PLEORA TECHNOLOGIES INC.

eBUS SDK

Programmer's Guide

Pleora Technologies

Document Number

EX001-017-0001, Version 3.0, 9/17/12

Table of Contents

Chapter 1

About this Guide

This chapter describes the purpose and scope of this guide, provides a list of complimentary guides, definitions for some of the terms used in this guide.

The following topics are covered in this chapter:

What this Guide Provides

This guide provides information and instructions to developers who are integrating the Pleora Technologies' eBUS SDK with their own application in order to communicate with Pleora's GigE Vision compliant products. In this guide, you can review information about the underlying technology being used, along with a high-level view of how the eBUS SDK fits into the system and your application.

Chapters 2-4 provide you with a high-level understanding of the eBUS SDK and of the standards and technology on which it is based.

Chapter 5, "Sample Code" on page 11, provides a list of sample applications included with the eBUS SDK installer.

Chapter 6, "Application Development" on page 17, provides you with the basic information you need to develop an application using the eBUS SDK, including Application Programming Interface (API) class descriptions and design guidelines.

Chapter 7, "System Level Information" on page 69, describes GigE Vision traffic and bandwidth implications.

Guidelines for building and distributing user applications on Linux and Window platforms is included in chapter 8, "Building and Distributing an eBUS SDK Application" on page 75.

The eBUS SDK supports C++, C#.NET, and VB.NET. This guide deals primarily with using the eBUS SDK in the C++ programming language. For information about the eBUS SDK implementation varies between C++ and C#.NET, see "Reference: C++ eBUS SDK and .NET SDK Comparison" on page 87.

Developers using this guide should have firm knowledge of PC hardware and software usage and architecture, along with an understanding of the C++, C#.NET, VB.NET programming languages.

Related Documents

The *eBUS SDK Programmer's Guide* is complemented by the following guides. It is not necessary to read these documents before using the eBUS SDK, but they do provide additional information.

- *GEVPlayer Quick Start Guide*
- *GEVPlayer User Guide*
- *eBUS SDK C++API Help File*
- *eBUS SDK .NET API Help File*

The eBUS SDK help files are available in the Windows Start menu (under **Pleora Technologies Inc.** > **eBUS SDK**) or in Linux (**/opt/pleora/ebus_sdk/share/doc**).

- *GigE Vision Specification* available from the Automated Imaging Association (AIA) located at www.visiononline.org.

You must be registered with the AIA and member of the GigE Vision standard group to have access to the GigE Vision Specification.

- *GenICam Specification* located at: www.genicam.org.
- *Pleora Custom Install Reference Guide*
- *eBUS Advanced Driver Configuration Reference Guide*
- *eBUS SDK Linux Software Guide*
- *Configuring Your Computer and Network Adapters for Best Performance Application Note*
- *Correcting Firewall Issues Application Note*
- *Establishing a Serial Bridge Application Note*

List of Terms

The following table includes some of the terms, along with their definitions, used in this guide.

Table 1: Terms and Definitions

Term	Definition
API	Application Programming Interface
CCP	Control Channel Privilege
GenICam™	GENeric Interface for CAMeras Standards group hosted by the European Machine Vision Association (EMVA)
GigE Vision®	GigE Vision is a camera interface standard developed using the Gigabit Ethernet communication protocol
SDK	Software Development Kit

Chapter 2

Introducing the eBUS SDK

The eBUS Software Development Kit (SDK) is the latest software development kit from Pleora Technologies. Supported under a variety of Linux and Windows operating systems, the eBUS SDK is fully GigE Vision and GenICam compliant, and allows for rapid development of high-performance vision applications. The performance of the eBUS SDK can be enhanced by using the eBUS Universal Pro driver, introduced as part of eBUS SDK 2.0. This driver, compatible with all network interface cards (NICs), decreases host CPU utilization and is optimized for image acquisition, in comparison to the NIC manufacturer's driver.

In addition to communication with video sources such as cameras and other sensors, the eBUS SDK can manage video receiver devices such as the vDisplay family of GigE Vision devices.

This document focuses on how developers can use the eBUS SDK to allow their application to communicate with GigE Vision devices.

