User Guide

Version 3.9

Fifth Edition

Jan 2026

A&B Software LLC
New London, CT 06320
USA

www.ab-soft.com
support@ab-soft.com

Table of Contents

1 Introduction

GigESim is a software package that includes a GigE Vision camera simulator and GigE Vision server SDK.

A computer running *GigESim* presents itself to the network as a GigE Vision and GenICam compliant camera. Any GigE Vision compliant software running at the same or other computers on the network will treat *GigESim* as an actual camera with adjustable features. The simulator can be used for prototyping a GigE Vision camera as well as performing GigE client application development and testing when actual cameras are not available.

GigESim can also be used as a camera type converter by turning cameras of different types into virtual GigE Vision cameras accessible from any computer on the network, with an ability of remote control over camera parameters. The main advantage of such a type conversion is the multicast feature of the GigE Vision standard which lets the video to be transferred to multiple computers on the network.

Finally, *GigESim* includes a powerful GigE Vision Server SDK that allows developers to assign images generated by their applications to one or several virtual GigE Vision cameras and stream their images to the network for further processing and analysis. Developers can define individual features provided by such simulated cameras and exercise a full remote control over each feature. The images can be received on any computer on the network by any standard GigE Vision viewer or SDK such as A&B Software's ActiveGigE.

In general, with *GigESim* you can :

- Make your computer behave like a camera fully compatible with the GigE Vision and GenICam standards.
- Switch between GEV 1.2 and GEV 2.0 standards.
- Create and run several virtual camera objects on one computer host.
- Perform the development and testing of a client GigE Vision application while an actual camera is not available.
- Stream individual images or video sequences to a remote computer and receive them with standard GigE Vision software.
- Utilize 1 Gbit and 10 Gbit network equipment.
- Extend your throughput by using several network connections in parallel.
- Stream GigE Vision video over Wi-Fi network.
- Assign arbitrary IP and MAC addresses to your virtual cameras.
- Transmit video data from each of your virtual cameras to multiple computers on the network in the multicast mode or by utilizing several stream channels.
- Convert an RGB color video to Bayer raw or YUV format to reduce the image payload size.
- Select from dozens of monochrome and color pixel formats including packed ones.
- Automatically compress outgoing image frames by utilizing built-in JPEG and H.264 encoders.
- Add GenICam-compatible features to your virtual camera and control them from remote computers.
- Integrate chunk data into each frame and associate them with GenICam chunk features.
- Send data events from the virtual camera to remote clients on the message channel.
- Receive action commands and scheduled action commands in accordance with IEEE 1588 Precision Time Protocol.
- Select generated patterns, static images (bmp, jpeg, tif) and AVI files as the video source.
- Convert cameras of standard types (analog, USB, 1394, CameraLink) into virtual GigE Vision cameras.
- Implement distributed image processing on several computers in the parallel or/and sequential modes.

- Prototype the development of a GigE Vision camera.
- Adjust the interpacket delay as well as camera response timing.
- Simulate rigorous network conditions by disordering and skipping UDP packets.

This document gives a detailed description of *GigESim* and explains how to use it to perform the most common tasks.

1.1 License Agreement

This legal document is an agreement between you, the Licensee, and A&B Software LLC. By installing *GigESim* software on your computer, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened package, together with all the other material which comprises the product, respectively delete all *GigESim* related files.

1. Subject of agreement

The subject of this agreement is the software *GigESim*, the operating manuals, and all other accompanying material. It will be referred to henceforth as *GigESim SDK*.

2. Grant of license

A&B Software LLC grants the Licensee a non-exclusive, non-transferable, personal and worldwide license to use one copy of *GigESim SDK* in the development of an end-user application, as described in section 3 (below). This license is for a single developer/one computer and not for an entire company. If additional programmers wish to use *GigESim SDK*, additional copies must be licensed.

3. End user application

An *end user application* is a specific application program that is licensed to a person or firm for business or personal use. The files which are not listed under section 5 must not be included with the end user application. Furthermore, the end user must not be in a position to be able to neither modify the program, nor to create *GigESim SDK* based programs. Likewise, the end user must not be given the *GigESim SDK* serial number.

4. Royalties

The *GigESim SDK* is NOT royalty free. The cost and licensing issue breaks down into the cost of the development environment and the cost for each run-time license that must be shipped with any product that embeds *GigESim SDK*.

5. Redistributable files

The redistributable components of *GigESim SDK* are those files specifically designated as being distributable in the distributing your application section of GigESim User's Guide.

6. Trial version

A&B Software LLC at its discretion may grant the Licensee a non-exclusive license to test *GigESim SDK* for 7 days on one computer system using the Trial version. The Trial version must be used solely for the trial and evaluation of the Software. The Licensee is not permitted to use the Trial version for any other purpose, including without limitation any use of the Software for productive purposes, hardware testing or in the operation of any Licensee's business.

At the end of the evaluation period the Licensee shall promptly remove all coding and other vestiges of the Trial version from the computer system, and make no further use of it, except to the extent that may be permitted under any subsequent agreements between the Licensee and A&B Software LLC

7. Third party software

Certain third party software included with the Software is subject to additional terms and conditions imposed by third party licensor(s).

8. Copyright

The Software is the property of A&B Software LLC. A&B Software LLC reserves all rights to the publishing, duplication, processing and utilization of *GigESim SDK*. A single copy may be

made exclusively for security and archiving purposes. Without the express written permission of *A&B Software LLC* it is forbidden to:

- reverse engineer, emulate, alter, translate, decompile, or disassemble *GigESim SDK*
- copy *GigESim SDK's* accompanying written documentation
- lend, hire out or lease *GigESim SDK*.

A permanent transference of *GigESim SDK* is only permitted when the Licensee retains no copies and the recipient declares his acceptance of the conditions of this agreement.

9. Exclusion of warranties

A&B Software LLC offers and the Licensee accepts the product 'as is'. A&B Software LLC does not warrant *GigESim SDK* will meet the Licensee's requirements, nor will operate uninterrupted, nor error free.

10. Liability

With the exception of damage caused by willful or gross negligence, neither A&B Software LLC nor its distributors are responsible for any damage whatsoever which is put down to the use of *GigESim SDK*. This is valid without exception, including loss of profits, lost working time, lost company information or other financial losses. In any event the liability of A&B Software LLC is limited to the purchase price.

11. Duration of Agreement

This agreement is valid for an indefinite period of time. The Licensee's rights as a user automatically expire if the conditions of this agreement are in any way violated. In this event all data storage material and all copies of *GigESim SDK* are to be destroyed.

1.2 System Requirements

Hardware requirements:

- 2.5 GHz Dual Core or better CPU recommended.
- 4 GB or more RAM recommended.
- One or more 1 Gbit or 10 Gbit Ethernet boards or notebook cards installed on the system.

Note - While it is possible to transfer images through Ethernet and Fast Ethernet ports, which support 10 Mbit/s and 100 Mbit/s respectively, this will only work at slow frame rates and small resolutions unless image compression formats are used. It is highly recommended that you use a Gigabit Ethernet Network Interface Controller.

Note - For 10 GigE video streaming a 3.5 GHz i7 or better CPU should be used.

Software requirements:

- Windows XP, Vista, Windows 7, Windows 10, Windows 11
- Linux Ubuntu 16.04 and later, Debian 10 and later, CentOS 7-9, OpenSUSE
- ARMv8 Ubuntu 16.04 and later, Debian 10 and later

1.3 Installation

To install *GigESim* under Windows, perform the following steps:

1. Save and exit out of all currently running applications.

2. Insert *GigESim* CD into the CD-ROM drive or run a *GigESim* installer from a link provided by your vendor. The **Windows Installer** box with a status bar will appear while setup prepares to start the installation process.

3. After the setup program has verified your system has the appropriate installer files, you are ready to start installing *GigESim.* The **Welcome** dialog box will appear. After reading the preliminary information, click **Next**.

4. The **License Agreement** dialog box will appear. To accept the license and continue, click **I Agree**, and then click **Next**. Note that the **Next** button is not available until you click **I Agree**. If you do not accept the license, click **I Do Not Agree**, and setup will terminate.

5. The **Select Installation Folder** dialog box will appear. The default location of *GigESim* files is *C:\Program Files\GigESim*. If you want to change the location, enter the path for the desired folder and then click **Next**.

6. When **Confirm Installation** dialog box appears, click **Next**. *GigESim* will begin installing on your system.

7. Once installation is finished, the **Installation Complete** dialog box will appear. Click **Close** to exit the installer.

To install *GigESim* under Linux Ubuntu, perform the following steps:

1. Go to *System Settings -> Software&Updates -> Other Software*. Verify if the *Canonical Partners* box is checked. If not, enable it, in which case you will see additinal packages being installed. Close the windows.

2. Download *GigeSim-debian.run* installer from a link provided by your vendor.

3. Right click on the file, go to *Properties->Permissions*, enable *"Allow executing file as program"*.

4. Open the terminal window and change the current directory to the installation file location using the "cd" command.

5. Run the installer using the following command:

    sudo ./GigeSim-debian.run

    You will be requested an administrative password.

6. The installer will run and install all necessary dependencies. GigESim files will be installed in ./opt/GigeSim directory

1.4 Registration

To start using the software, you must acquire a design-time license from your distributor. The design-time license is provided in form of a license file *gigesim.lic*

On Windows OS the license file must be saved in the Windows folder (typically, C:/Windows). If you installed GigESim under Linux, place the license file in */opt/GigeSim* directory

In addition, you should perform a run-time registration. This is done through the following procedure. When you first start an executable file that was created using *GigESim*, the registration dialog box will appear. The dialog will display a Control ID uniquely generated for your system. Based on this ID, your distributor will provide your with a Serial Number that will validate a run-time permission for *GigESim.* After the proper Serial Number is entered in the dialog and registration completed, all *GigESim* based application will become unlocked.

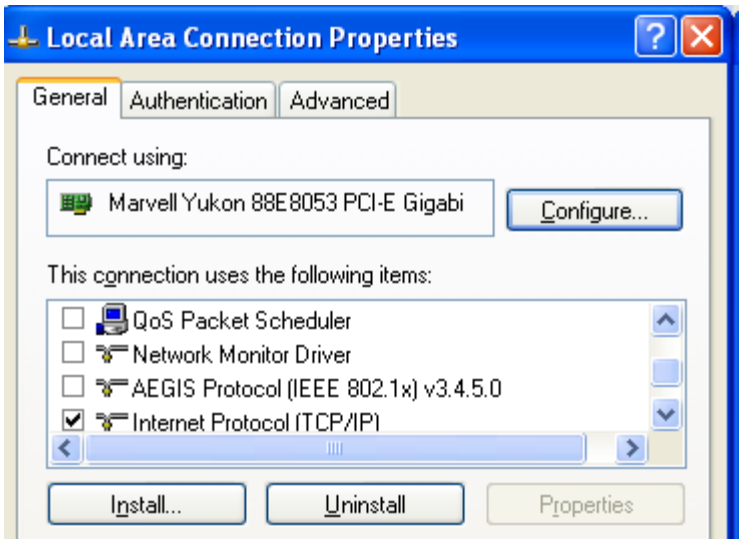
To register GigESim on Linux, run a *GigESim* based application from the terminal window using *sudo* command. You will be given a Control ID and will have to enter a matching Serial Number obtained from your distributor. Once you register GigESim, you will be able to run GigESim based applications without an administrative permission.

1.5 Network Setup

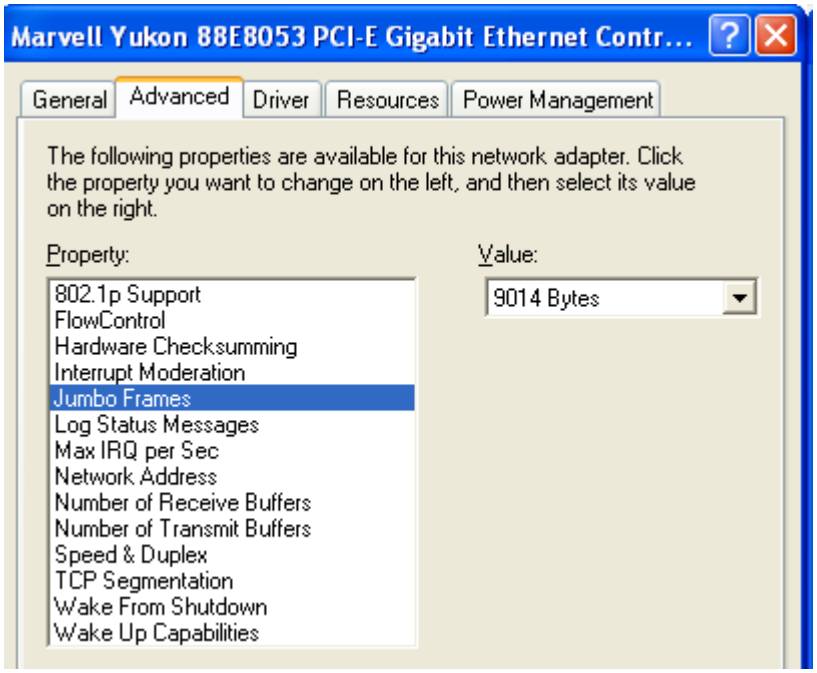
1.Make sure a Gigabit Ethernet card is installed on your system. To achieve the best performance, use a card that supports "Jumbo" frames of at least 9 KB size. You should only use a default Windows driver or the driver provided by the manufacturer of your network card.

Attention - do not use special drivers (such as "high-performance driver") provided by some camera manufacturers. Those drivers use their own protocols not compatible with GigESim and would prevent GigESim-based software from recognizing your network card.

2. From the **Start** menu open **Control Panel**, then click **Network Connections**. Right click the network connection which will be used with your camera. From the context menu select **Properties**. This will display **Local Area Connection Properties** window.

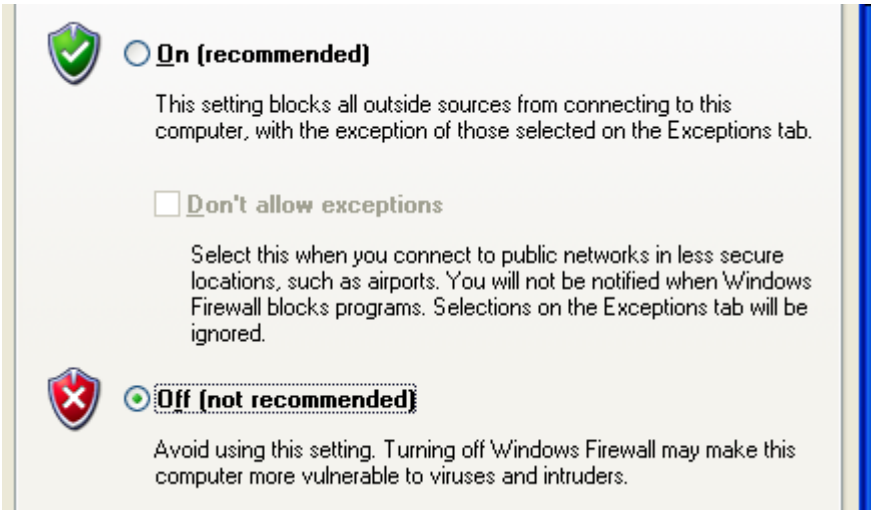


3. Click **Configure** button. The adapter properties window will be displayed. Select the **Advanced** tab. This will bring up the **Property** list which will be different depending on the model and type of the network card. Select **Jumbo Frames** and change the value to the maximum size allowed. If the **Jumbo Frames** option does not appear in the list, your network card might not support it, in which case the performance of the camera will be reduced.

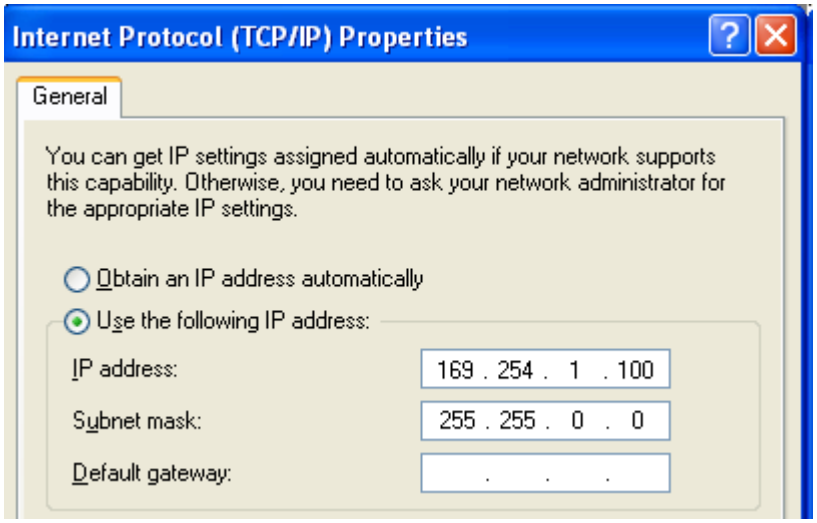


4. If the **Transmit Descriptors** (or **Transmit Buffers**) option is available in the list, change it to the highest possible value. Set **Max IRQ per Second** (if available) to 1000. Set **Interrupt Moderation** (if available) to **On**. Set **Interrupt Moderation Rate** (if available) to **Adaptive**. Click **OK**.

5. Reopen **Local Area Connection Properties** by right-clicking the network connection icon in the **Network Connections** window and selecting **Properties**. Select the **Advanced** tab at the top. Click the **Settings** button. The **Windows Firewall** window will be displayed. Select **Off** to turn off the Windows firewall. The camera will not communicate with the system if the firewall is active.



6. Reopen **Local Area Connection Properties** by right-clicking the network connection icon in the **Network Connections** window and selecting **Properties**. From the list of items select **Internet Protocol (TCP/IP)** and click the **Properties** button. The **Internet Protocol (TCP/IP) Properties** window will be displayed. Select **Use the following IP** address and enter **169.254.1.100** in the **IP address field**. Enter **255.255.0.0** in the **Subnet mask** field. Click **OK** to save your changes.



Note - the network card IP address can be any 169.254.xxx.yyy value as long as it is unique in your local network.

*Note - the configuration procedure above assumes that your virtual camera has the IP address in the **169.254.x.x** class B range, which is an address space assigned to GigE Vision cameras. If you use a different IP address, make sure that your client GigE Vision software is configured accordingly.*

7. Back in the **Local Area Connection Properties** window, in the list of items used by the connection, unselect every item except **Internet Protocol (TCP/IP)**, then click **OK**. The card is now fully configured for use with *GigESim*.

Note - some Gigabit network adapters have a limited transmit performance and are only capable at transmitting data at 600-700 Mbit/sec. Make sure to use a high-performance network card such as Intel PRO/1000 CT to get the transmit speed close to 1 Gbps.

Note - to transfer data at a speed higher than 1 Gbit/sec you have to use 10G adapters on both server (GigESim) and client side. The packet size for both adapters must be set to its highest value (typically 16000+) and both systems should utilize have high-performance CPUs.

Note - to enable higher packet sizes in Linux, perform the following steps:

- *open the terminal window*
- *display the list of network interfaces using ifconfig -a*
- *locate the interface to which GigESim will be connected (e.g. enp5s0)*
- *to set the maximum supported packet size to 8000, enter the following command:*
  ifconfig enp5s0 mtu 8000

1.6 Working with multiple cameras

If you want to run several virtual cameras on the same computer system, there are two ways of doing it:

1. You can use multiple network cards with each virtual camera connected to a separate network interface. This method is recommended when you want to extend your total bandwidth beyond the 1 Gigabit limit of an individual network card.

Before running your multiple camera application, you should configure your network for multiple camera setup. Each network card associated with a simulated camera must be assigned a unique subnet, preferably from the GigE Vision address space. For the two-camera connection through a pair of network adapters follow the procedure below.

a) Make sure two Gigabit Ethernet cards are installed on your GigE server system.

b) Refer to steps 1-7 of Network Setup and configure each network card with the following addresses:

Network card 1:	IP address: 169.254.100.1	Subnet mask: 255.255.255.0
Network card 2:	IP address: 169.254.200.1	Subnet mask: 255.255.255.0

Your system is now configured for running two virtual GigE Vision cameras.

c) Make sure that the IP addresses and subnet masks of the network cards installed on your client computer(s) match the IP addresses above. This check is not needed if the client applications are used on the local host.

d) Check your setup by running two instances of GigEmulator.exe, selecting Network card 1 for instance #1 and Network card 2 for instance #2, then clicking Connect. Your local or remote client application should be able to detect and operate two *GigESim* cameras.

e) If you are using *GigESim* as an SDK, creating two virtual GigE Vision cameras is as easy as instantiating two CGevCamera objects and connecting each one to a separate network adapter. See C++ API Reference for more details.

2. You can connect several virtual cameras to one network interface provided each of them is assigned a unique IP and MAC address. Since all the cameras will share the same network bandwidth, using this method is recommended for low-resolution or low-speed applications. It can also be used for high-speed applications when several network cards are teamed into one virtual NIC by means of the link aggregation.

For the multiple-camera connection through one network card follow the procedure below:

a) Make sure to use a separate Gigabit Ethernet card for your GigE client-server connection unless you are running server and client applications on the same local host. Theoretically it is possible to use your local area connection for the image transfer, but doing that is not recommended since the operation of your applications can be heavily affected by a traffic on your local network.

b) Refer to steps 1-7 of Network Setup to configure your Gigabit network card for the optimal performance.

c) Make sure that the IP addresses and subnet masks of the network cards installed on your client computer(s) match the IP address of your server NIC. This check is not needed if the client applications are used on the local host along with your virtual camera applications.

d) Verify your setup by running two or three instances of GigEmulator.exe, checking the Forced IP box and entering a unique IP addresses for each instance of the emulator. For example, if your NIC has IP address 169.254.100.1 and subnet mask 255.255.0.0, you can use the following IP configuration for a pair of virtual cameras:

Camera 1:	IP address: 169.254.200.1	Subnet mask: 255.255.0.0
Camera 2:	IP address: 169.254.200.2	Subnet mask: 255.255.0.0

In addition, you should assign a different MAC address to each instance of the emulator using the MAC Address text box. It is also recommended to assign different serial numbers to different instances.

Note - the latest version of GigEmulator supports an automatic execution of the above procedure for each instance of the application. All you have to do to simulate several GigE Vision cameras is run several instances of GigEmulator.

e) Click the Connect button on each instance of the emulator. Your local or remote client application should be able to detect and operate several *GigESim* cameras.

f) If you are using *GigESim* as an SDK, using two virtual GigE Vision cameras on one network adapter is as easy as instantiating two CGevCamera objects and assigning each of them a unique IP address, MAC address and serial number. See C++ API Reference for more details.

Note that you can use a combination of both methods by sharing some network connections between several virtual cameras and assigning individual connections to other virtual cameras. You can also stream images from any virtual camera to multiple client locations by using several stream channels or by activating the multicast mode.

1.7 Distributing your application

If you create your own applications using *GigESim SDK* under Windows, your distribution package should include *gigesimsdk.dll* for 32-bit application and/or *gigesimsdk64.dll* for 64-bit applications.

If you deploy GigESim-based applications under Linux, your distribution package should include libgigesimsdk.so library. The default installation of GigESim places this library into /usr/lib/GigeSimSDK/

When a *GigESim* based application is executed for the first time on an end-user's system, it will display the registration dialog (see Registration). You should instruct the user to provide you or your distributor with a Control ID displayed in the dialog. After the run-time license of the user is validated, the distributor will issue a unique serial number which will unlock the control on the user's machine.

