Zebra
Aurora™ Vision

Aurora Vision Studio 5.3

Technical Issues

Created: 6/8/2023

Product version: 5.3.4.94078

Table of content:

- Using TCP/IP Communication
- General Image Acquisition
- Working with Gen TL devices
- Working with National Instruments devices
- Working with Modbus TCP Devices
- Project Files
- C++ Code Generator
- .NET Macrofilter Interface Generator
- Remote USB License upgrade
- Working with Hilscher Devices

Using TCP/IP Communication

Introduction

Aurora Vision Studio has a set of filters for communication over the TCP/IP protocol. They are located in the Program I\O category of the Toolbox.

TCP/IP is actually a *protocol stack*, where TCP is on top of IP. These protocols are universally used from local to wide area networks and are fundamental to the communication over the Internet. They also constitute a basis for other protocols, like popular HTTP, FTP, as well as industrial standards Modbus TCP/IP or Modbus RTU/IP.

The TCP/IP protocol is a transport level communication protocol. It provides a bi-directional raw data stream, maintains a connection link and ensures data integrity (by keeping data in order and attempting to retransmit lost network packets). However raw TCP/IP protocol is usually not enough to implement safe and stable long term communication for a concrete situation and an additional communication protocol designed for a specific system must be formed on top of it. For example, when implementing a master-slave commands system with correct command execution check a command and acknowledge transaction must be used (first sending a command, than receiving with a timeout a command acknowledge where the timeout expiration informs that the command was not processed correctly). Single send operation is not able to detect whether data was properly received and processed by a receiver, because it is only putting data into a transmission buffer. Other situations may require different approach and usually required protocol will be imposed by a system on the other side of the connection. Because of the above basic TCP/IP and communication protocols knowledge is required to properly implement a system based on TCP/IP.

The communication is made possible with the use of *Sockets*. A network socket is an abstract notion of a bidirectional endpoint of a connection. A socket can be written to or read from. Data written to a socket on one end of the connection can be read from the opposite end. The mechanism of sockets is present in many programming languages, and is also the tool of choice for general network programming in Aurora Vision Studio.

Establishing a Connection

To communicate using the TCP/IP protocol stack, first a connection needs to be established. There are two ways of doing this: (1) starting a connection to a server and (2) accepting connections from clients. The sockets resulting from both of the methods are later indistinguishable - they behave the same: as a bidirectional communication utility.

A connection, once created, is accessed through its socket. The returned socket must be connected to all the following filters, which will use it for writing and reading data, and disconnecting.

Usually, a connection is created before the application enters its main loop and it remains valid through multiple iterations of the process. It becomes invalid when it is explicitly closed or when an I/O error occurs.

Connects to a specific port on a remote host.
The party to perform this operation is the client.

Opens a local TCP port and waits for incoming connections.
The party to perform this operation is the server.

Writing Data to Sockets

The filters for sending data take a SocketId value, which identifies an open connection and data, which is to be transferred to the other endpoint. The data can be of three different kinds: textual, binary or serialized objects.

The write operation is usually fast, but it can become problematic, when the other party is too slow in consuming the data, or if we attempt to write data faster than the available network throughput. The write operation is limited to putting data into a transmission buffer, without waiting for data delivery or receive confirmation. When data are being written faster than they can be delivered or processed by a receiver the amount of data held in a transmission buffer will start growing, causing significant delay in communication, and eventually the transmission buffer will overflow causing the write operation to rise and error.

Writes plain text in UTF-8 encoding to a socket.
An optional suffix* can be appended to the text.

Writes arbitrary binary data, given as a ByteBuffer, to a socket.

Writes a serialized object to a socket. The resulting data can only be read in another instance of Aurora Vision Studio/Executor with the TcpIp_ReadObject filter described below, and with using the same instantiation type.

Reading Data from Sockets

Reading data requires an open connection and can return the received data in either of ways:

- as String, using UTF-8 encoding
- as ByteBuffer
- as a de-serialized object

The time necessary to receive data can be dependent on the network RTT (round-trip time), transfer bandwidth and amount of received data, but much more on the fact, whether the other side of the connection is sending the data at all. If there is no data to read, the filter has to wait for it to arrive. This is, where the use of the **inTimeout** parameter makes the most sense.

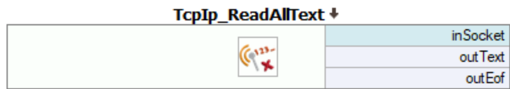
The filters for reading can optionally inform that the connection was closed on the other side and no more data can be retrieved. This mechanism can be used when connection closing is used to indicate the end of transmission or communication. The **outEof** output is used to indicate this condition. When the end of stream is indicated the socket is still valid and we should still close it on our side.

Reads text in UTF-8 encoding until reaching a specified delimiter.

The delimiter* can be either discarded, returned at the end of output, or left in the buffer to be processed by a subsequent read operation.

Reads a fixed-length chunk of binary data.

Reads a serialized object. The data can only come from another instance of Aurora Vision Studio/Executor executing the TcpIp_WriteObject filter described above, using the same type parameter.



Reads all text, until EOF (until other side closes the connection).

Reads all data, until EOF (until other side closes the connection).

* - the delimiter and the suffix are passed as *escaped strings*, which are special because they allow for so called *escape sequences*. These are combinations of characters, which have special meaning. The most important are "\n" (newline), "\r" (carriage return), and "\\" - a verbatim backslash. This allows for sending or receiving certain characters, which cannot be easily included in a String.

Closing Connections after Use

When a give socket is not needed anymore it should be passed to the socket closing filter, which will close the underlying connection and release the associated system resources. Every socket should be explicitly closed using the socket closing filter, even when the connection was closed on the other side or the connection was broken.

Typically, connections are being closed after the application exits its main loop macrofilter.

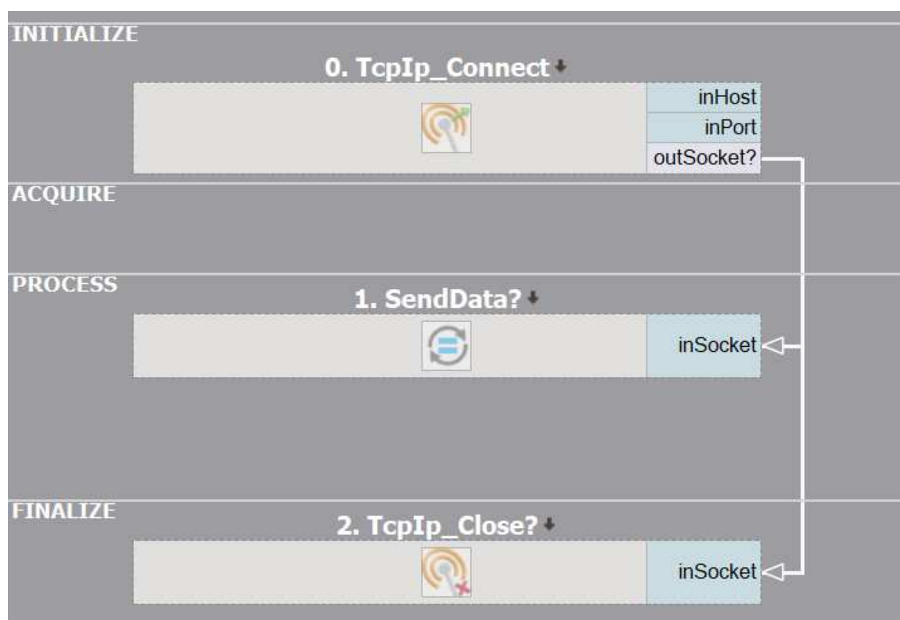
Closes the connection gracefully and releases the socket.

Application Structure

The most typical application structure consist of three elements:

1. A filter creating a socket (TcpIp_Connect or TcpIp_Accept) executed.
2. A macrofilter realizing the main loop of the program (task).
3. A filter closing the socket (TcpIp_Close).

If the main loop task may exit due to an I/O error, as described in the next section, there should be also a fourth element, a Loop filter (possibly together with some Delay), assuring that the system will attempt reconnection and enters the main loop again.



For more information see the "IO Simple TcpIp Communication" official example.

Error Handling and Recovery

In systems where an error detection is critical it is usually required to implement communication correctness checks explicitly based on used communication protocol and capabilities of a communication peer. Such checks will usually consists of transmitting operation acknowledge messages and receiving expected transmissions with limited time. To implement this a timeout system of receiving filters can be used.

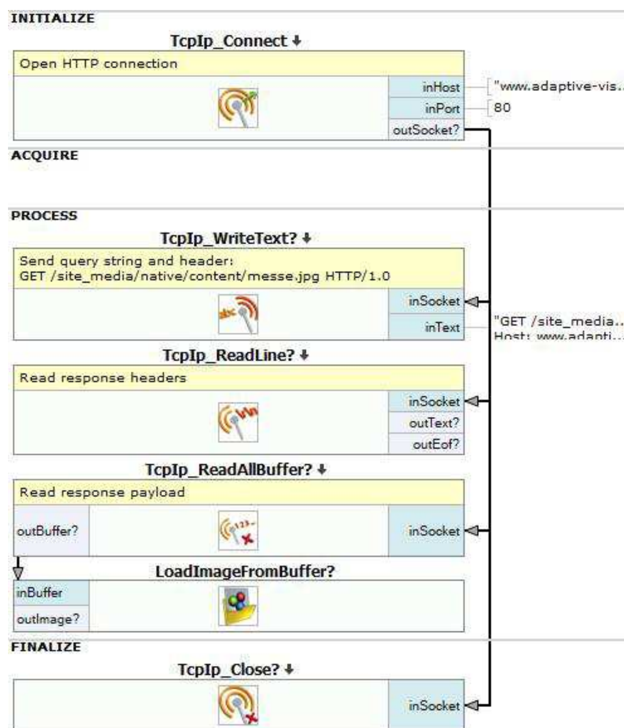
Additionally all TCP/IP filters can notify about unexpected situations or internal communication errors by rising an IoError. Such errors might result out of connection closing by a peer (when it is not expected in a given filter), system errors, network errors, or a broken connection state. The broken connection is usually the most common source of a communication error. Connection enters a broken state as a result of underlying system detecting that the other side of the communication is not confirming that it is receiving data (in some system defined time). After the system marks the connection as broken the next TCP/IP filter (that attempts to use this connection) will notify about that by rising an IoError. Relaying on detecting a broken connection is usually not the most reliable way of verifying valid connection state in error critical systems, as its detection may take a long time, detection time may vary on different systems and in some situations (like uni-directional transmission) it may not be detected at all. The TCP/IP Keep-Alive mechanism (available to activate in TcpIp_Connect and TcpIp_Accept) may help in increasing chances of detecting a broken connection.

IoError raised by filters can be handled using the macrofilter Error Handling mechanism (by putting TCP/IP filters into a separate Task). When a TCP/IP filter is terminated by an error its underlying connection is left in an undefined state, the connection should not be used anymore and the socket should be closed with a TcpIp_Close filter. The application can then attempt to recover from TCP/IP errors by initializing the connection from scratch. Specific possibilities and requirements of recovering from communication errors depends on a used protocol and a communication peer capabilities.

Using Tcp/Ip Communication with XML Files

Very often Tcp/Ip communication is used to exchange information structured as XML documents. For this type of communication, use filters for reading or writing text together with the filters from the System :: XML category for parsing or creating XML trees.

Example: Reading Data from HTTP



A sequence of filters to download an image through HTTP

This example consists of the following steps:

1. A connection is made, to port 80 (the HTTP standard port) of *www.adaptive-vision.com* host.
2. A HTTP query string is sent, to request an image from a specific URL.
3. The headers, which end with "\r\n\r\n" (double newline) are read (and not processed).
4. All the remaining data - the HTTP response body - is read and returned as binary data.
5. The bytes are converted to an image, using a utility function.

General Image Acquisition

Camera Acquisition Thread

In Aurora Vision image acquisition is done in the background. Due to that, regardless of what the vision program is doing a new image can be received from the camera as soon as possible.

When the communication with the camera is started (for example by using **GigEVision_StartAcquisition**, but this applies to different vendors as well) Aurora Vision creates a special background thread. This thread handles all communications with that particular camera, and in case of received images stores them in a special queue. While not exactly the same, in principle that queue is similar queues available to user in Aurora Vision. The size of the queue is determined by the filter (some camera interfaces do not support sizes larger than 1).

When the program executes a grabbing filter (e.g. **GigEVision_GrabImage:Synchronous**) that filter in fact grabs an image from the queue made by the background acquisition thread, not directly from the camera.

If there are no images in the queue, the grabbing filter waits until a new image arrives either for an indefinite amount of time (**synchronous variant**) or for a specified amount of time (**asynchronous variant**). This behavior is analogous to how the filters **Queue_Pop** and **Queue_Pop_Timeout** behave, respectively. After getting an image, it is removed from the queue. If the queue is full and the camera sends a new image, then the oldest image in the queue is replaced.

Images will remain in the queue until they are grabbed, are replaced by a newer image, or their camera thread was closed.

What is important to remember is that if the queue size is larger than 1, the grabbing filter will return the oldest image available. Depending on the application structure this may or may not be the desired behavior.

- If the application has to analyze every image, but the iteration time may be longer than the period between images the queue should be large enough to store all images until the application can catch up. For example, a camera is triggered multiple time in the burst, followed by a period of inactivity, the queue size should be equal to the number of triggers in one burst.
- Conversely, if the application does not need to inspect every image (common in application with a free-running camera) the queue size can usually be limited to one. This ensures the lowest possible lag between image acquisition and results.

As mentioned before, the background camera thread handles all communication with its assigned camera. If no thread for the specified camera exists, it will be created as soon as any camera filter is used. Camera can only have one thread assigned to it, so filters will always use the existing thread if one is available.

Camera threads also handle setting and reading parameters. Depending on the camera interface there may be optimizations, such as writing new parameter values only if the value has changed.

It is important to remember that the threads will be closed if the task in which they were created ends. When the thread is closed all data in it (including images in the queue) is removed.

