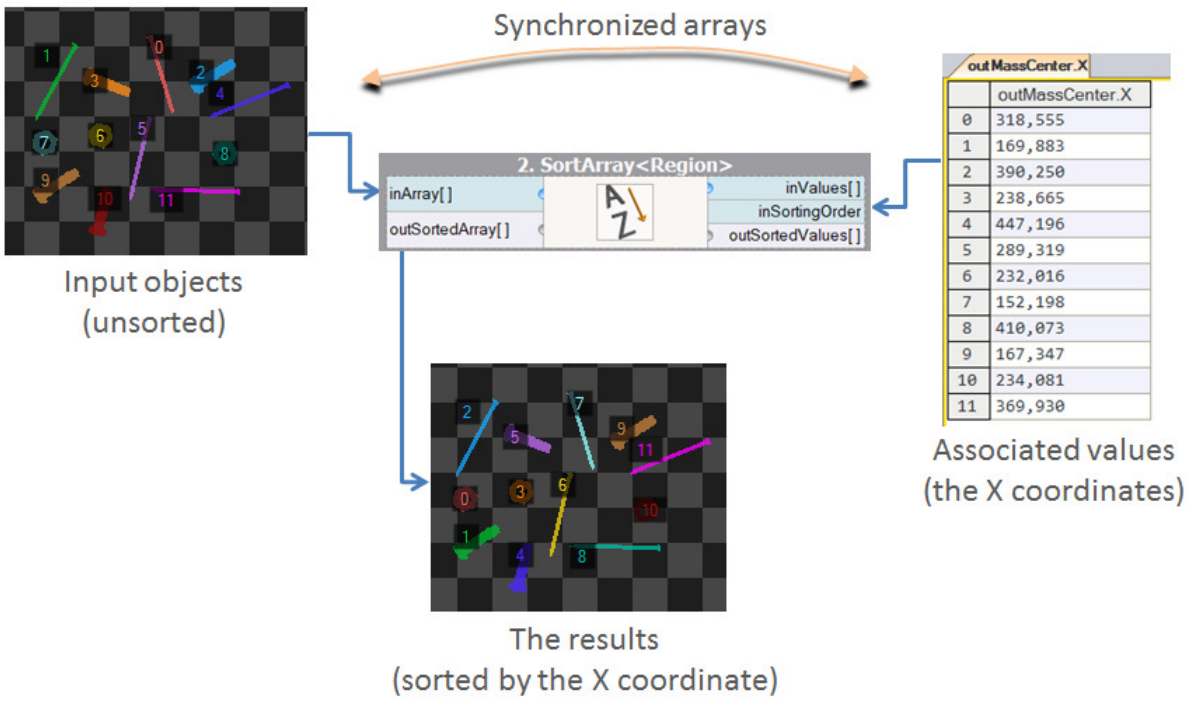
Synchronized Inputs

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:



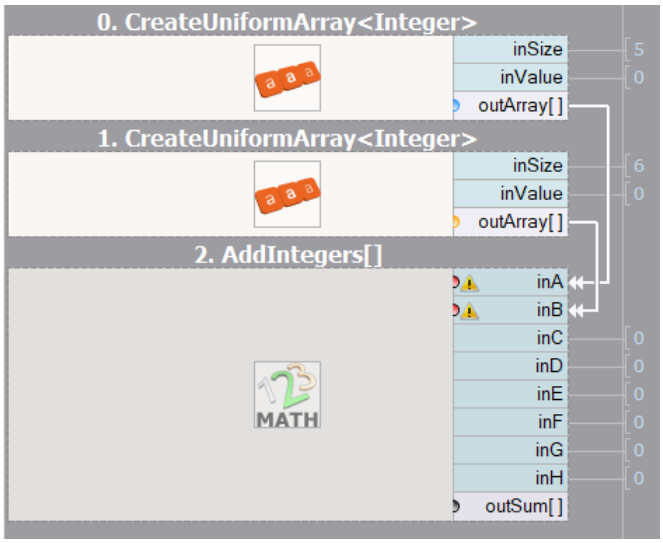
	outMassCenter.X
0	318,555
1	169,883
2	390,250
3	238,665
4	447,196
5	289,319
6	232,016
7	152,198
8	410,073
9	167,347
10	234,081
11	369,930

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:

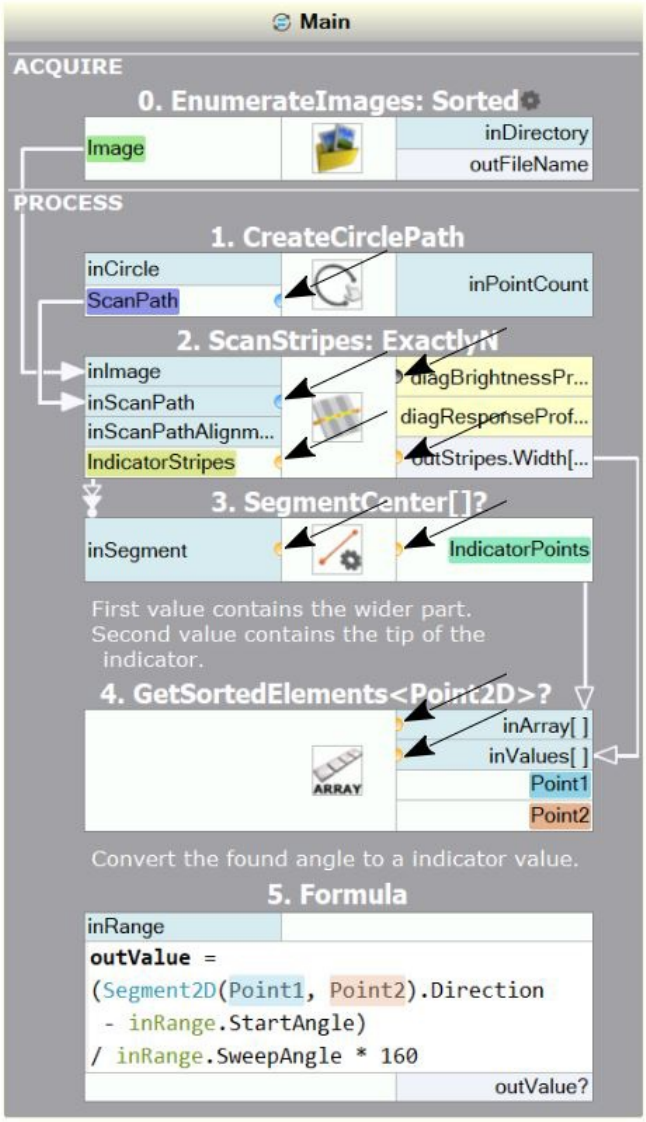


Visualization of an array synchronization problem.

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.

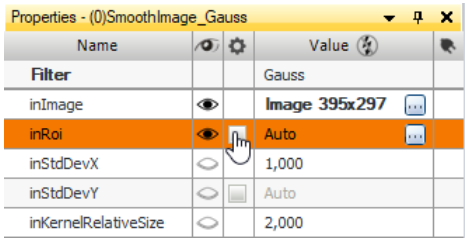
Note: Array synchronization analysis and its visualization is turned on by default, but this can be turned off in the settings of Aurora Vision Studio.

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? → T** are conditional, whereas *T? → 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

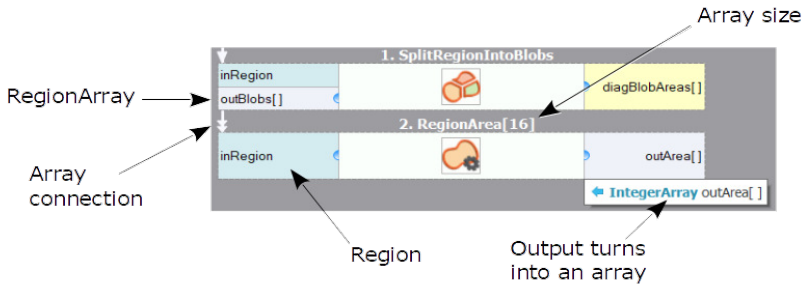
This is probably the most complicated part of the Aurora Vision programming model as it concentrates most of its computing flexibility. You will never define any connection logic explicitly as this is done by the application automatically, but still you will need to think a lot about it.