2 GigEmulator application

GigEmulator is a GUI application that behaves as a standard GigE Vision camera with a selectable version of the standard (GEV 1.2 or GEV 2.0), variable image size, multiple supported pixel formats, adjustable frame rate and exposure time. The simulated camera supports two streaming channels, multicast mode, chunk data, image compression (JPEG and H.264). It allows you to send message events to remote clients, both in the manual and automatic modes. It can also simulate a linescan camera with dynamically changing image height.

GigEmulator can be configured to output a generated moving pattern or video frames from an AVI file in a continuous loop. It can work in the free-run mode or trigger mode. Triggering can be performed via the Software Trigger or Action commands.

The application is very easy to use. You only have to select a network card which the emulator will be bound to and click the Connect button. This will make the virtual camera immediately available to all GigE Vision client applications on the network (such as A&B Software's *ActiveGigE*). Once a client application issues the *AcquisitionStart* command, the simulated video from *GigEmulator* will appear on the client application's screen.

If you need to simulate several GigE Vision cameras using one computer, just run several instances of *GigEmulator*. Each of them will be automatically assigned a unique IP address, MAC address and serial number, essentially providing you with several virtual cameras operating on the same network interface. Alternatively, you can manually bound each instance of *GigEmulator* to a separate network interface. See Working with multiple cameras for more details.

If you develop your own GigE Vision sever application using *GigESim SDK*, it is recommended to run *GigEmulator* first to make sure that all network components are configured correctly and the targeted network performance is achieved.

To start the emulator on Windows, run "GigEmulator.exe" from *GigESim* working folder. You can also run it through the Windows Start Menu: Start -> All Programs -> GigESim -> GigEmulator.

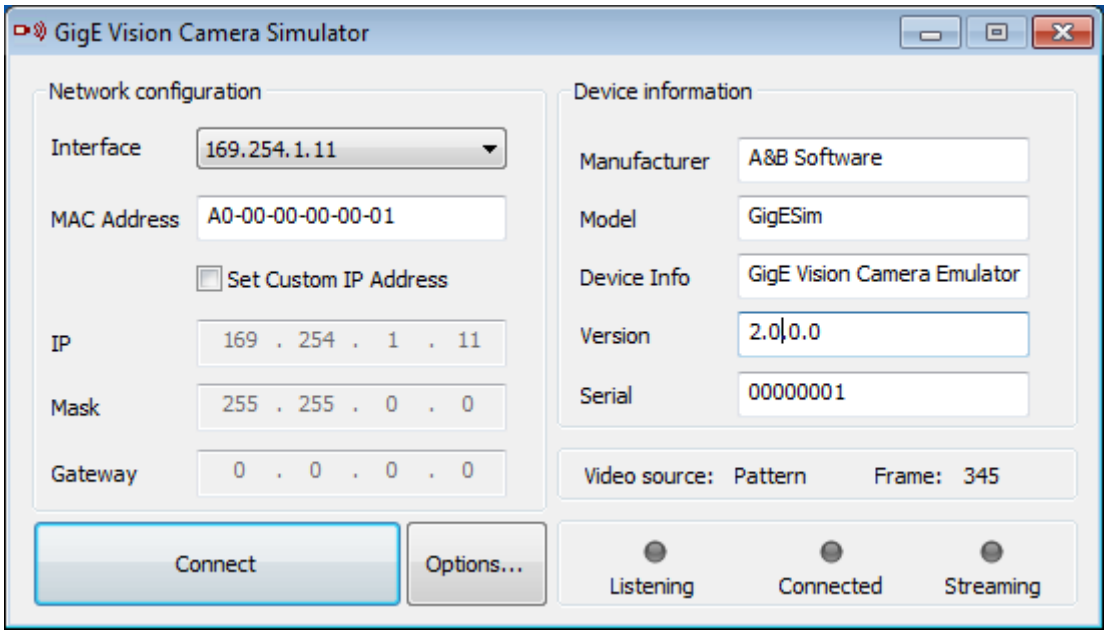
To start the emulator on Linux, run "GigEmulator" from .opt/gigesim/. For the first run use the terminal window and "sudo" command:
sudo ./GigEmulator

For the information on GigEmulator's user interface and functionality refer to the following topics:

User interface
General options
Advanced options

2.1 User Interface

When you start *GigEmulator*, the following user interface will be displayed.



Select from the following options:

Interface
Lets you select the network interface to which the emulated camera will be bound. The box contains the list of all network interfaces found in the system and listed by their IP addresses.

Custom IP
Check this box if you want the virtual camera's IP address to be different from the IP address of the network adapter it is bound to. If this option is selected, the *IP Address*, *Mask* and *Gateway* fields will become available. Make sure that the newly assigned IP address matches the subnet of the network adapter, or otherwise a client application may not be able to establish connection with the emulator. Using this option is necessary when you need to run several instances of the GigEmulator on the same network interface. See Working with multiple cameras for more details.

MAC Address
Use this option to enter the MAC address which will be assigned to the virtual camera. When several instances of the emulator are connected to the same network interface, their MAC addresses should be different.

Manufacturer
Use this option to simulate the name of the manufacturer associated with your virtual camera.

Model
Use this option to simulate the model name of the virtual camera.

Device Info
Use this option to enter the device information to be available to a client application.

Version
Use this option to enter the version of the device associated with the virtual camera.

Serial number
Use this option to enter the serial number of the virtual camera. When several instances of the emulator are used, it is recommended to assign each of them a different serial number.

Connect
Click this button to bind the emulator with the selected network card and bring the virtual camera online.

Options
Lets you configure the optional settings of the emulator. See General options and Advanced options for more details.

Video source
Shows the currently selected video source. See Options for more details.

Frame
Shows the index of the video frame being streamed at the moment on stream channel #0 and #1.

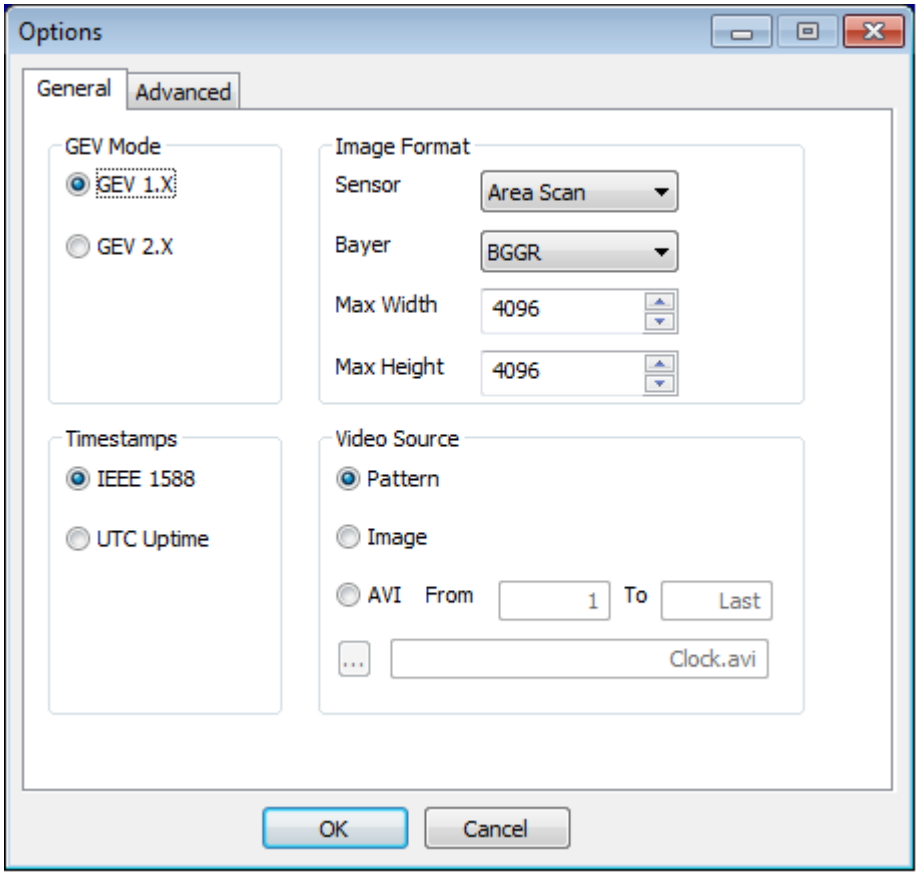
Listening
Indicates that the emulator is connected to the network interface and ready to communicate with GigE Vision clients.

Connected
Indicates that the emulator has established connection with at least one GigE Vision client.

Streaming
Indicates that the emulator is currently streaming video data. The green light corresponds to stream channel #0, blue light to stream channel #1, yellow light to both stream channels working in parallel.

2.2 General options

When you click the Options button on the GigEmulator main screen and select the General panel, the following interface will appear



GEV Mode
Lets you select the version of the standard under which GigEmulator will operate:

GEV 1.x
GigEmulator will operate in accordance with GigE Vision 1.2 specifications.

GEV 2.x
GigEmulator will operate in accordance with GigE Vision 2.0 specifications.

Timer
Lets you select the operational mode of GigEmulator's internal timer:

IEEE 1588
Internal camera timer is synchronized with the time elapsed since 00:00:00 January 1, 1970.

UTC Uptime
Internal camera timer is synchronized with the time elapsed since GigEmulator has started.

Sensor
Lets you select the sensor type of the virtual camera:

Area Scan
GigEmulator will simulate an area scan camera by streaming images of a constant horizontal and vertical size.

Line Scan
GigEmulator will simulate a line scan camera by streaming images of a constant horizontal and variable vertical size.

Bayer
Lets you select the sensor layout for the raw-Bayer type of camera:

BGGR - GigEmulator will simulate a camera with the BGGR sensor layout and support BayerBG formats.
RGGB - GigEmulator will simulate a camera with the RGGB sensor layout and support BayerRG formats.
GBRG - GigEmulator will simulate a camera with the GBRG sensor layout and support BayerGB formats.
GRBR - GigEmulator will simulate a camera with the GRBR sensor layout and support BayerGR formats.

Max Width
Lets you set the maximum horizontal size of images streamed by GigEmulator. The value of the Width feature will be limited by this setting.

Max Height
Lets you set the maximum vertical size of images streamed by GigEmulator. The value of the Height feature will be limited by this setting.

Video Source
Lets you select one of the following video sources:

Pattern
Generates a moving pattern of a monochrome or color wedge. One of the following pixel formats can be selected on a client application side: Mono8, Mono10, Mono10Packed, Mono12, Mono12Packed, Mono14, Mono16, Bayer8, Bayer10, Bayer10Packed, Bayer12, Bayer12Packed, Bayer16, YUV411_8_UYYVYY, YUV422_8_UYVY, YUV8_YUV, RGB8. In addition, the type of the pattern (vertical or horizontal) can also be selected on the client side through the TestPattern feature.

Image
Repeatedly streams the static image from a specified image file. Supported file formats are bmp, jpg and tif.

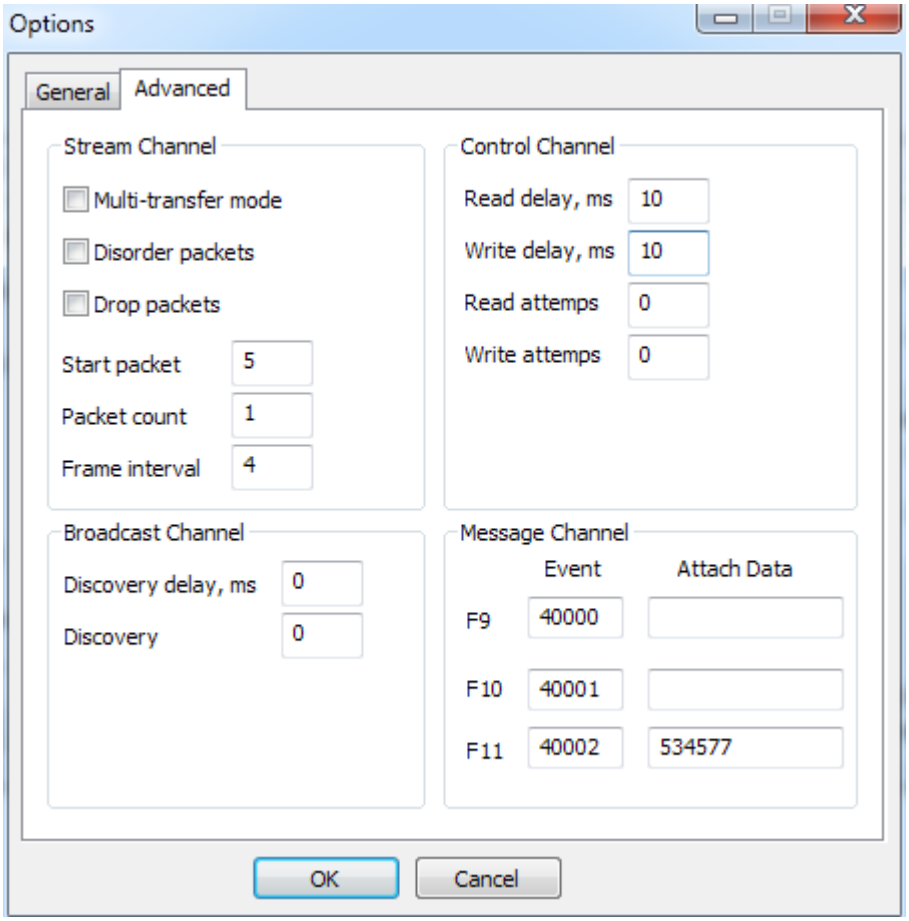
AVI
Streams images from a specified AVI file in a continuous loop. One of the following pixel formats can be selected on a client side: Mono8, RGB8, Bayer8.

From...To
Lets you set the index of the starting and ending frame in the AVI file that will be used in the continuous streaming loop.

File
Lets you select the name of the image file or AVI file whose content will be used as the video source.

2.3 Advanced options

When you click the Options button on the GigEmulator main screen and select the Advanced panel, the following interface will appear



Multi-transfer mode

Select this option if you use several instances of GigEmulator, each one connected to a separate network adapter. Note that using this mode for a single stream case may cause a link speed reduction on Windows 7 and Windows XP.

Disorder packets

Select this option to make the video packets transmitted in a random order instead of the sequential one. Using this option is recommended only for testing the robustness of GigE Vision client applications.

Drop packets

Select this option to make the emulator drop some video packets. Using this option is recommended only for testing the robustness of GigE Vision client applications. When this box is selected, the following options will become available:

Start packet

The number of the first packet to drop.

Packet count.
The amount of packet to drop in each frame.

Frame interval
The interval between dropped packets.

Discovery delay
Lets you set the time in milliseconds which will take the virtual camera to respond to a discovery request from a client application. Using this option is recommended only for simulating a camera with a slow discovery response. If this is not your intention, this value should be 0.

Discovery attempts
Lets you set the number of repetitions of each discovery request necessary for the virtual camera to respond with a discovery message. Using this option is recommended only for simulating erroneous conditions on the network. If this is not your intention, this value should be kept 0.

Read delay
Lets you set the time in milliseconds which will take for the virtual camera to reply to a read command. Using this option is recommended only for simulating a camera with a slow response. If this is not your intention, this value should be kept 0.

Write delay
Lets you set the time in milliseconds which will take for the virtual camera to respond to a write command. Using this option is recommended only for simulating a camera with a slow response. If this is not your intention, this value should be kept 0.

Read attempts
Lets you set the number of repetitions of each read request necessary for the virtual camera to respond. Using this option is recommended only for simulating erroneous conditions on the network. If this is not your intention, this value should be kept 0.

Write attempts
Lets you set the number of repetitions of each write request necessary for the virtual camera to respond. Using this option is recommended only for simulating erroneous conditions on the network. If this is not your intention, this value should be kept 0.

Read attempts
Lets you set the number of repetitions of each read requests necessary for the virtual camera to respond. Using this option is recommended only for simulating erroneous conditions on the network. If this is not your intention, this value should be 0.

Messages
Lets you assign event IDs and data to message events that will be generated by GigEmulator upon receiving F9, F10 or F11 key strokes. Note that the GigEmulator window must be in focus in order for the key strokes to be processed.

Event
Enter the Event ID that will be assigned to a message generated upon the selected key stroke

Data
Enter the data value that will be included into an EventData message generated upon the selected key stroke. If this filed is left empty, a regular Event message will be generated.

2.4 GenICam features

GigEmulator virtual camera exposes the following set of GenICam features to a GigE Vision client (default values are shown in brackets):

DeviceControl Category

Name	Interface	Access	Description
DeviceType	IEnumeration	R	Returns the device type [Transmitter]
DeviceScanType	IEnumeration	R	Scan type of the camera's sensor (Areascan or Linescan)
DeviceVendorName	IString	R	Name of the manufacturer of the device [A&B Software]
DeviceModelName	IString	R	Model of the device [GigESim]
DeviceVersion	IString	R	Version of the device [2.5.0.0]
DeviceFiremwareVersion	IString	R	Version of the firmware in the device [2.5.0.0]
DeviceSerialNumber	IString	R	Device serial number [00000001]
DeviceUserID	IString	R/W	User-programmable device identifier [empty]
DeviceMaxThroughput	IInteger	R	Maximum streaming throughput of the device in bps (1000000000)

SourceControl Category

Name	Interface	Access	Description
SourceSelector	IEnumeration	W	Selects the video source to control (Source0 or Source1)
SourceCount	IInteger	R	Returns the number of available video sources [2]
SourceStreamChannel	IInteger	R	Returns the index of the stream channel associated with the selected source

ImageFormat Category

Name	Interface	Access	Description
SensorWidth	IInteger	R	Effective width of the sensor in pixels [4096]
SensorHeight	IInteger	R	Effective height of the sensor in pixels [4096]
WidhtMax	IInteger	R	Maximum width of the image in pixels (same as SensorWidth)
HeightMax	IInteger	R	Maximum height of the image in pixels (same as SensorHeight)
Width	IInteger	R/W	Width of the transmitted image in pixels (from 1 to SensorWidth)
Height	IInteger	R/W	Height of the transmitted image in pixels (from 1 to SensorHeight)
OffsetX	IInteger	R/W	Horizontal offset from the top left pixel of the sensor to the region of interest
OffsetY	IInteger	R/W	Vertical offset from the top left pixel of the sensor to the region of interest
PixelFormat	IEnumeration	R	Maximum streaming throughput of the device in bps [1000000000]
PixelCoding	IEnumeration	R	Coding of pixels in the image (Mono, MonoPacked, Raw, RawPacked, YUV)
PixelSize	IEnumeration	R	Size of a pixel in the image in bits (Bpp8, Bpp10, Bpp12, Bpp16, Bpp24)
TestPattern	IEnumeration	R/W	Selects the test pattern (VertialRampMoving or HorizontalRampMoving)
ImageCompressionMode	IEnumeration	R/W	Selects an image compression mode (Off, JPEG, H264)
ImageCompressionQuality	IInteger	R/W	Controls the quality of the produced compressed stream (0-100)
ImageCompressionBitrate	IFloat	R/W	Controls the rate of the produced compressed stream in kbps [5000.]

AcquisitionControl Category

Name	Interface	Access	Description
AcquisitionMode	IEnumeration	R/W	Sets acquisition mode of the camera (SingleFrame, MultiFrame, Continuous)
AcquisitionStart	ICommand	W	Starts the acquisition on the currently selected stream channel
AcquisitionStop	ICommand	W	Stops the acquisition on the currently selected stream channel
AcquisitionFrameCount	IInteger	R/W	Number of frames to acquire in the MultiFrame acquisition mode
AcquisitionFrameRate	IFloat	R/W	The acquisition rate in Hertz at which the frames are captured
TriggerSelector	IEnumeration	R/W	Select the type of trigger to configure (FrameStart or AcquisitionStart)
TriggerMode	IEnumeration	R/W	Enables or disables the selected trigger (On, Off)
TriggerSoftware	ICommand	W	Generates an internal trigger (TriggerSource must be set to Software)
TriggerSource	IEnumeration	R/W	Specifies the signal to be used as the trigger source (FixedRate, Freerun, Software, Action1)
TriggerActivation	IEnumeration	R	Specified the activation mode of the trigger [RisingEdge]
TriggerDelay	IFloat	R/W	The delay in microseconds between the trigger reception and its activation [0]
ExposureMode	IEnumeration	R/W	Sets the operation mode of the exposure (Off, Continuous, Once)
ExposureTime	IFloat	R/W	Sets the simulated exposure time (2.-10000.) [100.]