Working with Gen TL devices

Introduction

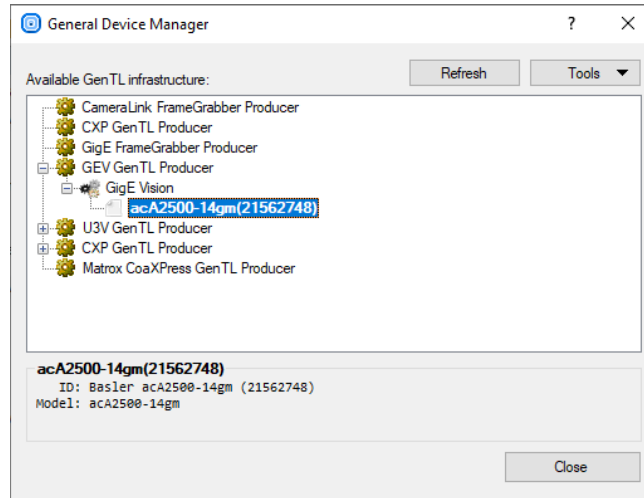
Aurora Vision Studio allows to communicate with devices which use Gen TL. For this purpose, you need to install the software provided by device vendor in your operating system (this step is highly dependant on device vendor so please refer to the device documentation, there is no general recipe for it). You can check if the installed software has added Gen TL provider to your system in the following way:

1. There is a definition of *GENICAM_GENTL32_PATH* and/or *GENICAM_GENTL64_PATH* in the environment variables section.
2. Catalogs created by the installed device vendor software should contain some .cti (Common Transport Interface) files.

Please make sure that the platform (32/64-bit) which the vendor software is intended for is the same as the version (32/64-bit) of Aurora Vision Studio you use.

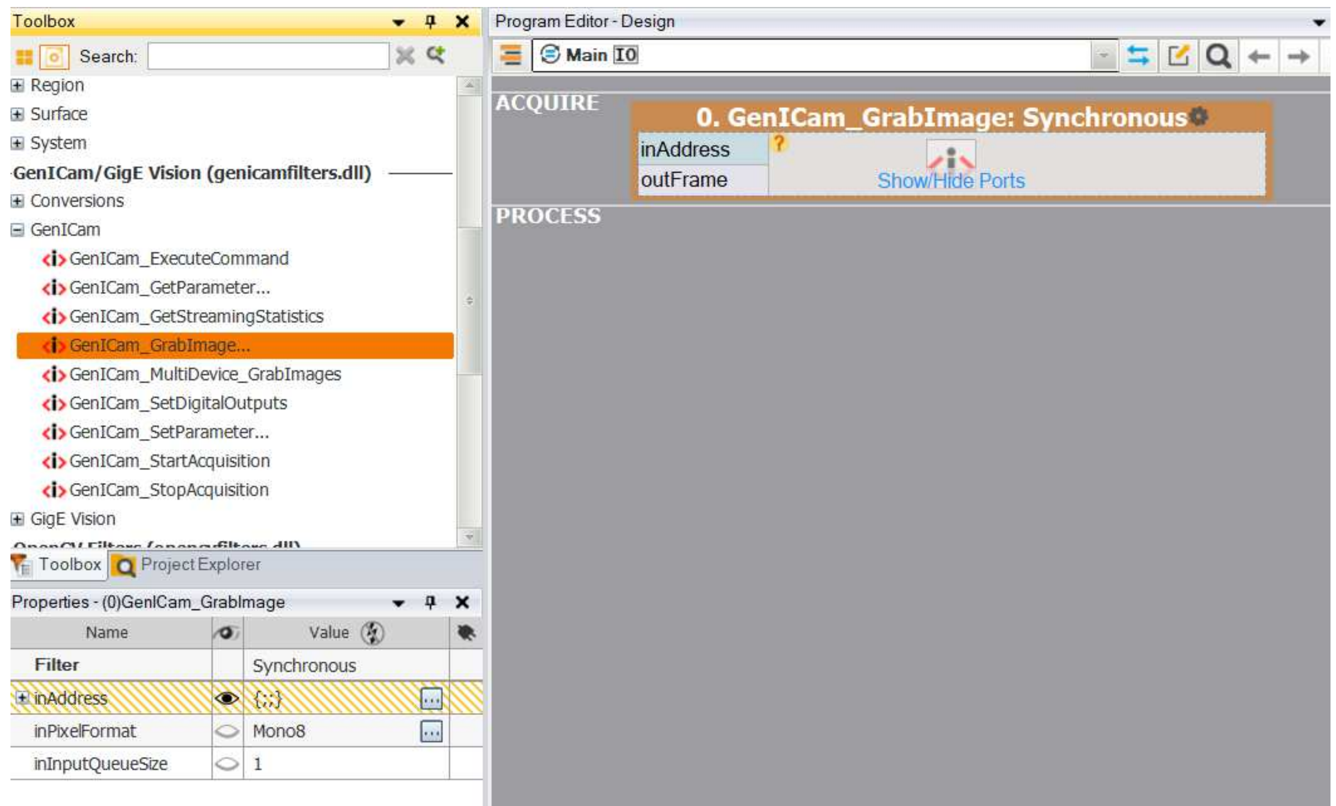
Using GenICam Device Manager

Once you have completed the steps above, you should be able to see your GenICam devices connected to the computer in the GenICam Device Manager (if they are not visible there, the cause is usually that device vendor's .cti files or proper entries in environment variables section are missing), like in the image below:

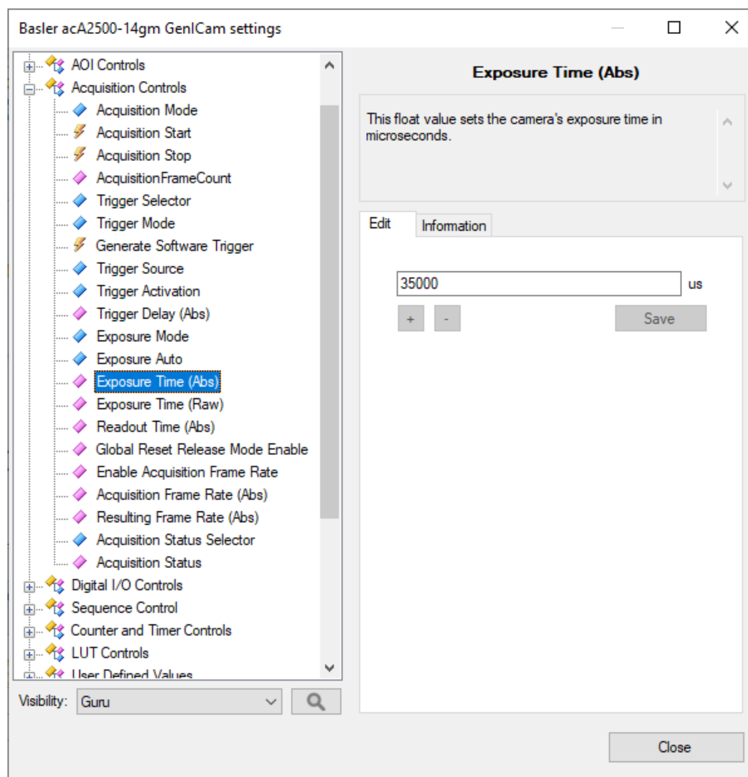


You can access the GenICam Device Manager in one of the two ways:

- Choose *Tools » Manage GenICam Devices...* from the Main Menu.
- Add one of GenICam filters to program and open the GenICam Device Manager directly from filter properties.

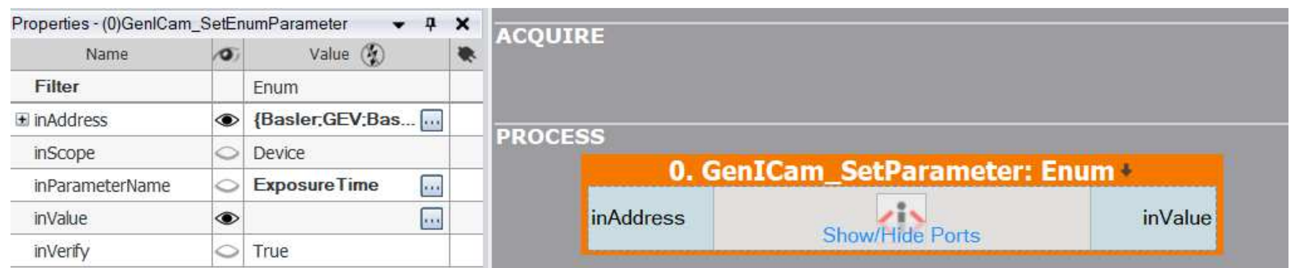


After choosing a device you can go to its settings tree by clicking *Tools » Device Settings* in the GenICam Device Manager. You should see device parameters grouped by category there. Some parameters are read-only, other are both readable and writable. You can set the writable parameters directly in the GenICam Settings Tree.



Another way to read or write parameters is by using the filters from the GenICam category. In order to do this you should:

1. Check of which type is the parameter you'd like to read/write (you can check it in the GenICam Settings Tree).
2. Add a proper filter from the GenICam category to the program.
3. Choose a device, define parameter name and new value (when you want to set the parameter's value) in the filter properties.
4. Execute the filter.



Known Issues Regarding Specific Camera Models

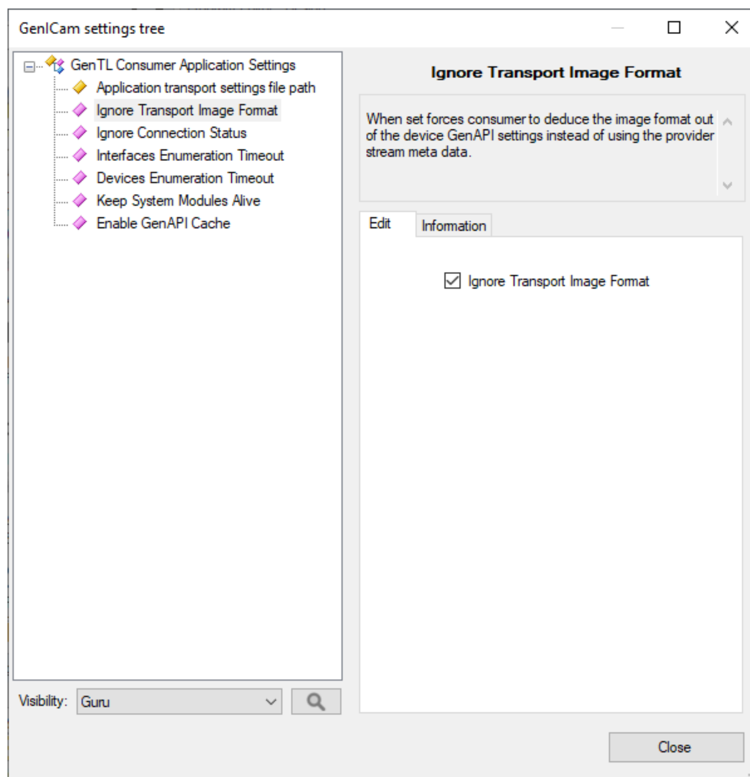
In this section you will find solutions to known issues that we have came across while testing communication between Aurora Vision products and different camera models through GenICam.

Manta G032C

We have encountered a problem with image acquisition (empty preview window with no image from camera) while testing this camera with default parameters. The reason for this is that the packet size in this camera is set by default to 8,8 kB. You need a network card supporting Jumbo Packets to work with packets of such size (the Jumbo Packets option has to be turned on in such case). If you don't have such network card, you need to change the packet size to 1.5 kB (maximal UDP packet size) in the GenICam Settings Tree for the Manta camera. In order to do this, please open the GenICam Device Manager, choose your camera and then click *Tools » Device Settings* (go to the GenICam Settings Tree). The parameter which needs to be changed is *GevSCPSPPacketSize* in the GigE category, please set its value to 1500 and save it. After doing this, you should be able to acquire images from your Manta G032C camera through GenICam.

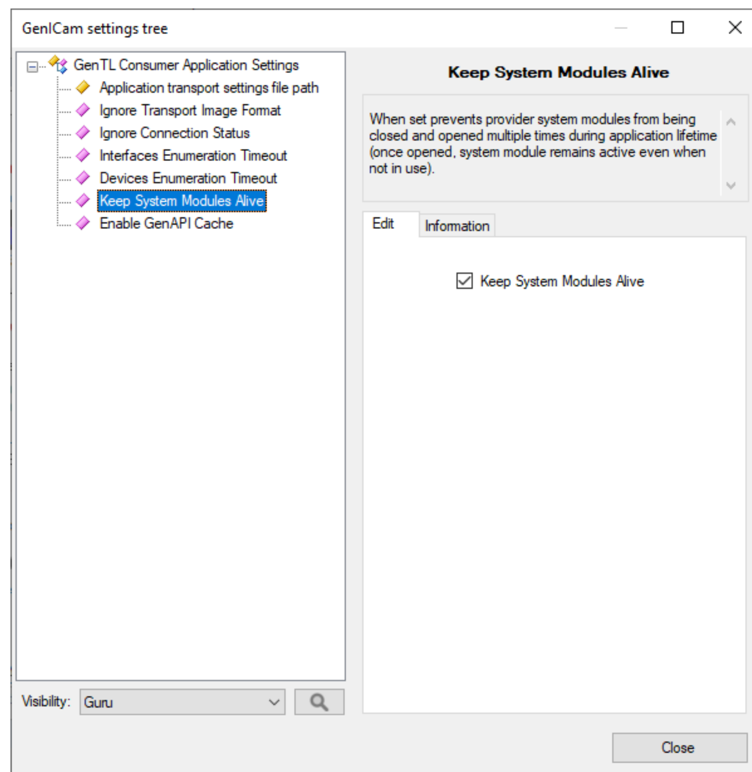
Adlink NEON

On devices with early version of the firmware, there are problems with starting image acquisition. The workaround is simple: just set the IgnoreTransportImageFormat setting to "yes". This option is available in the GenICam Device Manager (Tools → Application Settings...):



Daheng cameras

1. Open GentTL settings: *Tools » Manage GenICam Devices... » Tools » Application Settings*
2. Turn on "Keep System Modules Alive" option



3. Close the window and restart Aurora Vision Studio

It is possible to get IO Error: "Unable to open GenTL Provider System Module" after stopping and starting the application again. To solve that problem you should: Note, that it is a local setting, thus it needs to be repeated again on every PC on which the application would be run.

Working with National Instruments devices

Introduction

Aurora Vision Studio allows for communication with i/o devices from National Instruments. This is done by using filters from the Hardware Support :: National Instruments category.