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.

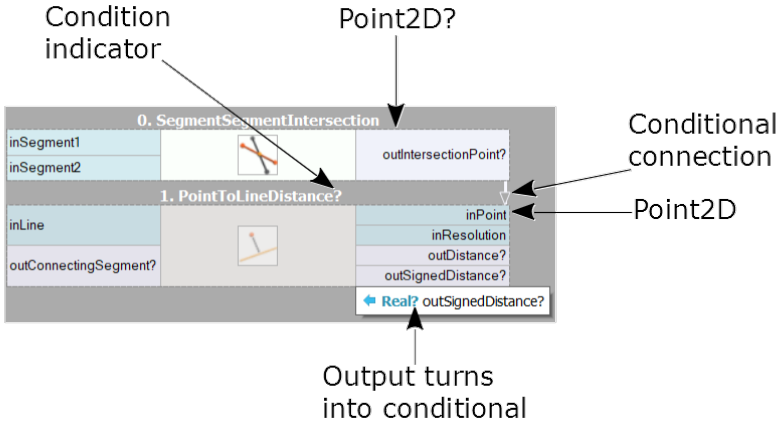
3. **Array Connection**
   TArray ⇉ *T*
   This logic creates an implicit loop on a connection from an array to a scalar type, e.g. from RegionArray to Region. All the individual results are then merged into arrays and so the outputs of the second filter turn into arrays as well (see also: Arrays).



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

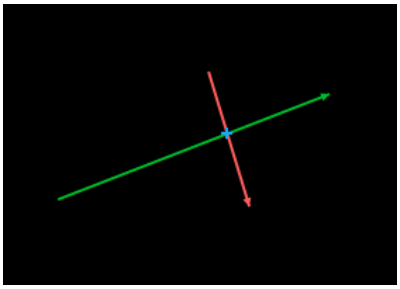
Conditional execution consists in executing a part of a program, or not, depending on whether some condition is met. There are two possible ways to achieve this in Aurora Vision Studio:

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.

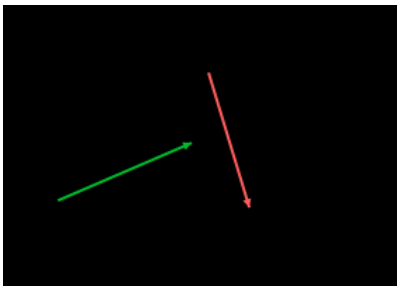
This section is devoted to the former, conditional connections.

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.

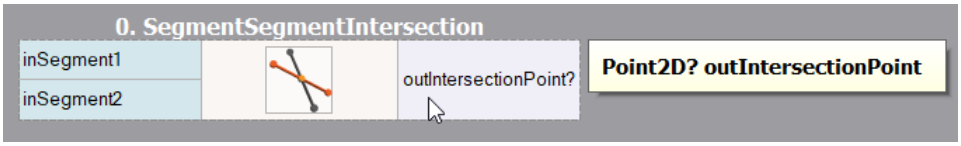


There is no intersection – the filter returns Nil value.

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.

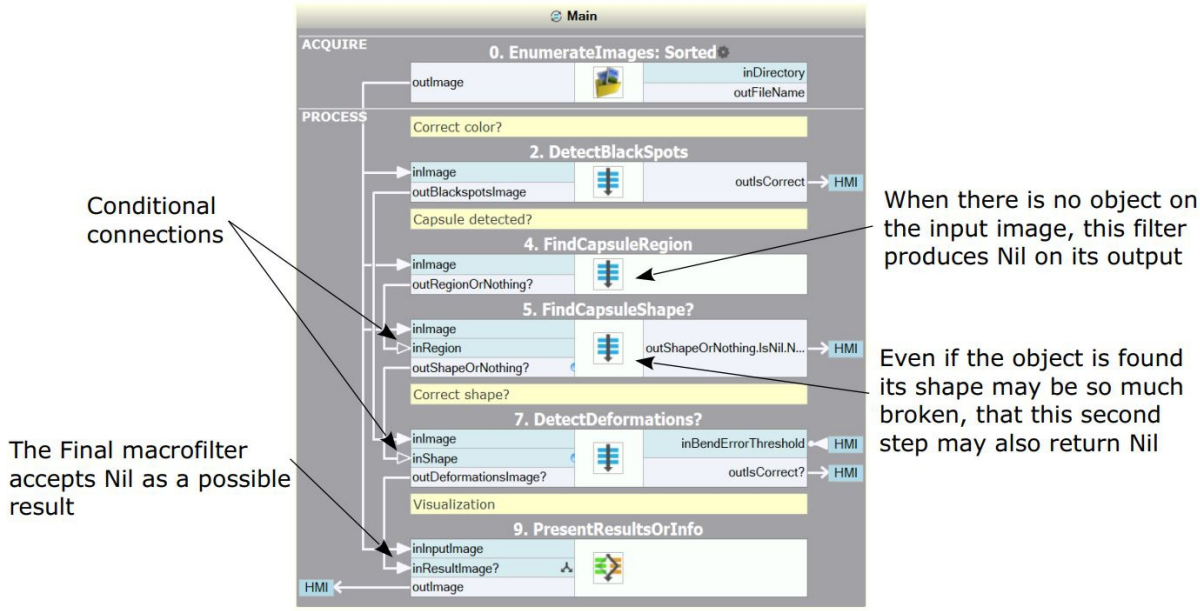
Resolving *Nil* Values

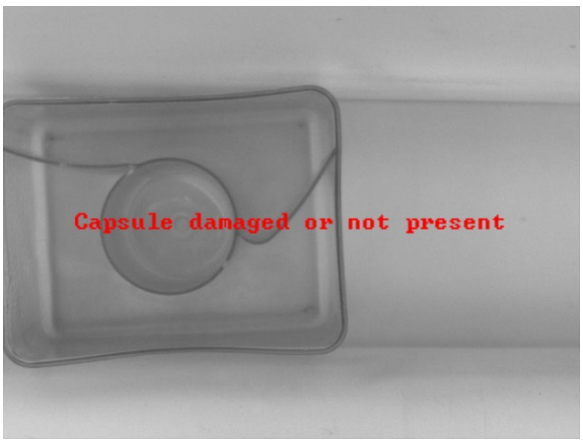
In general, if we have a conditional execution, a *Nil* value has to be anticipated, and it has to be resolved at some point. Here is a list of the most typical solutions:

- 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 *Nils* 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.