TransportLayerControl Category

Name	Interface	Access	Description
PayloadSize	IInteger	R	The number of bytes transferred for each frame on the current stream channel
GevStreamChannelSelector	Integer	R/W	Selects the stream channel to control (0 or 1)
GevVersionMajor	IInteger	R	Major version of the GEV specifications (1 or 2)
GevVersionMinor	IInteger	R	Minor version of the GEV specifications (2 or 0)
GevDeviceModeIsBigEndian	IBoolean	R	The endianess of the device registers [True]
GevDeviceModeCharacterSet	IEnumeration	R	The character set used by strings in the bootstrap registers [UTF8]
GevLinkSpeed	IInteger	R	The speed of transmission in mbps [1000]
GevMacAddress	IInteger	R	MAC address of the virtual camera
GevCurrentIpAddress	IInteger	R	Current IP address of the virtual camera
GevCurrentSubnetMask	IInteger	R	Current subnet mask of the virtual camera
GevCurrentDefaultGateway	IInteger	R	Current default gateway IP address for the virtual camera
GevPersistentIpAddress	IInteger	R/W	Controls persistent IP address of the virtual camera
GevPersistentSubnetMask	IInteger	R/W	Controls persistent subnet mask of the virtual camera
GevPersistentDefaultGateway	IInteger	R/W	Controls persistent default gateway IP address for the virtual camera
GevNumberOfInterfaces	IInteger	R	The number of logical interfaces supported by the virtual camera [1]
GevStreamChannelCount	IInteger	R	The number of stream channels supported by the virtual camera [2]
GevMessageChannelCount	IInteger	R	The number of message channels supported by the virtual camera [1]

TransportLayerControl Category (continued)

Name	Interface	Access	Description
GevHeartbeatTimeout	IInteger	R/W	Controls the current heartbeat timeout in milliseconds [3000]
GevHeartbeatTickFrequency	IInteger	R	The number of timestamp ticks in one second [1000000000]
GevTimestampControlReset	ICommand	W	Resets the timestamp counter to 0
GevTimestampControlLatch	ICommand	W	Latches the current timestamp counter into GevTimestampValue
GevTimestampValue	IInteger	R	Returns the latched 64-bit value of the timestamp counter
GevCCP	IEnumeration	R/W	Controls the device access privilege of a client (OpenAccess, ExclusiveAccess, ControlAccess, ControlAccessSwitchoverActive)
GevPrimaryApplicatoinSwitchoverKey	IInteger	W	The key to use to authenticate primary application switchover requests
GevSCPHostPort	IInteger	R/W	Destination port to which the data will be sent on the current stream channel
GevSCPFireTestPacket	IBoolean	R/W	When this feature is set, the device will fire one test packet
GevSCPPacketSize	IInteger	R/W	Controls the packet size in bytes for the current stream channel [1500]
GevSCPD	IInteger	R/W	Controls the delay in timestamp units to insert between each packet [0]
GevSCDA	IInteger	R/W	Destination IP to which the data will be sent on the current stream channel
GevMCPHostPort	IInteger	R/W	Destination port to which the device must send messages
GevMCDA	IInteger	R/W	Destination IP address to which the device must send messages
GevSupportedOptionSelector	IEnumeration	R/W	Selects the GEV option to interrogate for existing support
GevSupportedOption	IBoolean	R	Returns True if the selected option is supported
GevSecondURL	IString	R	Indicates the second URL for the GenICam XML description file
GevSCPInterfaceIndex	IInteger	R	Index of the logical link to use on the current stream channel
TLParamsLocked	IInteger	R/W	Used by the Transport Layer to prevent critical features from changing during the acquisition. If 0, no features

EventControl Category

Name	Interface	Access	Description
EventSelector	IEnumeration	R/W	Selects which event to fire to the client
EventNotification	IEnumeration	R/W	Activates or deactivates the notification of the selected event (Off or On)
EventAcquisitionStart	IInteger	R	The unique identifier of the AcquisitionStart event [36501]
EventAcquisitionStartTimestamp	IInteger	R	Returns the 64-bit timestamp of the AcquisitionStart event
EventAcquisitionStartChannelID	IInteger	R	Returns the stream channel index associated with the AcquisitionStart event
EventAcquisitionStartFrameID	IInteger	R	Returns the identifier of the frame that generated the AcquisitionStart event
EventAcquisitionEnd	IInteger	R	The unique identifier of the AcquisitionEnd event [36502]
EventAcquisitionEndTimestamp	IInteger	R	Returns the 64-bit timestamp of the AcquisitionEnd event
EventAcquisitionEndChannelID	IInteger	R	Returns the stream channel index associated with the AcquisitionEnd event
EventAcquisitionEndFrameID	IInteger	R	Returns the identifier of the frame that generated the AcquisitionEnd event
EventFrameStart	IInteger	R	The unique identifier of the FrameStart event [36503]
EventFrameStartTimestamp	IInteger	R	Returns the 64-bit timestamp of the FrameStart event
EventFrameStartChannelID	IInteger	R	Returns the stream channel index associated with the FrameStart event
EventFrameStartFrameID	IInteger	R	Returns the identifier of the frame that generated the FrameStart event
EventFrameStartExposureTime	IFloat	R	Returns the exposure time at which the FrameStart event was generated
EventFrameTrigger	IInteger	R	The unique identifier of the FrameTrigger event [36500]
EventFrameTriggerTimestamp	IInteger	R	Returns the 64-bit timestamp of the FrameTrigger event
EventMessageF9	IBoolean	R/W	Turns on an event generation upon pressing F9 key (Off or On)
EventMessageF10	IBoolean	R/W	Turns on an event generation upon

ChunkDataControl Category

Name	Interface	Access	Description
ChunkSelector	IEnumeration	R/W	Selects which chunk to enable or control
ChunkEnable	IEnumeration	R/W	Enables the inclusion of the selected chunk data in the frame (Off or On)
ChunkTimestamp	IInteger	R	Returns the timestamp of the image included in the frame
ChunkExposureTime	IFloat	R	Returns the exposure time used to capture the image
ChunkAcquisitionFrameRate	IFloat	R	Returns the frame rate at which the image was captured
ChunkWidth	IInteger	R	Returns the width of the image included in the frame
ChunkHeight	IInteger	R	Returns the height of the image included in the frame
ChunkOffsetX	IInteger	R	Returns the horizontal offset of the image included in the frame
ChunkOffsetY	IInteger	R	Returns the vertical offset of the image included in the frame
ChunkPixelFormat	IEnumeration	R	Returns the pixel format of the image included in the frame

UserSetControl Category

Name	Interface	Access	Description
UserSetSelector	IEnumeration	R/W	Selects the user set to load or save (Default, UserSet0, UserSet1)
UserSetLoad	ICommand	W	Loads the selected user set and applies it to the virtual camera's features
UserSetSave	ICommand	W	Saves the current values of the camera's features to the selected user set
UserSetDefault	IEnumeration	R/W	Selects the user set to load when the virtual camera is reset (Default, UserSet0, UserSet1)

ActionControl Category

Name	Interface	Access	Description
ActionDeviceKey	IInteger	W	Sets the device key for action commands [0]
ActionSelector	IInteger	R/W	Selects to which action signal further action features apply (1 or 2)
ActionGroupKey	IInteger	R/W	The group key that the device will use to validate incoming action commands [0]
ActionGroupMask	Integer	R/W	The mask that the device will use to validate incoming action commands ([1] for Action1, [2] for Action2)

3 C++ API Reference

3.1 Getting started

This chapter describes how to create a simple GigE Vision server application (virtual camera) with *GigESim SDK* in Visual C++ for both 32- and 64-bit platforms.

Creating the Project

In the Visual Studio development environment select *New -> Project.* The *New Project* Dialog box will appear. Select *Visual C++ projects* on the left and *MFC Application* on the right. In the *Name* filed below type the name of your application, for instance *GigeCam* and click *OK* . When *MFC Application Wizard* appears, click *Application Type* , select *Dialog based* radio button and click *Finish* . The project will be created, and the program dialog *GigeCam* will be displayed for editing.

Adding the CGevCam class

In the Solution Explorer right-click on your project (GigeCam) and select *Add -> Existing Item*.. Find *GigeSimSDK.h* file supplied with *GigESim* and copy it to your project folder. Depending on the platform of your application (32- or 64-bit), add a reference to either *gigesimsdk.lib* or *gigesimsdk64.lib* to your project.

Add the following line to the beginning of CGigeCamDlg.cpp:

```
#include "GigeSimSDK.h"
```

Add the following member variable to CGigeCamDlg.h:

```
CGevCamera* m_pCamera;
```

You are now ready to create and initialize a virtual camera object.

Initializing the camera object

Add the following lines to CGigeCamDlg::OnInitDialog

```
m_pCamera = createCamera();                    //create camera object
m_pCamera->SetImageSize(1024,768);             //define the image size
m_pCamera->AddPixelFormat("Mono8");            //define supported pixel formats
m_pCamera->AddPixelFormat("Mono16");
m_pCamera->SetPixelFormat("Mono8");            //select the current pixel format
```

You should also add the following line to the dialog's destructor:

```
CGigeCamDlg::~CGigeCamDlg()
{
    delete m_pCamera;
}
```

It is now time to prepare a function that will generate simulated images and transmit them to the network.

Creating the image transmission thread

Add the following method to the dialog class:

```
void CGigeCamDlg::videoGenerator()
{
  int width;
  int height;
  int i=0;
  char buffer[2048*2048];
  char *ptr;
  //the cycle below fills the image buffer with a moving pattern
  //and transmits it over the network
  while (!m_exitThread)
  {
    m_pCamera->LockFormat();    //this is needed to synchronize our
    cycle with the client
    width=m_pCamera->GetWidth();
    height=m_pCamera->GetHeight();
    for (ptr=buffer; ptr < buffer+width*height; ptr++)
     *ptr=i++;
    m_pCamera->SendImage(buffer);
    Sleep(20);
  }
}
```

We will also have to add the following function that will call the video transmission method in a separate thread:

```
ULONG CGigeCamDlg::videoTransmitThread(CGigeCamDlg* pDlg)
{
   pDlg->videoGenerator();
   return 0;
}
```

Connecting the camera to the network and starting the transmission thread

Add the "Connect" button to your dialog. Insert the following code into the button handler:

```
void CGigeCamDlg::Connect()
{
   //bind the camera with the GigE card
   int ret=m_pCamera->connect("169.254.1.100");
  if (ret == 0)
   {
         m_exitThread = false;
         //start the video transmission thread
         m_thread = CreateThread(NULL, 80000000,
(LPTHREAD_START_ROUTINE)videoTransmitThread, this, 0, NULL);
         if (m_thread == INVALID_HANDLE_VALUE)
               m_pCamera->Disconnect();
   }
}
```

Upon the button click the virtual camera (GigE Vision server) will get activated and become available to GigE Vision client applications operating on the same network. Once a client application issues the *AcquisitionStart* command, the virtual camera will enter the image streaming cycle and will keep transmitting images to the client until the latter issues the *AcquisitionStop* command.

Before running your application, make sure to place *gigesimsdk*.dll or/and *gigesimsdk64*.dll in the folder where your executable is located.

For more details refer to the sample projects provided with *GigESim*.

3.2 GevCamera Class

CGevCamera is the one and only class for the entire virtual camera object. It contains all the information and methods necessary to configure a simulation GigE Vision device, set up its features and stream images to the network. To instantiate an object of the **CGevCamera** class, use the CreateCamera function.

3.2.1 createCamera

Description

Creates an instance of the **CGevCamera** class.

[C++]
```
external CGevCamera* createCamera();
```

Parameters [C/C++]

none

Return Values

Pointer to an object of the **CGevCamera** class if successful. Zero if failed.

Example

This fragment of code instantiates a camera object and sets up initial values for the image transfer :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetImageSize (1024, 768);
m_pCamera->AddPixelFormat("Mono16");
```

Remarks

Use this function to instantiate a virtual camera object. Once the camera object is created, you can use its methods and callbacks for defining and configuring the camera's features, connecting it to the network and initiating an image transfer.

To work with multiple virtual cameras, you will have to create several CGevCam objects. See Working with multiple cameras for more details.

3.3 Methods

The following methods are available in the CGevCamera class:

GigE Vision Server Control

GetInterfaceCount	Returns the number of network interfaces available in the system
GetInterfaceAtIndex	Returns the IP address of the network interface in the system interface list
GetInterfaceInfoAtIndex	Returns the parameters of the network interface in the system interface list
SetDeviceInfo	Sets the device information fields and assigns the name to the XML information file
SetUserDefinedName	Sets the user-defined name of the virtual camera device
SetMacAddress	Assigns the specified MAC address to the virtual camera
SetIpAddress	Sets the IP configuration of the virtual camera device
GetIpAddress	Returns the current IP configuration of the virtual camera device
SetGevMode	Sets the version of the GigE Vision standard for the virtual camera
SetTimerMode	Sets the operational mode of the virtual camera timer
SetMaxImageSize	Sets the maximum horizontal and vertical size of outgoing images
SetStreamChannelCount	Sets the amount of stream channels in the virtual camera
SetActionCount	Sets the amount of action signals supported by the virtual camera
SetAdvancedOptions	Sets advanced operational parameters for the virtual camera
Connect	Brings the virtual camera online and binds it to the specified network interface
Disconnect	Disconnects the virtual camera from the currently selected interface and brings it offline
IsConnected	Return the network connection state of the virtual camera
GetTimer	Returns the current value of the virtual camera timer in nanoseconds
ResetTimer	Resets the internal timer of the virtual camera to zero
ReadRegister	Returns the value of the specified virtual camera register
WriteRegister	Sets the value of the specified virtual camera register
ReadMemory	Reads a block of data from the internal camera memory
WriteMemory	Writes the block of data to the internal camera memory

Image Format and Streaming Control

SetImageSize	Sets the horizontal and vertical size of outgoing images
GetWidth	Returns the current horizontal size of outgoing images
GetHeight	Returns the current vertical size of outgoing images
AddPixelFormat	Adds the specified pixel format to the list of formats supported by the virtual camera
SetPixelFormat	Sets the specified pixel format for outgoing images
GetPixelFormatString	Returns the string value of the current pixel format of outgoing images
GetPixelFormatValue	Returns the numerical value of the current pixel format of outgoing images
GetFormatStringFromValue	Returns the name of the pixel format specified by its numerical ID
GetFormatValueFromString	Returns the numerical ID of the specified pixel format
GetPayloadSize	Returns the current payload size of outgoing images
PixelFormatConvert	Converts image data from a plain pixel format to the specified output format
SetImageCompression	Sets the image compression mode
SetCompressionQuality	Sets the compression quality and bitrate for JPEG and H.264 compressed streams
LockFormat	Synchronizes the image streaming thread with a client application
SendImage	Streams the specified image frame to the network

GenICam Feature Control

CreateCategory	Creates a GenICam category with the specified name
CreateFeature	Creates a GenICam feature with the specified name, type and access mode
CreateAdvancedFeature	Creates a custom GenICam feature with advanced functionality
DeleteFeature	Deletes the existing feature and frees the memory associated with it
AddEnumEntry	Adds an entry to the specified enumerated feature and sets up the parameters of the entry
DeleteEnumEntry	Deletes the specified entry from the enumerated feature
SetFeatureIntValue	Sets the integer value of the specified feature
SetFeatureFloatValue	Sets the value of the specified floating point feature
SetFeatureStingValue	Sets the string of the specified feature
GetFeatureIntValue	Returns the integer value of the specified feature
GetFeatureFloatValue	Returns the value of the specified floating point feature
GetFeatureStringValue	Returns the string value of the specified feature
GetFeatureEnumList	Returns the array of string values representing the specified enumerated feature
SetFeatureIntRange	Assigns the minimum, maximum and increment attributes to the specified feature
SetFeatureRange	Assigns the minimum and maximum attributes to the specified feature
GetFeatureIntRange	Returns the minimum, maximum and increment attributes of the specified feature
GetFeatureRange	Returns the minimum and maximum attributes of the specified feature
SetFeatureAccess	Sets the access mode for the specified feature
SetFeatureDescription	Assigns a description to the specified feature
SetFeatureElement	Sets the value of the specified element associated with the feature
GetFeatureElement	Returns the value of the specified element associated with the feature
DeleteFeatureElement	Deletes the specified element associated with the feature
CreateElementAttribute	Creates an attribute associated with the specified element
GetFeatureRegister	Returns the address of the virtual hardware register associated with the specified feature

Event and Chunk Data Control

CreateEventCategory	Creates the a GenICam category under which event-related features will be grouped
CreateEvent	Creates a GenICam event sub-category with related features, registers and ports
CreateEventFeature	Creates a GenICam feature associated with a data field in the specified event
SendEvent	Sends the specified event message to the network
SendEventData	Sends the specified data event message to the network
SendEvents	Sends multiple event messages to the network
CreateChunkCategory	Creates the a GenICam category under which chunk-related features will be grouped
CreateChunkFeature	Creates a GenICam feature associated with the chunk data field
AddChunkData	Adds a chunk data field to the frame buffer
SetChunkMode	Enables/disables the chunk streaming mode and sets the size of the chunk data block

3.3.1 AddChunkData

Description

Adds a chunk data field to the frame buffer.

[C++]

```
int AddChunkData (constant char* frame , constant char* data , int length,
  unsigned int chunkId, unsigned int channelIndex=0);
```

Parameters [C/C++]

[in] `constant char* frame`
Pointer to the beginning of the frame buffer (image data). Must not be NULL or empty

[in] `constant char* data`
Pointer to the chunk data field. Must not be NULL or empty.

[in] `int length`
Integer value specifying the size of chunk data in bytes.

[in] `unsigned int chunkID`
Integer value specifying the numerical ID of the chunk.

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code creates the "camera temperature" chunk feature, adds a corresponding data field to the frame buffer and sends the frame to the network:

```
m_pCamera->SetStreamChannelCount(1);
m_pCamera->SetChunkMode (4,1);
m_pCamera->CreateChunkCategory ("ChunkDataControl");
 m_pCamera->CreateChunkFeature (0xa8dc50a0, "ChunkTemperature", "ChunkDataControl",
 FEATURE_TYPE_FLOAT, 4, "Camera temperature");
 .........
 m_pCamera->LockFormat();
 buffer=GenerateImageFrame();
 data=GetVirtualTemperature();
 m_pCamera->AddChunkData (buffer, (char*) data, 4, 0xa8dc50a0);
 m_pCamera->AddChunkData (buffer, NULL, imageWidth*imageHeight*BytesPerPixel, 0);
 m_pCamera->SendImage(buffer);
```

Remarks

This method is used to add chunk data to the frame buffer when the chunk mode is activated. It should be called before the image frame is sent to the network. You can use several calls to AddChunkData to add several chunk fields to the frame. Note that in the chunk mode the very last call to AddChunkData should always be made for the image buffer with the *data* and *chunkID* parameters set to zero and *length* equal to the size of the image in bytes.

For GenICam compatibility It is recommended to associate each chunk data field with a corresponding chunk feature by a prior call to CreateChunkFeature.

3.3.2 AddEnumEntry

Description

Adds an entry to the specified enumerated feature and sets up the parameters of the entry.

[C++]
```
int AddEnumEntry(const char* feature, const char* name, unsigned int
value);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the enumerated type.

[in] `const char* name`
Name of the entry to be added. Must not be NULL or empty.

[in] `unsigned int value`
Numerical value associated with the entry.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Entry with this value already exist

Example

This fragment of code instantiates a camera object, creates an enumerated feature and sets up a list of three entries:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();
m_pCamera->CreateFeature(FEATURE_TYPE_ENUMERATION, "TestImageSelector", "Root",
FEATURE_ACCESS_RW);

m_pCamera->AddEnumEntry("TestImageSelector", "First pattern", 1);
m_pCamera->AddEnumEntry("TestImageSelector", "Second pattern", 2);
m_pCamera->AddEnumEntry("TestImageSelector", "Third pattern", 3);
```

Remarks

This method should be used after calling CreateFeature in order to finalize the setup of a feature of the enumerated type.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.3 AddPixelFormat

Description

Adds the specified pixel format to the list of formats supported by the virtual camera.

[C++]
```
int AddPixelFormat(const char* format, const char* pisImplemented =
NULL, const char* pisAvailable = NULL);
```

Parameters [C/C++]

[in] `const char* format`
String specifying the pixel format to add to the list of supported formats. See **Remarks** for the list of possible values.

[in] `const char* pisImplemented`
If not NULL, a <pisImplemented> tag with the specified string will be added to the description of the pixel format entry in the XML file.

[in] `const char* pisAvailable`
If not NULL, a <pisAvailable> tag with the specified string will be added to the description of the pixel format entry in the XML file.

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Unsupported format

Example 1

This fragment of code instantiates a camera object and sets up initial values for the image transfer :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetImageSize (1024, 768);
m_pCamera->AddPixelFormat("Mono16");
m_pCamera->AddPixelFormat("RGB8");
m_pCamera->SetPixelFormat("Mono16");
```

Example 2

This line of code ads the BayerGR12Packed pixel format and makes its availability dependent on two variables:

```
m_pCamera->AddPixelFormat("BayerGR12Packed", "isColorCamera",  "isGRLayoutAvaliable");
```

As a result, the following code will be added to the XML file

```
<EnumEntry Name="BayerGR12Packed">
        <pIsImplemented>isColorCamera</pIsImplemented>
        <pIsAvailable>isGRLayoutAvailable</pIsAvailable>
        <Value>17563690</Value>
</EnumEntry>
```

Remarks

This method adds an entry to the *PixelFormat* feature of the virtual camera. *PixelFormat* is a mandatory GigE Vision enumerated feature which is automatically created when a virtual camera object is instantiated.

By default, the *PixelFormat* contains one entry corresponding to the "Mono8" format. To add more pixel formats to the list of supported formats, use one of the following string values as the *format* parameter for **SetPixelFormat**:

Format	ID	Description	Bits per pixel
"Mono8"	0x01080001	8-bit monochrome unsigned	8
"Mono8s"	0x01080002	8-bit monochrome signed	8
"Mono10"	0x01100003	10-bit monochrome unpacked	16
"Mono10Packed"	0x010C0004	10-bit monochrome packed	12
"Mono12"	0x01100005	12-bit monochrome unpacked	16
"Mono12Packed"	0x010C0006	12-bit monochrome packed	12
"Mono14"	0x01100025	14-bit monochrome pixel	16
"Mono16"	0x01100007	16-bit monochrome	16
"BayerGR8"	0x01080008	8-bit raw Bayer, GRBG layout	8
"BayerRG8"	0x01080009	8-bit raw Bayer, RGGB layout	8
"BayerGB8"	0x0108000A	8-bit raw Bayer, GBRG layout	8
"BayerBG8"	0x0108000B	8-bit raw Bayer, BGGR layout	8
"BayerGR10"	0x0110000C	10-bit raw Bayer unpacked, GRBG layout	16
"BayerRG10"	0x0110000D	10-bit raw Bayer unpacked, RGGB layout	16
"BayerGB10"	0x0110000E	10-bit raw Bayer unpacked, GBRG layout	16
"BayerBG10"	0x0110000F	10-bit raw Bayer unpacked, BGGR layout	16
"BayerGR12"	0x01100010	12-bit raw Bayer unpacked, GRBG layout	16
"BayerRG12"	0x01100011	12-bit raw Bayer unpacked, RGGB layout	16
"BayerGB12"	0x01100012	12-bit raw Bayer unpacked, GBRG layout	16
"BayerBG12"	0x01100013	12-bit raw Bayer unpacked, BGGR layout	16
"BayerGR10Packed"	0x010C0026	10-bit raw Bayer packed, GRBG layout	12
"BayerRG10Packed"	0x010C0027	10-bit raw Bayer packed, RGGB layout	12
"BayerGB10Packed"	0x010C0028	10-bit raw Bayer packed, GBGR layout	12
"BayerBG10Packed"	0x010C0029	10-bit raw Bayer packed, BGGR layout	12
"BayerGR12Packed"	0x010C002A	12-bit raw Bayer packed, GRBG layout	12
"BayerRG12Packed"	0x010C002B	12-bit raw Bayer packed, RGGB layout	12
"BayerGB12Packed"	0x010C002C	12-bit raw Bayer packed, GBRG layout	12
"BayerBG12Packed"	0x010C002D	12-bit raw Bayer packed, BGGR layout	12
"BayerGR16"	0x0110002E	16-bit raw Bayer unpacked, GRBG layout	16
"BayerRG16"	0x0110002F	16-bit raw Bayer unpacked, RGGB layout	16
"BayerGB16"	0x01100030	16-bit raw Bayer unpacked, GBRG layout	16
"BayerBG16"	0x01100031	16-bit raw Bayer unpacked, BGGR layout	16
"RGB8"	0x02180014	24-bit RGB color	24
"BGR8"	0x02180015	24-bit BGR color	24
"RGBa8"	0x02200016	24-bit RGB color with alpha channel	32
"BGRa8"	0x02200017	24-bit BGR color with alpha channel	32
"RGB10"	0x02300018	30-bit RGB color	48
"BGR10"	0x02300019	30-bit BGR color	48

To populate the list of formats, this method should be called repeatedly with different values of the first argument. After the list of formats is created, a specific format can be assigned to the virtual camera by calling SetPixelFormat.

The use of *pisImplemented* and *pisAvailable* arguments requires an advanced knowledge of the GenICam standard and XML syntax. For more information please refer to the description of the GenICam standard available at www.genicam.org.

Note that the camera must be in the disconnected state in order for **AddPixelFormat** to work.

3.3.4 Connect

Description

Brings the virtual camera "online" and binds it to the specified network interface.

[C++]
```
int Connect(const char* ipAddr);
```

Parameters [C/C++]

[in] `const char* ipAddr`
String containing the IP address which will be assigned to the virtual camera. Must be in the same subnet as the interface selected in the first parameter. If zero or omitted, the camera's IP address will coincide with the IP address of the selected interface.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Interface does not exist
E_INVALIDARG
Wrong IP address

Example

This fragment of code creates a virtual camera object and binds it to a specific network interface :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();
m_pCamera->Connect ("169.254.1.100");
```

Remarks

This method should be called after all camera features have been set up and configured. After the virtual camera gets to the connected state, it becomes available to client GigE Vision applications operating on the same network.

In a typical configuration a GigE network interface must be selected for a remote GigE Vision connection. It is possible however to select any other interface available in the system as long as a client application is running on the local host.

No feature can be added or removed while the camera is in the connected state. Use Disconnect to bring the camera offline.

3.3.5 CreateAdvancedFeature

Description

Creates a custom GenICam feature with advanced functionality.

[C++]
```
int CreateAdvancedFeature (const char* type, const char* name, const char* category,
    unsigned short namespace = FEATURE_NAMESPACE_CUSTOM);
```

Parameters [C/C++]

[in] `const char* type`
String indicating the type of the feature, such as "SwissKnife", "Converter", "ConfRom".

[in] `const char* name`
Name of the feature to be created. Must not be NULL or empty.

[in] `const char* category`
GenICam category under which the feature will be created. This parameter should be either "Root" or the name of the existing category created via CreateCategory.

[in] `unsigned short namespace`
Namespace to which the feature will belong. Can be one of the following values:
FEATURE_NAMESPACE_STANDARD - feature will be created with the "Standard" namespace tag (recommended for features which follow the GenICam SFNC specifications).
FEATURE_NAMESPACE_CUSTOM - feature will be created with the "Custom" namespace tag (recommended for features with proprietary names).

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Category does not exist

Example

This fragment of code creates an advanced feature "MaxFrameRate" of the SwissKnife type, then adds several elements and attributes to it.