Working with Tasks

The entire support for National Instruments devices is based on *tasks* (do not confuse with Task macrofilters). A task in Aurora Vision Studio is a single channel. Every task is associated with an ID, which is necessary for configuring tasks, reading states from inputs or writing values to device outputs.

Creating Tasks

To start working with National Instruments' devices, a task that will perform a specific operation must be created.
For this purpose one of the filters from list shown below should be selected.

- DAQmx_CreateDigitalPort
- DAQmx_CreateAnalogChannel
- DAQmx_CreatePulseChannelFreq
- DAQmx_CreateCountEdgesChannel

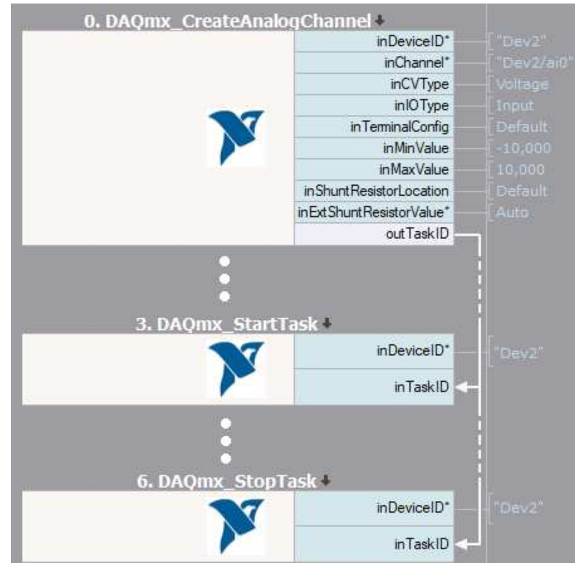
These filters return *outTaskID*, which is the identifier of a created task.

Starting Tasks

To start a task, filter DAQmx_StartTask should be used.

Finishing Tasks

There are two ways of finishing tasks: by using the DAQmx_StopTask filter or by waiting for the entire program to finish.
It should be noted that you cannot start two tasks using the same port or channel in a device.



Basic scheme of program

Reading and Writing Values

Aurora Vision Studio provides two kinds of filters for reading and writing values. DAQmx_WriteAnalogArray, DAQmx_WriteDigitalArray and DAQmx_ReadAnalogArray, DAQmx_ReadDigitalArray, DAQmx_ReadCounterArray filters allow to work with multiple values. For reading or writing a single value, DAQmx_WriteAnalogScalar, DAQmx_WriteDigitalScalar and DAQmx_ReadAnalogScalar, DAQmx_ReadDigitalScalar, DAQmx_ReadCounterScalar filters should be used.

Configuring Tasks

After creating a task, it can be configured using the filters DAQmx_ConfigureTiming, DAQmx_ConfigAnalogEdgeTrigger or DAQmx_ConfigDigitEdgeTrigger. Note that some configuration filters should be used before the task is started.

Sample Application

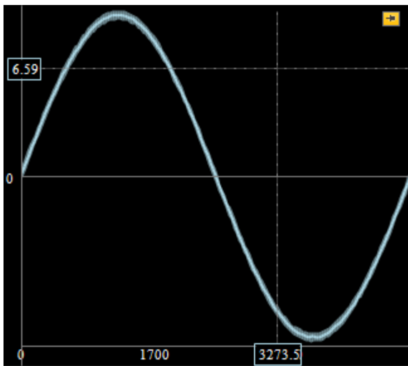
The following example illustrates how to use DAQmx filters. The program shown below reads 5000 samples of voltage from analog input. However, data is acquired on rising edge of a digital signal.

At first, analog channel is created. In this sample, input values will be measured, so **inCVType** input must be set to *Voltage* and **inIOType** should be set to *Input*. The rest of inputs depends on used device.

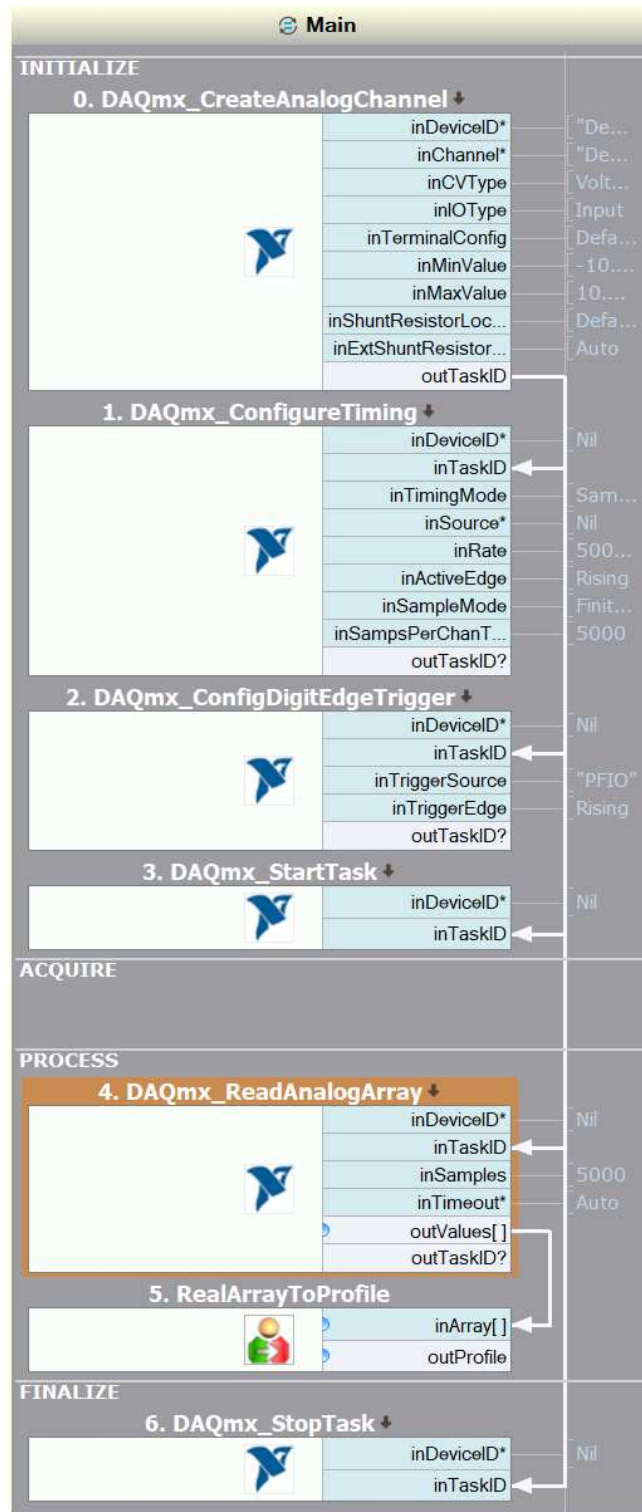
Afterwards, sample clock to task is assigned. **inDeviceID** input might be set to *Auto*, because it is the only one device used in this program. Other parameters could be set according to user's camera.

Sample application should start acquiring data after a digital edge occurs. Because an active edge of digital signal should be rising, **inTriggerEdge** input must be set to *Rising* (for falling edge, value *Falling* must be chosen). Task is ready to start.

Last step in presented program is to acquire multiple data from selected device. Filter DAQmx_ReadAnalogArray could be used for this. DAQmx_StopTask is not required, because this program doesn't use one channel in two different tasks. Acquired array of values might be represented e.g. as a profile (shown below).



Acquired data from DAQ device



Sample application for acquiring data

Working with Modbus TCP Devices

Introduction

The filters for Modbus over TCP communication are located in the System :: Modbus TCP

Establishing Connection

To communicate with a ModbusTCP device, first a connection needs to be established. The ModbusTCP_Connect filter should be used for establishing the connection. The default port for ModbusTCP protocol is 502. Usually, a connection is created before the application enters its main loop and it remains valid through multiple iterations of the process. It becomes invalid when it is explicitly closed using ModbusTCP_Close filter or when an I/O error occurs. The Modbus filters work on the same socket as TCP IP filters. If Modbus device supports Keep-Alive mechanism it is recommended to enable it, to increase chance of detecting a broken connection. For more details please read Using TCP/IP Communication.

Reading and Writing Values

Aurora Vision Studio provides set of filters to manage Modbus objects. The following is table of object types provided by a modbus:

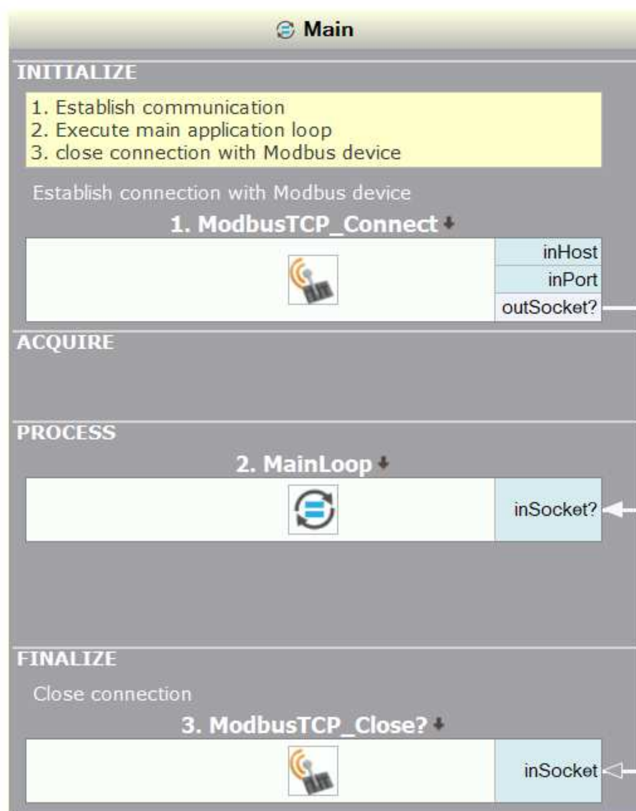
Object Type	Access	Size
Coil	Read-Write	1-bit
Discrete Input	Read-Only	1-bit
Input Register	Read-Only	16-bits
Holding Register	Read-Write	16-bits

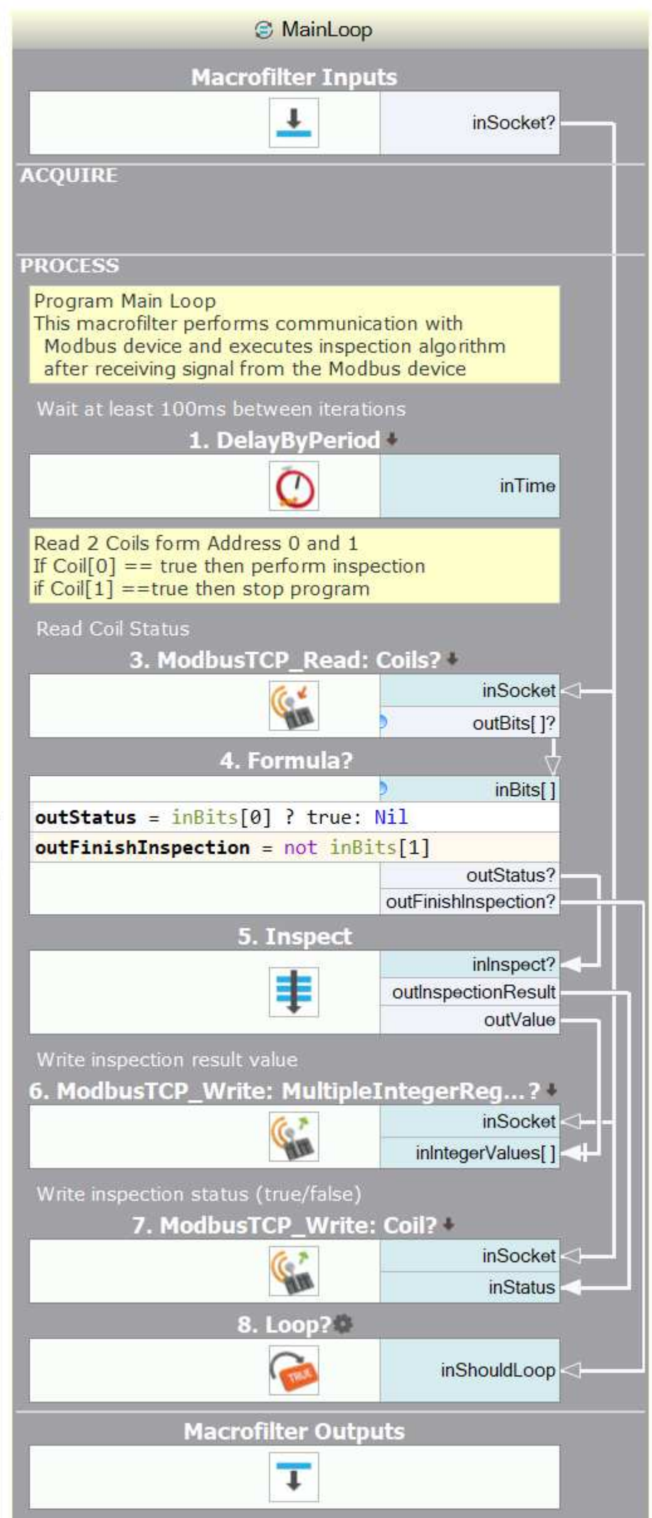
The Coils, Inputs and Registers in ModbusTCP protocol are addressed starting at zero. The table below lists filters to control object types mentioned above.