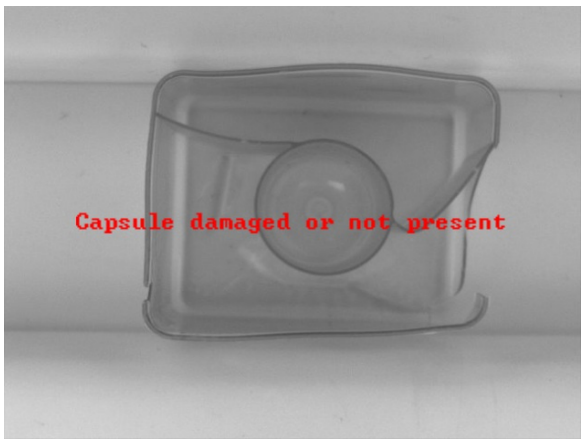
Example

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

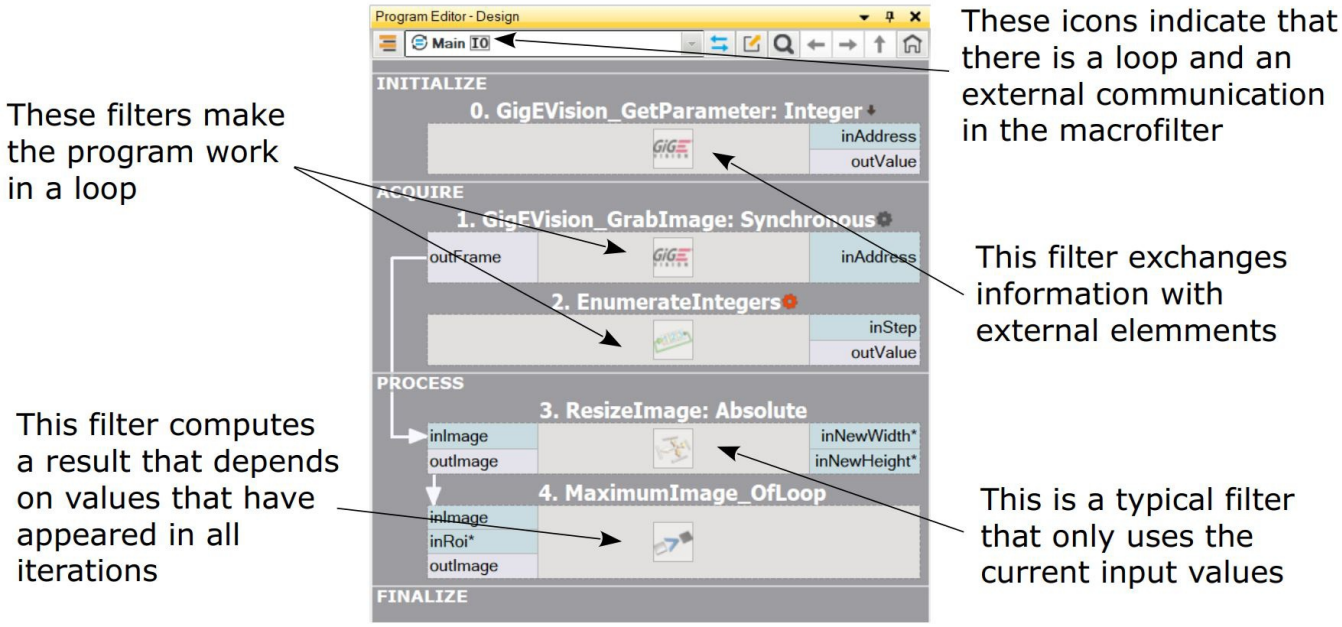
Read before: Introduction to Data Flow Programming.

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.

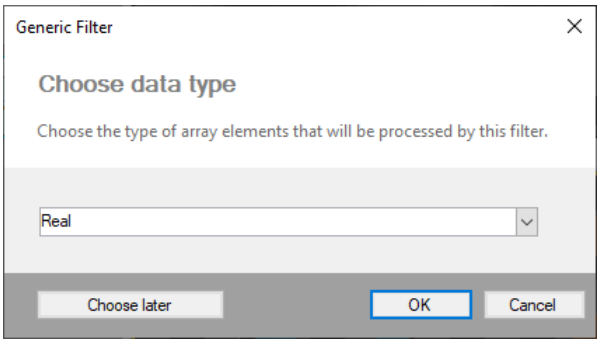
Type Definitions

Each generic filter is defined using a type variable *T* that represents any valid type. For instance, the GetArrayElement filter has the following ports:

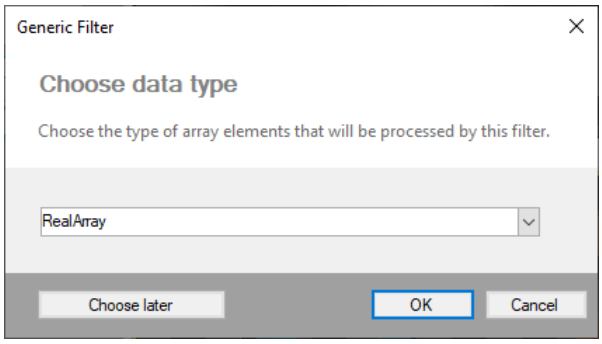
	Port	Type	Description
→	inArray	**T**Array	Input array
→	inIndex	Integer	Index within the array
←	outValue	**T**	Element from the array

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.

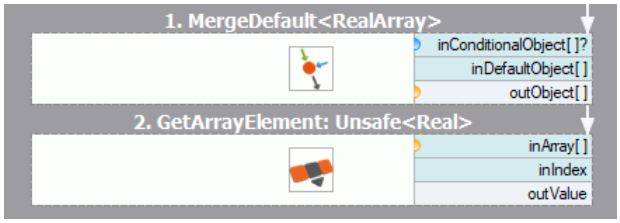


This window appears for filters not related to arrays, e.g. MergeDefault. The user has to choose the type of the entire object.

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.

Table of Contents

- Macrofilter Structures
- Steps
- Variant Steps
- Tasks
- Worker Tasks
- Queues
- Macrofilters Ports
- Sequence of Filter Execution

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.

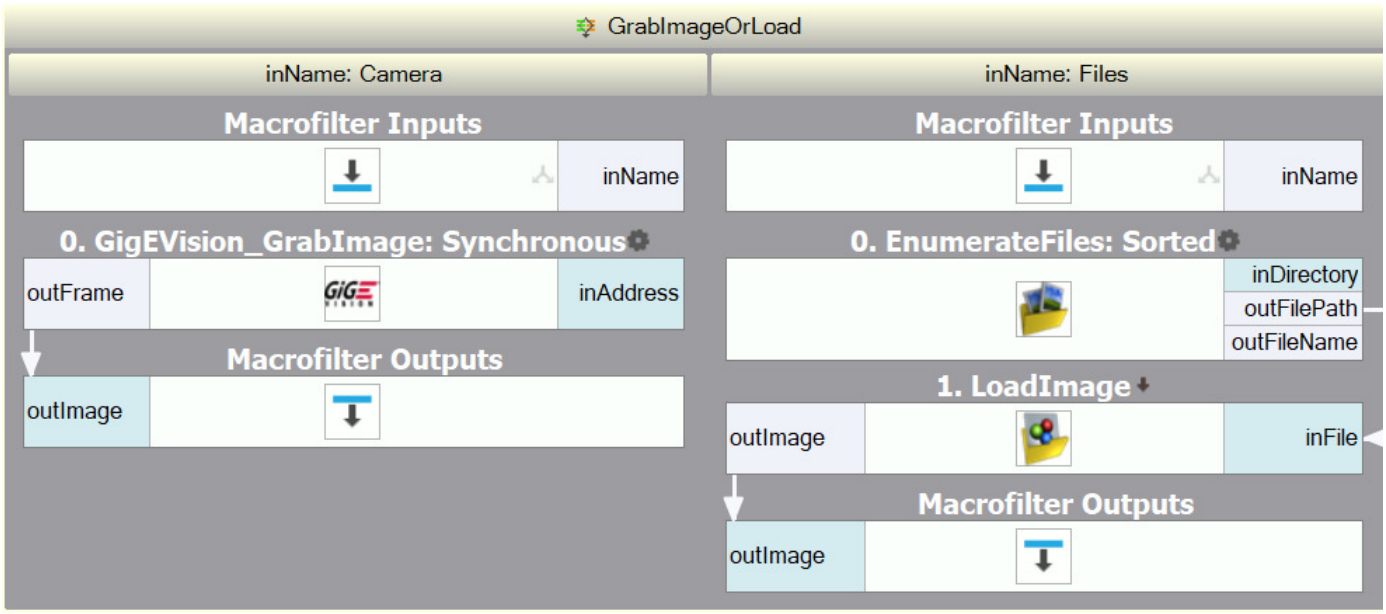
c++

Variant Step macrofilters correspond to the *switch* statement in C++.