```
CreateCustomFeature("SwissKnife", "MaxFrameRate", "");

SetFeatureElement("MaxFrameRate", "pVariable", "regPayloadSize", true);
SetFeatureElement("MaxFrameRate", "pVariable", "regExposureTime", true);
SetFeatureElement("MaxFrameRate", "pVariable", "regMultiplierFPS", true);
SetFeatureElement("MaxFrameRate", "Formula", "( (1000000000 / 8 / PS < 1000000 / ET) ?
(1000000000 / 8 / PS) * MUL : (1000000 / ET) * MUL)", true);

CreateElementAttribute("MaxFrameRate", "pVariable", "regPayloadSize", "Name", "PS");
```

```
CreateElementAttribute("MaxFrameRate", "pVariable", "regExposureTime", "Name", "ET");
CreateElementAttribute("MaxFrameRate", "pVariable", "regMultiplierFPS", "Name", "MUL");
```

As a result, the following feature description will appear in the camera XML file::

```
<SwissKnife Name="MaxFrameRate" NameSpace="Custom">
  <pVariable Name="PS">regPayloadSize</pVariable>
  <pVariable Name="ET">regExposureTime</pVariable>
  <pVariable Name="MUL">regMultiplierFPS</pVariable>
 <Formula>( (1000000000 / 8 / PS &amp;lt; 1000000 / ET) ? (1000000000 / 8 / PS) * MUL : (1000000
/ ET) * MUL)</Formula>
</SwissKnife>
```

Remarks

Use this method to create features of advanced types not covered by CreateFeature. Advanced types of features include "SwissKnife", "IntSwissKnife", "Converter", "IntConverter", "ConfRom", "TextDesc", "IntKey". A creation of an advanced feature is typically followed by setting up its elements and attributes with SetFeatureElement and CreateElementAttribute.
.
The use of this method requires an advanced knowledge of the GenICam standard and XML syntax. For more information please refer to the description of the GenICam standard available at www.genicam.org.

Note that the camera must be in the disconnected state in order for this method to work.

It is recommended to change the version of your virtual camera device each time you modify the set of features (see SetDeviceInfo). If this is not done, a client GigE Vision application may use a copy of an old XML information file from its cache folder and remain unaware of your changes. If you do not want to use the device version control, make sure to locate an XML cache folder of your client application and delete an XML file in it each time the set of features of the virtual camera has been modified. This should be done on each computer running the client software.

3.3.6 CreateCategory

Description

Creates a GenICam category with the specified name.

[C++]
```
int CreateCategory (const char* name, const char* parentCategory);
```

Parameters [C/C++]

[in] `const char* name`
Name of the category to be created. Should not be NULL or empty.

[in] `const char* parentCategory`
Parent category under which the category or sub-category will be created. To create a sub-category in the root category, use "Root" for this argument.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Parent category does not exist

Example

This fragment of code instantiates a camera object, creates an integer and enumerated features and sets up their attributes :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_INTEGER, "TestFeatureInt", "Root",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureRange("TestFeatureInt", 0, 1000, 1);

m_pCamera->CreateFeature(FEATURE_TYPE_ENUMERATION, "TestFeatureEnum", "Root",
FEATURE_ACCESS_RW);
m_pCamera->setEnumEntry("TestEnumFeature", "First item", 1);
m_pCamera->setEnumEntry("TestFeatureEnum", "Second item", 2);
m_pCamera->setEnumEntry("TestFeatureEnum", "Third item", 3);
```

Remarks

Use this method to assign features to your virtual camera object. The assigned features will be exposed to GigE Vision client applications allowing them to remotely control parameters of the virtual camera.

A creation of a feature should be followed by setting up its attributes. See for more details.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.7 CreateChunkCategory

Description

Creates the a GenICam category under which chunk-related features will be grouped.

[C++]
```
int CreateChunkCategory (const char* name);
```

Parameters [C/C++]

[in] `const char* name`
Name of the chunk data category to be created. Should not be NULL or empty.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Parent category does not exist

Example

The following line of code creates a chunk category with the standard SFNC name "ChunkDataControl".

```
m_pCamera->CreateChunkCategory ("ChunkDataControl");
```

As a result, the following code will be added to the XML file:

```
<Category Name="ChunkDataControl" NameSpace="Standard">
        <pFeature>ChunkSelector</pFeature>
        <pFeature>ChunkEnable</pFeature>
</Category>
```

Remarks

This method is used in conjunction with CreateChunkFeature to automate the process of adding GenICam compliant chunk features to the virtual camera. Per GenICam standard, chunk data fields included in a frame buffer should be associated with features grouped under a common category which should also contain the *ChunkSelector* and *ChunkEnable* features. In the example above a single call to **CreateChunkCategory** creates the *ChunkDataControl* category and adds *ChunkSelector* and *ChunkEnable* features to it.

Note that this method does not handle inclusionsion of chunks into frame buffers but only adds chunk-related members to the camera's XML file. To insert chunk data into a frame buffer, use AddChunkData.

The camera must be in the disconnected state in order for this method to work.

3.3.8 CreateChunkFeature

Description

Creates a GenICam feature associated with the chunk data field.

[C++]
```
int CreateChunkFeature (unsigned int chunkId, const char* name , const
char* category ,
  unsigned short type, unsigned int length, const char* description = NULL);
```

Parameters [C/C++]

[in] `unsigned int chunkID`
Integer value specifying the numerical ID of the chunk.

[in] `const char* name`
Name of the data field in the chunk. Must not be NULL or empty.

[in] `const char* category`
Parent chunk category under which the chunk feature will be created. The category must be created by a prior call to CreateChunkCategory.

[in] `unsigned short type`
Type of the feature to be created. Can be one of the following values:
FEATURE_TYPE_UINTEGER - unsigned 32-bit integer feature, maps to a slider with value, minimum, maximum and increment
FEATURE_TYPE_INTEGER - signed 32-bit integer feature, maps to a slider with value, min, maximum, increment and physical unit
FEATURE_TYPE_INTEGER64 - signed 62-bit integer feature, maps to a slider with value, min, maximum, increment and physical unit
FEATURE_TYPE_FLOAT - floating point 32-bit feature, maps to a slider with value, minimum, maximum and physical unit
FEATURE_TYPE_STRING - string feature, maps to an edit box showing a string of text
FEATURE_TYPE_BOOLEAN - boolean feature, maps to a check box
FEATURE_TYPE_ENUMERATION - enumeration feature, maps to dropdown box with a list of selectable items

`unsigned int length`
[in] Integer value specifying the length (in bytes) of the data field associated with the chunk feature.

`const char* description`
[in] String containing the description of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Category does not exist
E_INVALIDARG

Wrong feature type

Example

This fragment of code creates the "OffsetY" chunk feature:

```
m_pCamera->CreateChunkFeature (0xa8dc50e0, "ChunkOffsetY", "ChunkDataControl",
FEATURE_TYPE_INTEGER, 4, "Vertical offset");
```

As a result, the following code will be added to the XML file:

```
<Enumeration Name="ChunkSelector" NameSpace="Standard">
   <EnumEntry Name="ChunkOffsetY">
      <Value>0</Value>
   </EnumEntry>
   <pValue>regChunkSelector</pValue>
</Enumeration>

<Integer Name="ChunkOffsetY" NameSpace="Standard">
   <Description>Vertical offset </Description>
   <pValue>regChunkOffsetY</pValue>
   <Min>0</Min>
   <Max>4294967295</Max>
</Integer>

<IntReg Name="regChunkOffsetY" NameSpace="Custom">
   <Address>0x0</Address>
   <Length>4</Length>
   <AccessMode>RO</AccessMode>
   <pPort>ChunkOffsetYPort</pPort>
   <Cachable>NoCache</Cachable>
   <Sign>Unsigned</Sign>
   <Endianess>BigEndian</Endianess>
</IntReg>

<Port Name="ChunkOffsetYPort">
   <ChunkID>a8dc50e0</ChunkID>
</Port>
```

Remarks

This method is used in conjunction with CreateChunkCategory to automate the process of adding GenICam compliant chunk features to the virtual camera. Per GenICam standard, each chunk data field included in a frame should be associated with a feature grouped under the chunk category. In the example above a single call to **CreateChunkFeature** creates the *ChunkOffsetY* feature, maps it to a virtual port and adds a corresponding entry into the *ChunkSelector* feature.

Note that this method does not insert chunk data into frame buffers but only adds chunk-related members to the camera's XML file. To add chunk data to a frame buffer, use AddChunkData.

The camera must be in the disconnected state in order for this method to work.

3.3.9 CreateElementAttribute

Description

Creates an attribute associated with the specified element.

[C++]
```
int CreateElementAttribute(const char* feature, const char* element, const char* elemValue, const char* attr, const char* value);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to which the element belongs.

[in] `const char* element`
Name of the element for which the attribute will be created.

[in] `const char* elemValue`
String representing the value of the element.

[in] `const char* attr`
Name of the attribute to be created.

[in] `const char* value`
String representing the value of the attribute.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature or element does not exist

Example

```
CreateAdvancedFeature("SwissKnife", "MaxFrameRate", "");

SetFeatureElement("MaxFrameRate", "pVariable", "regPayloadSize", true);
SetFeatureElement("MaxFrameRate", "pVariable", "regExposureTime", true);
SetFeatureElement("MaxFrameRate", "pVariable", "regMultiplierFPS", true);
SetFeatureElement("MaxFrameRate", "Formula", "( (1000000000 / 8 / PS < 1000000 / ET) ?
(1000000000 / 8 / PS) * MUL : (1000000 / ET) * MUL)", true);

CreateElementAttribute("MaxFrameRate", "pVariable", "regPayloadSize", "Name", "PS");
CreateElementAttribute("MaxFrameRate", "pVariable", "regExposureTime", "Name", "ET");
CreateElementAttribute("MaxFrameRate", "pVariable", "regMultiplierFPS", "Name", "MUL");
```

As a result, the following feature description will appear in the camera XML file::

```
<SwissKnife Name="MaxFrameRate" NameSpace="Custom">
```

```
        <pVariable Name="PS">regPayloadSize</pVariable>
        <pVariable Name="ET">regExposureTime</pVariable>
        <pVariable Name="MUL">regMultiplierFPS</pVariable>
       <Formula>( (1000000000 / 8 / PS &lt; 1000000 / ET) ? (1000000000 / 8 / PS) * MUL : (1000000
/ ET) * MUL)</Formula>
    </SwissKnife>
```

Remarks

This method allows you to directly program an XML-file linked to the virtual camera object. An element is part of the XML-file encapsulated between a starting and ending tag. The element can contain multiple unique attributes which give more information about the element. The above sample code creates an advanced *SwissKnife*-type feature named MaxFrameRate with several elements, including a formula for calculating the value of the feature and three elements of the *pVariable* type pointing to certain registers (regPayloadSize, regExposureTime, regMultiplierFPS). Then the code uses **SetFeatureElement** to add a *Name* attribute to *pVariable* elements thus associating them with PS, ET and MUL variables used in the formula.

The use of this method requires an advanced knowledge of the GenICam standard and XML syntax. For more information please refer to the description of the GenICam standard available at www.genicam.org.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.10 CreateEventCategory

Description

Creates the a GenICam category under which event-related features will be grouped.

[C++]
```
int CreateEventCategory (const char* name);
```

Parameters [C/C++]

[in] `const char* name`
Name of the event category to be created. Should not be NULL or empty.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Parent category does not exist

Example

The following line of code creates an event category with the standard SFNC name "EventControl".

```
m_pCamera->CreateEventCategory ("EventControl");
```

As a result, the following code will be added to the XML file:

```
<Category Name="ChunkDataControl" NameSpace="Standard">
        <pFeature>EventSelector</pFeature>
        <pFeature>EventNotification</pFeature>
</Category>
```

Remarks

This method automates the process of adding GenICam compliant events to the virtual camera. Per GenICam standard, camera events are represented by sub-categories and features grouped under a common category which should also contain the *EventSelector* and *EventNotification* features. In the example above a single call to **CreateEventCategory** creates the *EventControl* category and adds *EventSelector* and *EventNotification* features to it.

To add actual events to the event category, use CreateEvent.

Note that this method does not handle the event generation, but only adds event-related members to the camera's XML file. To send events to the network, use SendEvent, SendEvents, SendEventData.

The camera must be in the disconnected state in order for this method to work.

3.3.11 CreateEvent

Description

Creates a GenICam event sub-category with related features, registers and ports.

```
[C++]
int CreateEvent (unsigned int msgId, const char* name, const char*
parentCategory,
    bool evTimestamp, bool evChannelId, bool evFrameId);
```

Parameters [C/C++]

[in] `unsigned int msgID`
Integer value specifying the numerical ID of the event.

[in] `const char* name`
Name of the event to be created. Should not be NULL or empty.

[in] `const char* parentCategory`
Parent category under which the event sub-category will be created. The category must be created by a prior call to CreateEventCategory.

[in] bool evTimestamp
If TRUE, the event will contain the Timestamp feature.

[in] bool evChannelID
If TRUE, the event will contain the ChannelID feature.

[in] bool evFrameID
If TRUE, the event will contain the FrameID feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Parent category does not exist

Example

The following line of code creates the "AcquisitionTrigger" event sub-category in the "EventControl" category and adds related features, registers and ports.

```
m_pCamera->CreateEventCategory ("EventControl");
m_pCamera->CreateEvent (0x8e94, "AcquisitionTrigger", "EventControl", True, False, False);
```

As a result, the following code will be added to the XML file:

```
<Enumeration Name="EventSelector" NameSpace="Standard">
    <EnumEntry Name="AcquisitionTrigger">
    <Value>0</Value>
```

```
        </EnumEntry>
        <pValue>regEventSelector</pValue>
</Enumeration>

<Category Name="EventAcquisitionTriggerData" NameSpace="Standard">
        <pFeature>EventAcquisitionTrigger</pFeature>
        <pFeature>EventAcquisitionTriggerTimestamp</pFeature>
</Category>

<IntReg Name="regEventAcquisitionTrigger">
    <Address>0x1014C</Address>
    <Length>4</Length>
    <AccessMode>RO</AccessMode>
     <pPort>Device</pPort>
    <Sign>Unsigned</Sign>
    <Endianess>BigEndian</Endianess>
 </IntReg>

 <IntReg Name="regEventAcquisitionTriggerTimestamp">
    <Address>0x8</Address>
    <Length>8</Length>
    <AccessMode>RO</AccessMode>
     <pPort>EventAcquisitionTriggerPort</pPort>
    <Sign>Unsigned</Sign>
    <Endianess>BigEndian</Endianess>
 </IntReg>

 <Port Name="EventAcquisitionTriggerPort">
    <EventID>8e94</EventID>
 </Port>
```

Remarks

This method automates the process of adding GenICam compliant events to the virtual camera. Per GenICam standard, for each event there should be a sub-category grouping all data members related to the particular event. In the example above a single call to **CreateEvent** creates the *EventAcquisitionTriggerData* sub-category, adds *EventAcquisitionTrigger* and *EventAcquisitionTriggerTimestamp* features to it, maps them to available registers and ports and adds a corresponding entry into the *EventSelector* feature..

Calls to **CreateEvent** should be preceded by a call to CreateEventCategory. To add additional data fields to the event, use CreateEventFeature.

Note that this method does not handle the event generation, but only adds event-related members to the camera's XML file. To send events to the network, use SendEvent, SendEvents, SendEventData.

The camera must be in the disconnected state in order for this method to work.

3.3.12 CreateEventFeature

Description

Creates a GenICam feature associated with a data field in the specified event.

```
[C++]
int CreateEventFeature (const char* name, const char* event, unsigned short
type,
      unsigned int offset, unsigned int length, const char* description =
NULL);
```

Parameters [C/C++]

[in] `const char* name`
Name of the data field in the event. Must not be NULL or empty.

[in] `const char* event`
Name of the event.

[in] `unsigned short type`
Type of the feature to be created. Can be one of the following values:
FEATURE_TYPE_UINTEGER - unsigned 32-bit integer feature, maps to a slider with value, minimum, maximum and increment
FEATURE_TYPE_INTEGER - signed 32-bit integer feature, maps to a slider with value, min, maximum, increment and physical unit
FEATURE_TYPE_INTEGER64 - signed 62-bit integer feature, maps to a slider with value, min, maximum, increment and physical unit
FEATURE_TYPE_FLOAT - floating point 32-bit feature, maps to a slider with value, minimum, maximum and physical unit
FEATURE_TYPE_STRING - string feature, maps to an edit box showing a string of text
FEATURE_TYPE_BOOLEAN - boolean feature, maps to a check box
FEATURE_TYPE_COMMAND - command feature, maps to a command button
FEATURE_TYPE_ENUMERATION - enumeration feature, maps to dropdown box with a list of selectable items

`unsigned int offset`
[in] Integer value specifying the offset (in bytes) of the data field associated with the feature relative to the

`unsigned int length`
[in] Integer value specifying the length (in bytes) of the data field associated with the feature.

`const char* description`
[in] String containing the description of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Event does not exist

E_INVALIDARG
Wrong feature type or access

Example

This line of code adds the "ExposureTime" field to the "FrameStart" event:

```
m_pCamera->CreateEventFeature ("ExposureTime", "FrameStart", FEATURE_TYPE_FLOAT,
16, 4);
```

As a result, the following code will be added to the XML file:

```
<Category Name="EventFrameStartData" NameSpace="Standard">
        ..................................................
      <pFeature>EventFrameStartExposureTime</pFeature>
</Category>

<Float Name="EventFrameStartExposureTime">
      <pValue>regEventFrameStartExposureTime</pValue>
</Float>
<FloatReg Name="regEventFrameStartExposureTime" NameSpace="Custom">
      <Address>0x10</Address>
      <Length>4</Length>
      <AccessMode>RO</AccessMode>
       <pPort>EventFrameStartPort</pPort>
      <Endianess>BigEndian</Endianess>
</FloatReg>
```

Remarks

This method is used in conjunction with CreateEventCategory to automate the process of adding GenICam compliant events to the virtual camera. Per GenICam standard, each data field encapsulated in the event should be associated with a feature grouped under the event category. While CreateEventCategory can be used to create three standard data fields (Timestamp, ChannelID and FrameID), this method allows you to add additional data fields to the event.

Note that this method does not handle the event generation, but only adds event-related members to the camera's XML file. To send events to the network, use SendEvent, SendEvents, SendEventData.

The camera must be in the disconnected state in order for this method to work.

3.3.13 CreateFeature

Description

Creates a GenICam feature with the specified name, type and access mode.

[C++]
```
int CreateFeature (unsigned short type, const char* name, const char*
category, unsigned short access,
   const char* description = NULL, unsigned short namespace =
FEATURE_NAMESPACE_STANDARD);
```

Parameters [C/C++]

[in] `unsigned short type`
Type of the feature to be created. Can be one of the following values:
FEATURE_TYPE_UINTEGER - unsigned 32-bit integer feature, maps to a slider with value, minimum, maximum and increment
FEATURE_TYPE_INTEGER - signed 32-bit integer feature, maps to a slider with value, min, maximum, increment and physical unit
FEATURE_TYPE_INTEGER64 - signed 62-bit integer feature, maps to a slider with value, min, maximum, increment and physical unit
FEATURE_TYPE_FLOAT - floating point 32-bit feature, maps to a slider with value, minimum, maximum and physical unit
FEATURE_TYPE_STRING - string feature, maps to an edit box showing a string of text
FEATURE_TYPE_BOOLEAN - boolean feature, maps to a check box
FEATURE_TYPE_COMMAND - command feature, maps to a command button
FEATURE_TYPE_ENUMERATION - enumeration feature, maps to dropdown box with a list of selectable items

[in] `const char* name`
Name of the feature to be created. Must not be NULL or empty.

[in] `const char* category`
GenICam category under which the feature will be created. This parameter should be either "Root" or the name of the existing category created via CreateCategory.

[in] `unsigned short access`
Access mode for the feature to be created. Can be one of the following values:
FEATURE_ACCESS_NA - feature is not available
FEATURE_ACCESS_RO - feature is available only for reading
FEATURE_ACCESS_WO - feature is available only for writing
FEATURE_ACCESS_RW - feature is fully available

[in] `const char* description`
String containing the description of the feature.

[in] `unsigned short namespace`
Namespace to which the feature will belong. Can be one of the following values:
FEATURE_NAMESPACE_STANDARD - feature will be created with the "Standard" namespace tag (recommended for features which follow the GenICam SFNC specifications).
FEATURE_NAMESPACE_CUSTOM - feature will be created with the "Custom" namespace tag (recommended for features with proprietary names).

Return Values

S_OK
  Success
E_FAIL
  Failure
E_NOINTERFACE
  Category does not exist
E_INVALIDARG
  Wrong feature type or access

Example

This fragment of code instantiates a camera object, creates an integer and enumerated features and sets up their attributes :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_INTEGER, "TestFeatureInt", "Root",
FEATURE_ACCESS_RW, "GigESim demo feature");
m_pCamera->SetFeatureRange("TestFeatureInt", 0, 1000, 1);

m_pCamera->CreateFeature(FEATURE_TYPE_ENUMERATION, "TestFeatureEnum", "Root",
FEATURE_ACCESS_RW);
m_pCamera->AddEnumEntry("TestEnumFeature", "First item", 1);
m_pCamera->AddEnumEntry("TestFeatureEnum", "Second item", 2);
m_pCamera->AddEnumEntry("TestFeatureEnum", "Third item", 3);
```

Remarks

Use this method to assign features to your virtual camera object. The assigned features will be exposed to GigE Vision client applications allowing them to remotely control parameters of the virtual camera.

The camera must be in the disconnected state in order for this method to work.

The creation of a feature should be followed by setting up its parameters. See SetFeatureRange, AddEnumEntry, SetFeatureIntValue, SetFeatureStingValue, SetFeatureElement, for more details.

Note that there are seven features mandatory for all GigE Vision cameras. They are created automatically along with a virtual camera object and therefore they do not require separate calls to CreateFeature. Those features are:

Standard Feature	Type	Default Value
Width	Integer	640
Height	Integer	480
PayloadSize	Integer	SizeX * SizeY
PixelFormat	Enumerated	"Mono8"
AcquisitionMode	Enumerated	Continuous
AcquisitionStart	Command	N/A
AcquisitionStop	Command	N/A

The default values of the mandatory features can be modified by calling SetImageSize and SetPixelFormat.

It is recommended to change the version of your virtual camera device each time you modify the set of features (see SetDeviceInfo). If this is not done, a client GigE Vision application may use a copy of an old XML information file from its cache folder and remain unaware of your changes. If you do not want to use the device version control, make sure to locate an XML cache folder of your client application and delete an XML file in it each time the set of features of the virtual camera has been modified. This should be done on each computer running the client software.