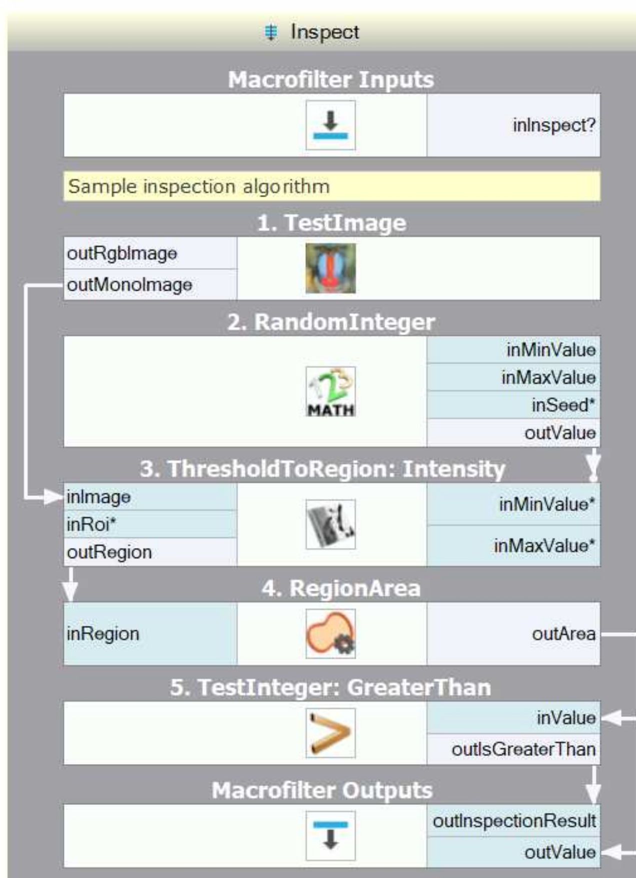
Object Type	Read	Write
Coil	ModbusTCP_ReadCoils	ModbusTCP_WriteCoil ModbusTCP_ForceMultipleCoils
Discrete Input	ModbusTCP_ReadDiscreteInputs	Read-Only
Input Register	ModbusTCP_ReadInputRegisters_AsByteBuffer ModbusTCP_ReadInputIntegerRegisters ModbusTCP_ReadInputRealRegisters	Read-Only
Holding Register	ModbusTCP_ReadMultipleRegisters_AsByteBuffer ModbusTCP_ReadMultipleIntegerRegisters ModbusTCP_ReadMultipleRealRegisters	ModbusTCP_WriteSingleRegister ModbusTCP_WriteMultipleRegisters_AsByteBuffer ModbusTCP_WriteMultipleIntegerRegisters ModbusTCP_WriteMultipleRealRegisters

Sample Application

The following example illustrates how to use ModbusTCP filters.







Main macrofilter - establish and close connection

Main Loop - communication with Modbus device

Inspection - sample inspection algorithm

Custom Functions

To support device specific function Aurora Vision Studio provides ModbusTCP_SendBuffer filter. This filter allows to create a custom Modbus frame.

Project Files

When you save a project, several files are created. Most of them have textual or xml formats, so they can be edited also in a notepad and can be easily managed with version control systems.

A project may consist of the following file types:

- Single ***.avproj** file (XML) – this is the main file of the project; plays a role of the table of content.
- Single ***.avcode** file (textual) – this is the main source code file.
- Zero or more ***.avlib** files (textual) – these are additional source code modules.
- Zero or more ***.avdata** files (binary) – these files contain binary data edited with graphical tools, such as Regions or Paths.
- Single ***.avview** file (XML) – this file stores information about the layout of Data Previews panels.
- Zero or one ***.avhmi** file (XML) – this file stores information about the HMI Design.

C++ Code Generator

- Introduction
- User interface
- Generated Code Organization
- Generated Code Usage
    - Stateful functions
    - Error handling
    - Global parameters
- Compiling a program with generated code
- Running a compiled program
- Generated sample Microsoft Visual Studio solution

Introduction

A program accepted by Aurora Vision Studio and Aurora Vision Executor is executed in a virtual machine environment. Such a machine consecutively executes (with preservation of proper rules) filters from the source program. The C++ Code Generator allows to automatically create a C++ program, which is a logical equivalent of the job performed by the virtual machine of Aurora Vision Studio executor. As part of such programs, consecutive filter calls are changed to calls of functions from Aurora Vision Library.

To generate, compile and run programs in this way it is necessary to own also a Aurora Vision Library license.

User interface

The C++ Code Generator functionality is available in the Main Menu in *File » Generate C++ Code...*. After choosing this command, a window with additional generator options and parameters (divided into tabs) will be opened.

In the *Output* tab basic generator parameters can be set:

- **Program name** - it determines the name of the generated C++ program (it doesn't have to be related to Aurora Vision Studio project name). It is a base name for creating names of source files and (after choosing a proper option) it is also the name of the newly created project.
- **Output directory** - it determines the folder in which a generated program and project files will be saved. Such folder has to exist before generation is started. The path can be either absolute or relative (when an Aurora Vision Studio project is already saved) starting from the project directory. All program and project files will be saved in this folder with no additional subfolders. Before overwriting any files in the output directory, a warning message with a list of conflicted files will be displayed.
- **Code namespace** - an optional namespace, in which the whole generated C++ code will be contained. A namespace can be nested many times using the "::" symbol as a separator (e.g. "MyNamespace" or "MyProject::MyLibrary::MyNamespace"). Leaving the code namespace field empty will result in generating code to the global namespace.
- **Create sample Microsoft Visual Studio solution** - enabling this option will result in generating a new sample tentatively configured solution for compiling generated code in the Microsoft Visual Studio environment. Each time when this option is enabled and code is generated (e.g. when generating code again after making some changes in an Aurora Vision project), a new solution will be created and any potential changes made in an already existing solution with the same path will be overwritten.
  This option is enabled by default when generating code for the first time for the selected folder. It is disabled by default when code is generated again (updated) in the selected output folder.
  Details regarding Microsoft Visual Studio solution configuration required by generated code are described in this document below.

In the *Modules* tab there is a list of project modules which can participate in code generation. It is possible to choose any subset of modules existing in an Aurora Vision Studio project, as long as this doesn't cause breaking the dependencies existing in them. Selecting a subset of modules will cause that only macrofilters and global parameters present in the selected modules will be generated to C++ code. If disabling a module causes breaking dependencies (e.g. a macrofilter being generated refers to an other macrofilter which exists in a module which is disabled in code generation), it will be reported with an error during code generation.

In the *Options* tab there are additional and advanced options modifying the behavior of the C++ Code Generator:

- **Include instances in diagnostic mode** - diagnostic instances (instances processing data from diagnostic outputs of preceding filter instances) are meant to be used to analyze program execution during program development, that's why they aren't included by default in generated programs. In order to include such instances in a generated program this option has to be enabled (**Note: in order to make diagnostic outputs of functions work correctly, enabling diagnostic mode in Aurora Vision Library has to be taken into account in a C++ program**).
- **Generate macrofilter inputs range checks** - the virtual machine controls data on macrofilter inputs in terms of assigned to them allowed ranges. Generated code reproduces this behavior by default. If such control is not necessary in a final product, it is possible to uncheck this option to remove it from a C++ program.
- **Enable function block merging** - language constructions of conditional blocks and loops, which map virtual machine functions in terms of conditional and array filter execution, are placed in generated code. The C++ Code Generator uses optimization which places, where possible, a couple of filter calls in common blocks (merging their function blocks). Unchecking this option will disable such optimization. This functionality is meant to help solve problems and usually should be kept enabled.

The *Generate* button at the bottom of the window starts code generation according to the chosen configuration (in case of a need to overwrite existing files an additional window to confirm such action will be displayed) and when the generation operation is completed successfully it closes the window and saves the parameters. The *Close* button closes the window and saves the chosen configuration. The *Cancel* button closes the window and ignores the changes made to the configuration.

All parameters and generation options are saved together with an Aurora Vision Studio project. At the next code generation (when the configuration window is opened again), the parameters chosen previously for the project will be restored.

In order to keep names between Aurora Vision Studio project files and C++ program files synchronized, it is advised to save a project each time before generating C++ code.

Generated Code Organization

A program is generated to C++ code preserving program's split into modules. For each program module, chosen for generation, there are two files (.cpp and .h) created. For the main module (or when there is only one module) these files have names equal to the program name (with appropriate extensions) which was chosen in generation options. For other modules file names are created according to the template `program-name.module-name.cpp/h`.

Macrofilters contained in modules are generated in form of C++ functions. Each public macrofilter, or a private macrofilter used by an other macrofilter contained in generated code, is included in a .cpp file as a function definition. For each public macrofilter its declaration is included also in an .h file. Analogical rules are applied to global parameters. Macrofilter visibility level (public/private) can be set in its properties.

Other project elements, e.g. an HMI structure, don't have their correspondents in generated code.

Generated function interface corresponds to a set of macrofilter inputs and outputs. In the first place inputs are consecutively mapped to function arguments (sometimes using constant reference type), next arguments of reference types from outputs (function output arguments) are also mapped, through them a function will assign results to objects passed to a function from the outside. E.g. for a macrofilter containing two inputs (inValue1, inValue2) and one output (outResult) a function interface may look like this:

```
void MyMacro( int inValue1, int inValue2, int& outResult )
```

When a macrofilter contains facilities requiring preserving their states in consecutive macrofilter iterations (e.g. a Step macrofilter containing registers, loop generators or accumulators), a state argument will be added to the function interface on the first position:

```
bool MyMacro( MyMacroState& state, int inValue1, int inValue2, int& outResult )
```

Data types of inputs, outputs and connections will be mapped in code to types, structures and constructions based on the templates (e.g. `atl::Array<>` or `atl::Conditional<>`) coming from Aurora Vision Library. Filter calls will be mapped to function calls coming also from Aurora Vision Library. In order to get an access to proper implementations and declarations, there are Aurora Vision Library headers included in generated code. Such includes can appear as well in .cpp files, as in .h files (because references to Aurora Vision Library types appear also in function signatures generated from macrofilters).

In generated program there are also static constants, including constant compound objects, which require initial initialization or loading from a file (e.g. if in the IDE there is a hand-edited region on a filter input, then such region will be saved in an .avdata file together with generated code). A program requires preparing constants before using any of its elements. In order to do this, in the main module in generated code an additional function `Init` with no arguments is added, as part of this function compound constants are prepared. This function has to be called before using any element from generated code (also before constructing a state object).

Generated Code Usage

The basis of generated code are file pairs emerging from modules with the structure described above. These modules can be used as part of a vision program, e.g. as a set of functions implementing the algorithms designed in the Aurora Vision Studio environment.

Before calling any generated function, constructing a function state object or accessing a global parameter, it is necessary to call the `Init()` function added to the main module code. The Init function must be called once at the beginning of a program. This function is added to generated code even when a program does not contain any compound constants which require initialization, in order not to have to modify the schema of generated code after modifying and updating the program.

Stateful functions

As described above, sometimes a function may have an additional first argument named *state*, passed by reference. Such object preserves a function state between consecutive iterations (each call of such function is considered as an iteration). In such objects there are, among others, kept generator positions (e.g. the current position of filters of *Enumerate** type), register states of Step macrofilters or internal filter data (e.g. a state of a connection with a device in filters which acquire images from cameras).

A state object type is a simple class with a parameterless constructor (which initializes the state for the first iteration) and with a destructor which frees state resources. It is the task of applications using functions with a state to prepare a state object and to pass it through a parameter to a function. **An application may construct a state object only after calling the `Init()` function.** An application should not modify such object by itself. One instance of a state object is intended for a function call as part of one and the same task (it is an equivalent of a single macrofilter instance in an Aurora Vision Studio program). Lifetime of a single object should be sustained for the time when such task is performed (e.g. you should not construct a new state object to perform the same operation on consecutive video frames). A single state object cannot be shared among different tasks (e.g. if as part of one iteration the same function is called twice, then both calls should use different instances of a state object).

State object type name is generated dynamically and it can be found in the declaration of a generated function. The C++ Code Generator, if possible, creates names of state types according to the template *Macrofilter-name*State.

Freeing state resources is performed in a state class destructor. If a function established connections with external devices, then a state object destructor is used also to close them. The recommended way of state handling is using a local variable on the stack, in such way that a destructor is called automatically after stepping out of a program block.

Error handling

The functions of Aurora Vision Library and generated code report the same errors which can be observed in the Aurora Vision Studio environment. On the C++ code level error reporting is performed by throwing an exception. To throw an exception, an object of type derived from the `atl::Error` class is used. Methods of this class can be used to get the description of reported problems.

Note: the `Init()` function can also throw an exception (e.g. when an external .avdata file containing a static constant could not be loaded).

Global parameters

Simple global parameters (only read by the vision application) are generated as global variables in the C++ program (with the same name as the global parameter). It is possible to modify those variables from the user code in order to change the application configuration, however such modification is **not thread safe**.

When thread safe modification of the global parameters is needed, global parameters should only be accessed with WriteParameter and ReadParameter filter blocks in the vision application. In order to allow access from user code to such operations, appropriate read/write should be encapsulated in a public macrofilter participating in the code generation. This will give access to the parameters from the user code in form of a function call.

Compiling a program with generated code

Generated C++ code is essentially a project basing on Aurora Vision Library and has to follow its guidelines. Compilation requires this product to be installed in order to get the access to proper headers and .lib files. Aurora Vision Library requires that a project is compiled in the Microsoft Visual Studio environment.

A program being compiled requires the following settings:

- The header file search path is set to the *include* folder of the Aurora Vision Library package.
- The search paths of .lib files for respective platforms are set to the corresponding subfolders in the *lib* directory of the Aurora Vision Library package (e.g. the Release|Win32 configuration compiled in the Microsoft Visual Studio package has to use the libraries from the *lib\Win32* subfolder).

- The AVL.lib library is linked to a project (in order to link a dynamic library - AVL.dll).

Running a compiled program

A program created basing on generated code and the Aurora Vision Library product requires the following components to run:

- All .avdata files generated with C++ code (if any were generated at all), located in the current application folder during the `Init()` function call.
- If in the Aurora Vision Studio project (from which code is generated) there are any filters reading external data (e.g LoadImage) or enclosures of data files to filter inputs, then all such files have to be available during program execution. The files will be searched according to the defined path, in case of a relative path the search will begin starting from the current application folder.
- A redistributable package of Visual C++ (from the version of Microsoft Visual Studio used to compile the program and Aurora Vision Library) installed in the destination system.
- The AVL.dll file ready to work with the used platform. This module can be found in a subfolder of the *bin* directory of the Aurora Vision Library package (analogically to the use of the search paths of .lib files).
- If the generated code uses third party packages (e.g. cameras, I/O cards, serial ports), then additional .dll files (intermediating in the access to the drivers of such devices) are required. Such files are identified with names ending with the "_Kit" suffix, they can be found in the same directory as the used AVL.dll file (also in this case it is necessary that this file is compatible with given platform). In case when a required file is missing, the program being executed will report an error pointing the name of the missing module in a message. Such libraries are loaded dynamically during program execution. In such case, for correct execution it is also necessary to install proper redistributable packages or SDKs for the used devices.
- A valid license for Aurora Vision Library activated in the destination system.