Example 1

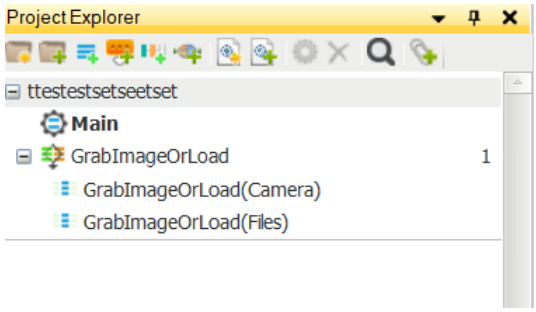
A *Variant Step* can be used to create a subprogram encapsulating image acquisition that will have two options controlled with a String-typed input:

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.

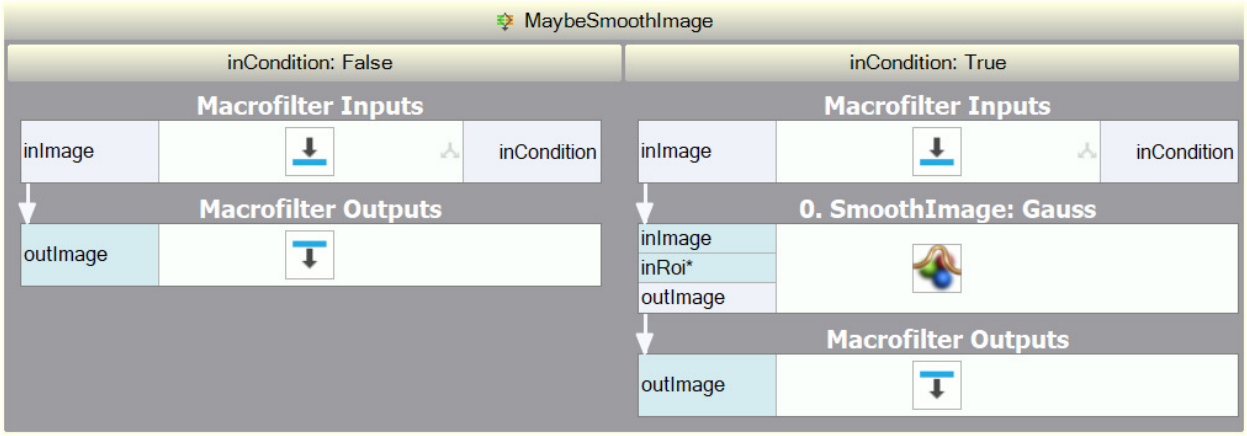
In the Project Explorer, the variants will be accessible after expanding the macrofilter item:



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).

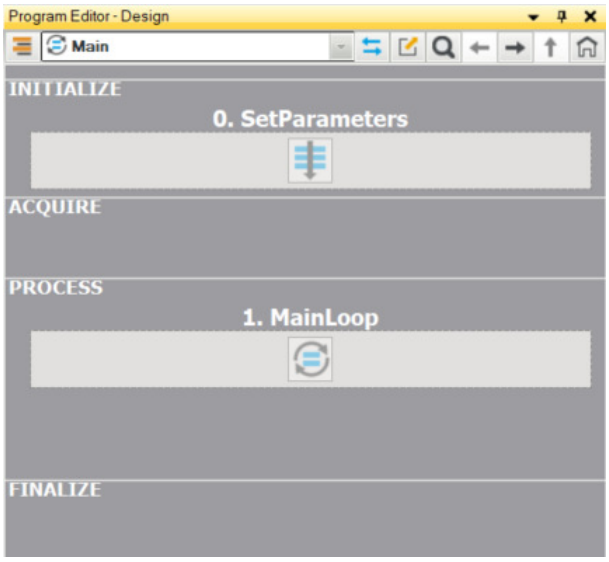
What is also very important, *Tasks* define the notion of an *iteration* of the program. During program execution there is always a single task that is the most nested and currently being executed. We call it *the current* task. When the execution process comes to the end of this task, we say that an 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.

Every program will contain at least one *Worker Task*, which will act as "Main". To run programs with more than one *Worker Task* it is necessary to own also a Parallel Add-on license.

If there are multiple Worker Tasks in application that use data analysis tools, then it is necessary to have some multi-threaded runtime license to deploy it. Tools belonging to modules others than following ones which can be used without restrictions: FoundationLite, Genicam, ThirdParty and OpenCV. List of all macrofilters using data analysis tools can be check on Licensing tab of Program Statistics tool in Studio.

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.

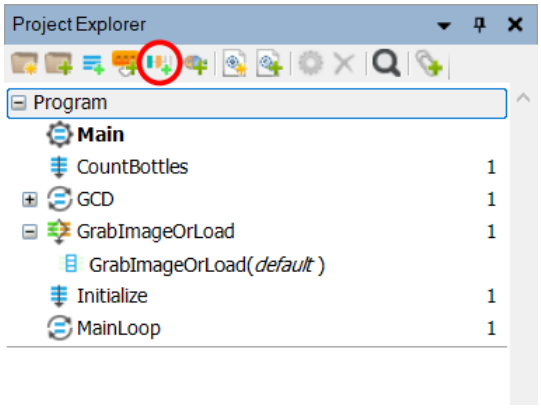
Please note, that different arrays are not synchronized with each other. To process complex data or pass data that needs to be synchronized (like image and its information) you should use user types. We also highly advise not to use queues as an replacement of global parameters.

Creating Queues

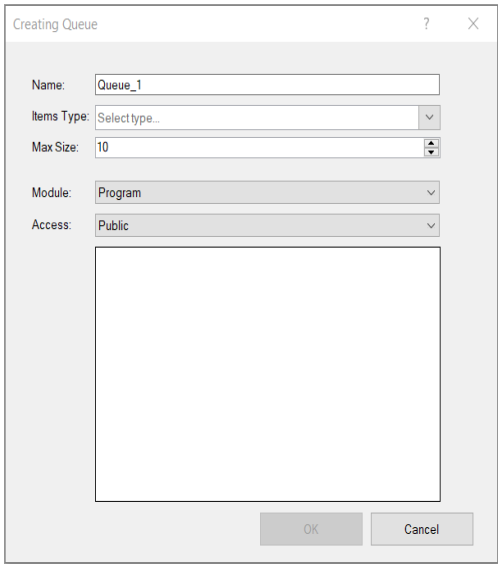
To create a new queue in the Project Explorer click the *Create New Queue...* icon. A new window will appear, allowing you to select the name and parameters of the new queue such as:

- Items Type that specifies the type of data passed by the queue
- Maximum size of the queue
- Module to which it belongs

- Access – public or private



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.