Assumed Knowledge

This guide is intended for users with knowledge of:

- Computer hardware
- Software usage and architecture
- Computer programming (C++, C#.NET, or VB.NET)

Chapter 3

Technical Overview

Knowledge of the GigE Vision and GenICam specifications is not required when using Pleora products; however a high-level understanding of these specifications can be helpful to eBUS SDK users.

GigE Vision

GigE Vision is an image interface standard for video transfer and device control over Ethernet networks. It ensures full interoperability between compliant products from different vendors.

More information about the GigE Vision standard can be found at www.visiononline.org.

GenICam

The goal of GenICam is to provide a generic programming interface for all kinds of cameras. Regardless of the interface technology (for example, GigE Vision, Camera Link, 1394 DCAM, or USB, and so forth) or the features that are implemented, the Application Programming Interface (API) should always remain the same. It is important to note that the eBUS SDK acts like a wrapper around the GenICam library.

The GenICam standard consists of multiple modules according to the main tasks to be solved. The eBUS SDK uses the following modules:

- **GenApi**. Used for configuring the camera
- **Standard Feature Naming Convention (SFNC)**. Provides recommended names and types for common features

More information about the GenICam standard can be found at http://www.genicam.org.

The eBUS SDK provides video player software examples, such as GEVPlayer and NetCommand, that you can use to access the available XML nodes.

The features exposed in the GenICam interface of a camera, or in other video sources, are controlled by the camera manufacturer. If the camera you are working with does not expose a particular feature, we recommend that you contact the manufacturer of that camera who may be able to modify the GenICam XML file (and potentially the device's firmware) to include the needed feature.

Supported Integrated Development Environments (IDEs), Compilers, and Operating Systems

The eBUS SDK supports the following IDEs, compilers, and operating systems:

- An appropriate compiler or integrated development environment (IDE):
  - Visual Studio 8 or Visual Studio 9 (if using C++)
  - Version 4.0 of the .NET Framework, Microsoft Visual Studio 2010 (or later) (if using C#.NET or VB.NET)
  - Default development toolchain for Red Hat Enterprise Linux 6.1
- One of the following operating systems:
  - Microsoft® Windows 7, 32 bit or 64-bit
  - Microsoft Windows XP with Service Pack 3 (or later)
  - Red Hat Enterprise Linux 6.1 32 bit and 64-bit
-

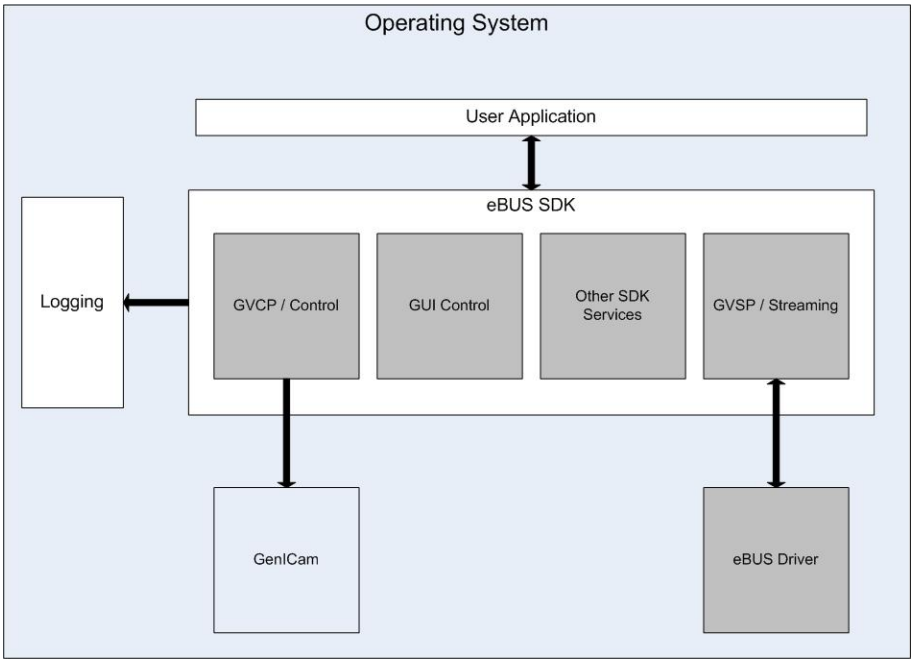
Depending on the incoming and outgoing bandwidth requirements, as well as the performance of each NIC, you may require two NICs. For example, even though Gigabit Ethernet is full duplex (that is, it can simultaneously manage 1 Gbps incoming and 1 Gbps outgoing), the computer's bus may not have enough bandwidth to support this. This means that while your NIC can accept four cameras at 200 Mbps each incoming, and output a 750 Mbps stream on a single NIC (in theory), the NIC you choose may not support this level of performance. A conventional PCI bus (32-bit at 33 MHz) does not provide enough bandwidth to transmit a Gigabit Ethernet stream.