Generated sample Microsoft Visual Studio solution

After choosing a proper option in the *Output* tab, the C++ Code Generator, apart from modules' code, will also create a sample tentatively configured Microsoft Visual Studio project together with sample user code using generated code. The configuration of such project follows the compilation requirements described above, including attaching headers and .lib files from the Aurora Vision Library package (it uses the AVL_PATHXX environment path created by the installer of Aurora Vision Library). This means that to compile a sample project, a properly installed package of Aurora Vision Library is required.

To make the generated project's structure clear and distinguish its elements, there are two filters (solution directories) created in such project:

- **Generated Code** - here are located files deriving from generation of project modules. Such elements are not supposed to be further manually modified by the user, because in case of project update and code re-generation such modifications will be overwritten.
- **User Code** - here are located files from the sample user application. It's assumed that such code should be created by the user and implement a final application which uses generated code.

Project configuration covers:

- Including paths to the headers of the Aurora Vision Library package.
- Linking with the AVL.lib library.
- Copying the AVL.dll file to the destination folder.

The configuration does not cover providing proper _Kit files (if they are required).

As sample user code there is one simple "main.cpp" file created, it performs the following actions:

- Calling the Init function.
- Activating the diagnostic mode of Aurora Vision Library (if in project options usage of diagnostic instances is enabled for code generation).
- Calling the Main macrofilter's function (if it took part in code generation during creation of the main.cpp file).
- Catching all exceptions thrown by the program and Aurora Vision Library and reporting error messages on the standard output before closing the program.

A project and sample user code are generated once. It means that neither project configuration nor user code which calls generated code will be updated during project update (code re-generation). They can be only completely overwritten in case of choosing again the option to create a sample Microsoft Visual Studio solution. A project and sample code are created basing on names and options from the code generation configuration in Aurora Vision Studio.

In case of developing an application using generated code, the duty to adjust a project to changes in generated code is assigned to the user. To such tasks belong among others:

- Adding to the project and removing from it files with code of generated modules and adjusting `#include` directives in case of modifying modules in an Aurora Vision Studio project.
- Adjusting the code which calls generated code in case of modifications of interfaces of public macrofilters.
- Providing proper _Kit files in the scope of the resulting program.

Please note that projects generated from Aurora Vision Studio up to version 3.2 and thus prepared for Aurora Vision Library up to version 3.2 required linking with additional library named "AvCodeGenRuntime.lib". This library is not required anymore and can be removed from project configuration when updating to newer version of Aurora Vision Library.

.NET Macrofilter Interface Generator

- Introduction
- Requirements
- .NET Macrofilter Interface Assembly Generator
  - Output page
  - Interface page
  - Compiler page
  - Advanced page
- .NET Macrofilter Interface Assembly Usage
  - Initialization
  - Finalization
  - Bitness
  - Diagnostic mode
  - Dialogs
  - Full AVL.NET
- Example
  - Introduction
  - Creating Aurora Vision Project
  - Generating .NET Macrofilter Interface Assembly
  - Using .NET Macrofilter Interface Assembly in a Visual C# application
- Hints

Introduction

Generation of macrofilter interfaces allows to design complex applications without loosing comfort of visual programming which comes from the environment of Aurora Vision Studio.

The most common reasons people choose .NET Macrofilter Interfaces are:

- Creating applications with very complex or highly interactive HMI.
- Creating applications that connect to external databases or devices which can be accessed more easily with .NET libraries.
- Creating flexible systems which can be modified without recompiling the source code of an application.

Macrofilters can be treated as mini-programs which can be run independently of each other. Aurora Vision Studio enables to use those macrofilters in such programming languages as C# or C++/CLI and execute them as regular class methods. Because those methods are just interfaces to the macrofilters, there is no need to re-generate the assembly after each change made to the AVCODE (the graphical program).

Requirements

In order to build and use a .NET Macrofilter Interface assembly, the following applications must be present in the user's machine:

Building	Running
Aurora Vision Professional 5.3	**Aurora Vision Professional 5.3** or **Aurora Vision Runtime 5.3**
Microsoft Visual Studio 2015 (or greater)	**Microsoft Visual C++ Redistributable Package** of the same bitness (32/64) as the generated assembly and the same version as Microsoft Visual Studio used to build the assembly.

.NET Macrofilter Interface Assembly Generator

Macrofilters may be interfaced for use in such managed languages as C# or Visual Basic. Such interface is generated into a Dynamic Link Library (dll), which can be then referenced in a Visual C# project. To generate a dll library for the current Aurora Vision Studio project, fill in all necessary options in the .NET Macrofilter Interface Generator form that can be opened from *File » Generate .NET Macrofilter Interface...*

Output page

Namespace
: Defines the name of the main library class container.

Macrofilter Interface Class Name
: Defines the name of the class, where all macrofilters will be available as methods (with the same signatures as macrofilters).

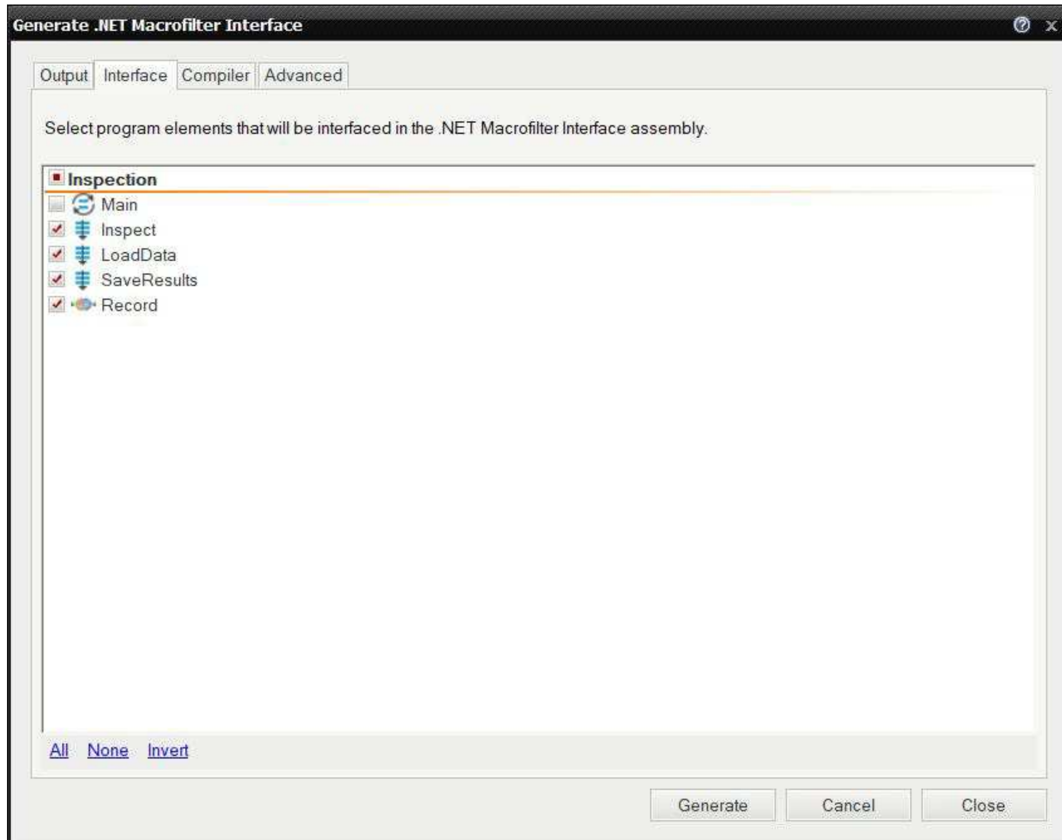
Generate sample Microsoft Visual Studio solution
: Generates empty Microsoft Visual Studio C# WinForms project that uses to-be created Macrofilter .NET Interface assembly.

Link Aurora Vision Project Files
: Includes current Aurora Vision project files to the C# project as links. This way Aurora Vision project files (*.avproj, *.avcode, *.avlib) are guaranteed to be easily accessible from the application output directory, e.g. with following line of code (assuming the project is Inspection.avproj):

```
macrofilters = InspectionMacrofilters.Create(@"auroravision\Inspection.avproj");
```

Path to the generated dll library
: Location of the to-be generated assembly.

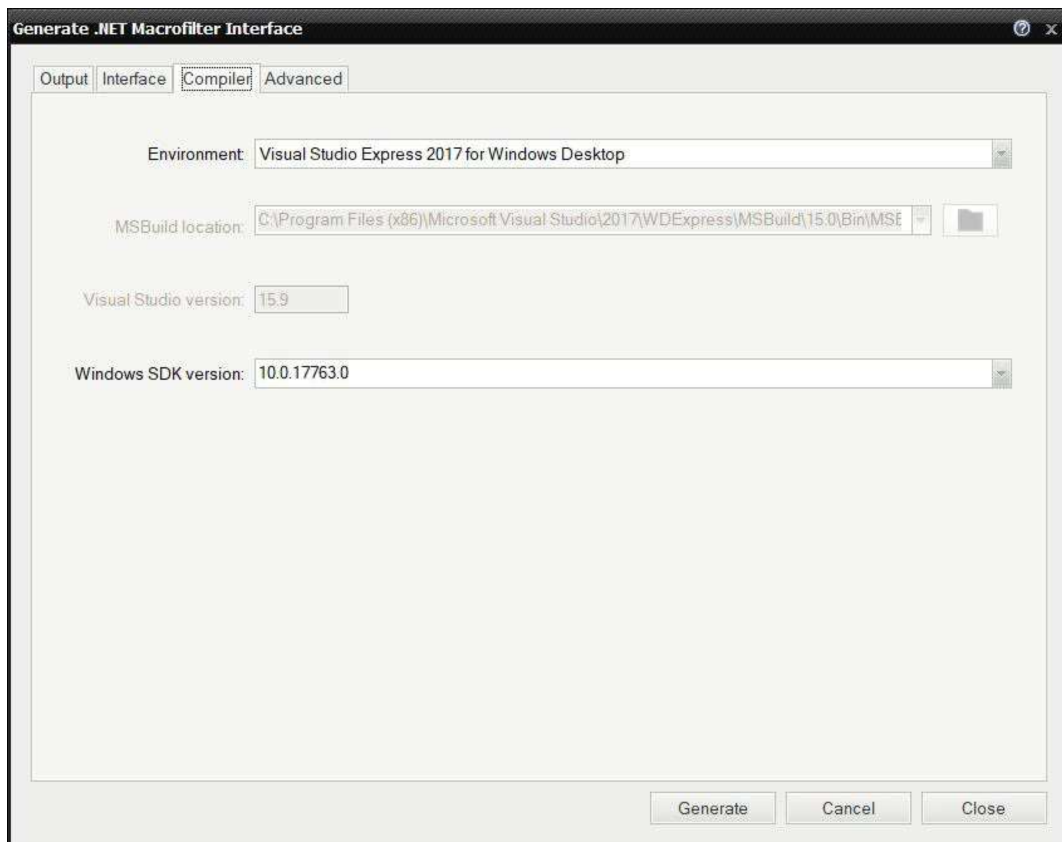
Interface page



Check box list

List of all macrofilters and User Types defined in the current Aurora Vision project that a .NET Interface may be generated for.

Compiler page



Environment

Selection of which Microsoft Visual Studio build tools should be used to build an assembly. The drop down list is populated with all detected compatible tools in the system, including Microsoft Visual Studio and Microsoft Visual Studio Build Tools environments. If none of detected are suitable, custom environment may be used. Then, MSBuild.exe location and target Microsoft Visual Studio version properties need to be defined manually.

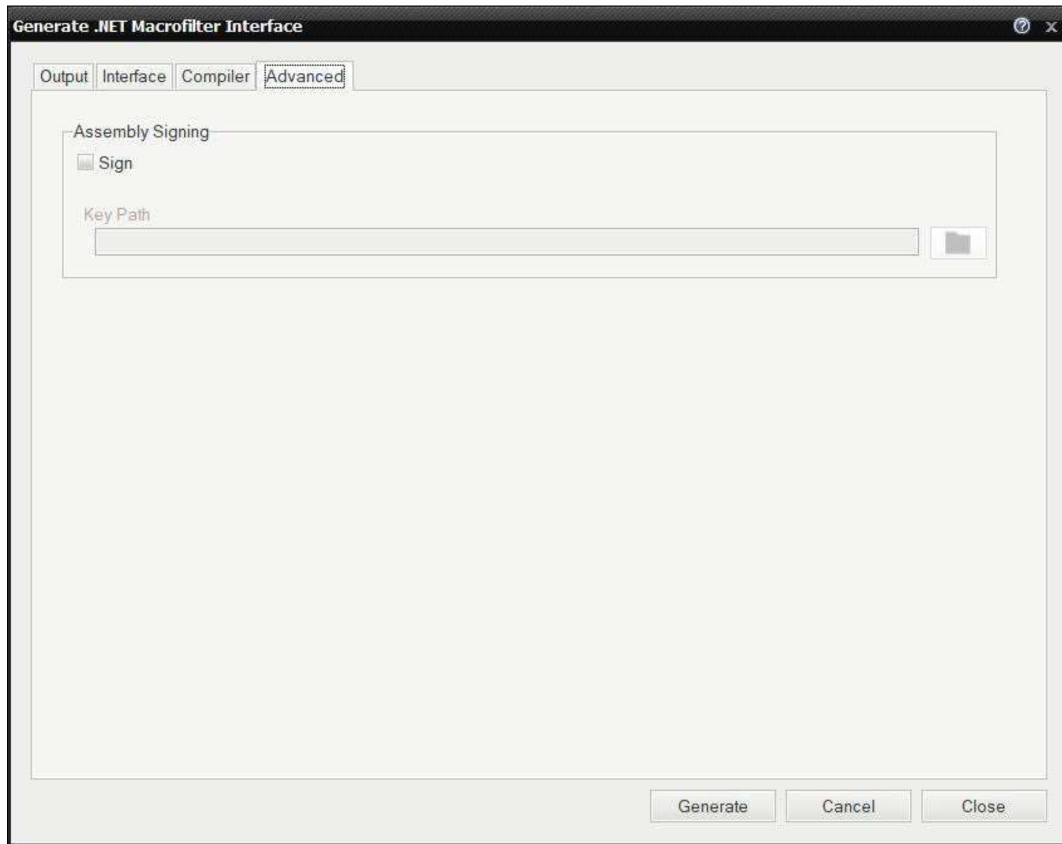
MSBuild location

Shows the path to the actual MSBuild.exe according to the selected environment. Editable, when Custom environment is selected.