3.3.14 DeleteEnumEntry

Description

Deletes the specified entry from the enumerated feature.

Syntax

[C++]
```
int DeleteEnumEntry (const char* feature, const char* name);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the enumerated type.

[in] `const char* name`
Name of the entry to be deleted. Should not be NULL or empty.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature or entry does not exist

Example

This fragment of the code disconnects the virtual camera and deletes two previously created entries from an enumerated feature:

```
m_pCamera->Disconnect();
m_pCamera->DeleteEnumEntry("TestImageSelector", "Pattern 2");
m_pCamera->DeleteEnumEntry("TesImageSelector", "Pattern 3");
```

Remarks

Use this method to delete previously created features before creating a new set of features. Note that the camera must be in the disconnected state in order for **DeleteFeature** to work.

This method can also be used for deleting a category created by CreateCategory.

3.3.15 DeleteFeature

Description

Deletes the feature created by CreateFeature and frees the memory associated with it

Syntax

[C++]
```
int DeleteFeature (const char* name);
```

Parameters [C/C++]

[in] `const char* name`
Name of the feature to be deleted. Should not be NULL or empty.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist

Example

This fragment of the code disconnects the camera and deletes two previously created features:

```
m_pCamera->Disconnect();
m_pCamera->DeleteFeature("TestFeatureInt")
m_pCamera->DeleteFeature("TestFeatureEnum")
```

Remarks

Use this method to delete previously created features before creating a new set of features. Note that the camera must be in the disconnected state in order for **DeleteFeature** to work.

This method can also be used for deleting a category created by CreateCategory.

3.3.16 DeleteFeatureElement

Description

Deletes the specified element associated with the feature.

Syntax

[C++]
```
int DeleteFeatureElement ((const char* feature, const char* attribute,
const char* value = NULL);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to which the element belongs.

[in] `const char* element`
Name of the element to delete.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature or element does not exist

Example

This fragment of the code deletes the ToolTip element:

```
m_pCamera->Disconnect();
m_pCamera->DeleteFeatureElement("ExposureTime","ToolTip");
```

Remarks

The use of this method requires an advanced knowledge of the GenICam standard and XML syntax. For more information please refer to SetFeatureElement.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.17 Disconnect

Description

Disconnects the virtual camera from the currently selected interface and brings it offline.

Syntax

```
int Disconnect();
```

Parameters [C/C++]

None

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of a code disconnects a camera from the network and reconfigures its features:

```
m_pCamera->Disconnect();
m_pCamera->DeleteFeature("TestFeatureInt")
m_pCamera->CreateFeature(FEATURE_TYPE_STRING, "TestFeatureString", "Root",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureString("TestFeatureString", "Hello, World")
```

Remarks

Calling this method is equivalent to disconnecting a GigE Vision camera from the network. It stops all the acquisition and control threads and return the virtual camera into the configuration state. In this state new features can be added and existing features can be deleted.

To bring the camera back online, use Connect.

3.3.18 GetFeatureElement

Description

Returns the value of the specified element associated with the feature.

[C++]
```
int GetFeatureElement(const char* feature, const char* element, const char*
pValue);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to which the element belongs.

[in] `const char* element`
Name of the element. Must be part of the feature.

[in] `const char* value`
Pointer to a buffer that receives the string value of the element.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature or element does not exist

Example

The following fragment of the MFC code displays the value of the *ToolTip* element:

```
CString str;
char buf[256]
m_pCamera->GetFeatureElement("ExposureTime", "ToolTip", buf);
str.Format("%s", buf);
SetDlgItemText(IDC_EDIT, str);
return false;
```

Remarks

This method allows you to directly access elements of the XML-file linked to the virtual camera object.

The use of this method requires an advanced knowledge of the GenICam standard and XML syntax. For more information please refer to SetFeatureElement.

3.3.19 GetFeatureEnumList

Description

Returns the array of string values representing the specified enumerated feature.

[C++]

```
const char** GetFeatureEnumList(const char* feature, int* pSize);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the enumerated type.

[in] `int* pSize`
Pointer to a variable that receives the number of entries in the enumerated list.

Return Values

Array of strings containing the list of enumerated entries.

Example

This fragment of code instantiates a camera object, creates an enumerated feature and sets up a list of three entries:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();
m_pCamera->CreateFeature(FEATURE_TYPE_ENUMERATION, "TestImageSelector", "Root",
FEATURE_ACCESS_RW);

m_pCamera->setEnumEntry("TestImageSelector", "First pattern", 1);
m_pCamera->setEnumEntry("TestImageSelector", "Second pattern", 2);
m_pCamera->setEnumEntry("TestImageSelector", "Third pattern", 3);
```

Remarks

If the specified feature is not of the enumerated type or not existent, the method will return zero.

3.3.20 GetFeatureIntRange

Description

Returns the minimum, maximum and increment attributes of the specified feature.

[C++]
```
int GetFeatureIntRange(const char* feature, __int64* pMin, __int64* pMax,
int* pInc);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the attributes to. Must be an existing feature of the integer or floating point type.

[in] `__int64* pMin`
Pointer ot a value that receives the feature's minimum.

[in] `__int64* pMax`
Pointer to a value that receives the feature's maximum.

[in] `__int* pInc`
Pointer to a value that receives the feature's increment.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type

Example

This fragment of code instantiates a camera object, creates an integer feature, sets up its range and verifies the values of the minimum, maximum and increment:

```
static CGevCamera* m_pCamera; __int64 Min, Max; int Inc;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_INTEGER, "TestFeatureInt", "Root",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureRange("TestFeatureInt", 0, 1000, 5);
m_pCamera->SetFeatureIntValue("TestFeatureInt", 500);
m_pCamera->GetFeatureIntRange("TestFeatureInt", &min, &max, &inc);
```

Remarks

This method will work only if SetFeatureRange or SetFeatureIntRange was previously called for the specified feature.

3.3.21 GetFeatureIntValue

Description

Returns the integer value of the specified feature.

[C++]
```
int GetFeatureIntValue(const char* feature, __int64* pValue);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the integer, enumerated or boolean type.

[out] `__int64* pValue`
Pointer to a variable that receives the value of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type

Example

This fragment of an MFC code uses a feature-write callback to request and display the name and value of a feature after it has just been modified by a client application:

```
bool CGigemuDlg::onFeatureWrite(const char* feature)
{
 CString str; __int64 iVal;
 m_pCamera->GetFeatureIntValue(feature, &iVal);
 str.Format(_T("Feature: %S - %i"), feature, iVal);
 SetDlgItemText(IDC_FEATURE_STATIC, str);
 return false;
}
```

Remarks

Depending on the type of the feature the *pValue* argument has the following meaning:

Feature Type	Value
Integer	Integer value of the feature
Boolean	0 if False, 1 if True
Enumerated	Numerical value of a currently selected item

This method is typically used as part of a feature-read callback in order to obtain the new value of a feature after it has been modified by an external client application. See SetWriteCallback for more details.

3.3.22 GetFeatureFloatValue

Description

Returns the value of the specified floating point feature.

[C++]
```
int GetFeatureIntValue(const char* feature, double* pValue);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the floating point type.

[out] `double* pValue`
Pointer to a variable that receives the value of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type

Example

This fragment of an MFC code uses a feature-write callback to request and display the name and value of a feature after it has just been modified by a client application:

```
bool CGigemuDlg::onFeatureRead(const char* feature)
{
 CString str; double fVal;
 m_pCamera->GetFeatureFloatValue(feature, &fVal);
 str.Format(_T("FeatureRead: %S - %7.2f"), feature, fVal);
 SetDlgItemText(IDC_FEATURE_STATIC, str);
 return false;
}
```

Remarks

This method is typically used as part of a feature-read callback in order to obtain the new value of a feature after it has been modified by an external client application. See SetWriteCallback for more details.

3.3.23 GetFeatureRange

Description

Returns the minimum and maximum attributes of the specified feature.

[C++]
```
int GetFeatureRange(const char* feature, float* pMin, float* pMax);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the attributes to. Must be an existing feature of the integer or floating point type.

[out] `float* pMin`
Pointer to a value that receives the feature's minimum.

[out] `float* pMax`
Pointer to a value that receives the feature's maximum.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist

Example

This fragment of code instantiates a camera object, creates a floating point feature, sets up its range and verifies the values of the minimum and maximum :

```
static CGevCamera* m_pCamera; double min, max;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_FLOAT, "TestFeatureFloat", "Root",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureRange("TestFeatureFloat", 0, 1000);
m_pCamera->SetFeatureFloatValue("TestFeatureFloat", 3.1415926);
m_pCamera->GetFeatureRange("TestFeatureFloat", &min, &max);
```

Remarks

This method will work only if SetFeatureRange or SetFeatureIntRange was previously called for the specified feature.

3.3.24 GetFeatureRegister

Description

Returns the address of the virtual hardware register associated with the specified feature.

[C++]
```
int GetFeatureRegister(const char* feature, unsigned int* pAddr);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the integer, enumerated or boolean type.

[out] `unsigned int* pAddr`
Pointer to a variable that receives a 32-bit address of the register.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist

Example

This fragment of an MFC code displays the address and value of the register associated with the Gain feature:

```
CString str;
unsigned int addr; int iVal
m_pCamera->GetFeatureRegister("Gain", &addr);
m_pCamera->ReadRegister(addr,&iVal)
str.Format(_T("Register address: %X \n Register value: %i"), addr,
iVal);
SetDlgItemText(IDC_REGISTER_STATIC, str);
```

Remarks

This method allows you to obtain the address the register associated with the feature. The address can be further used for a direct register access by calling ReadRegister and WriteRegister.

3.3.25 GetFeatureStringPointer

Description

Returns the string value of the specified feature.

[C++]
```
int GetFeatureStringPointer(const char* feature, char* pValue);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the string or enumerated type.

[out] `const char** pValue`
Pointer to a memory location that will receive a pointer to the string value of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type

Example

This fragment of an MFC code uses a feature-write callback to request and display the name and value of a string feature after it has just been modified by a client application:

```
bool CGigemuDlg::onFeatureRead(const char* feature)
{
 CString str; int iVal;
 char* buf;
 m_pCamera->GetFeatureStringPointer(feature, &buf);
 str.Format(_T("FeatureRead: %S - %s"), feature, buf);
 SetDlgItemText(IDC_FEATURE_EDIT, str);
 return false;
}
```

Remarks

Depending on the type of the feature the *pValue* argument has the following meaning:

Feature Type	Value
String	String value of the feature
Enumerated	String value of a currently selected item

This method is typically used as part of a feature-read callback in order to obtain the new value of a feature after it has been modified by an external client application. See SetWriteCallback for more details.

Unlike GetFeatureStringValue, **GetFeatureStringPointer** does not copy the string data to the destination buffer.

3.3.26 GetFeatureStringValue

Description

Returns the string value of the specified feature.

[C++]
```
int GetFeatureStringValue(const char* feature, char* value, unsigned int
dstSize);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature. Must be an existing feature of the string or enumerated type.

[out] `const char* value`
Pointer to a buffer that receives the string value of the feature.

[in] `unsigned int dstSize`
Maximum size of the buffer to receive the string value.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type

Example

This fragment of an MFC code uses a feature-write callback to request and display the name and value of a string feature after it has just been modified by a client application:

```
bool CGigemuDlg::onFeatureRead(const char* feature)
{
 CString str; int iVal;
 char buf[256]
 m_pCamera->GetFeatureStringValue(feature, buf, 256);
 str.Format(_T("FeatureRead: %S - %s"), feature, buf);
 SetDlgItemText(IDC_FEATURE_EDIT, str);
 return false;
}
```

Remarks

Depending on the type of the feature the *pValue* argument has the following meaning:

Feature Type	Value
String	String value of the feature
Enumerated	String value of a currently selected item

This method is typically used as part of a feature-read callback in order to obtain the new value of a feature after it has been modified by an external client application. See SetWriteCallback for more details.

3.3.27 GetFormatStringFromValue

Description

Returns the name of the pixel format specified by its numerical ID.

[C++]
```
const char* GetFormatStringFromValue(unsigned int formatID = 0);
```

Parameters [C/C++]

[in] `unsigned int formatID`
Integer containing the numerical ID of the pixel format.

Return Values

String containing the name of the pixel format.
NULL if the format is not recognized.

Example

The following line of code converts a format ID into a string :

```
char* FormatId=m_pCamera->GetFormatValueFromString(0x010C0027);
```

Remarks

This method converts the numerical value of the pixel format to its string name. For the list of possible format values refer to the table in SetPixelFormat.

3.3.28 GetFormatValueFromString

Description

Returns the numerical code of the specified pixel format.

[C++]
```
unsigned int GetFormatValueFromSting(const char* formatName = 0);
```

Parameters [C/C++]

[in] `const char* formatName`
String containing the name of the pixel format.

Return Values

Integer containing the GigE Vision value of the current pixel format.
E_INVALIDARG if the format is not recognized.

Example

The following line of code returns the numerical code of the "Mono12Packed" format :

```
int FormatId=m_pCamera->GetFormatValueFromString("Mono12Packed");
```

Remarks

This method converts the string value of the pixel format to its numerical GigE Vision format ID. For the list of possible format values refer to the table in SetPixelFormat.

3.3.29 GetHeight

Description

Returns the current vertical size of outgoing images.

[C++]
```
int GetHeight(unsigned int channelIndex = 0);
```

Parameters [C/C++]

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

Currently selected vertical size of outgoing images

Example

This fragment of an MFC code uses the image size information to generate a periodic image pattern:

```
void GenerateImage(unsigned char* buffer)
{
   for (unsigned int i = 0; i < m_pCamera->GetHeight(); i++)
   {
         unsigned char *ptr = buffer + i*m_pCamera->GetWidth();
         unsigned char *end = ptr + m_pCamera->GetWidth();
         while(ptr < end)
               *ptr++=i;
   }
}
```

Remarks

This method returns the value of the *Height* feature of the virtual camera. It can be modified externally by a client application or internally by calling SetImageSize.

Width, Height, PixelFormat and *PayloadSize* are mandatory GigE Vision features which are automatically created when a virtual camera object is instantiated.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.30 GetInterfaceCount

Description

Returns the number of network interfaces available in the system.

Syntax

[C++]
```
int GetInterfaceCount();
```

Parameters [C/C++]

None

Return Values

Integer value containing the number of network interfaces found in the system. If zero, no

Example

This fragment of an MFC code fills out a combo box with IP addresses of network interfaces installed on the system.

```
for (int i = 0; i < m_pCamera->GetInterfaceCount(); i++)
{
    ULONG32 ip = m_pCamera->GetInterfaceAtIndex(i);
    TCHAR ipstr[32];
    ip2str(ipstr, 32, ip);
    m_interfaceList.AddString(ipstr);
}
m_interfaceList.SetCurSel(0);
```

Remarks

This method is typically used in combination with GetInterfaceAtIndex to list all network interfaces installed on the system. A virtual camera can then be bound to one of the interfaces by calling Connect.

3.3.31 GetInterfaceInfoAtIndex

Description

Returns the parameters of the network interface at the specified index in the system interface list.

Syntax

[C++]
```
int GetInterfaceInfoAtIndex(unsigned int index, InterfaceInfo* pInfo)
```

Parameters [C/C++]

[in] `unsigned int index`
Index of the requested interface in the system interface list.

[out] `const InterfaceInfo* advOpt`
Pointer to an InterfaceInfo structure containing the following fields:
`unsigned int ip`
The IPv4 address of the interface in network byte order.
`unsigned char macAddr[8]`
The MAC address of the interface in the IEEE-802 format (six groups of two hexadecimal digits separated by hyphens).
`unsigned int mask`
The subnet mask of the interface in network byte order.
`unsigned int gateway`
The IPv4 address of the gateway in network byte order.
`unsigned int speed`
The speed of the interface in bits per second.
`unsigned int mtu`
The Maximum Transmission Unit (MTU) size in bytes.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Index outside of range

Example

This fragment of code browses through the network interfaces and fills out an array of records with the information about each interface.

```
InterfaceInfo Info[16];
for (int i = 0; i < m_pCamera->GetInterfaceCount(); i++)
{
    char buf[20];
    m_pCamera->GetInterfaceInfoAtIndex(i, Info);
}
```

Remarks

This method is typically used in combination with GetInterfaceCount to list all network interfaces installed on the system. A virtual camera can then be bound to one of the interfaces by calling Connect.

The list of system interfaces contains all network interfaces including Wi-Fi and some virtual interfaces that are not compatible with the GigE Vision standard. They may be utilized for connecting to a client application on the local host, but only GigE interfaces should be used for remote connections.

Remarks

This method is typically used in combination with GetInterfaceAtIndex to retrieve information about network interfaces installed on the system. A virtual camera can then be bound to one of the interfaces by calling Connect.

The list of system interfaces contains all network interfaces including Wi-Fi and some virtual interfaces that are not compatible with the GigE Vision standard. They may be utilized for connecting to a client application on the local host, but only GigE interfaces should be used for remote connections.

3.3.32 GetInterfaceAtIndex

Description

Returns the IP address of the network interface at the specified index in the system interface list.

Syntax

[C++]
```
int GetInterfaceAtIndex(unsigned int index, static char* pIP)
```

Parameters [C/C++]

[in] `unsigned int index`
Index of the requested interface in the system interface list.

[out] `const char* pIP`
Pointer to a 16-byte buffer that receives the string value of the IP address in the IPv4 format (four decimals in the range of 0-255 separated by dots).

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Index outside of range

Example

This fragment of an MFC code fills out a combo box with IP addresses of network interfaces installed on the system.

```
for (int i = 0; i < m_pCamera->GetInterfaceCount(); i++)
{
    char buf[20];
    m_pCamera->GetInterfaceAtIndex(i, buf);
    m_interfaceList.AddString(buf);
}
m_interfaceList.SetCurSel(0);
```

Remarks

This method is typically used in combination with GetInterfaceCount to list all network interfaces installed on the system. A virtual camera can then be bound to one of the interfaces by calling Connect.

The list of system interfaces contains all network interfaces including Wi-Fi and some virtual interfaces that are not compatible with the GigE Vision standard. They may be utilized for connecting to a client application on the local host, but only GigE interfaces should be used for remote

connections.

3.3.33 GetIpAddress

Description

Returns the current IP configuration of the virtual camera device.

[C++]
```
int GetIpAddress(char ip[16], char mask[16], char gateway[16]);
```

Parameters [C/C++]

[out] `char ip[16]`
Pointer to a buffer that receives the string value of the ip address of the camera in the IPv4 format (four decimals in the range of 0-255 separated by dots).

[out] `const mask[16]`
Pointer to a buffer that receives the string value of the subnet mask in the IPv4 format.

[out] `const gateway[16]`
Pointer to a buffer that receives the string value of the gateway in the IPv4 format.

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Wrong IPv4 format

Example

This fragment of code retrieves the current IP configuration of the virtual camera:

```
char ip[16], mask[16], gateway[16];
m_pCamera->GetIpAddress (ip, mask, gateway);
```

Remarks

Typically the IP configuration of a virtual camera device coincides with the IP configuration of the network adapter it is connected to. There are two cases though when the camera's IP address can differ from the NIC's IP: 1) if an external client application modified the camera's IP address through the Force IP protocol; 2) if the camera's IP address is set by the SetIpAddress method.

To obtain the IP address of the network adapter (as opposed to the camera's IP), use GetInterfaceAtIndex.

3.3.34 GetPayloadSize

Description

Returns the current payload size of outgoing images.

[C++]
```
int GetPayloadSize(unsigned int channelIndex = 0);
```

Parameters [C/C++]

[out] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

Currently selected payload size of outgoing images

Example

This fragment of an MFC code uses the image size information to generate a periodic image pattern:

```
void GenerateImage(unsigned char* buffer)
{
   unsigned char *ptr = buffer;
   for (unsigned int i = 0; i < m_pCamera->GetPayloadSize(); i++)
   {
               *ptr++=i;
   }
}
```

Remarks

The payload size is typically equal to the size of each image in bytes, but can be larger when chunk data are appended to each image frame.

Width, Height, PixelFormat and *PayloadSize* are mandatory GigE Vision features which are automatically created when a virtual camera object is instantiated.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.35 GetPixelFormatValue

Description

Returns the numerical value of the current pixel format of outgoing images.

[C++]
```
unsigned int GetPixelFormatValue(unsigned int channelIndex = 0);
```

Parameters [C/C++]

[out] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

Integer containing the GigE Vision value of the current pixel format.

Example

This fragment of code sets and verifies the current pixel format :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetImageSize (1024, 768);
m_pCamera->AddPixelFormat("Mono16");
m_pCamera->AddPixelFormat("RGB8Packed");
m_pCamera->SetPixelFormat("Mono16");
int FormatId=m_pCamera->GetPixelFormatValue();
```

Remarks

This method returns the numerical value of the *PixelFormat* feature of the virtual camera. It can be modified externally by a client application or internally by calling SetPixelFormat.

Width, Height, PixelFormat and *PayloadSize* are mandatory GigE Vision features which are automatically created when a virtual camera object is instantiated.

For the list of possible format values refer to the table in SetPixelFormat.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.36 GetPixelFormatString

Description

Returns the string value of the current pixel format of outgoing images.

[C++]
```
const char* GetPixelFormatString(unsigned int channelIndex = 0);
```

Parameters [C/C++]

[out] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

String containing the name of the current pixel format.

Example

This fragment of code sets and verifies the current pixel format :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetImageSize (1024, 768);
m_pCamera->AddPixelFormat("Mono16");
m_pCamera->AddPixelFormat("RGB8Packed");
m_pCamera->SetPixelFormat("Mono16");
CString  Fmt=m_pCamera->GetPixelFormatString();
```

Remarks

This method returns the string value of the *PixelFormat* feature of the virtual camera. It can be modified externally by a client application or internally by calling SetPixelFormat.

Width, Height, PixelFormat and *PayloadSize* are mandatory GigE Vision features which are automatically created when a virtual camera object is instantiated.

For the list of possible format names refer to the table in SetPixelFormat.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.37 GetWidth

Description

Returns the current horizontal size of outgoing images.

[C++]
```
int GetWidth(unsigned int channelIndex = 0);
```

Parameters [C/C++]

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

Currently selected horizontal size of outgoing images

Example

This fragment of an MFC code uses the image size information to generate a periodic image pattern:

```
void GenerateImage(unsigned char* buffer)
{
   for (unsigned int i = 0; i <  m_pCamera->GetHeight(); i++)
   {
         unsigned char *ptr = buffer + i* m_pCamera->GetWidth();
         unsigned char *end = ptr +  m_pCamera->GetWidth();
         while(ptr < end)
               *ptr++=i;
   }
}
```

Remarks

This method returns the value of the *Width* feature of the virtual camera. It can be modified externally by a client application or internally by calling SetImageSize.

Width, Height, PixelFormat and *PayloadSize* are mandatory GigE Vision features which are automatically created when a virtual camera object is instantiated.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.38 IsConnected

Description

Returns the network connection state of the virtual camera.

[C++]
```
bool IsConnected();
```

Parameters [C/C++]

none

Return Value

TRUE if the virtual camera is in the connected state; FALSE if in the disconnected state.

Example

This fragment of code checks a dialog button depending on the state of the camera:

```
bool bConnected=m_pCamera->IsConnected();
CheckDlgButton(IDB_CONNECTED, bConnected);
```

Remarks

For more information on binding the camera to a network adapter refer to the Connect and Disconnect methods.

3.3.39 LockFormat

Description

Blocks the execution of the image streaming thread until the image format change requested by a client application is finalized and the acquisition mode is turned on.