If you use the Linux operating system, you must install the SDK as Root.

Chapter 4

eBUS SDK Architecture

The eBUS SDK can communicate with any GigE Vision-enabled device. The following diagram demonstrates how the eBUS SDK communicates with a Pleora transmitter connected to a camera.



In this example, a user application using the eBUS SDK acts as a receiver that can be used to control and stream video images from the GigE Vision transmitter, for example, a GigE Vision camera. When an application using the eBUS SDK connects to a GigE Vision device, it caches a copy of the XML database to the local memory. Device commands are sent and received though the GVCP command port. Images are received through the GVSP streaming port.

The eBUS SDK software components interface with the following components

- User application
- Operating system
- Logging (to log event information)
- Driver (to receive image stream data) (optional)
- GenApi (to interpret the GenICam XML file format)

The GVCP/Device Control and GenICam software components make use of the GigE Vision Control Protocol (GVCP) to communicate and control a GigE Vision Device. The GVSP/Streaming and eBUS Universal Pro driver software component make use of the GigE Vision Stream Protocol (GVSP) to receive data blocks such as images from the GigE Vision transmitter device.

Chapter 5

Sample Code

To illustrate how you can use the eBUS SDK to acquire and transmit images and data, the SDK includes sample code that you can use. The sample applications can be found in the Samples directory of the eBUS SDK installation directory.

Most of the sample code provided for the eBUS SDK is written using the C++ programming language, with the exception of the PvPipelineSample, PvSimpleUISample, and PvTransmitTestPatternSample applications, which are also available as C#.NET and Visual Basic® .NET (VB.NET) samples.

Sample Applications

The following table provides a description of the sample code that is available for the eBUS SDK.

The following samples are available as C# samples in this release. Some samples are also available as VB.NET samples.

Table 2: Sample Code

Sample code	Function	Type of application that is created
GEVPlayerSample	Provides the source code for the GEVPlayer application, used to detect, connect, configure and display GigE Vision devices and stream images from GEV transmitter devices.	User interface (UI)-based
NetCommandSample	Provides the source code for the NetCommand application, used to detect, connect and configure multiple GEV devices.	UI-based
PvBufferWriterSample	Writes the image content of a PvBuffer object to the disk as either raw data or a standard color BMP file. This sample is only available for C++.	Command line
PvCamHeadSerialComLogSample	Provides basic device control and streaming capabilities. It also allows monitoring the serial communication between a Pleora GigE Vision device and its camera head. This sample is only available for C++.	UI-based
PvConfigurationReaderSample	This sample code illustrates how to use the PvConfigurationReader and PvConfigurationWriter classes. It illustrates how to persist the state of your applications: devices, streams configuration, custom strings, and so forth. This sample is only available for C++.	Command line
PvDeviceFindingSample	This sample illustrates how to use the PvSystem, PvInterface, PvDeviceInfo and PvDeviceFinderWnd classes to detect and enumerate GigE Vision devices. This sample is only available for C++.	Command line
PvDualSourceSample	Provides a user interface that lets you view image streams from two sources simultaneously. This sample is only available for C#.	UI-based

Table 2: Sample Code (Continued)