Microsoft Visual Studio version
Defines the generated sample C# project format (sln and csproj files).

Windows SDK version
Allows choosing the appropriate SDK version when generating Macrofilter .Net Interface. The list contains all detected SDK versions in the system.

Advanced page



Assembly Signing
Enables signing the generated Macrofilter .NET Assembly with given private key and makes it a Strong-Named assembly. Keys may be generated e.g. in Strong Name Tool or in Microsoft Visual Studio IDE (C# project properties page).

.NET Macrofilter Interface Assembly Usage

Initialization

Once a generated library is referenced in a Visual C# project, macrofilters chosen in the .NET Macrofilter Interface Assembly Generator form are available as instance methods of the class defined in the Macrofilter Interface Class Name text box (`MacrofilterInterfaceClass`), in the Output page. However, in order to successfully execute these methods, a few Aurora Vision libraries must be initialized, i.e. Executor.dll and available Filter Libraries. It is done in a static method named `MacrofilterInterfaceClass.Create`, so no additional operations are necessary. To achieve the best performance, such initialization should be performed only once in application lifetime, and only one instance of the `MacrofilterInterfaceClass` class should exist in an application.

`MacrofilterInterfaceClass.Create` static method receives a path to either *.avcode, *.avproj or *.avexe path, for which the dll library was generated. It is suggested to wrap `MacrofilterInterfaceClass` instantiating with a `try-catch` statement, since the `Create` method may throw exceptions, e.g. when some required libraries are missing or are corrupted.

Finalization

`MacrofilterInterfaceClass` class implements the IDisposable interface, in which the Dispose() method, libraries' releasing and other cleaning ups are performed. It is a good practice to clean up on application closure.

Bitness

Generated assembly bitness is the same as the used Aurora Vision Studio bitness. That is why the user application needs to have its target platform adjusted to avoid libraries' format mismatch. For example, if Aurora Vision Studio 64-bit was used, the user application needs to have its target platform switched to the *x64* option.

Diagnostic mode

The macrofilter execution mode can be modified with `DiagnosticMode` property of the generated Macrofilter .NET Interface class. It enables both checking and enabling/disabling the diagnostic mode.

Dialogs

It is possible to edit geometrical primitives the same way as in Aurora Vision Studio. All that need to be done is to use appropriate classes from the Avl.NET.Designers.dll assembly. Dialog classes are defined in AvlNet.Designers namespace. For more info see the AVL.NET Dialogs article.

Full AVL.NET

With Aurora Vision Library installed one can take advantage of full AVL.NET functionality in the applications that use Macrofilter .NET Interface assemblies. Just follow the instructions from Getting Started with Aurora Vision Library .NET.

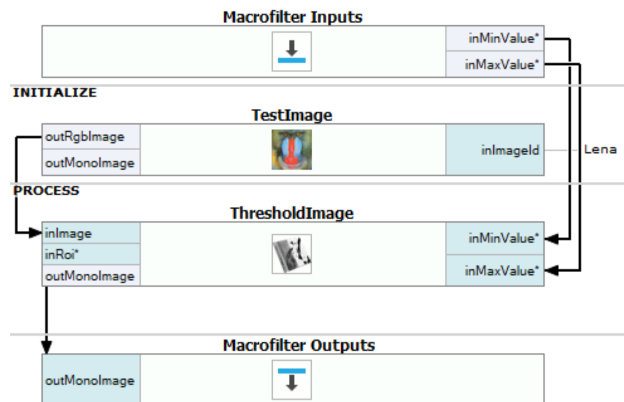
Example

Introduction

This example is a step-by-step demonstration of the .NET Macrofilter Interface Assembly generation and usage. To run this example at least Aurora Vision Studio Professional 5.3 and Microsoft Visual Studio 2015 are required. Visual C# will be used to execute macrofilters.

Creating Aurora Vision Project

We have to have some Aurora Vision Studio project, which macrofilter we want to use in our application. For demonstrative purposes the project will be as simple as possible, just thresholding Lena's image with parameters provided in a Visual C# application's GUI. The macrofilter we would like to run in a C# application will be **ThresholdLena** (see: Creating Macrofilters). The whole macrofilter consists of two filters as in the image below:

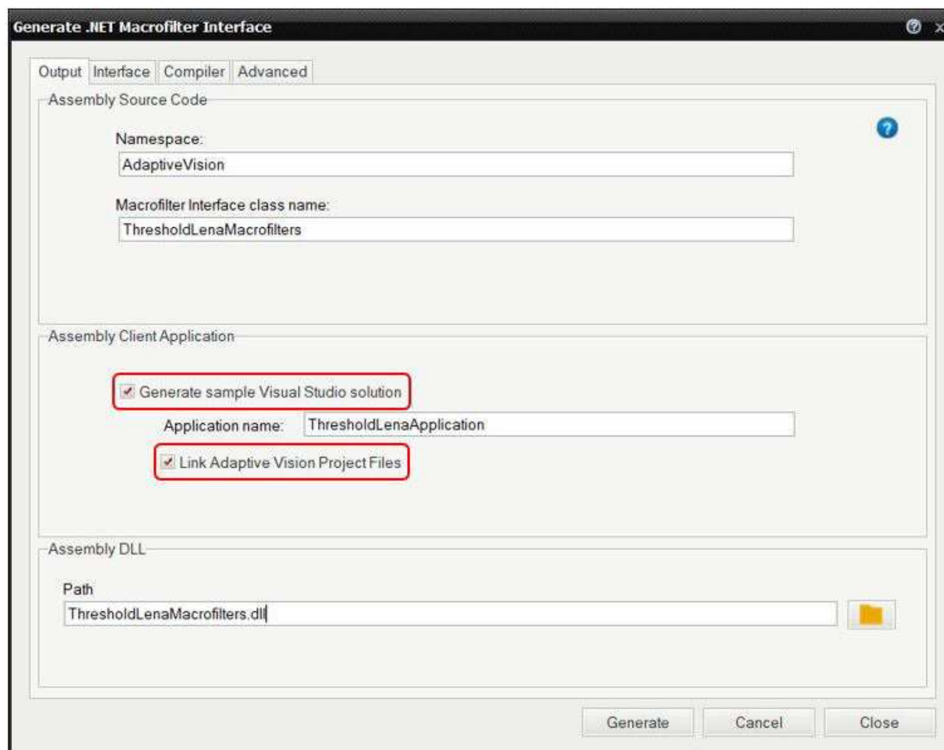


*A basic **ThresholdLena** macrofilter used in example.*

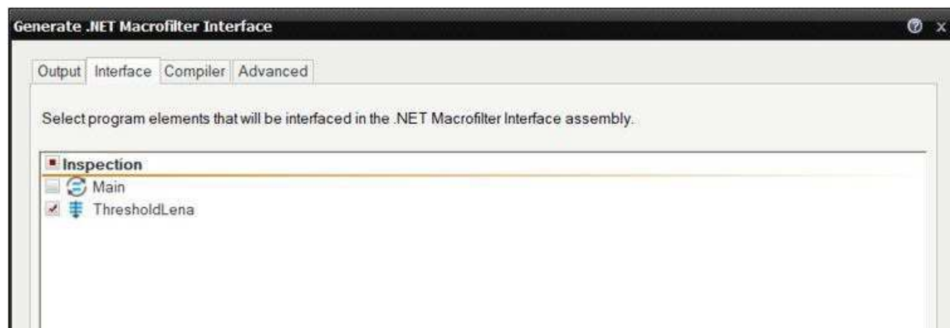
Generating .NET Macrofilter Interface Assembly

Having a ready Aurora Vision Studio project, a .NET Macrofilter Interface assembly may be generated from *File » Generate .NET Macrofilter Interface...*. If the project has not been saved yet, you will be prompted to do it now.

In the Output page of the .NET Macrofilter Interface dialog check the *Generate Visual Studio Solution* option to create a default C# project with required configuration properly set for the current Aurora Vision Studio installation. It is suggested to check also the *Link Aurora Vision Project Files* check box to include necessary files to the C# project.



In the Interface page check the ThresholdLena to generate .NET Macrofilter Interface for the macrofilter. This macrofilter will be accessible as C# method later on.



When ready, click *Generate* to generate an assembly, which will allow us to run the ThresholdLena macrofilter in a Visual C# application.

~ThresholdLenaMacrofilters()
Create(string)
Dispose()
Dispose(bool)
Exit()
ResetThresholdLena()
SetAssertionErrorThrowingMode(bool)
ThresholdLena(float?, float?, AvlNet.Image)
DiagnosticMode

Methods provided by macrofilter interface class.

Using .NET Macrofilter Interface Assembly in a Visual C# application

Having generated the ExampleMacrofilters.dll we may include it into Visual C# application references and have access to the only type exposed by the library, i.e. `AuroraVision.ExampleMacrofilters` class (if you have checked the *Generate Visual Studio Solution* option in the *Generate .NET Macrofilter Interface* dialog, you may proceed to the next paragraph since all references are already set, as well as a ready-to-run starter application). Aside from generated ExampleMacrofilters.dll there is also the Avl.NET.TS.dll assembly which has to be referenced if any of the AVL types are used in the application. In this example the `AvlNet.Image` type will be used to obtain an image from ThresholdLena macrofilter so Avl.NET.TS.dll is essential here.

As it was mentioned before, instantiation of this class should be performed once during application lifetime. As so, `Form1`'s constructor is a perfect place for such an initialization. Instance of the `ExampleMacrofilters` should be then kept in the `Form1` class as a private member:

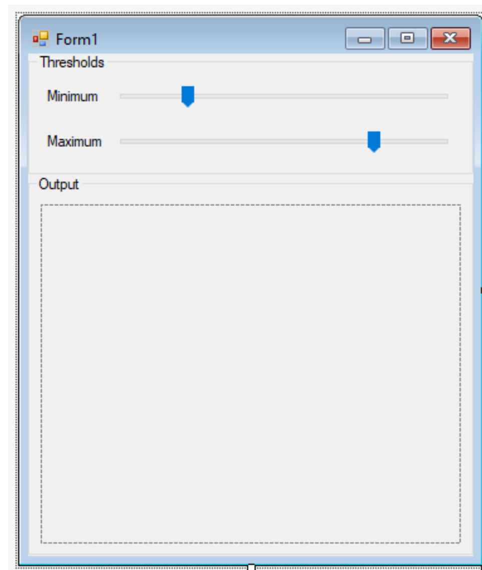
```
using AvlNet;
using AuroraVision;
    public partial class Form1 : Form
    {

        /// <summary>
        /// Object that provides access to the macrofilters defined in the Aurora Vision Studio project.
        /// </summary>
        private readonly ThresholdLenaMacrofilters macros;

        public Form1()
        {
            InitializeComponent();
            try
            {
                string avsProjectPath = @"auroravision\ThresholdLena.avproj";
                macros = ThresholdLenaMacrofilters.Create(avsProjectPath);
            }
            catch (Exception e)
            {
                MessageBox.Show(e.Message);
            }
        }
```

`ExampleMacrofilters` class does not provide a public constructor. Instead, instances of the `ExampleMacrofilters` class can only be obtained through it's static method `Create` accepting a path to either *.avcode, *.avproj or *.avexe file. In this example it takes the path to the *.avcode file with the definition of ThresholdLena macrofilter. *Example.avcode* passed to the `Create` method means, that the runtime will look for the *.avcode file in the application output directory. To guarantee that this file will be found, it should be included in the project and its "Copy to Output Directory" property should be set to either "Copy always" or "Copy if newer".

The C# project is prepared to run the macrofilters as methods. Since ThresholdLena outputs an image taking two optional float values, which stand for threshold's minimum and maximum values, let's add one `PictureBox` and two `TrackBar` controls with value range equal to 0-255:



Changing a value of either of track bars calls the `UpdateImage()` method, where ThresholdLena macrofilter is executed and calculated image is printed in the PictureBox control:

```
private void UpdateImage()
{
    try
    {
        //create an empty image buffer to be populated in the ThresholdLena macrofilter
        using (var image = new Image())
        {
            //call macrofilter
            macros.ThresholdLena(minTrackBar.Value, maxTrackBar.Value, image);

            //dispose previous background if necessary
            pictureBox1.Image?.Dispose();

            //get System.Drawing.Bitmap object from resulting AvlNet.Image
            pictureBox1.Image = image.CreateBitmap();
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(
            ex.Message,
            "Macrofilter Interlace Application",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

On application closing, all resources loaded in `ExampleMacrofilters.Create(...)` method, should be released. It is achieved in `ExampleMacrofilters.Dispose()` instance method, which should be called during disposing of containing form, in `Form1.Dispose(bool)` override:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        if (components != null)
            components.Dispose();

        //Release resources held by the Macrofilter .NET Interface object
        if (macros != null)
            macros.Dispose();
    }

    base.Dispose(disposing);
}
```

Hints

- Generated macrofilter interface class offers also **`Exit()`** method which allows user to disconnect from Aurora Vision environment.
- Every step macrofilter contains specific resetting method, which resets an internal state. For example method **`ResetLenaThreshold()`** resets internal register values and any iteration states.
- Best way to create an application using macrofilter interfaces is to use encrypted avexe files. It secures application code from further modifications.
- Diagnostic mode can be switched on and off with IsDiagnosticModeEnabled static property of the AvlNet.Settings class.
- The application using Macrofilter .NET interface may also reference the AVL's AvlNet.dll assembly to enable direct AVL function calls from the user code (see Getting Started with Aurora Vision Library .NET for referencing the AvlNet.dll in the user applications).

Remote USB License upgrade

Introduction

This guide will show steps, which are to be taken, when dongle license upgrade is considered. This may happen in various situations, like purchase of new products or new major software release.