Syntax

int LockFormat(unsigned int channelIndex = 0);

Parameters [C/C++]

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This function runs the image generation and transfer cycle automatically synchronized with the client application:

```
void CGigemuDlg::videoGenerator()
{
  int width;
  int height;
  int i=0;
  char buffer[MAXWIDTH*MAXHEIGHT];
  char *ptr;
  while (!m_exitThread)
  {
    m_pCamera->LockFormat();
    width=m_pCamera->GetWidth();
    height=m_pCamera->GetHeight();
    for (ptr=buffer; ptr < buffer+width*heigh; ptr++)
     *ptr=i++;
    m_pCamera->SendImage(buffer);
  }
}
```

Remarks

Use this method to automatically synchronize your image streaming cycle with an external client application. **LockFormat** serves two purposes:
1) It allows you to start your image streaming cycle without waiting for the client application to issue the *AcquisitionStart* command. LockFormat will block the execution of the thread until the *AcquisitionStart* command is received from the client.
2) It lets you generate your images based on the image size and format returned by GetWidth,

GetHeight and GetPixelFormat without being concerned about those parameters being modified by the client application in the middle of the image transfer. If a format change request is received from the client, **LockFormat** will block the execution of the thread until the virtual camera finalizes the format change procedure.

Using this method requires you to run the image streaming cycle in a separate thread in order to prevent a lock up of the application.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.40 PixelFormatConvert

Description

Converts image data from a plain pixel format to the specified output format.

[C++]
```
unsigned int PixelFormatConvert(const char* pSrc, char* pDst,
unsigned int width, unsigned int height, unsigned int inputFormat,
   unsigned int outputFormat, unsigned int offsetX =
0, unsigned int offsetY = 0, unsigned int sizeX = width, unsigned int sizeY
= height);
```

Parameters [C/C++]

[in] `const char* pSrc`
Pointer to the buffer containing the input image (top left pixel). The image data must be compatible with the pixel format specified by the *inputFormat* parameter.

[in] `const char* pDst`
String containing the name of the pixel format.

[in] `unsigned int width`
Integer value specifying the width of the image.

[in] `unsigned int height`
Integer value specifying the height of the image.

[in] `unsigned int inputFormat`
Integer value specifying the numerical ID of the input image format.

[in] `unsigned int outputFormat`
Integer value specifying the numerical ID of the output image format.

[in] `unsigned int offsetX`
Optional integer value specifying the horizontal offset of the upper left corner of the output image relative to the input one's.

[in] `unsigned int offsetY`
Optional integer value specifying the vertical offset of the upper left corner of the output image relative to the input one's.

[in] `unsigned int sizeX`
Optional integer value specifying the horizontal size of the output image. If zero or omitted, the right bottom corner of the output image will coincide with the input one's.

[in] `unsigned int sizeY`
Optional integer specifying the vertical size of the output image. If zero or omitted, the right bottom corner of the output image will coincide with the input one's.

Return Values

S_OK
Success

E_FAIL
Failure
E_INVALIDARG
Unsupported format ID

Example

The following fragment of code converts the image data from RGB8 to BayerGB8 pixel format :

```
char* pOutputImage;
pOutputImage=new char*  [m_iWidth*m_iHeight];
int inputFormat=GetFormatValueFromString("RGB8");
int outputFormat=GetFormatValueFromString("BayerGB8");
PixelFormatConvert (pInputImage, pOutputImage, m_iWidth, m_iHeight, inputFormat,
outputFormat)
```

Remarks

This method converts an image data array presented in a plain pixel format into an output image data array encoded in a specified pixel format. It is typically used to reduce the streaming bandwidth by converting an input image into one of the packed pixel formats. For example, converting an input image of the RGB8 type into a raw Bayer8 image will reduce the size of each frames by a factor of 3.

The following input pixel formats are currently supported: Mono8, Mono16, RGB8, RGB16

The following output pixel formats are currently supported: Mono10Packed, Mono12Packed, Bayer**8, Bayer**10, Bayer**12, Bayer**10Packed, Bayer**12Packed.

Optional parameters *offsetX*, *offsetY*, *sizeX* and *sizeY* allow you to optimize the processing speed by performing the format conversion on a selected rectangular region in the input pixel array. If this parameters are used, they will define the size of the output image array.

For the list of pixel format values and associated number of bits per pixel refer to the table in SetPixelFormat.

3.3.41 ReadMemory

Description

Reads a block of data from the internal camera memory starting from the specified bootstrap address.

[C++]
```
int ReadMemory(unsigned int addr, void* pAddr, unsigned int size);
```

Parameters [C/C++]

[in] `unsigned int* addr`
Starting address in the camera bootstrap address space.

[out] `unsigned int* pAddr`
Pointer to a buffer that receives the content of the memory.

[in] `unsigned int size`
Size of memory to read in bytes.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Bootstrap address does not exist

Example

This fragment of an MFC code reads a block of data from the camera's bootstrap memory:

```
CString str;
unsigned char buffer[1024];
m_pCamera->ReadMemory(0xA000,buffer,1024)
```

3.3.42 ReadRegister

Description

Returns the value of the specified virtual camera register.

[C++]
```
int ReadRegister(unsigned int addr, unsigned int* pValue);
```

Parameters [C/C++]

[in] `unsigned int* addr`
Address of the register to read.

[out] `unsigned int* pAddr`
Pointer to a variable that receives the value of the register.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Register does not exist

Example

This fragment of an MFC code displays the address and value of the register associated with the Gain feature:

```
CString str;
unsigned int addr; int iVal
m_pCamera->GetFeatureRegister("Gain", &addr);
m_pCamera->ReadRegister(addr,&iVal)
str.Format(_T("Register address: %X \n Register value: %i"), addr,
iVal);
SetDlgItemText(IDC_REGISTER_STATIC, str);
```

Remarks

This method allows you to read the value of the virtual camera register. To obtain the address of the register associated with a specific feature, use GetFeatureRegister.

3.3.43 ResetTimer

Description

Resets the internal timer of the virtual camera to zero.

[C++]
```
int ResetTimer();
```

Parameters [C/C++]

None

Return Values

S_OK
Success
E_FAIL
Failure

Example

This line of code resets the internal camera timer:

```
m_pCamera->ResetTimer();
```

Remarks

The internal *GigESim* timer is used to integrate timestamps into each frame streamed to the network by the SendImage function. It is also used to handle schedule action commands.

Note that this method will work only if the Timer Mode is set to UDP. When the timer is set to the UNIX (IEEE-1588) mode, it is always synchronized with the astronomical time and therefore cannot be reset.

3.3.44 SendEvent

Description

Sends the specified event message to the network.

[C++]
```
int SendEvent(unsigned short eventId, unsigned short channel = 0xFFFF, bool requireAck = true);
```

Parameters [C/C++]

[in] `unsigned short eventID`
Integer value specifying the numerical ID of the event.

[in] `unsigned int channel`
Integer value specifying the index of the streaming channel associated with the event. Default value of 0xFFFF specifies that no streaming channel is involved.

[in] `bool requireAck`
If TRUE, the virtual camera will wait for the acknowledge reply from the GigE Vision client.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This line of code generates a single event and sends it over the network:

```
m_pCamera->SendEvent(0x8E94);
```

Remarks

This method generates a standard EVENT message per GigE Vision specifications. It is typically associated with a certain action of the virtual camera of which a GigE Vision client should be notified. The event will contain a timestamp indicating the exact time at which the event was generated. Depending on the Timer Mode, the timestamp will be reported either in the astronomical time or time elapsed from the start of the virtual camera application.

The *channelIndex* parameter should be used only if your virtual camera supports multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

If you want to associate your event messages with GenICam event features, you should call CreateEventCategory and CreateEvent prior to calling **SendEvent**.

Note that this method generates an EVENT message as opposed to EVENTDATA message generated by SendEventData.

3.3.45 SendEvents

Description

Sends multiple event messages to the network.

[C++]
```
int SendEvents(unsigned int eventCount, unsigned short eventIds[], unsigned short channel = 0xFFFF, bool requireAck = true);
```

Parameters [C/C++]

[in] `unsigned short eventCount`
Integer value specifying the amount of events to be sent.

[in] `unsigned short eventIDs[]`
Integer array containing the numerical IDs of the events.

[in] `unsigned int channel`
Integer value specifying the index of the streaming channel associated with the events. Default value of 0xFFFF specifies that no streaming channel is involved.

[in] `bool requireAck`
If TRUE, the virtual camera will wait for the acknowledge reply from the GigE Vision client.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code fires three event messages to the network:

```
unsigned short eventIDs[0x8E94,0x8E95,0x8E96];
m_pCamera->SendEvents(3, eventIDs);
```

Remarks

This method generates standard EVENT messages per GigE Vision specifications. It is typically associated with a certain action of the virtual camera of which a GigE Vision client should be notified. Each event will contain a timestamp indicating the exact time at which the event was generated. Depending on the Timer Mode, the timestamp will be reported either in the astronomical time or time elapsed from the start of the virtual camera application.

The *channelIndex* parameter should be used only if your virtual camera supports multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

If you want to associate your event messages with GenICam event features, you should call CreateEventCategory and CreateEvent prior to calling **SendEvents**.

Note that this method generates EVENT messages as opposed to EVENTDATA messages generated by SendEventData.

3.3.46 SendEventData

Description

Sends the specified data event message to the network.

[C++]
```
int SendEventData(unsigned short eventId, const char* data, unsigned int
size,
        unsigned short channel = 0xFFFF, bool requireAck = true);
```

Parameters [C/C++]

[in] `unsigned short eventID`
Integer value specifying the numerical ID of the event.

[in] `const char* data`
Pointer to the block of memory containing data fields of the event.

[in] `unsigned int size`
Integer value specifying the size of the data block in bytes. Must not exceed 540.

[in] `unsigned int channel`
Integer value specifying the index of the streaming channel associated with the event. Default value of 0xFFFF specifies that no streaming channel is involved.

[in] `bool requireAck`
If TRUE, the virtual camera will wait for the acknowledge reply from the GigE Vision client.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment code generates a data event containing the current timestamp and sends it over the network:

```
__int64 time;
time=m_pCamera->GetTimer();
SendDataEvent(0x8E94, (char*) time, 8);
```

Remarks

This method generates a standard EVENTDATA message per GigE Vision specifications. It is typically associated with a certain action of the virtual camera of which a GigE Vision client should be notified. The event will contain a timestamp indicating the exact time at which the event was generated. Depending on the Timer Mode, the timestamp will be reported either in the astronomical time or time elapsed from the start of the virtual camera application.

The *channelIndex* parameter should be used only if your virtual camera supports multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

If you want to associate your data event message with GenICam event features and data members, you should call CreateEventCategory, CreateEvent and CreateEventFeature prior to calling **SendEventData**.

Note that this method generates an EVENTDATA message as opposed to EVENT message generated by SendEvent.

3.3.47 Send3DImage

Description

Streams the specified 3D point cloud image array to the network.

[C++]
```
int Send3DImage(const char* frame, unsigned __int64 points, unsigned int channelIndex = 0);
```

Parameters [C/C++]

[in] `const char* frame`
Pointer to the beginning of the point cloud array.

[in] `unsigned int points`
The amount of 3D points in the array. Each point must be represented by three coordinate values (X, Y, Z).

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code repeatedly sends a 3D point cloud image array over the network.

```
   m_pCamera->SetImageSize(width, height);
   m_pCamera->SetMaxImageSize(width, height);
   m_pCamera->SetPixelFormat("Coord3D_ABC32f");
   m_pCamera->SetGevMode(CGevCamera::GEV_VERSION_MODE::GEV_2);
   m_pCamera-
>SetPayloadType(CGevCamera::PAYLOAD_TYPE::PAYLOAD_3DPOINTCLOUD);


   if (m_pCamera->connect(m_interfaceList.GetCurSel())
   {
    m_exitThread = false;
    m_thread = CreateThread(NULL, NULL,
   (LPTHREAD_START_ROUTINE)videoGenThread, this, 0, NULL);
   }

   void CGigemuDlg::videoGenerator()
   {
      int ret;
      while (!m_exitThread)
      {
```

```
            ret=m_pCamera->LockFormat();
            ret=m_pCamera->Send3DImage(m_points3D.get(), m_total3DPoints);
            Sleep(20);
        }
    }
```

Remarks

This function is typically called in a cycle to stream a series of 3D point cloud frames from the virtual camera to the network. The camera must be in the connected state and the acquisition must have been started by a client application.

Note that for the correct operation of this function GevMode must be set to GEV_2, PixelFormat must be set to Coord3D_ABC16 (16-bit integer coordinates) or Coord3D_ABC32f (32-bit floating point coordinates) and the type of data in the array must correspond to the selected pixel format.

If the **Send3DImage** cycle is called before the acquisition is started by the client, the LockFormat method must be called prior to calling **Send3DImage** in order to lock the execution of the thread until the *AcquisitionStart* command is issued by the client.

Each image streamed with this function will contain a timestamp per GigE Vision standard indicating the exact time at which the image was generated. Depending on the Timer Mode, the timestamp will be reported either in the astronomical time or time elapsed from the start of the virtual camera application.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.48 SendImage

Description

Streams the specified image frame to the network.

[C++]
```
int SendImage(const char* frame, unsigned int channelIndex = 0, unsigned int sizeY
= 0);
```

Parameters [C/C++]

[in] `const char* frame`
Pointer to the beginning of the image data (top left pixel). The image data format must be compatible with the one set via SetImageSize and SetPixelFormat.

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

[in] `unsigned int sizeY`
The vertical size of the frame to send. Can be used for simulating a line scan camera with a variable frame size. If zero or omitted, the full-size frame will be sent.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code generates sequential monochrome video frames with a random pattern and sends them over the network

```
if (m_pCamera->connect(m_interfaceList.GetCurSel());
m_exitThread = false;
m_thread = CreateThread(NULL, NULL,
(LPTHREAD_START_ROUTINE)videoGenThread, this, 0, NULL);

void CGigemuDlg::videoGenerator()
{
  int width;
  int height;
  char buffer[MAXWIDTH*MAXHEIGHT];
  char *ptr;
  int i=0;
  while (!m_exitThread)
  {
    m_pCamera->LockFormat();
    width=m_pCamera->GetWidth();
    height=m_pCamera->GetHeight();
    for (ptr=buffer; ptr < buffer+width*heigh; ptr++)
     *ptr=i++;
```

```
      m_pCamera->SendImage(buffer);
      Sleep(20);
    }
  }
```

Remarks

This function is typically called in a cycle to stream a series of image frames from the virtual camera to the network. The camera must be in the connected state and the acquisition must have been started by a client application.

If the **SendImage** cycle is called before the acquisition is started by the client, the LockFormat method must be called prior to calling **SendImage** in order to lock the execution of the thread until the *AcquisitionStart* command is issued by the client.

Each image streamed with this function will contain a timestamp per GigE Vision standard indicating the exact time at which the image was generated. Depending on the Timer Mode, the timestamp will be reported either in the astronomical time or time elapsed from the start of the virtual camera application.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.49 SetActionCount

Description

Sets the amount of action signals supported by the virtual camera.

[C++]
```
int SetActionCount(unsigned int count);
```

Parameters [C/C++]

[in] `unsigned int count`
Integer value specifying the amount of stream channels in the virtual camera. Possible values are 1 or 2.

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Value out of range

Example

This fragment of code instantiates a camera object and sets the amount of action signals to 2:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetActionCount(2);
```

Remarks

This method allows you to configure the virtual camera for the amount of separate action signals it will be able to process. It must be called in order for action commands to be supported.

For more information on action commands refer to SetActionCallback.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.50 SetAdvancedOptions

Description

Sets advanced operational parameters for the virtual camera.

[C++]
```
int SetAdvancedOptions(const AdvancedOptions* advOpt);
```

Parameters [C/C++]

[in] `const AdvancedOptions* advOpt`
Pointer to an AdvancedOptions structure containing the following fields:

`bool` *`disorderEnabled [false]`*
If true, enables packets to be sent in a random order.

`bool` *`dropEnabled [false]`*
If true, enables the packet drop simulation.

`unsigned int` *`dropPeriod`* `[1]`
Interval in frames between packet drops.

`unsigned int` *`dropFrom [0]`*
Index of the first packet in a frame to be dropped.

`unsigned int` *`dropCount [1]`*
The amount of sequential packets to drop in a frame.

`unsigned int` *`interPacketDelay [0]`*
The delay to insert between sequential stream packets in timestamp units (reversed byte order).

`unsigned int` *`readDelay [0]`*
Time in milliseconds which will take for the virtual camera reply to a read command.

`unsigned int` *`writeDelay [0]`*
Time in milliseconds which will take for the virtual camera reply to a write command.

`unsigned int` *`readAttempts [0]`*
The number of repetitions of each read request necessary for the virtual camera to respond.

`unsigned int` *`writeAttempts [0]`*
The number of repetitions of each write request necessary for the virtual camera to respond.

`unsigned int` *`discoveryDelay [0]`*
Time in milliseconds which will take the virtual camera to respond to a discovery request from a client application

`unsigned int` *`discoveryAttempts [0]`*
The number of repetitions of each discovery request necessary for the virtual camera to respond with a discovery message

Return Values

S_OK
  Success
E_FAIL
  Failure
E_INVALIDARG
  Unsupported format

Example

This line of code sets up the packet drop options. As a result, 2 consecutive packets will be dropped in every 4th frame starting from packet #10:

```
CGevCamera::AdvancedOptions options;
memcpy(&options, 0, sizeof(CGevCamera::AdvancedOptions));
options.dropEnabled=true;
options.dropPeriod=3;
options.dropFrom=10;
options.dropCount=2;
options.interPacketDelay=htonl(1000);
m_pCamera->SetAdvancedOptions(&options);
```

Remarks

Use this method allows you to simulate unusual and erroneous conditions for a virtual camera.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.51 SetChunkMode

Description

Enables/disables the chunk streaming mode and sets the size of the chunk data block.

[C++]
```
int SetChunkMode(unsigned int chunkSize, unsigned int chunkCount, unsigned int channelIndex=0);
```

Parameters [C/C++]

[in] `unsigned int chunkSize`
Integer value specifying the total size of the chunk data fields in bytes. If zero, no chunk data will be added to image frames.

[in] `unsigned int chunkCount`
Integer value specifying the amount of chunk data fields. If zero, no chunk data will be added to image frames.

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code instantiates a camera object and initiates the chunk mode with 2 data fields of the integer and float types:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();
m_pCamera->SetStreamChannelCount(1);
.......
int iSize=sizeof (int) + sizeof (float);
m_pCamera->SetChunkMode(iSize, 2);
```

Remarks

When the chunk streaming mode is enabled, each frame will contain one or several chunk data fields following the image data. This method informs *GigESim* of the size of the chunk data block that will be appended to image data in each frame. As a result, the payload size will change accordingly.

Note that you must explicitely call SetStreamChannelCount prior to calling **SetChunkMode**.

Note that the camera must be in the disconnected state in order for this method to work.

The *channelIndex* parameter should be used only if your virtual camera must support multiple

streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.52 SetCompressionQuality

Description

Sets the compression quality and bitrate for JPEG and H.264 compressed streams.

[C++]
```
int SetCompressionQuality(unsigned int quality, unsigned int bitrate=5000,
unsigned int channelIndex=0);
```

Parameters [C/C++]

[in] `unsigned int quality`
Relative value of the compression quality. The higher the quality is, the higher the streaming bandwidth will be. Can be in the following range:
JPEG compression: 0 - 100
H.264 compression 0 - 6

[in] `unsigned int bitRate`
The value of the bitrate in kbps. Applies only to H.264 compression.

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code generates sequential video frames with a random RGB pattern and sends them over the network as JPEG images with the compression quality factor of 75.

```
if (m_pCamera->connect(m_interfaceList.GetCurSel());
m_pCamera->SetImageCompression(COMPRESSION_JPEG);
m_pCamera->SetCompressionQuality(75);
m_exitThread = false;
m_thread = CreateThread(NULL, NULL,
(LPTHREAD_START_ROUTINE)videoGenThread, this, 0, NULL);

void CGigemuDlg::videoGenerator()
{
  int width;
  int height;
  char buffer[MAXWIDTH*MAXHEIGHT*3];
  char *ptr;
  int i=0;
  while (!m_exitThread)
  {
    m_pCamera->LockFormat();
    width=m_pCamera->GetWidth();
```

```
    height=m_pCamera->GetHeight();
    for (ptr=buffer; ptr < buffer+width*heigh; i++)
    {
     *ptr++=i; *ptr++=i/2; *ptr++=i%255;
    m_pCamera->SendImage(buffer);
    Sleep(20);
  }
}
```

Remarks

This method is used in conjunction with SetImageCompression.

The *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

If your computer has an Intel graphics card, GigESim will use hardware acceleration for H.264 compression. This will allow you to achieve a higher image resolution and/or frame rate as opposed to the software compression.

Note that a GigE Vision client must be able to decompress incoming image frames in real time in order to display and process compressed video generated by *GigESim*. We recommend using our ActiveGigE SDK which supports both JPEG and H.264 decompression on the fly.

3.3.53 SetDeviceInfo

Description

Sets the device information fields and assigns a name to the XML information file.

[C++]
```
int SetDeviceInfo(const char* manufacturer, const char* model, const char*
version, const char* info, const char* serial);
```

Parameters [C/C++]

[in] `const char* manufacturer`
String specifying the manufacturer of the virtual camera device.

[in] `const char* model`
String specifying the model of the virtual camera device.

[in] `const char* version`
String specifying the version of the virtual camera device.

[in] `const char* info`
String specifying the manufacturer's info of the virtual camera device.

[in] `const char* serial`
String specifying the serial number of the virtual camera device.

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Unsupported format

Example

This line of code sets the device information fields:

```
m_pCamera->SetDeviceInfo("gigecam","simulator", "v.2.5", "GigE Vision Camera Emulator",
"00000-01");
```

Remarks

Use this method to assign an identity of your virtual camera device.

SetDeviceInfo has a dual purpose. It fills out the device information registers used by external GigE Vision client applications to uniquely identify each GigE Vision device found on the network. When a client application lists all connected devices, it will typically use the vendor and model name in its user interface. **SetDeviceInfo** also assigns a unique name to the XML information file. The XML file is created in the *GigESim* memory and contains the information about all the features exposed by the

camera. In the example above the following name will be assigned to the XML file of the virtual camera device: "gigecam_simulator_v.1.0.xml"

When a client application connects to a GigE Vision device, it retrieves the XML file and stores it in its database. As a rule, when another connection to the same device is made, the client application will not repeat the retrieval process, but will use the copy of the XML file previously stored, unless the name of the XML file reported by the device has changed. If you implement custom features in your virtual camera application, it is very important that you call **SetDeviceInfo** with a different value of the *version* parameter each time you add a new feature or modify an existing one. If you fail to do it, the client application may continue using the stored version of the XML file which wouldn't contain an updated information about your revised feature set.

If you do not implement any custom features, the version control is not required. In this case you can use **SetDeviceInfo** to assign your own name to the virtual camera or may not call this method at all. By default the following fields are preset in *GigESim*:

Manufacturer	"A&B Software"
Model	"GigeSim"
Version	"2.5.0.0"
Info	"GigE Vision Camera Simulator"
XML file name	"A&B_GigeSim_2.5.0.0.xml"

Note that the camera must be in the disconnected state in order for this method to work.

3.3.54 SetEnumElement

Description

Creates and/or sets an element associated with an entry in the enumeration feature.