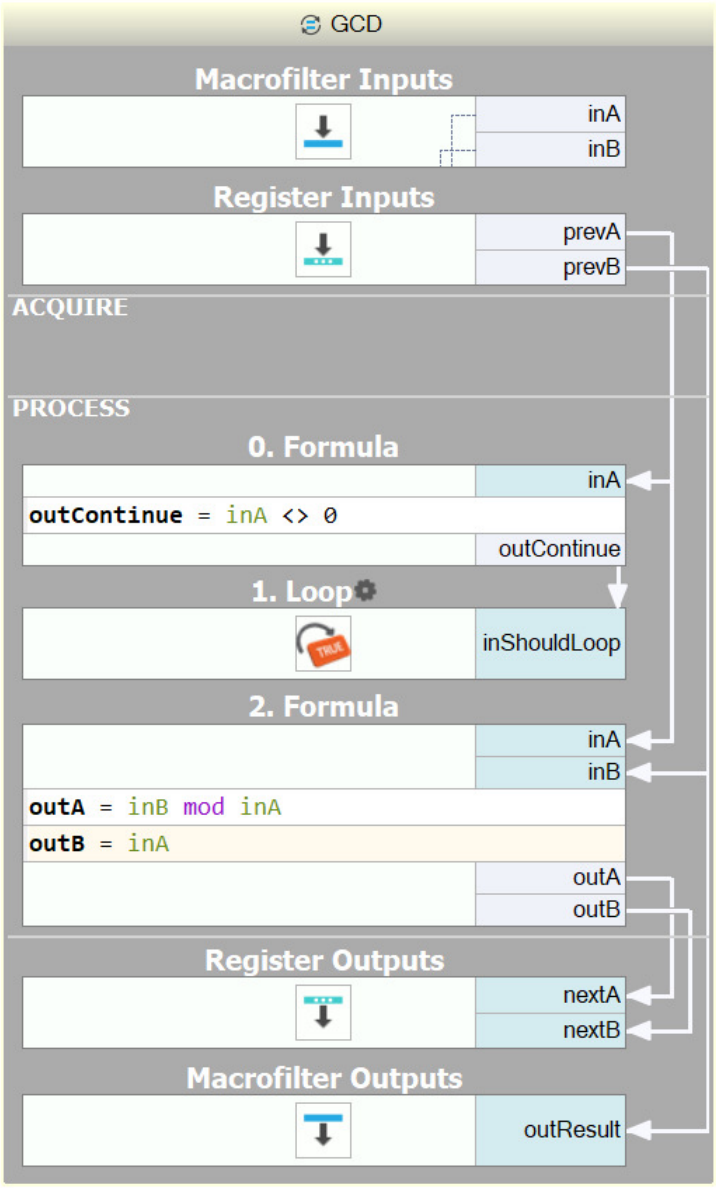
Yet another alternative to macrofilter registers is the *prev* operator in the formula blocks.

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
- Data types
- Literal constants
- Predefined constant values
  - Enumeration constants
- Arithmetic conversions
- Automatic implicit conversions
- Operators
  - Unary operators
  - Binary operators
  - Special operators
  - Operator precedence
- Operator processing modes
  - Conditional processing
  - Array processing
  - Array and conditional processing combination
- Functions
  - Functions list
  - Mathematical functions
  - Conversion functions
  - Statistic and array processing functions
  - Geometry processing functions
  - Functions of type String
  - Structure constructors
  - Typed Nil constructors
- Examples

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:

$$\begin{bmatrix} i_{1,1} & i_{1,2} \\ i_{2,1} & i_{2,2} \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' & y' \end{bmatrix}$$

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:

$$\begin{bmatrix} i_{1,1} & i_{1,2} & i_{1,3} \\ i_{2,1} & i_{2,2} & i_{2,3} \\ i_{3,1} & i_{3,2} & i_{3,3} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' & y' & w' \end{bmatrix} \rightarrow \begin{bmatrix} \frac{x'}{w'} & \frac{y'}{w'} \end{bmatrix}$$

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. Negative values indicate to read from the end of the array, path, matrix row or column (e.g. *arr*[-1] reads the last element of the array, *arr*[-2] reads the second to last element of the array, and so on). If an indexer returns a value pointing out of -ArrayLength...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
atan2
clamp
lerp

Conversion functions

integer
real
long
double
toString
parse*Value*
tryParse*Value*

Statistic and array processing functions

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
group
crop
trimEnd
rotate
pick
sequence
array
createArray
join

Geometry processing functions

angleNorm
angleTurn
angleDiff

distance
area
dot
normalize
createVector
createSegment
createLine
translate
toward
scale
rotate

Complex object creation

Matrix
identityMatrix
Path

Functions of type String

Substring
Trim
ToLower
ToUpper
Replace
StartsWith
EndsWith
Contains
Find
FindLast
IsEmpty

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) = 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) = $\sqrt{a^2 + b^2}$

atan2

```
Real atan2( Real y, Real x )
Double atan2( Double y, Double x )
```

Calculates the angle of the coordinates vector by computing the arc tangent of y / x using the signs of the arguments to determine the quadrant of the vector. Returns the angle in degrees in the range [-180...+180] (use the angleNorm function to change the range).

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 )
Point3D lerp( Point3D a, Point3D b, Real lambda )
Vector2D lerp( Vector2D a, Vector2D b, Real lambda )
Vector3D lerp( Vector3D a, Vector3D b, Real lambda )
```

Computes a linear interpolation between two numeric values, two geometric points or two geometric vectors. 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).

When interpolating between data types of Vector2D or Vector3D, the resulting vector is computed by linearly interpolating each of its coordinates independently, not by rotating the vector. This means that the length of the vector is not maintained along the interpolation range and that the angle of the vector is not interpolated linearly.

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: $^{1}/_{n}\Sigma(\bar{x} - x_i)^2$. 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.

group