To perform remote license upgrade following will be needed:

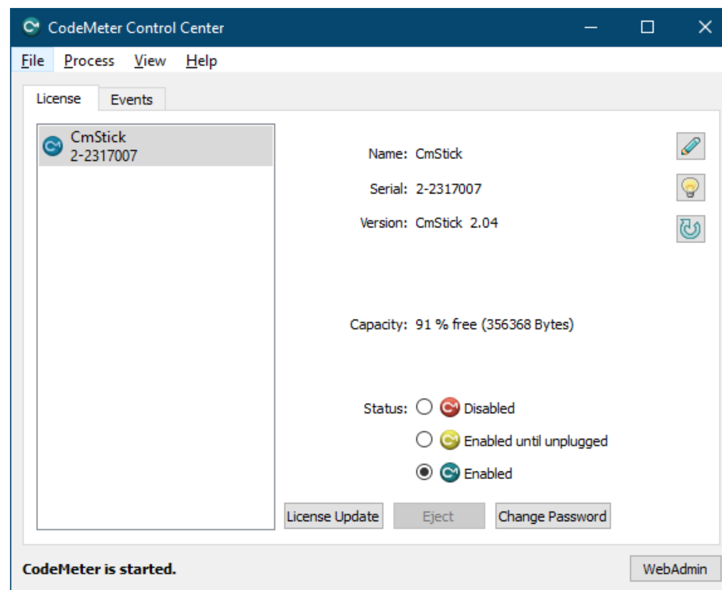
- Computer with Wibu's CodeMeter runtime. This can be installed with Aurora Vision Studio or downloaded from https://www.wibu.com/support/user/downloads-user-software.html.
- Access to Internet.
- Dongle, which will be upgraded.

Process of remote update consists of three steps:

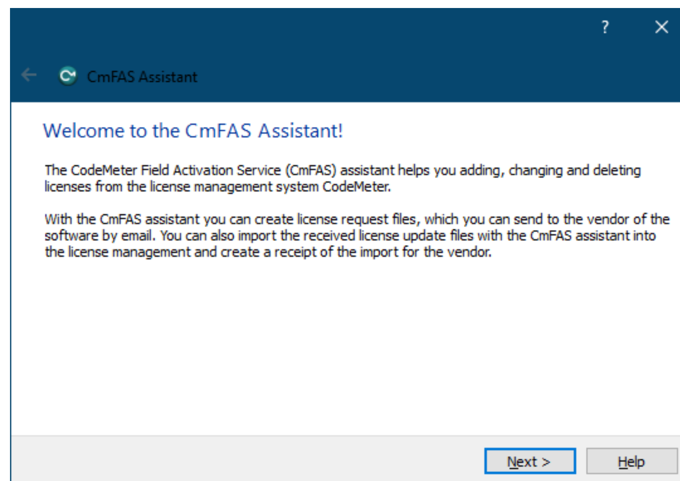
1. Creation of WibuCmRac file, which identifies your dongle.
2. Sending created file to Aurora Vision team.
3. Receiving WibuCmRau file, which is used to update dongle.

WibuCmRac file creation

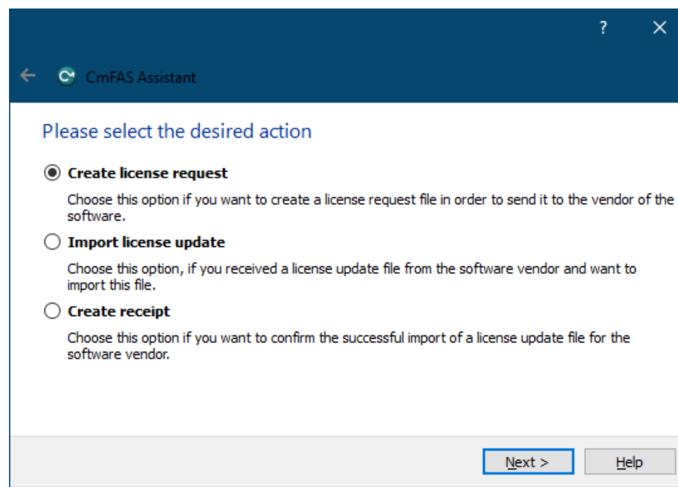
First step is to open CodeMeter Control Center, by clicking on its icon in systems' tray. Opened window should like below:



Next step is to choose dongle, that is about to be upgraded and then click on "License Update" button, which will bring CmFas Assistant window:

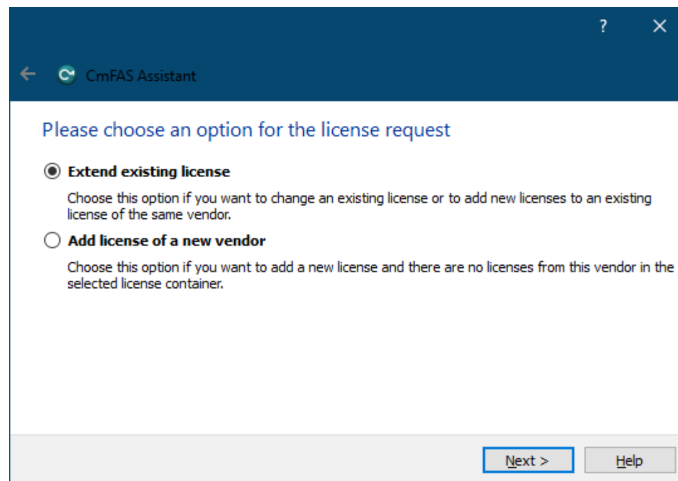


If "Next" button is chosen, window with possible operations will appear:

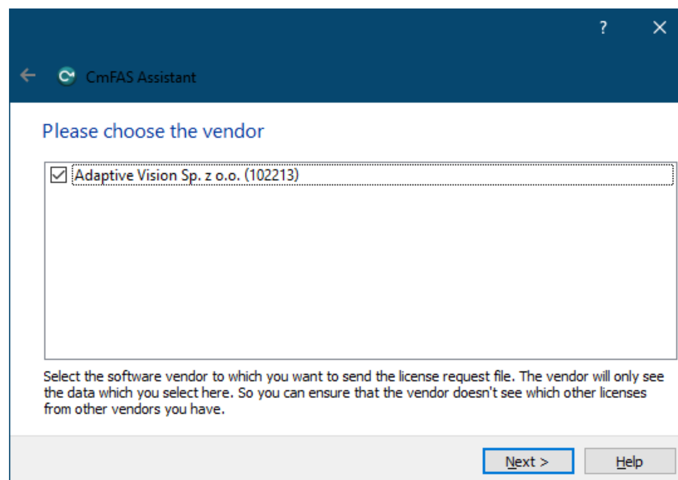


To create WibuCmRac file, which is necessary for process of update, first option should be selected, "Create license request".

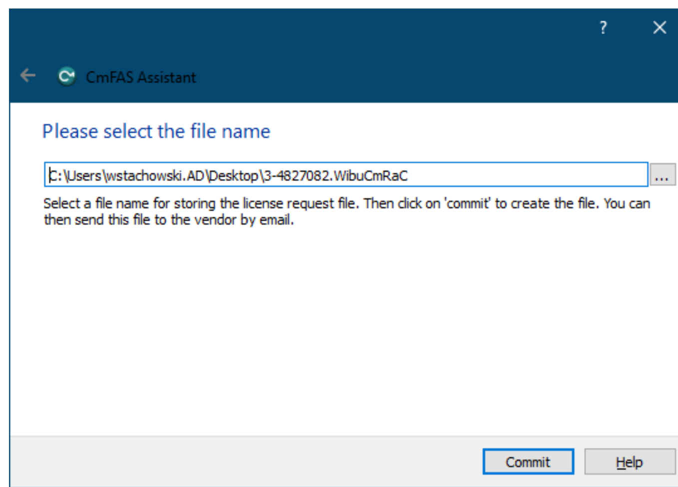
Because there is already Aurora Vision Studios' license in dongle being upgraded, in next step "Extend existing license" should be chosen.



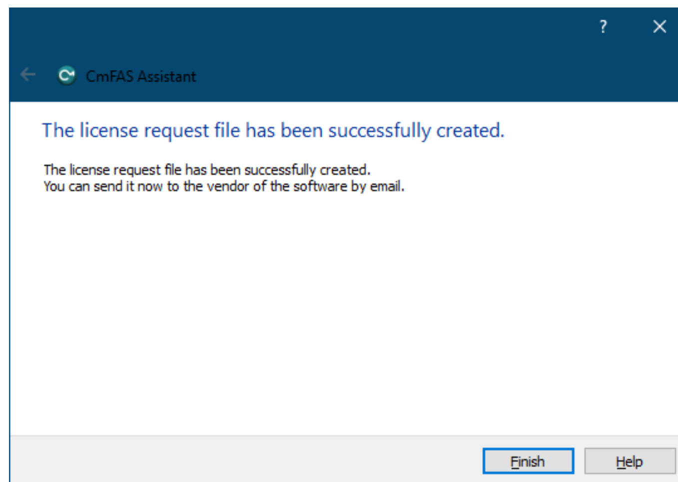
In next window one can choose which vendors' license is wanted to be updated. "Adaptive Vision Sp. z o.o." should be selected.



The last step of creation WibuCmRac file is to choose its name (default is fine) and localization.

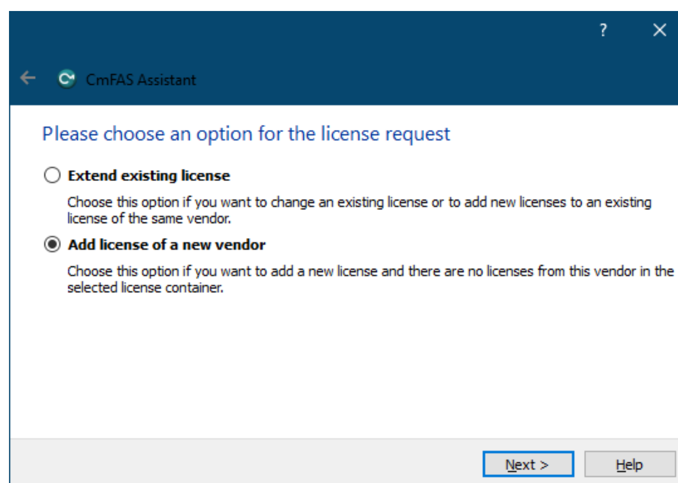


All settings need to be applied by clicking "Commit". If everything went well, this window will appear:

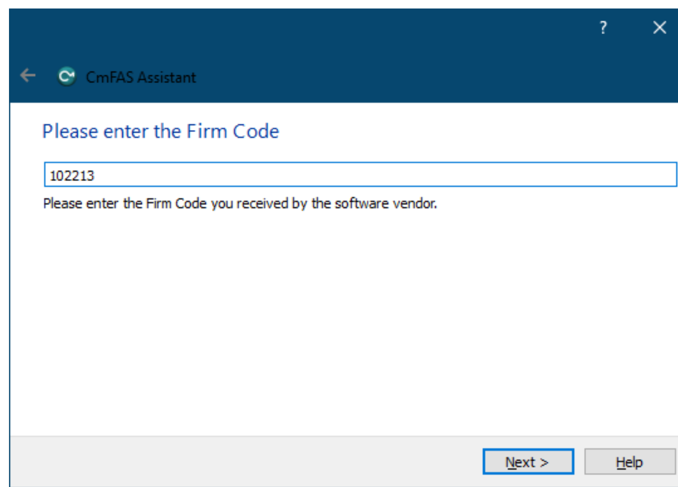


Now its time to send created file to Aurora Vision team.

If your dongle happens to be empty (no prior Aurora Vision license is programmed, and list with available producers is empty), you need to select "Add license of a new vendor" option in CmFAS Assistant:



In next window one will be prompted for entering the FirmCode of vendor. In case of "Adaptive Vision Sp. z o.o." one should type 102213, as shown on image below:



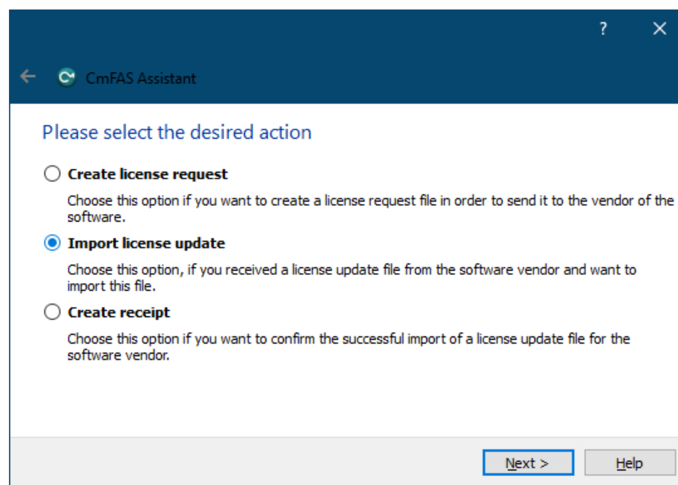
After committing these steps, request file will be generated as described earlier in this section.

Using WibuCmRau file to upgrade USB dongle

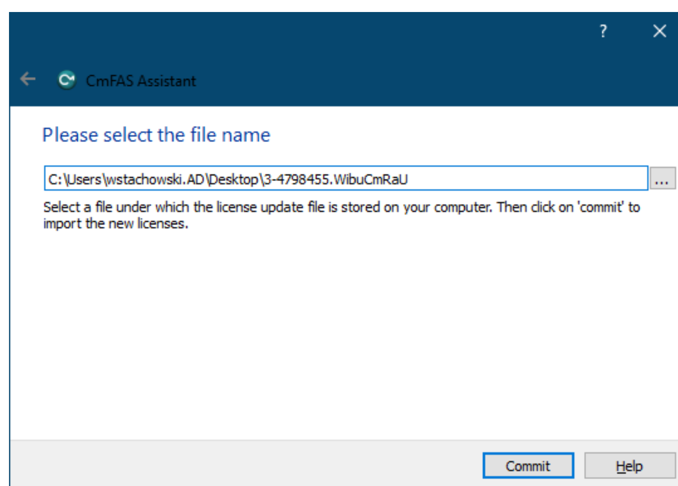
When WibuCmRau file is received, one can transfer it to USB dongle, which has to be present in one's computer's USB port.

Using CodeMeter Control Center to upgrade USB dongle

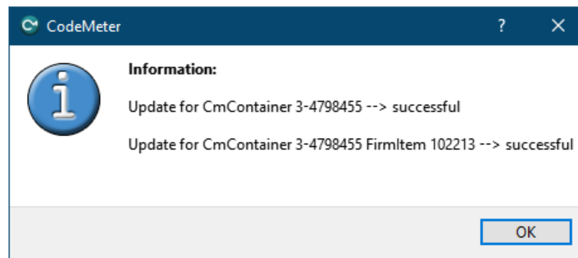
As previously, Control Center needs to be opened by clicking on its icon in tray. Dongle needs to be selected, and "License update" should be clicked. Note that WibuCmRau file can only be used once, and will work only with dongle, from which the corresponding WibuCmRac was created.