[C++]
```
int SetEnumElement(const char* feature, const char* enumName, const char*
element, const char* value, bool forceNew = false);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the enumerated feature. to assign the element to.

[in] `const char* enumName`
Name of the entry to assign the element to.

[in] `const char* element`
Name of the element.

[in] `const char* value`
String representing the value of the element.

[in] `bool* forceNew`
This parameter should be used only if the element with the same name already exists. If TRUE, the new element will be created regardless. If FALSE, the method will not create a new element, but will only modify the value of the existing element.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist

Example

The following fragment of code creates an enumerated feature with several entries and sets up associated elements:

```
m_pCamera->CreateFeature(FEATURE_TYPE_ENUMERATION, "LineSource",
"DigitalControl", REG_ACCESS_RW);
m_pCamera->SetFeatureDescription("LineSource", "Selects output
signal on digital line");
m_pCamera->AddEnumEntry("LineSource", "AcquisitionTrigger",0);
m_pCamera->AddEnumEntry("LineSource", "UserOutput",1);
m_pCamera->SetFeatureElement("LineSource", "Streamable", "Yes",
true);
m_pCamera->SetEnumElement("LineSource", "AcquisitionTrigger",
"Description", "Line source is acquisition trigger", true);
```

```
m_pCamera->SetEnumElement("LineSource", "UserOutput", "Description",
"Line source is user output", true);
m_pCamera->SetEnumElement("LineSource", "AcquisitionTrigger",
"pIsAvailable", "TriggerMode", true);
```

As a result, the following feature description will appear in the camera XML file:

```
<Emumeration Name="LineSource" NameSpace="Standard">
<Description>Selects output signal on digital line</Description>
<pStreamable>Yes</pStreamable>
 <EnumEntry Name="AcquisitionTrigger" NameSpace="Standard">
   <Description>Line source is acquisition trigger</Description>
   <pIsAvailable>TriggerMode</pIsAvailable>
   <Value>0</Value>
 </EnumEntry>
 <EnumEntry Name="UserOutput" NameSpace="Standard">
   <Description>Line source is user output</Description>
   <Value>1</Value>
 </EnumEntry>
<Value>regLineSource</Value>
</Enumeration>
```

Remarks

This method allows you to directly program an XML-file linked to the virtual camera object. An element is part of the XML-file encapsulated between a starting and ending tag. In the example above those elements are *Streamable*, *Description* and *pIsAvailable*.

Elements can contain one or several attributes. See CreateElementAttribute for more details.

The use of this method requires an advanced knowledge of the GenICam standard and XML syntax. For more information please refer to the description of the GenICam standard available at www.genicam.org.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.55 SetFeatureAccess

Description

Sets the access mode for the specified feature.

[C++]
```
int SetFeatureAccess(const char* feature, unsigned int access);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the description to.

[in] `unsigned int access`
Integer value specifying the access mode:
FEATURE_ACCESS_RO - feature will be accessible in the read-only mode.
FEATURE_ACCESS_WO - feature will be accessible in the write-only mode.
FEATURE_ACCESS_RW - feature will be accessible for reading and writing.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist

Example

This fragment of code instantiates a camera object, creates an integer feature and sets the read-only access mode:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_FLOAT, "Temperature", "Analog");
m_pCamera->SetFeatureAccess("Temperature", FEATURE_ACCESS_RO);
```

Remarks

Note that the camera must be in the disconnected state in order for this method to work.

3.3.56 SetFeatureDescription

Description

Assigns a description to the specified feature.

[C++]
```
int SetFeatureDescription(const char* feature, const char* description);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the description to.

[in] `const char* description`
Siring containing the description of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist

Example

This fragment of code instantiates a camera object, creates an integer feature and sets its description:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_INTEGER, "Contrast", "Analog",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureDescription("Offset", "Contrast of the image in percent");
```

Remarks

The descriptions of features is available to remote client applications as part of the GenICam protocol.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.57 SetFeatureElement

Description

Creates and/or sets an element associated with the feature.

[C++]
```
int SetFeatureElement(const char* feature, const char* element, const char*
value, bool forceNew = false);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the element to.

[in] `const char* element`
Name of the element.

[in] `const char* value`
String representing the value of the element.

[in] `bool* forceNew`
This parameter should be used only if the element with the same name already exists. If TRUE, the new element will be created regardless. If FALSE, the method will not create a new element, but will only modify the value of the existing element.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist

Example

The following fragment of code creates a floating point feature and sets up several associated elements:

```
m_pCamera->CreateFeature(FEATURE_TYPE_FLOAT, "ExposureTime",
"Analog", REG_ACCESS_ALL);
m_pCamera->SetFeatureRange("ExposureTime", 10., 100000000.);
m_pCamera->SetFeatureDescription("ExposureTime", "Exposure duration,
in microseconds");
m_pCamera->SetFeatureElement("ExposureTime", "ToolTip","Exposure
time");
m_pCamera->SetFeatureElement("ExposureTime", "Representation",
"Logarithmic");
m_pCamera->SetFeatureElement("regExposureTime", "Cachable",
"NoCache");
```

As a result, the following feature description will appear in the camera XML file:

```
 <Float Name="ExposureTime" NameSpace="Standard">
<ToolTip>Exposure time.</ToolTip>
<Description>Exposure duration, in microseconds.</Description>
<pValue>RegExposureTime</pValue>
<Min>10.</Min>
<Max>100000000.</Max>
<Representation>Logarithmic</Representation>
</Float>
<FloatReg Name="regExposureTime" NameSpace="Standard">
<Address>0x10044</Address>
<Length>4</Length>
<AccessMode>RW</AccessMode>
<pPort>Device</pPort>
<Cachable>NoCache</Cachable>
<Endianess>BigEndian</Endianess>
<FloatReg>
```

Remarks

This method allows you to directly program an XML-file linked to the virtual camera object. An element is part of the XML-file encapsulated between a starting and ending tag. In the example above those elements are: *ToolTip, Description, pValue, Min, Max, Representation*. The *Min* and *Max* elements are assigned by using the SetFeatureRange method, while **SetFeatureElement** is used to assign the remaining elements. The *Cachable* element is assigned not to the feature itself, but to its register ("regExposureTime") as required by the GenICam standard.

Elements can contain one or several attributes. See CreateElementAttribute for more details.

The use of this method requires an advanced knowledge of the GenICam standard and XML syntax. For more information please refer to the description of the GenICam standard available at www.genicam.org.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.58 SetFeatureFloatValue

Description

Sets the value of the specified floating point feature.

[C++]
```
int SetFeatureFloatValue(const char* feature, double value);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the value to. Must be an existing feature of the floating point type.

[in] `double value`
New value of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type

Example

This fragment of code instantiates a camera object, creates a feature of the floating point type and assigns a default value to it:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_FLOAT, "ExposureTime", "Analog",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureRange("ExposureTime", 0.f, 1000000.f);
m_pCamera->SetFeatureFloatValue("ExposureTime", 50000.f);
```

Remarks

This method is typically used to assign a default value to a feature. It can also be used as part of a feature-read callback in order to modify the value of the feature before it gets transmitted to a client application. See SetReadCallback for more details.

3.3.59 SetFeatureIntValue

Description

Sets the integer value of the specified feature.

[C++]
```
int SetFeatureIntValue(const char* feature, __int64 value);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the value to. Must be an existing feature of the integer, enumerated, boolean or command type.

[in] `__int64 value`
New value of the feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type or value out of range

Example

This fragment of code instantiates a camera object, creates a feature of the integer type and assigns a default value to it:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_INTEGER, "Offset", "Root",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureRange("Offset", 0, 1000, 1);
m_pCamera->SetFeatureIntValue("Offset", 512);
```

Remarks

Depending on the type of the feature the *Value* argument has the following meaning:

Feature Type	Value
Integer	Integer value to be set
Boolean	0 if False, 1 if True
Enumerated	Numerical value of the item to select
Command	Use non-zero value to execute

This method is typically used to assign a default value to a feature. It can also be used as part of a feature-read callback in order to modify the value of the feature before it gets transmitted to a client application. See SetReadCallback for more details.

3.3.60 SetFeatureIntRange

Description

Assigns the minimum, maximum and increment attributes to the specified feature.

[C++]
```
int SetFeatureRange(const char* feature, __int64 min, __int64 max, int
inc=1);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the attributes to. Must be an existing feature of the integer or floating point type.

[in] `__int64 min`
Value of the feature's minimum.

[in] `__int64 max`
Value of the feature's maximum.

[in] `int inc`
Value of the feature's incremental step.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type or value out of range

Example

This fragment of code instantiates a camera object, creates an integer feature and sets up its range :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_INTEGER, "TestFeatureInt", "Root",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureRange("TestFeatureInt", 0, 1000, 5);
m_pCamera->SetFeatureIntValue("TestFeatureInt", 500);
```

Remarks

This method should be used after calling CreateFeature for integer and enumerated features in order to finalize their instantiation.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.61 SetFeatureRange

Description

Assigns the minimum and maximum attributes to the specified feature.

[C++]
```
int SetFeatureRange(const char* feature, float min, float max);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the attributes to. Must be an existing feature of the integer or floating point type.

[in] `float min`
Value of the feature's minimum.

[in] `float max`
Value of the feature's maximum.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type or value out of range

Example

This fragment of code instantiates a camera object, creates a floating point feature and sets up its range:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_FLOAT, "TestFeatureFloat", "Root",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureRange("TestFeatureFloat", 0, 1000);
m_pCamera->SetFeatureFloatValue("TestFeatureFloat", 3.1415926);
```

Remarks

This method should be used after calling CreateFeature for floating point, integer and enumerated features in order to finalize their instantiation.

When **SetFeatureRange** is used for an integer feature, the parameters will be rounded to the nearest integer values.

Note that the camera must be in the disconnected state in order for this method to work.

3.3.62 SetFeatureStingValue

Description

Sets the string value of the specified feature.

[C++]
```
int SetFeatureStringValue(const char* feature, const char* value);
```

Parameters [C/C++]

[in] `const char* feature`
Name of the feature to assign the value to. Must be an existing feature of the string or enumerated type.

[in] `const char* value`
New value of the string feature or selected string value of the enumerated feature.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Feature does not exist
E_INVALIDARG
Wrong feature type

Example

This fragment of code instantiates a camera object, creates a feature of the string type and assigns a default value to it:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->CreateFeature(FEATURE_TYPE_STRING, "CameraModel", "DeviceInformation",
FEATURE_ACCESS_RW);
m_pCamera->SetFeatureStringValue("CameraMode", "Simulator 1.2");
```

Remarks

Depending on the type of the feature the *Value* argument has the following meaning:

Feature Type	Value
String	String value to be set
Enumerated	String value of an item to be selected in the enumerated list

This method is typically used to assign a default value to a feature. It can also be used as part of a

feature-read callback in order to modify the value of the feature before it gets transmitted to a client application. See SetReadCallback for more details.

3.3.63 SetGevMode

Description

Sets the version of the GigE Vision standard under which the virtual camera will operate.

[C++]
```
int SetGevMode(unsigned short gevMode);
```

Parameters [C/C++]

[in] `unsigned short gevMode`
Type of the timer. Can be one of the following values:
GEV_1 - virtual camera will comply with version 1.2 of the GigE Vision specifications.
GEV_2 - virtual camera will comply with version 2.0 of the GigE Vision specifications.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code instantiates a camera object and sets the GEV mode:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetGevMode(GEV_2);
```

Remarks

The main difference between GEV 2.x and GEV 1.x modes is the use of 64-bit block IDs and 32-bit packet IDs as opposed to 32-bit block IDs and 16-bit packet IDs. In addition, certain functionality such as JPEG compression and IEEE 1588 Precision Time Protocol are available only in GEV 2.x mode. For more information refer to *GigE Vision Video Streaming and Device Control Over Ethernet Standard* available from Advanced Imaging Association.

If this function is not called, *GigESim* will be using the GEV 1.2 mode.

Note that the camera must be in the disconnected state in order for **SetGevMode** to work.

3.3.64 SetImageCompression

Description

Sets the image compression mode.

[C++]
```
int SetImageCompression(unsigned short compression,  unsigned int
channelIndex=0);
```

Parameters [C/C++]

[in] `unsigned short compression`
Image compression mode. Can be one of the following values:
COMPRESSION_NONE - virtual camera will stream uncompressed images
COMPRESSION_JPEG - virtual camera will stream JPEG compressed images
COMPRESSION_H264 - virtual camera will stream H.264 compressed video

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code generates sequential video frames with a random RGB pattern and sends them over the network as JPEG images.

```
if (m_pCamera->connect(m_interfaceList.GetCurSel());
m_pCamera->SetImageCompression(COMPRESSION_JPEG);
m_exitThread = false;
m_thread = CreateThread(NULL, NULL,
(LPTHREAD_START_ROUTINE)videoGenThread, this, 0, NULL);

void CGigemuDlg::videoGenerator()
{
  int width;
  int height;
  char buffer[MAXWIDTH*MAXHEIGHT*3];
  char *ptr;
  int i=0;
  while (!m_exitThread)
  {
    m_pCamera->LockFormat();
    width=m_pCamera->GetWidth();
    height=m_pCamera->GetHeight();
    for (ptr=buffer; ptr < buffer+width*heigh; i++)
    {
     *ptr++=i; *ptr++=i/2; *ptr++=i%255;
```

```
      m_pCamera->SendImage(buffer);
      Sleep(20);
    }
  }
```

Remarks

This method is used to switch the virtual camera to a compressed streaming mode per GigE Vision 2.0 specifications. When JPEG or H.264 payload type is selected, the SendImage function will apply an internal compression to image frames before streaming them to the network. To adjust compression settings, use SetCompressionQuality.

The *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

Note that only 8-bit monochrome and RGB images can be compressed. You need to make sure that image buffer submitted to SendImage have data in a correct format.

If your computer has an Intel graphics card, GigESim will use hardware acceleration for H.264 compression. This will allow you to achieve a higher image resolution and/or frame rate as opposed to the software compression.

Note that a GigE Vision client must be able to decompress incoming image frames in real time in order to display and process compressed video generated by *GigESim*. We recommend using our ActiveGigE SDK which supports both JPEG and H.264 decompression on the fly.

3.3.65 SetImageSize

Description

Sets the horizontal and vertical size of outgoing images.

[C++]
```
int SetImageSize(int width, int height,  unsigned int channelIndex=0);
```

Parameters [C/C++]

[in] `int width`
Horizontal size of the image in pixels.

[in] `int height`
Vertical size of the image in pixels.

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code instantiates a camera object and sets up initial values for the image transfer :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetImageSize (1024, 768);
m_pCamera->AddPixelFormat("Mono16");
```

Remarks

This method modifies the values of the *Width* and *Height* features of the virtual camera and recalculates the value of the *PayloadSize* feature based on the current *PixelFormat*. It does not do any manipulation with an actual image buffer. You have to make sure that the image data in the buffer are compliant with the specified horizontal and vertical size of the image.

Width, Height, PixelFormat and *PayloadSize* are mandatory GigE Vision features which are automatically created when a virtual camera object is instantiated.

The actual size of each transmitted frame can be variable and smaller than the current image size. See SendImage for more details.

The *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default

zero value.

3.3.66 SetMaxImageSize

Description

Sets the maximum horizontal and vertical size of outgoing images.

[C++]
```
int SetMaxImageSize(int maxWidth, int maxHeight);
```

Parameters [C/C++]

[in] `int maxWidth`
Maximum horizontal size of the image in pixels.

[in] `int maxHeight`
Maximum vertical size of the image in pixels.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code instantiates a camera object and sets up the maximum image size :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetImageSize (4000, 3000);
```

Remarks

This method defines the maximum values of the *Width* and *Height* features of the virtual camera and uses them to reserve internal image buffers. In the real world this corresponds to the maximum size of the camera sensor.

Width and *Height* are mandatory GigE Vision features which are automatically created when a virtual camera object is instantiated. They define the current image size which represents a region of interest and be smaller than the maximum image size. See SendImage for more details.

If this method is not called, *GigESim* will be using the default maximum image size of 4096x4096.

Note that if the maximum image size is too large, the 32-bit version of *GigESim* may not be able to reserve the memory needed for its operation. If this happens, use the 64-bit version.

The camera must be in the disconnected state in order for **SetMaxImageSize** to work.

3.3.67 SetIpAddress

Description

Sets the IP configuration of the virtual camera device.

[C++]
```
int SetIpAddress(const char* ip, const char* mask=NULL, const char*
gateway=NULL);
```

Parameters [C/C++]

[in] `const char* ip`
String specifying the ip address of the camera in the IPv4 format (four decimals in the range of 0-255 separated by dots).

[in] `const char* mask`
String specifying the subnet mask in the IPv4 format. If this parameter is NULL or omitted, the mask of a connected interface will be used.

[in] `const char* gateway`
String specifying the gateway in the IPv4 format. If this parameter is NULL or omitted, the gateway of a connected interface will be used.

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Wrong IPv4 format

Example

This fragment of code sets the IP address, MAC address and serial number of the virtual camera and connects it to a matching network interface:

```
m_pCamera->SetIpAddress ("169.254.101.102", "255.255.0.0");
m_pCamera->SetMacAddress("AA-11-12-13-14-15");
m_pCamera->SetSerialNumber("10001");
m_pCamera->Connect("169.254.1.100");
```

Remarks

When a virtual camera device is connected to a network interface, it is automatically assigned the IP address and subnet mask of the interface. **SetIpAddress** allows you to modify the default IP configuration of the virtual camera. This is especially useful when you need to run several virtual camera devices on the same network interface. Assigning each virtual camera a unique IP address (as well as MAC address and serial number) will effectively simulate multiple GigE Vision cameras connected to one network card through a Gigabit switch.

Note that the IP configuration assigned to the virtual camera should match the IP address and subnet mask of the network adapter to which the camera will be connected to. If the camera is assigned a non-matching IP address, an external client software would have to use the Force IP protocol to correct the camera's IP configuration before it could connect to the camera.

The camera must be in the disconnected state in order for **SetIpAddress** to work.

If you have only one virtual camera connected to each network adapter, using this method is not recommended other than for testing purposes.

3.3.68 SetMacAddress

Description

Assigns the specified MAC address to the virtual camera.

[C++]
```
int SetMacAddress(const char* mac);
```

Parameters [C/C++]

[in] `const char* mac`
String specifying the MAC address of the virtual camera in the IEEE-802 format (six groups of two hexadecimal digits separated by hyphens)

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Wrong MAC format

Example

This command sets the MAC address of the camera to a specified value:

```
m_pCamera->SetMacAddress("AB-AC-AD-AE-AF-00");
```

Remarks

If you are connecting several virtual camera devices to the same network interface (see SetIpAddress for more details), you have to make sure that each device is assigned a unique MAC address. One way to do it is by implementing a counter of instances of your application in the system and setting a MAC address of an instance based on its ordinal number.

Note that the camera must be in the disconnected state in order for this method to work.

By default the MAC address of a virtual camera object is preset to AA-00-00-00-00-00.

3.3.69 SetPixelFormat

Description

Sets the specified pixel format for outgoing images.

[C++]
```
int SetPixelFormat(const char* format, unsigned int channelIndex=0);
int SetPixelFormat(unsigned int value, unsigned int channelIndex=0);
```

Parameters [C/C++]

[in] `const char* format`
String specifying the pixel format of the outgoing images. See **Remarks** for the list of possible values.

[in] `unsigned int value`
Value specifying the GigE Vision numerical identificator of the pixel format. See **Remarks** for the list of possible values.

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Unsupported format

Example

This fragment of code instantiates a camera object and sets up initial values for the image transfer :

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetImageSize (1024, 768);
m_pCamera->AddPixelFormat("Mono16");
m_pCamera->AddPixelFormat("RGB8");
m_pCamera->SetPixelFormat("Mono16");
```

Remarks

This method modifies the value of the *PixelFormat* feature of the virtual camera and recalculates the value *PayloadSize* feature based on the current *Width* and *Height*. It does not do any manipulation with an actual image buffer. You have to make sure that the image data in the image buffer are compliant with the specified pixel format.

Width, Height, PixelFormat and *PayloadSize* are mandatory GigE Vision features which are

automatically created when a virtual camera object is instantiated.

The following strings and values can be used as parameters for **SetPixelFormat**:

Format	Value	Description	Bits per pixel
"Mono8"	0x01080001	8-bit monochrome unsigned	8
"Mono8s"	0x01080002	8-bit monochrome signed	8
"Mono10"	0x01100003	10-bit monochrome unpacked	16
"Mono10Packed"	0x010C0004	10-bit monochrome packed	12
"Mono12"	0x01100005	12-bit monochrome unpacked	16
"Mono12Packed"	0x010C0006	12-bit monochrome packed	12
"Mono14"	0x01100025	14-bit monochrome pixel	16
"Mono16"	0x01100007	16-bit monochrome	16
"BayerGR8"	0x01080008	8-bit raw Bayer, GRBG layout	8
"BayerRG8"	0x01080009	8-bit raw Bayer, RGGB layout	8
"BayerGB8"	0x0108000A	8-bit raw Bayer, GBRG layout	8
"BayerBG8"	0x0108000B	8-bit raw Bayer, BGGR layout	8
"BayerGR10"	0x0110000C	10-bit raw Bayer unpacked, GRBG layout	16
"BayerRG10"	0x0110000D	10-bit raw Bayer unpacked, RGGB layout	16
"BayerGB10"	0x0110000E	10-bit raw Bayer unpacked, GBRG layout	16
"BayerBG10"	0x0110000F	10-bit raw Bayer unpacked, BGGR layout	16
"BayerGR12"	0x01100010	12-bit raw Bayer unpacked, GRBG layout	16
"BayerRG12"	0x01100011	12-bit raw Bayer unpacked, RGGB layout	16
"BayerGB12"	0x01100012	12-bit raw Bayer unpacked, GBRG layout	16
"BayerBG12"	0x01100013	12-bit raw Bayer unpacked, BGGR layout	16
"BayerGR10Packed"	0x010C0026	10-bit raw Bayer packed, GRBG layout	12
"BayerRG10Packed"	0x010C0027	10-bit raw Bayer packed, RGGB layout	12
"BayerGB10Packed"	0x010C0028	10-bit raw Bayer packed, GBGR layout	12
"BayerBG10Packed"	0x010C0029	10-bit raw Bayer packed, BGGR layout	12
"BayerGR12Packed"	0x0000002A	12-bit raw Bayer packed, GRBG layout	12
"BayerRG12Packed"	0x0000002B	12-bit raw Bayer packed, RGGB layout	12
"BayerGB12Packed"	0x0000002C	12-bit raw Bayer packed, GBRG layout	12
"BayerBG12Packed"	0x0000002D	12-bit raw Bayer packed, BGGR layout	12
"BayerGR16"	0x0110002E	16-bit raw Bayer unpacked, GRBG layout	16
"BayerRG16"	0x0110002F	16-bit raw Bayer unpacked, RGGB layout	16
"BayerGB16"	0x01100030	16-bit raw Bayer unpacked, GBRG layout	16
"BayerBG16"	0x01100031	16-bit raw Bayer unpacked, BGGR layout	16
"RGB8"	0x02180014	24-bit RGB color	24
"BGR8"	0x02180015	24-bit BGR color	24
"RGBa8"	0x02200016	24-bit RGB color with alpha channel	32
"BGRa8"	0x02200017	24-bit BGR color with alpha channel	32
"RGB10"	0x02300018	30-bit RGB color	48
"BGR10"	0x02300019	30-bit BGR color	48

"RGB12"	0x0230001A	36-bit RGB color	48
"BGR12"	0x0230001B	36-bit BGR color	48
"RGB16"	0x02300033	48-bit RGB color	48
"RGB10V1Packed"	0x0220001C	30-bit BGR color packed	32
"RGB12V1Packed"	0x02240034	36-bit BGR color packed	36
"YUV411_8_UYYVYY"	0x020C001E	12-bit YUV color (YUV 4:1:1)	12
"YUV422_8_UYVY"	0x0210001F	16-bit YUV color (YUV 4:2:2)	16
"YUV8_UYV"	0x02180020	24-bit YUV color (YUV 4:4:4)	24
"RGB8Planar"	0x02180021	24-bit RGB in form of three 8-bit planes	24
"RGB10Planar"	0x02300022	30-bit BGR in form of three 16-bit planes	48
"RGB12Planar"	0x02300023	36-bit BGR in form of three 16-bit planes	48
"RGB16Planar"	0x02300024	48-bit BGR in form of three 16-bit planes	48
"Coord3D_ABC16"	0x023000B9	48-bit 3D-cloud in form of 3x16-bit XYZ coordinates	48
"Coord3D_ABC32f"	0x026000C0	96-bit 3D-cloud in form of 3x32-bit float XYZ coordinates	96

The camera must be in the disconnected state in order for this method to work.

"Coord3D" formats are available only when GevMode is set to GEV 2.0.

Note the *channelIndex* parameter should be used only if your virtual camera must support multiple streaming channels. For a regular GigE Vision transfer this parameter should remain in its default zero value.

3.3.70 SetStreamChannelCount

Description

Sets the amount of stream channels in the virtual camera.

[C++]
```
int SetStreamChannelCount(unsigned int count);
```

Parameters [C/C++]

[in] `unsigned int count`
Integer value specifying the amount of stream channels in the virtual camera. Possible values are 1 or 2.

Return Values

S_OK
Success
E_FAIL
Failure
E_INVALIDARG
Value out of range

Example

This fragment of code instantiates a camera object and sets the amount of stream channels to 2:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetStreamChannelCount(2);
```

Remarks

This method allows you to configure the virtual camera for multi-channel streaming. If this method is not used, only one stream channel will be supported.