```
<T>ArrayArray group( <T>Array array, IntegerArray values )
```

Groups elements of an array into sub-arrays according to associated integer keys.

Both arguments must have the same number of elements. Elements that share the same key value are collected into the same sub-array. The output is an array of arrays where each sub-array corresponds to one distinct key value, ordered by first occurrence of that key in the *values* array. Elements within each sub-array preserve their original relative order.

Example: group([a, b, c, d], [1, 2, 1, 3]) returns [[a, c], [b], [d]].

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:

- **inA** of type **Integer**
- **inB** of type **Integer**

Filter outputs:

- **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:

- **inA** of type **Integer**
- **inB** of type **Integer**
- **inPos** of type **Real**

Filter outputs:

- **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:

- **inBox** of type **Box**

Filter outputs:

- **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:

- **inBox** of type **Box**
- **inFrame** of type **Integer**

Filter outputs:

- **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:

- **inA** of type **Real**
- **inB** of type **Real?**
- **inC** of type **Integer**

Filter outputs:

- **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:

- **inA** of type **Real**
- **inB** of type **Real?**
- **inC** of type **Integer**

Filter outputs:

- **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:

- **inElements** of type **IntegerArray**

Filter outputs:

- **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:

- **inParameter** of type **Real**

Filter outputs:

- **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:

- **inReverse** of type **Bool**

Filter outputs:

- **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:

- **inValue** of type **Integer**

Filter outputs:

- **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.

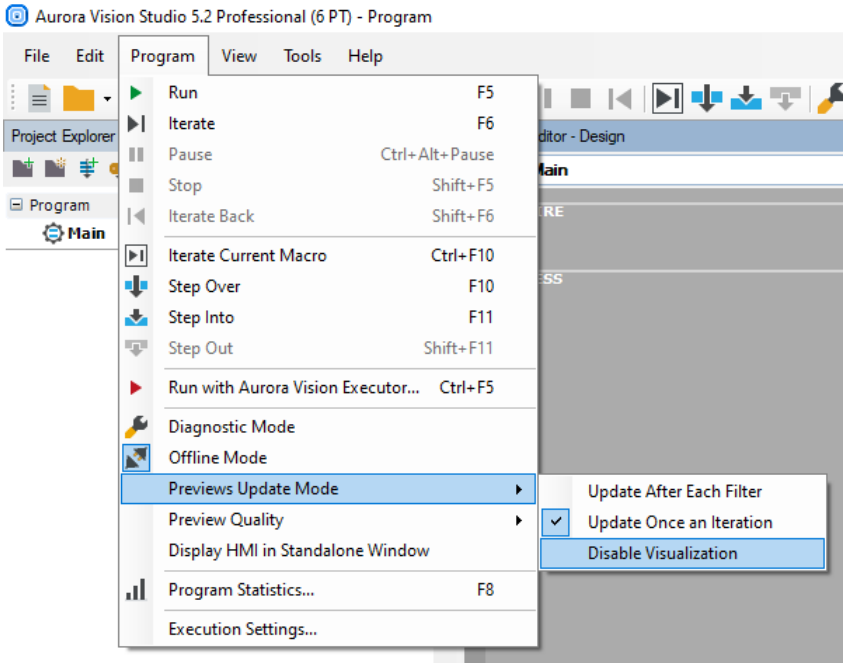
Those features are enabled by default, which makes Aurora Vision Studio behave different than Aurora Vision Runtime. Differences can affect both performance and program flow. As such, those settings are not recommended when testing how the final version works.

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.

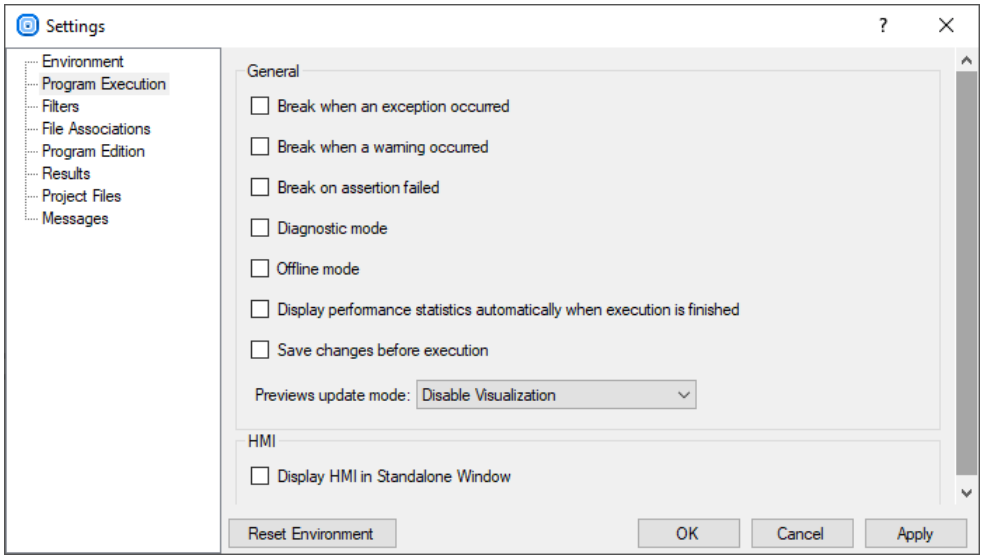
Disabling the *Diagnostic Mode* along with previews makes Studio execute programs nearly as fast as Runtime.

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
- 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 in which it was launched:

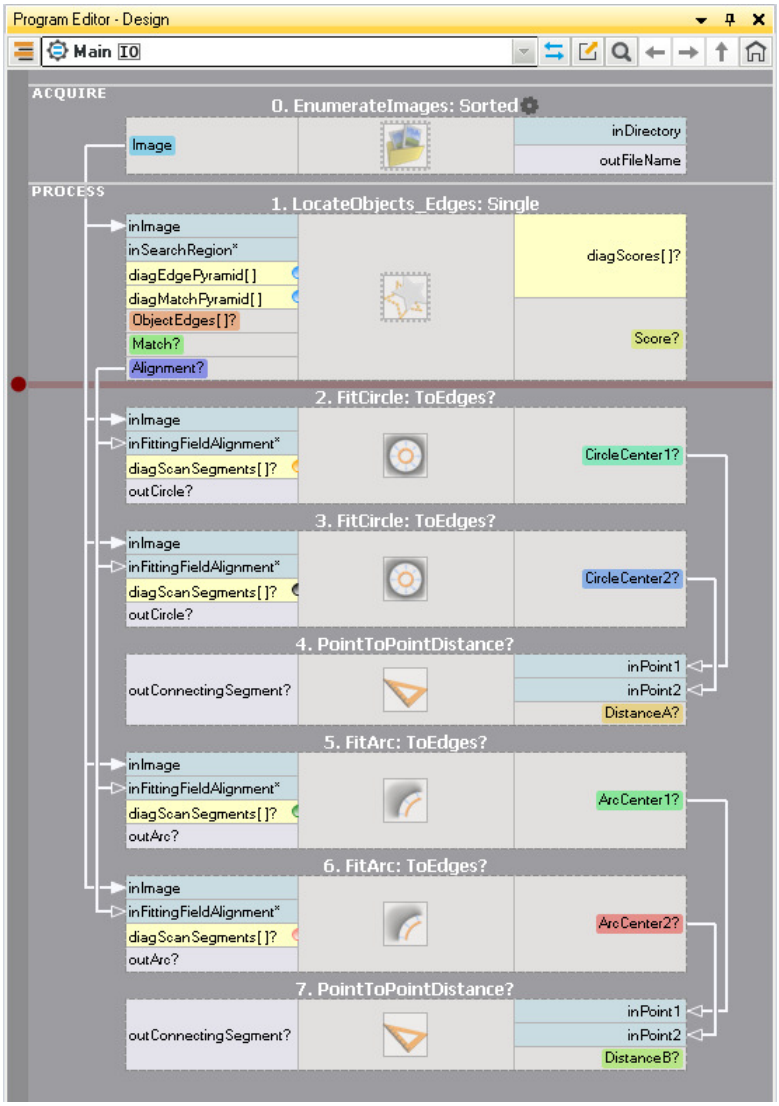
- Breakpoints,
- Ending of the tracked Worker Task,
- Failed Assertions (can be changed in settings),
- Exception (can be changed in settings),
- Warnings (can be changed in settings).

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 saved in the project and they will still be present after loading the program again. To remove them it is necessary to select *Edit » Remove All Breakpoints* option.