Sample code	Function	Type of application that is created
PvGenBrowserWndSample	Displays the GenICam features and settings for the IP engine to which you connect, and provides a display window for a single video source. This sample is only available for C#.	UI-based
PvGenParameterArraySample	This sample shows how to discover and access features of a GenApi node map built from a GenICam XML file programmatically using PvGenParameterArray.	UI-based
PvMulticastMasterSample	Connects to a GigE Vision device and initiates a multicast stream (by default it transmits to 239.192.1.1, port 1042).* This sample code is used in conjunction with the PvMulticastSlaveSample, which listens to the multicast stream.	UI-based
PvMulticastSlaveSample	Receives an image stream as a multicast slave from a multicast master.* This sample is used in conjunction with the PvMulticastMasterSample, which initiates the multicast stream.	UI-based
PvPipelineSample, PvPipelineSampleVB	In step-by-step order, connects to a device, displays an image stream, stops streaming, and disconnects from the device.	UI-based. Available as a C#.NET sample and a VB.NET sample.
PvPlcAndGevEvents	This sample shows how to control and handle PLC states and handle GEV events. This sample is only available for C++.	UI-based
PvRecoverySample	Performs image acquisition with automatic recovery support from problems such as accidental disconnects, power interrupts, and so on. This sample is only available for C++.	Command line
PvRGBFilterSample	Grabs an image, converts it to RGB32, applies filters and saves it disk. Places an image in a buffer, converts it to RGB32, applies an RGB and white balance filter, and converts the image to a bitmap file. This sample is only available for C++.	Command line

Table 2: Sample Code (Continued)

Sample code	Function	Type of application that is created
PvSerialBridgeSample	Controls a serial device (usually a camera) connected to a GigE Vision IP Engine from either a camera configuration application or a CLProtocol GenICam interface. This sample is only available for C++.	Command line
PvSimpleUISample, PvSimpleUISampleVB	Provides a basic user interface to connect to a device, and receive and display an image stream. This sample is a good starting point for creating your own UI project.	UI-based. Available as a C#.NET sample and a VB.NET sample.
PvStreamSample	Creates a PvStream for image acquisition. This sample is only available for C++.	Command line
PvTransmitTestPatternSample, PvTransmitTestPatternSampleVB	Transmits a test pattern to a given destination.	Command line. Available as a C++ sample and a .NET (UI-based) sample.
PvTransmitTiledImage	Receives image streams from up to four GigE Vision compatible transmitters (typically cameras), tiles them into a single image feed, and then transmits the tiled image stream to a given destination.	UI-based Available as a C++ sample and a .NET sample.
PvTransmitVideoSample	Captures images from either a file (such as a WMV file) or a capture device (such as a webcam) using OpenCV, and transmits it to a given destination.	Command line
PvTransmitScreenSample	Captures the contents of the screen and transmits it to a given destination.	Command line
PvTransformAndTransmitSample	Receives images from a GigE Vision device, resamples it (to RGB 24 bits per pixel, 640x480 resolution), prints text on it using OpenCV, and transmits it to a given destination.	Command line
PvTransmitRawSample	Transmits raw data to a given destination. This sample code is used in conjunction with the PvReceiveRawSample application.	Command line
PvReceiveRawSample	Receives raw data from a GigE Vision transmitter. This sample code is used in conjunction with the PvTransmitRawSample application.	Command line

Accessing the Sample Applications

The sample code is installed on your computer in the Pleora Technologies Inc folder, as part of the eBUS SDK.

To access the sample code (Windows operating system)

- To access the C++ sample code, on the Windows **Start** menu, click **All Programs** > **Pleora Technologies Inc.** > **eBUS SDK** > **C++ Code Samples Directory**.
- To access the sample code for .NET based languages, on the Windows **Start** menu, click **All Programs** > **Pleora Technologies Inc.** > **eBUS SDK** > **.NET Code Samples Directory**.

  Windows Explorer opens to the location of the sample code.

You can also access the C++ sample code by navigating to the following location (on the computer running the eBUS SDK):

C:\Program Files\Pleora Technologies Inc\eBUS SDK\Samples

And the sample code for .NET-based languages is available by navigating to the following location:

C:\Program Files\Pleora Technologies Inc\eBUS SDK\SamplesDotNet

To access the sample code (Linux operating system)

- Navigate to the following location:

  /opt/pleora/ebus_sdk/share/samples

Chapter 6

Application Development

This chapter provides the information you need to successfully develop an application using the eBUS SDK's API.

This chapter is based on using the eBUS SDK in the C++ programming language. If you are creating a C#.NET or VB.NET application, please consult "Reference: C++ eBUS SDK and .NET SDK Comparison" on page 87 as you read the information in this chapter.

The following topics are covered in this chapter:

API Class Description

The eBUS SDK API is comprised of a set of classes. The main classes are listed in the following table.