Note that this method must be called in order for the Chunk mode to work.

The camera must be in the disconnected state in order for this method to work.

3.3.71 SetTimerMode

Description

Sets the operational mode of the virtual camera timer.

[C++]
```
int SetTimerMode(unsigned short timerMode);
```

Parameters [C/C++]

[in] `unsigned short timerMode`
Type of the timer. Can be one of the following values:
TIMER_UNIX - the internal camera timer counts the time elapsed since 00:00:00 January 1, 1970.
TIMER_UDP - the internal camera timer counts the time elapsed since the current *GigESim*-based application has started.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code instantiates a camera object and sets the UNIX timer mode:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetTimerMode(TIMER_UNIX);
```

Remarks

The internal *GigESim* timer is used to integrate timestamps into each frame streamed to the network by the SendImage function. It is also used to handle schedule action commands.

If this function is not called, *GigESim* will be using the UNIX timer mode.

Note that the camera must be in the disconnected state in order for **SetTimerMode** to work.

3.3.72 SetTransferMode

Description

Selects the packet transfer optimization mode.

[C++]
```
int SetTransferMode(unsigned short transferMode);
```

Parameters [C/C++]

[in] `unsigned short gevMode`
Transfer mode to be set. Can be one of the following values:
TRANSFER_SINGLE - transfer of network packets is optimized for streaming via a single network interface
TRANSFER_MULTI - transfer of network packets is optimized for streaming via multiple network interfaces

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code instantiates a camera object and sets the packet transfer mode:

```
static CGevCamera* m_pCamera;
m_pCamera = createCamera();

m_pCamera->SetTransferMode(TRANSFER_MULTI);
```

Remarks

GigESim utilizes two different methods for transferring images to the network. The single transfer mode results in efficient transmission when a single network interface is used to stream images to a remote node, but it can cause a reduction of the link speed on Windows XP and Windows 7 when several virtual cameras are streaming simultaneously via multiple network interfaces. In the latter case the multi-transfer mode must be selected for the optimal performance.

If this function is not called, *GigESim* will be using the single transfer mode.

This function is not used in the Linux version of *GigESim*.

Note that the camera must be in the disconnected state in order for **SetTransferMode** to work.

3.3.73 SetUserDefinedName

Description

Sets the user-defined name for the virtual camera.

[C++]
```
int SetUserDefinedName(const char* userDefined);
```

Parameters [C/C++]

[in] `const char* userDefined`
String specifying the user-defined name of the virtual camera device.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of code sets the device information fields and assigns a user-defined name:

```
    m_pCamera->SetDeviceInfo("gigecam","simulator", "v.2.5", "GigE Vision Camera Emulator",
"00000-01");
    m_pCamera->SetUserDefinedName("Cam001");
```

Remarks

Per GigE Vision specifications, the user-defined name is broadcast as part of the Discovery acknowledgement message along with the model name and other device information fields.

This method can be used in both connected and disconnected states of the camera.

For GenICam SFNC compliance it is recommended to have user defined name linked to the value of the DeviceUserID feature.

3.3.74 WriteMemory

Description

Writes the block of data to the internal camera memory starting from the specified bootstrap address.

[C++]
```
int WriteMemory(unsigned int addr, void* pAddr, unsigned int size);
```

Parameters [C/C++]

[in] `unsigned int* addr`
Starting address in the camera bootstrap address space.

[in] `unsigned int* pAddr`
Pointer to a buffer that contains the data to be written.

[in] `unsigned int size`
Size of memory to write in bytes.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Bootstrap address does not exist

Example

This fragment of an MFC code writes a block of data to the camera's bootstrap memory:

```
CString str;
unsigned char LUT[256];
for (int i=0; i<256; i++)
 LUT[i]=i;
m_pCamera->WriteMemory(0xF100,LUT,256)
```

3.3.75 WriteRegister

Description

Sets the value of the specified virtual camera register.

[C++]
```
int WriteRegister(unsigned int addr, unsigned int Value);
```

Parameters [C/C++]

[in] `unsigned int* addr`
Address of the register to read.

[in] `unsigned int* pAddr`
Value to be written to the register.

Return Values

S_OK
Success
E_FAIL
Failure
E_NOINTERFACE
Register does not exist

Example

This fragment of an MFC code modifies the value of the register associated with the Gain feature:

```
CString str;
unsigned int addr; int iVal
m_pCamera->GetFeatureRegister("Gain", &addr);
m_pCamera->WriteRegister(addr,8)
```

Remarks

This method allows you to read the value of the virtual camera register. To obtain the address of the register associated with a specific feature, use GetFeatureRegister.

3.4 Events

The **CGevCamera** class provides the callback functionality that can be used for intercepting remote client requests associated with reading or writing the values of the features of a virtual camera object. The following callback functions are available:

SetReadCallback	Sets up a callback function associated with a feature read request from a client
SetWriteCallback	Sets up a callback function associated with a feature write request from a client
SetFormatChangedCallback	Sets up a callback function to be called when the image format is changed by a client
SetActionCallback	Sets up a callback function to be called upon receiving an action command from a client
SetScheduledActionCallback	Sets up a callback function to be called upon receiving a scheduled action command

3.4.1 SetActionCallback

Description

Sets up a callback function which will be called when a client application issues an action command.

[C++]
```
int SetActionCallback(void* context, ActionCallback callback);
```

where ActionCallback type is defined as

```
 typedef int (*ActionCallback)(void* context, const ActionParameters*
params);
```

Parameters [C/C++]

[in] `void* context`
Address of the context in which the callback function will be called.

[in] `ActionCallback callback`
Address of the callback function

[in] `const ActionParameters* params`
Pointer to an ActionParameters structure containing the following fields:

`unsigned int` *device_key*
The 32-bit value of the device key in the received action command.

`unsigned int` *group_key*
The 32-bit value of the group key in the received action command.

`unsigned int` *group_mask*
The 32-bit value of the group mask in the received action command.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of an MFC code uses an action callback to validate the action command, determine what type of action is requested and implement it.

```
bool onActionCommand(void* context, const ActionParameters* params)
{
  CGigemuDlg* dlg = (CGigemuDlg*) context;
  return dlg->onActionCommand(params);
}

 bool CGigemuDlg::onActionCommand(const ActionParameters* params)
```

```
{
  if((params.device_key==m_DeviceKey) && (params.group_key==m_GroupKey))
  {
    //action command is intended for our virtual camera, implement
corresponding action and return true
    if(params.group_mask & 1))
    {
      ImplementAction1();
      return true;
    }
    else if(params.group_mask & 2))
    {
      ImplementAction2();
      return true;
    }
  }
  //action parameters do not match action key settings of the virtual
camera, ignore it
  return false;
}

bool CGigemuDlg::OnInitDialog()
{
    ....
    static CGevCamera* m_pCamera;
    m_pCamera = createCamera();
    m_pCamera->SetActionCount(1);
    m_pCamera->SetActionCallback(this, &::onActionCommand);
     ....
}
```

Remarks

Action commands are used by a client application to trigger a simultaneous action on multiple devices at roughly the same time. Each action command contains information for the device to validate the requested operation: device key to authorize the action on this device, group key to define a group of devices on which actions have to be executed, and group mask to define different types of action. You should check these values from an incoming action command in the body of the ActionCommand callback and react to the action command accordingly. If the action command is ignored, your callback function must return False, otherwise it must return True. In the latter case GigESim will return an action acknowledge packet to the client application.

To receive schedule action commands, use SetScheduledActionCallback.

If you are using a callback function in the main thread of the simulation application, make sure that no lengthy processing is done in the body of the function as otherwise the connection to the remote client may become broken.

3.4.2 SetHeartbeatTimeoutCallback

Description

Sets up a callback function which will be called when a client application connects to the virtual camera or disconnects from it.

[C++]
```
int SetHeartbeatTimeoutCallback(void* context, HeartbeatTimeoutCallback
callback);
```

where HeartbeatTimoutCallback type is defined as

```
 typedef void (*HeartbeatTimeoutCallback)(void* context, bool
connected);
```

Parameters [C/C++]

[in] `void* context`
Address of the context in which the callback function will be called.

[in] `HeartbeatTimeoutCallback callback`
Address of the callback function

[in] `bool connected`
TRUE when the client application connects to the virtual camera, FALSE when disconnects.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of an MFC code uses the HeartbeatTimeout callback to inform the virtual camera application about the connection/disconnection status of a remote client application.

```
void heartbeatTimeoutCallback(void* context, bool connected)
{
   return dlg->onHeartbeatTimeout(connected);
}

bool CGigemuDlg::onHeartbeatTimeout(bool connected)
{
  if(connected)
      printf("External client connected to virtual camera\n");
  else
      printf("External client disconnected from virtual camera\n");

  return true;
}
```

```
bool CGigemuDlg::OnInitDialog()
{
      ....
      static CGevCamera* m_pCamera;
      m_pCamera = createCamera();
      m_pCamera->SetHeartbeatTimeoutCallback(this, &::heartbeatTimeoutCallback);
       ....
}
```

Remarks

The write callback function is called immediately after an external client applicatoin connects to the camera and starts sending the heartbeat to it. It is also called when *GigESim* detects the absense of the heartbeat which indicates that the client application is no longer connected to the virtual camera.

If you are using a callback function in the main thread of the simulation application, make sure that no lengthy processing is done in the body of the function as otherwise the connection to the remote client may become broken.

3.4.3 SetScheduledActionCallback

Description

Sets up a callback function which will be called when a client application issues a scheduled action command.

[C++]
```
int SetActionCallback(void* context, ScheduledActionCallback callback);
```

where ScheduledActionCallback type is defined as

```
 typedef int (*ScheduledActionCallback)(void* context, const
ActionParameters* params);
```

Parameters [C/C++]

[in] `void* context`
Address of the context in which the callback function will be called.

[in] `ScheduledActionCallback callback`
Address of the callback function

[in] `const ActionParameters* params`
Pointer to an ActionParameters structure containing the following fields:

`unsigned int` *`device_key`*
The 32-bit value of the device key in the received action command.

`unsigned int` *`group_key`*
The 32-bit value of the group key in the received action command.

`unsigned int` *`group_mask`*
The 32-bit value of the group mask in the received action command.

`unsigned __int64` *`action_time`*
The 64-bit value of the scheduled action time in nanoseconds.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of an MFC code uses a scheduled action callback to set up an exact time for a frame trigger:

```
   bool onScheduledActionCommand(void* context, const ActionParameters*
params)
   {
```

```
  CGigemuDlg* dlg = (CGigemuDlg*) context;
  return dlg->onActionCommand(params);
}

bool CGigemuDlg::onScheduledActionCommand(const ActionParameters*
params)
{
   if((params.device_key==m_DeviceKey) && (params.group_key==m_GroupKey))
   {
    //action command is intended for our virtual camera, convert action
time into windows file time
    #define SEC_TO_UNIX_EPOCH 11644473600LL
    #define WINDOWS_TICK 10000000
    unsigned __int64 ll;
    LPFILETIME pft;
    ll = (params->action_time/1000000000 + SEC_TO_UNIX_EPOCH) *
WINDOWS_TICK;
    pft->dwLowDateTime = (DWORD)ll;
    pft->dwHighDateTime = (DWORD)(ll >> 32);
    ScheduleTriggerTimer(pft);  //User-defined function for scheduling a
frame transfer
    return true;
   }
   //action parameters do not match action key settings of the virtual
camera, ignore it
   return false;
}

bool CGigemuDlg::OnInitDialog()
{
    ....
    static CGevCamera* m_pCamera;
    m_pCamera = createCamera();
    m_pCamera->SetActionCount(1);
    m_pCamera->SetScheduledActionCallback(this, &::onScheduledActionCommand);
     ....
}
```

Remarks

A scheduled action is used by a client application to trigger an action on the virtual camera at the specified time. Each scheduled action command contains a precise time for an action as well as information for the device to validate the requested operation: device key to authorize the action on this device, group key to define a group of devices on which actions have to be executed, and group mask to define different types of action. You should check these values from an incoming action command in the body of the ActionCommand callback and react to the action command accordingly. If the action command is ignored, your callback function must return False, otherwise it must return True. In the latter case GigESim will return an action acknowledge packet to the client application.

The *scheduled_time* is specified in the same time domain in which the internal GigESim timer operates. If the timer mode is set to UNIX time, *scheduled_time* should be interpreted as the amount of nanoseconds elapsed since 00:00:00 January 1, 1970. Otherwise it should be interpreted as the amount of nanoseconds elapsed since the execution of the current *GigESim*-based application or the last call to ResetTimer.

If you are using a callback function in the main thread of the simulation application, make sure that no

lengthy processing is done in the body of the function as otherwise the connection to the remote client may become broken.

3.4.4 SetFormatChangedCallback

Description

Sets up a callback function which will be called when the image format is changed by a client application.

[C++]
```
int SetFormatChangedCallback(void* context, unsigned int channelIndex,
FormatChangedCallback callback);
```

where FormatChangedCallback type is defined as

```
 bool (*FormatChangedCallback)(void* context);
```

Parameters [C/C++]

[in] `void* context`
Address of the context in which the callback function will operate

[in] `unsigned int channelIndex`
Index of the associated stream channel. Possible values are 0 or 1.

[in] `FormatChangedCallback callback`
Address of the callback function

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of an MFC code uses a format changed callback to intercept requests for the change of the image size and modifies the image buffer and acquisition parameters accordingly:

```
int onFormatChangedCallback(void* context, const char* feature)
{
   CGigemuDlg* dlg = (CGigemuDlg*) context;
   return dlg->onFormatChanged();
}

int CGigemuDlg::onFormatChanged()
{
   delete imgBuf;
   imgBuf=new (m_pCamera->GetPayloadSize())
   return 0;
  }
}

BOOL CGigemuDlg::OnInitDialog()
{
```

```
  ....
  m_pCamera->SetFormatChangedCallback(this, 0,
&::onFormatChangedCallback);
  ....
}
```

Remarks

The write callback function is called after one or more features that affect the image format have been changed by a client application. Typically these features are SizeX, SizeY and PixelFormat. This callback can be used to reallocate the memory required by image buffers.

If you are using the callback function in the main thread of the simulation application, make sure that no lengthy processing is done in the body of the function as otherwise the connection to the remote client may become broken.

3.4.5 SetReadCallback

Description

Sets up a callback function which will be called when a client application issues a feature read request.

[C++]
```
int SetReadCallback(void* context, FeatureCallback callback);
```

where FeatureCallback type is defined as

```
typedef bool (*FeatureCallback)(void* context, const char* feature);
```

Parameters [C/C++]

[in] `void* context`
Address of the context in which the callback function will be called.

[in] `FeatureCallback callback`
Address of the callback function

[in] `const char* feature`
String containing the name of the feature whose value is about to be read by a client application.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of an MFC code uses a feature-read callback to obtain the current temperature value from a thermal sensor and modify the value of the Temperature feature before it gets transmitted to the client application.

```
int onFeatureRead(void* context, const char* feature)
{
 CGigemuDlg* dlg = (CGigemuDlg*) context;
 return dlg->onFeatureRead(feature);
}

int CGigemuDlg::onFeatureRead(const char* feature)
{
  if(CString(feature)=="Temperature")
  {
    float temp;
    GetTemperatureFromSensor(&temp);
   m_pCamera->SetFeatureFloatValue(feature, temp);
  }
  return 0;
}
```

```
BOOL CGigemuDlg::OnInitDialog()
{
  ....
  m_pCamera->SetReadCallback(this, &::onFeatureRead);
  ....
}
```

Remarks

The read callback function is called before the value of the feature is transmitted to the client application. Therefore, the callback can be used to update the value of the feature at the time of the read request. This is especially useful when the feature represents a real-world measurement such as the temperature, as shown in the example above.

If you are using a callback function in the main thread of the simulation application, make sure that no lengthy processing is done in the body of the function as otherwise the connection to the remote client may become broken.

3.4.6 SetWriteCallback

Description

Sets up a callback function which will be called when a client application modifies a feature value.

[C++]
```
int SetWriteCallback(void* context, FeatureCallback callback);
```

where FeatureCallback type is defined as

```
bool (*FeatureCallback)(void* context, const char* feature);
```

Parameters [C/C++]

[in] `void* context`
Address of the context in which the callback function will operate.

[in] `FeatureCallback callback`
Address of the callback function

[in] `const char* feature`
String containing the name of the feature that has just been modified by a client application.

Return Values

S_OK
Success
E_FAIL
Failure

Example

This fragment of an MFC code uses a feature write callback to intercept requests for the change of the image size and modifies the image buffer and acquisition parameters accordingly:

```
int onFeatureWrite(void* context, const char* feature)
{
   CGigemuDlg* dlg = (CGigemuDlg*) context;
   return dlg->onFeatureWrite(feature);
}

int CGigemuDlg::onFeatureWrite(const char* feature)
{
   if(CString(feature)=="SizeX" || CString(feature)=="SizeY")
   {
      delete imgBuf;
      m_pCamera->GetFeatureValue("SizeX",&m_SizeX);
      m_pCamera->GetFeatureValue("SizeY",&m_SizeY);
      imgBuf=new (m_SizeX * m_SizeY)
      return 0;
   }
}
```

```
BOOL CGigemuDlg::OnInitDialog()
{
  ....
  m_pCamera->SetWriteCallback(this, &::onFeatureWrite);
  ....
}
```

Remarks

The write callback function is called after the value of the feature has just been modified by a client application. Therefore, the callback can be used to update those parameters of virtual camera application that are associated with the modified feature.

If the write command is ignored, your callback function must return False, otherwise it must return True. In the latter case GigESim will return an acknowledge packet to the client application.

If you are using a callback function in the main thread of the simulation application, make sure that no lengthy processing is done in the body of the function as otherwise the connection to the remote client may become broken.

4 Samples

The GigESim distribution package includes the following sample applications:

Programming language	Project name	Description
Visual C++ (Windows) QT, C++ 11 (Linux)	GigeConsole	Virtual camera console application with Mono8 and Mono16 pixel formats, test pattern generator, event generator and several remotely-controlled features of different types.
Visual C++ for Visual Studio 6.0, 2005, 2008, 2010, 2013, 2017	GigeCam	Virtual camera GUI-based application with Mono8 and Mono16 pixel formats, test pattern generator, event generator and several remotely-controlled features of different types.
Visual C++ for Visual Studio 6.0, 2005, 2008, 2010, 2013, 2017	GigeCamDualChannel	Virtual camera GUI-based application generating asynchronous patterns on two streaming channels and supporting several remotely-controlled features. The project can be built for both 32- and 64-bit platform.
QT, Linux	GigeSimQT	Virtual camera GUI-based simulator with Mono8 and Mono16 pixel formats, test pattern generator, event generator and several remotely-controlled features of different types.

Note that running sample executables on Windows may require VC++ 2005 Redistributable Package *installed on your system.*

5 Troubleshooting

Below is the list of the most frequently encountered issues and remedies for their resolutions:

Problem description	Cause	Resolution
The simulated camera is not detected by the remote GigE Vision client application	The network adapters bound with the simulator and client application have non-matching IP-addresses. Firewall is enabled. The client and server PCs have a secondary network connection, possibly through the local network Several virtual cameras are connected to the same network adapter, but their IP or MAC addresses are the same. A third-party application is blocking the GigE Vision traffic.	Make sure that the network adapters on both ends have IP addresses assigned to the same subnet (typically 169.254.x.x). Refer to Network Setup or manufacturer's documentation for details. Disable the Windows Firewall or any additional third party firewall on the NIC used by the virtual camera. Make sure that your server and client PCs are connected only through a single network path. Assign a unique IP and MAC address to each simulated camera. See Working with multiple cameras for more details. Some network applications such as Dropbox may block an outgoing GigE Vision traffic. Try to disable processes running in the system one by one until you find the one that causes the problem.
Live video from the simulated camera would not start or occasionally freezes.	Firewall is enabled. UDP packet size exceeds the maximum value allowed by your network configuration. The computer/network card/cabling has an intermittent hardware issue. Both GigESim-based server application and client application are running on the same computer, and the client application uses a network filter driver.	Disable the Windows Firewall or any additional third party firewall on the NIC connected to the camera. Configure your network adapter for Jumbo frames. If Jumbo option is not available for your NIC, reduce the packet size for the virtual camera down to 1500. Replace the cable or try to run the camera/software on a different system. The filter driver prevents the client application from receiving video packets. Deactivate the filter driver associated with the client application in the properties of the network card used by *GigESim*.

The video is corrupted (frames are broken into parts, or synchronization is lost). Some video formats and frame rates do not work.	UDP packet size is incorrect	Configure your network adapter for Jumbo frames. If Jumbo option is not available for your NIC, reduce the packet size for the virtual camera down to 1500.
The transfer speed between my virtual camera and client application is about 600 Mbps, way below 1 Gbps	Network adapter on the server side is not configured for the optimal performance Network adapter on the server side has a mediocre transmission performance	Use our Network Setup instructions to configure the network card. Make sure to use a high-performance network card such as Intel PRO/1000 CT.
I work with multiple virtual cameras, and they get randomly disconnected from the client application.	Excessive activity on the network prevents discovery messages from being received in time.	Set the Broadcast Channel Timeout in your client application to a larger value.
I added new features to my virtual camera object, but the client application does not see them.	The client application uses an old XML file from an xml cache folder.	Use SetDeviceInfo in your code to modify the version of the XML file. As an alternative solution, locate the xml cache folder of your client GigE Vision software and delete an xml file that corresponds to your virtual camera.

Index

- A -

- C -

- D -

- E -

- G -

- I -

- L -

- M -

- N -

- O -

- P -

- R -

- S -

- T -

- W -