- 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 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.

HMI events can also be debugged. Breakpoints can be placed inside event handling macrofilters and the 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

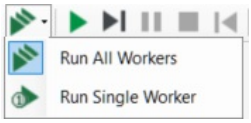
To run in Studio programs with more than one *Worker Task* it is necessary to own also a Parallel Add-on license. If there are multiple Worker Tasks in application that use data analysis tools, then it is necessary to have some multi-threaded runtime license to deploy it. Tools belonging to modules others than following ones which can be used without restrictions: FoundationLite, Genicam, ThirdParty and OpenCV. List of all macrofilters using data analysis tools can be check on Licensing tab of Program Statistics tool in Studio.

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.

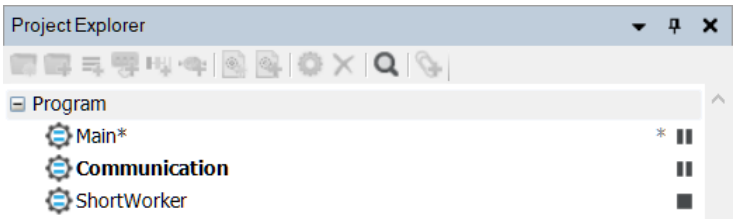


It is only possible to switch it before the program is started. When the program is paused, all Worker Tasks are paused.

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 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.
- 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.
- 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
  - EnumerateFiles
  - EnumerateImages

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).

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).
- **LICENSE_ERROR**
  An error caused by lack of a license required to use the library (AVL).
- **ANY_ERROR**
  Includes all of the above kinds as well as unspecified runtime errors arising during program execution.

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.

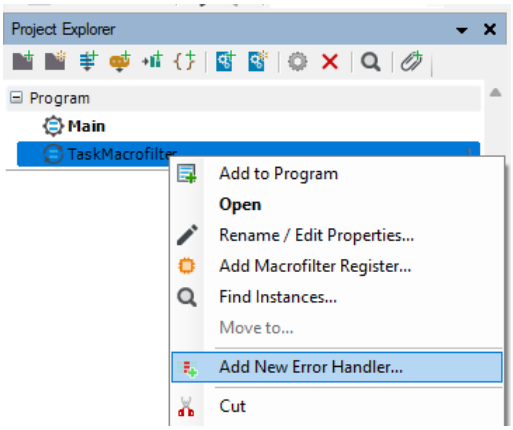
Each Task Macrofilter can have multiple handlers, for different error kinds. If an error occurs, first a handler for the specific kind is checked, and if there is no such handler, the ANY_ERROR handler will be used (if it is defined).

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.

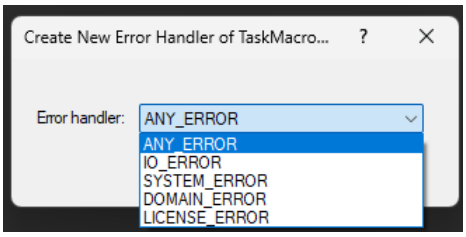
Error handlers for ANY_ERROR should be treated as a way to inform the user about the problem (e.g. write to console, save text logs) before quitting the application.

Adding Error Handlers

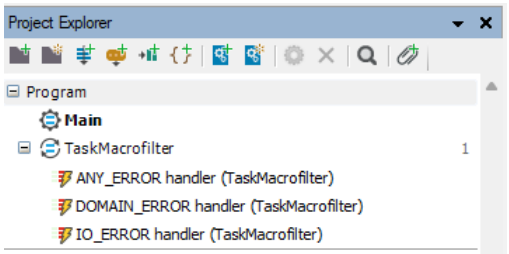
Error handlers can be added by right-clicking a Task Macrofilter in the Project Explorer and choosing "Add New Error Handler ...".



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:

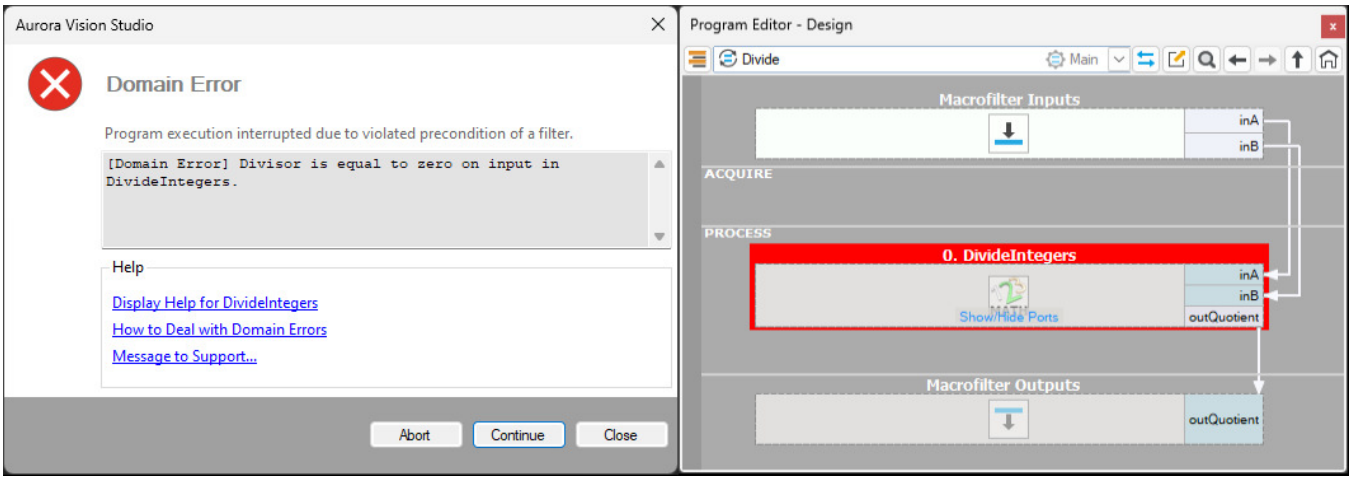


After double-clicking the error handler can be edited.

Program Execution

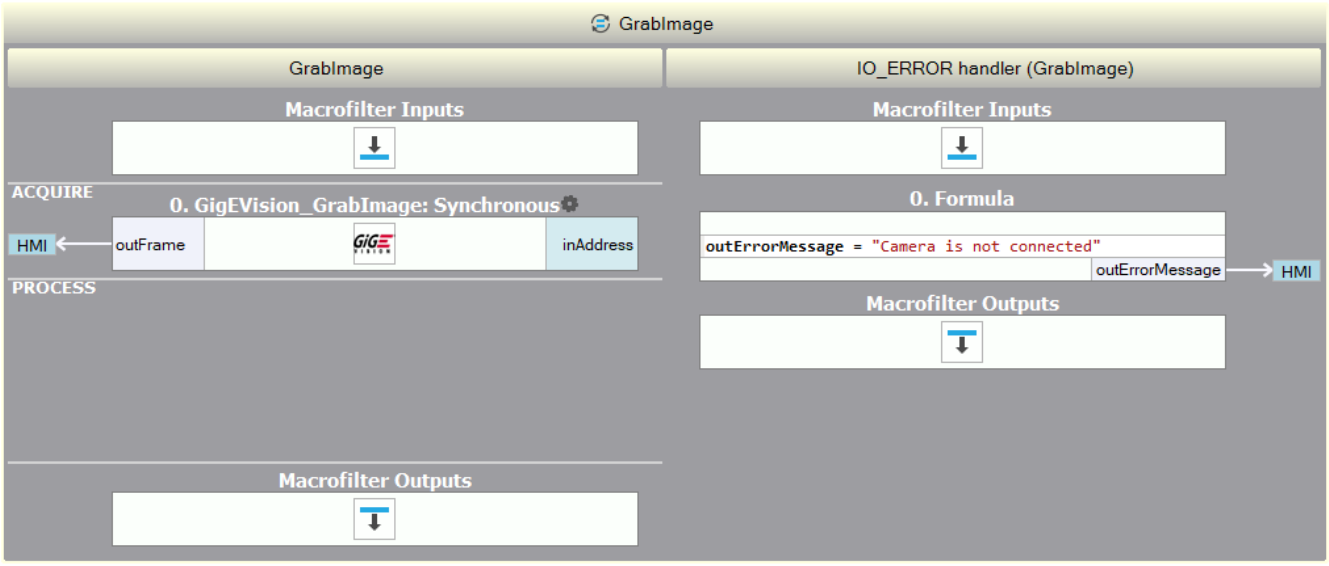
During program execution in the IDE, after an error occurs, there will be a message pop-up, with the option to continue (that is, run the error handler). This message can be turned off in the IDE settings, by changing: *Tools » Settings » Program Execution* : "Break when exception occurred".

In the Runtime environment, understandably, there will be no message and the error handling will be executed immediately.

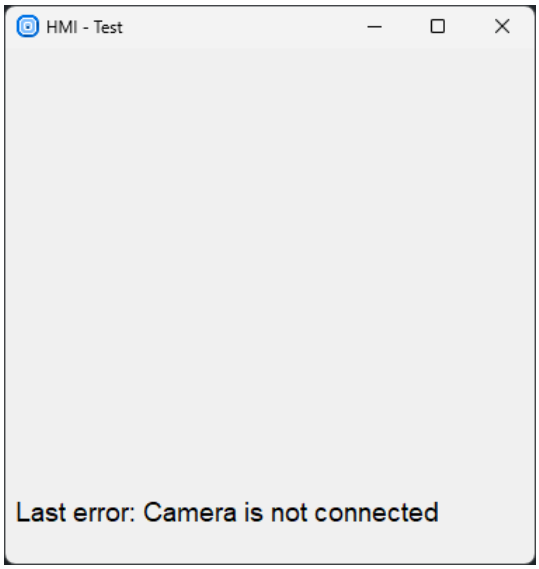


Example

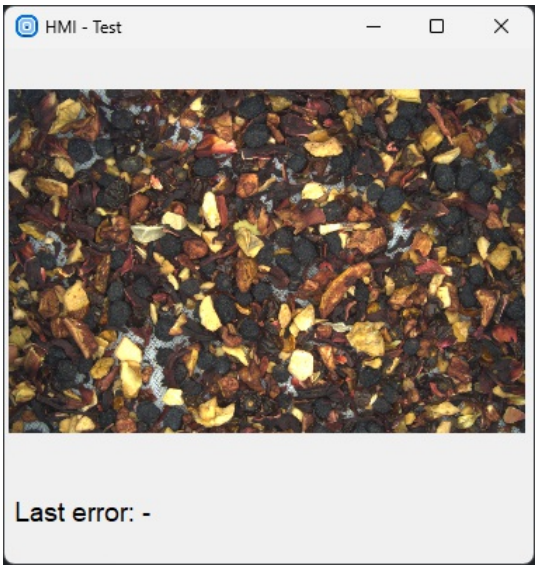