This time "Import license update" needs to be chosen.

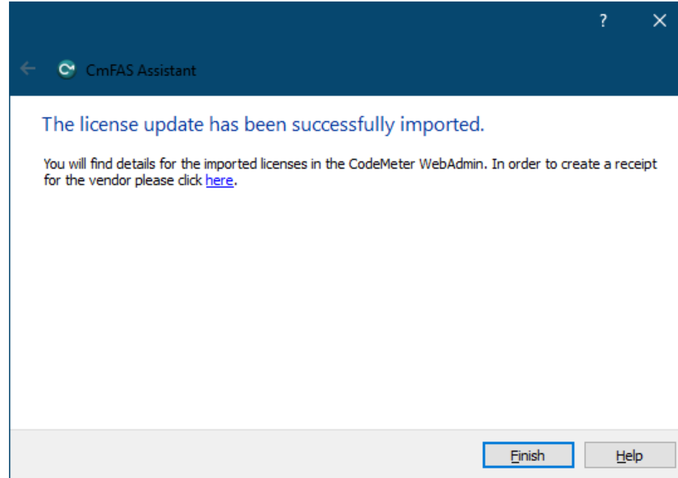


Navigate to file, which was received from Aurora Vision team, and click "Commit".



If everything went well, similar information will pop out.

Last window of assistant contains "Finish" button, which need to be clicked in order to exit.

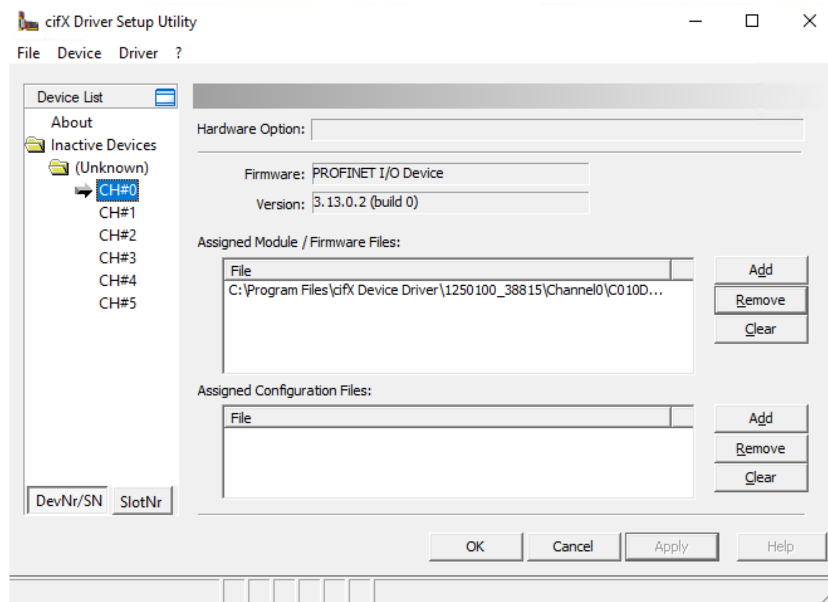


Working with Hilscher Devices

Introduction

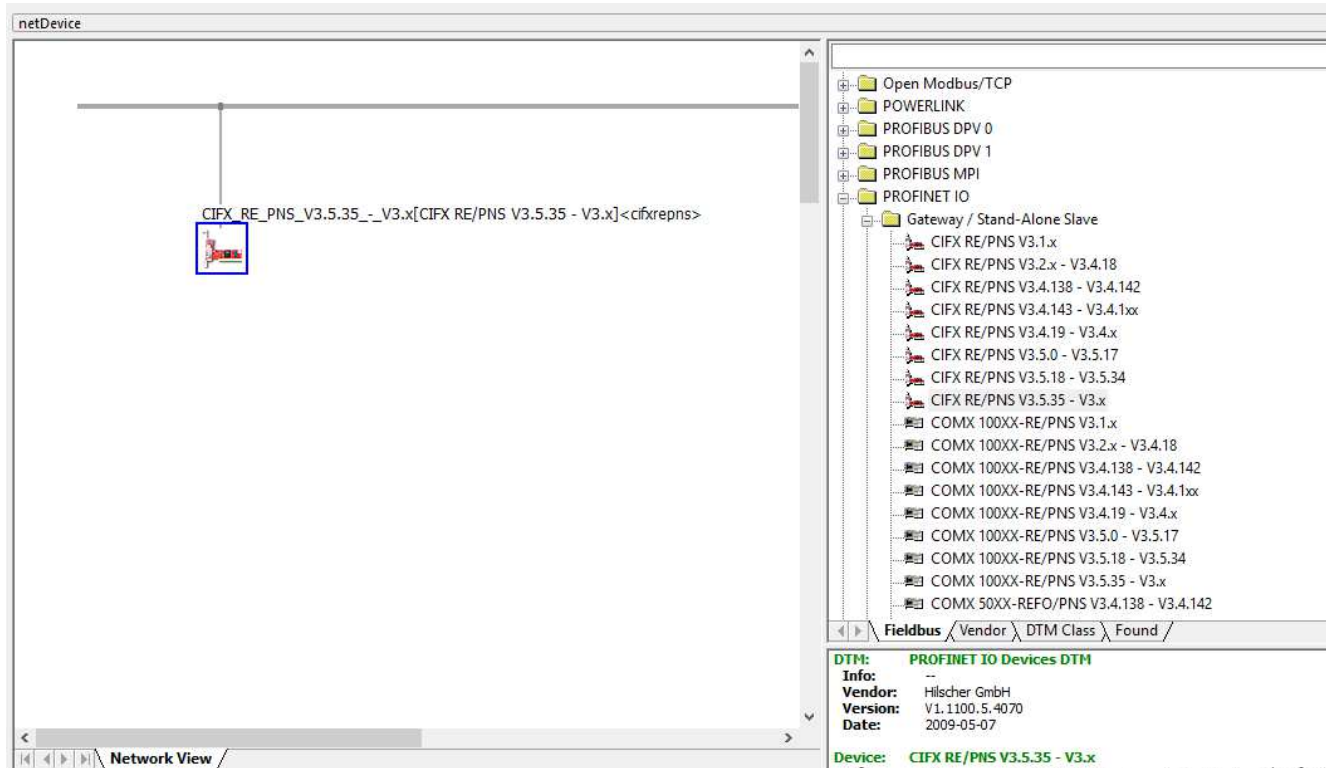
To work with a Hilscher device make sure that you have the recommended driver installed, newest SYCON.net, and firmware prepared. Profinet requirements are written down in Hilscher_Channel_Open_Profinet documentation.

The easiest way to install firmware to the cifX Driver is using the cifX Driver Setup software - the Windows Search in the start menu should find this tool, otherwise it is located in `C:\Program Files\ cifX Device Driver` directory. In the cifX Driver Setup Utility select the channel you want to configure (usually channel 0 - CH#0) and remove preexisting firmware by clicking "Clear". To add a new firmware file click "Add", navigate to the downloaded profinet slave firmware and click "Open". Finally, click "Apply" - the button should turn disabled.



Generating channel configuration files

Channel configuration files are generated in SYCON.net software. Firstly, pick a proper slave device using a PCI card, here we use the Profinet Stand-Alone Slave, and drag it to the gray bus on the "Network View".



To open card configuration, double click on its icon and select the "Configuration/Modules" category in the Navigation Area on the left side of the dialog. In the main window you can add individual modules. **Remember that in Profinet, Slot configuration must match the configuration from your master device, otherwise the connection will not work.** For boolean indicators we recommend "1 Byte Output" or "1 Byte Input".

netDevice - Configuration CIFX_RE_PNS_V3.5.35_-_V3.x[CIFX RE/PNS V3.5.35 - V3.x]<cifxrepns>

IO Device: CIFX RE/PNS V3.5.35 - V3.x
Vendor: Hilscher Gesellschaft für Systemautomation mbH
Device ID: 0x0103
Vendor ID: 0x011E

Navigation Area

- Settings
  - Driver
    - netX Driver
  - Device Assignment
  - Firmware Download
- Configuration
  - General
  - Modules
  - Signal Configuration
  - Address Table
  - Device Settings
- Description
  - Device Info
  - Module Info

Modules

Slot	Sub Slot	!	Module
0			CIFX RE/PNS V3.5.35 - V3.x [1250.100]
1			1 Byte Input
2			2 Bytes Input
3			4 Bytes Output
4			1 Byte Output
5			1 Byte Output
6			1 Byte Output

Add Module | Add Submodule | Remove

Use of slots: 7/256
State of data length: Input 3/1440 Octets, Output 7/1440 Octets, In-Output 10/2880 Octets

Submodule details
Dataset: I/O data
Display mode: Decimal

Direction	Consistence	Data type	Text ID	Length

OK | Cancel | Apply | Help

Disconnected | Data Set

You can see the final address table (addresses on the Hilscher device where input or output data will be available) in the "Configuration/Address table" category. If you plan to use IORead and IOWrite filters, for example Hilscher_Channel_IORead_SInt8, note the addresses. You can switch the display mode to decimal, as Aurora Vision Studio accepts decimal addressing, not hexadecimal. Aurora Vision Studio implementation of Profinet checks whether the address and data size match. In the sample configuration below, writing a byte to address 0x002 would not work, because that module address starts at 0x001 and spans 2 bytes. Moreover, Aurora Vision Studio prohibits writing with a SlotWrite filter to input areas, and reading with a SlotRead filter from output areas. Click "OK" when you have finished the proper configuration.

netDevice - Configuration CIFX_RE_PNS_V3.5.35_-_V3.x[CIFX RE/PNS V3.5.35 - V3.x]<cifxrepns>

IO Device: CIFX RE/PNS V3.5.35 - V3.x
Vendor: Hilscher Gesellschaft für Systemautomation mbH
Device ID: 0x0103
Vendor ID: 0x011E

Navigation Area
- Settings
  - Driver
    - netX Driver
  - Device Assignment
  - Firmware Download
- Configuration
  - General
  - Modules
  - Signal Configuration
  - Address Table
  - Device Settings
- Description
  - Device Info
  - Module Info

Address Table

Display mode: Hexadecimal | CSV Export

Inputs:

Module	Submodule	Type	Length	Address
4 Bytes Output	4 Bytes Output	IB	0x0004	0x0000
1 Byte Output	1 Byte Output	IB	0x0001	0x0004
1 Byte Output	1 Byte Output	IB	0x0001	0x0005
1 Byte Output	1 Byte Output	IB	0x0001	0x0006

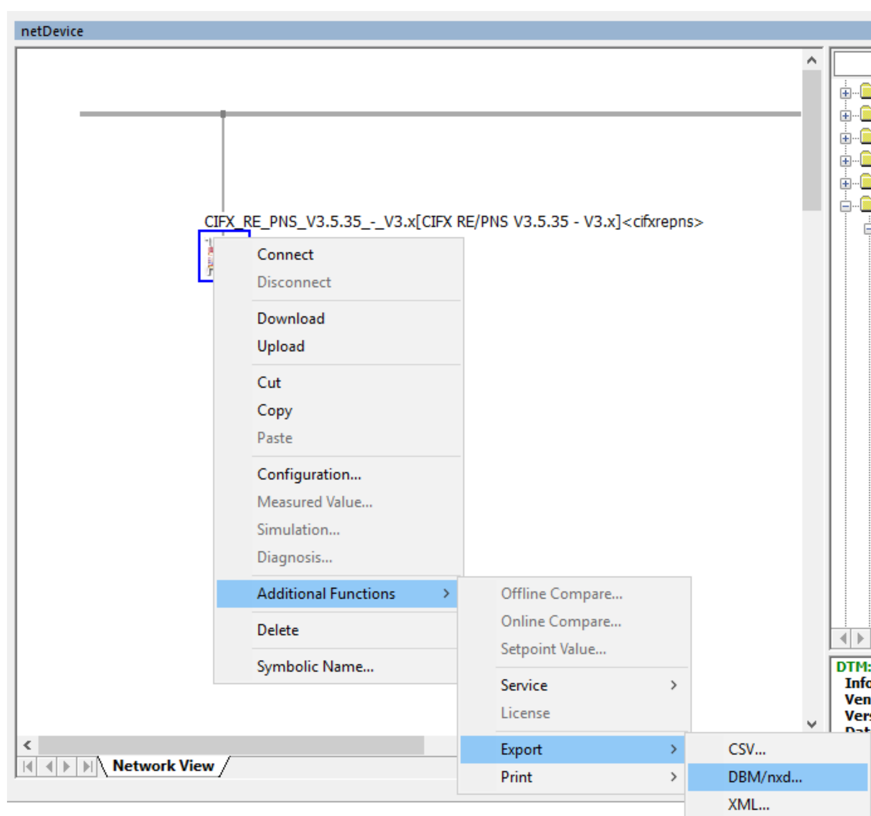
Outputs:

Module	Submodule	Type	Length	Address
1 Byte Input	1 Byte Input	QB	0x0001	0x0000
2 Bytes Input	2 Bytes Input	QB	0x0002	0x0001

OK | Cancel | Apply | Help

Disconnected | Data Set

The final step is to generate configuration files for Aurora Vision Studio. You can to this by right clicking on the device icon, then navigating to "Additional Functions->Export->DBM/nxd...", entering your configuration name and clicking "Save". You can now close SYCON.net for this example, **remember to save your project before, so that it is easier to add new slots.**



Filters in Aurora Vision Studio

At the beginning of every program where you want to use filters intended for communication over a Hilscher card you need to add Hilscher_Channel_Open_Profinet filter. The configuration files generated in the previous step are now required in inConfig (`xxx.nxd`) and inNwid (`xxx_nwid.nxd`) properties of that filter. These files are used when there is no connection between a card and a Profinet master. In that case Aurora Vision Studio updates the card configuration and starts the communication. For IO, we recommend SlotRead and SlotWrite filters, as they are more convenient. For example, the Hilscher_Channel_SlotWrite_SInt8 filter writes 8 bytes of signed data to the selected slot. Slot numbers match those in the "Configuration/Modules" category of card configuration in SYCON.net program.

Zebra
Aurora™ Vision

This article is valid for version 5.3.4
©2007-2023 Aurora Vision