Below is a simple example, acquiring images from a camera in a loop. If the camera is not connected, we enter the error handler, where a message is returned until the error is no longer being raised. Without error handling the program would stop after the first failed image acquisition:



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. Automatic Conversions
2. Singleton Connections
3. Array Connections
4. Conditional Data
5. Conditional Connections
6. Other Alternatives to Conditional Execution
7. Instantiation
8. Macrofilter Structures
9. Steps
10. Variant Steps
    1. Example 1
    2. Example 2
11. Tasks
    1. Execution Process
    2. Example: Initial Computations before the Main Loop
12. Worker Tasks
13. Queues
14. Macrofilters Ports
    1. Inputs
    2. Outputs
    3. Registers
    4. Example: Computing Greatest Common Denominator
15. Sequence of Filter Execution
16. Execution and Performance
    1. Performance-Affecting Settings
    2. Execution Flow
17. Operation Mode
18. Execution Pausing
19. Breakpoints
20. Single-Threaded Debugging and Testing
21. Multi-Threaded Debugging and Testing
22. Program ComboBox
23. Iterating Program
    1. Iterate Program
    2. Iterate Current Macro
    3. Iterate Back
    4. Step Buttons
24. Introduction
25. Workflow Example
26. Online-Only Filters
27. Accessing the Offline Data
    1. Binding Online-Only Filter Outputs
    2. The ReadFilmstrip Filter
28. Offline Data Structure
29. Workspace and Dataset Assignment
30. Modifying the Offline Data
    1. Structural Modifications
    2. Content Modifications

31. Activation and Appearance
    1. Main Window
    2. Program Editor
32. See Also

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.

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

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.

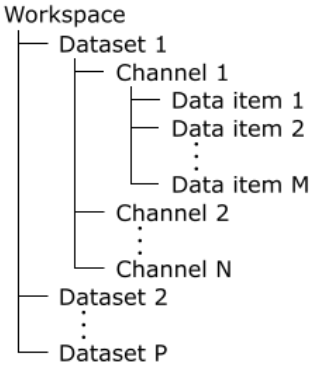
Just like any other filter, the ReadFilmstrip can be inserted into the program with the standard filter lookup and insert facilities like dragging the filter from the Toolbox. Apart from that, the ReadFilmstrip filter can be inserted by drag-and-drop the channel from the Filmstrip control onto the empty 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.

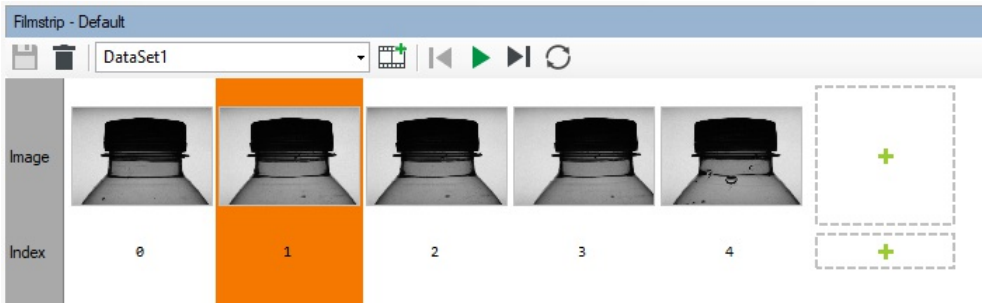
Offline Data Structure

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.:

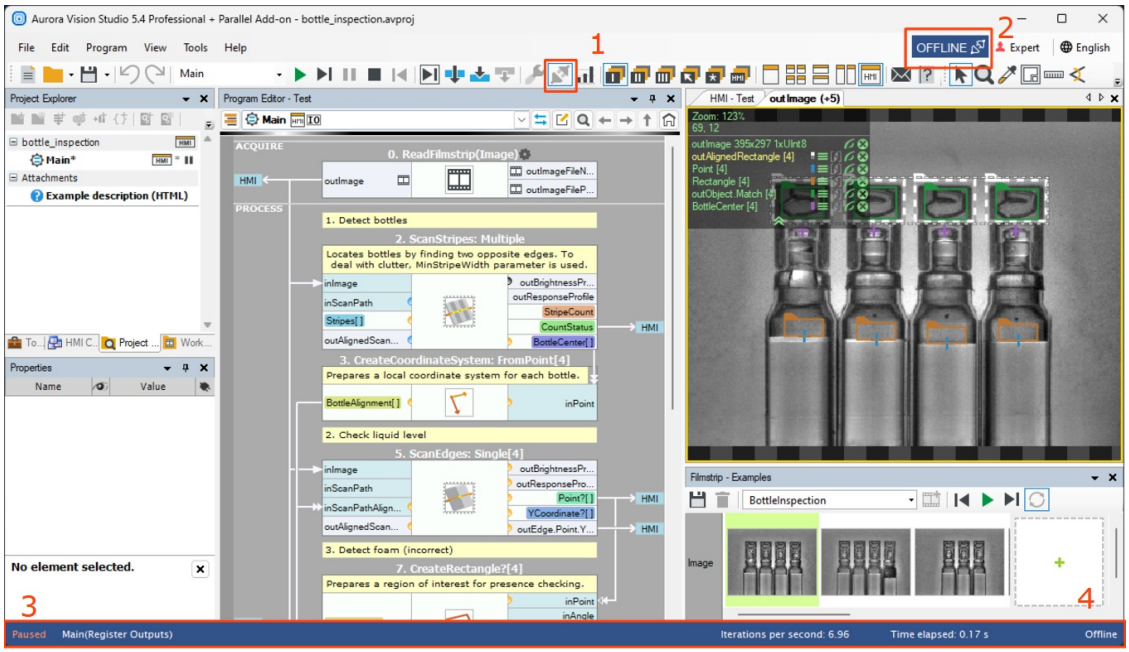


Editing the channel item.

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.

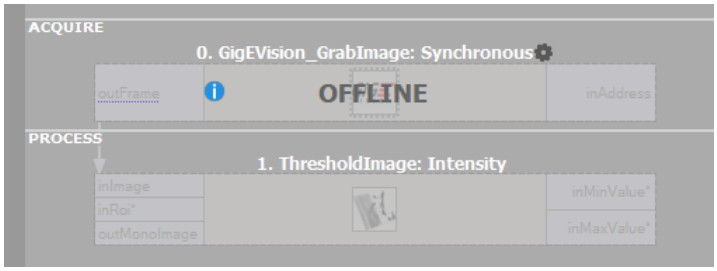
1. The Offline mode button
   
   at the main toolbar. If selected, the Offline mode is active,
2. The button at the top-right corner which visually represents the Mode and Execution:

	Offline Mode	Online Mode
Idle	OFFLINE	ONLINE
Executing	OFFLINE	ONLINE

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 an additional *Offline* label at the bottom-right corner of 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.

Zebra
Aurora™ Vision

This article is valid for version 5.7.5

Legal Terms of Use Privacy Policy

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