Table 3: API Class Descriptions

Class	Description
PvSystem, PvInterface, PvDeviceInfo	Locates devices on the network.
PvDevice (A class referring to a GigE Vision device)	GigE Vision controller.
PvTransmitterRaw	GigE Vision Transmitter used to transmit GigE Vision blocks. This feature is GigE Vision compliant.
PvVirtualDevice	Used to provide basic GigE Vision device capabilities, such as device discovery.
PvGenBoolean, PvGenCommand, PvGenEnum, PvGenEventSink, PvGenFloat, PvGenFile, PvGenInteger, PvGenRegister and PvGenString.	Provides access to various GenICam parameters based on the class type. For example, the PvGenBoolean class provides access to parameters that are either true or false.
PvConfigurationReader, PvConfigurationWriter	Control saving and loading settings and state information. You can save the following in the same file: • One or more device configurations (all parameters) • One or more stream configurations • One or more generic PvGenParameterArray(s) (GenICam nodemap state) • User-specific custom strings
PvSerialPort, PvSerialPortIPEngine	Provides access to a serial port on a Pleora GigE Vision device.
PvIPEngineI2CBus (I2C serial controller)	Sends commands to your camera and receives the camera's replies over an I2C bus connected to a Pleora GigE Vision device. **Note**: This class is not supported by non-Pleora devices.
PvBuffer, PvBufferConverter, PvBufferWriter	Controls the memory used to store blocks of data (typically images) received from the GigE Vision device.
PvFilter, PvFilterDeinterlace, PvFilterRGB	Filters images.
PvStream, PvStreamRaw, PvStreamBase	Receives and controls the data stream from the GigE Vision device to the PC.
PvStatistics	Provides access to stream statistics.
PvPipeline (PvBuffer management and acquisition thread)	Makes driving a PvStream easier, but only applies to simple acquisition scenarios.
PvWnd, PvDeviceFinderWnd, PvDisplayWnd, PvGenBrowserWnd, PvTerminalIPEngineWnd	Provides access to various user interface classes. For example, PvDeviceFinderWnd provides a user interface class that locates GigE Vision devices.

Table 3: API Class Descriptions (Continued)

Class	Description
PvString (String class)	Used to input and output strings to/from the eBUS SDK. Supports both multi-byte and Unicode strings.
PvResult	Provides result information.

For more information, refer to,"Appendix A — eBUS SDK Primitives and Classes" on page 77, or the *eBUS SDK C++ Reference Guide*.

"Appendix B — Layered Representation of the eBUS SDK" on page 83 describes how the each class is organized by library layers.

Design and Development Guidelines

The following table provides important design and development guidelines.

Table 4: Design and Development Guidelines

Design component	Guideline
Asynchronous application	To avoid polling, and blocking, event listening classes are available to asynchronously handle events.
Logging	During application development and debugging, logging can be an extremely useful tool. The eBUS SDK makes use of log4cxx libraries to log events to a file or the console. When an application is deployed, it should perform a minimal amount of application logging. • If more logging information is required to debug field issues, the logging level can be increased as required to isolate the problem. The logging level should be returned to normal once the problem is resolved. Refer to "Appendix B — Layered Representation of the eBUS SDK" on page 83 for information on how to change the logging level. • Serial commands between the Pleora GigE Vision device and the camera head can also be monitored through the **PvDeviceEventSink** Class (Refer to **PvCamHeadSerialComLogSample**). Use the **PvTerminalIPEngineWnd** class to directly communicate with the camera head.
Competing resources	It is important to choose the appropriate hard drives, NICs and other devices so that you do not heavily load the system. Pleora supplies a driver that can be used to improve the transfer between the NIC and the user application. Please refer to the *Configuring Your Computer and Network Adapters for Best Performance Application Note,* available at the Pleora Technologies Support Center, for more details.

Table 4: Design and Development Guidelines (Continued)

Design component	Guideline
Coding guidelines	Most methods return a value, which should be checked. Null pointers should always be checked for reference types. For **PvResult**, you should test the result for success. In most cases IsOK() should be used to test against specific **PvResult::Code values**. **IsSuccess**(), **IsFailure**(), and **IsPending**() should only be used when queuing a buffer in a **PvStream**.
Debugging guidelines	While debugging an application and setting a breakpoint, you might need to increase the heartbeat timeout so that the GigE Vision device does not think it has lost communication with the user application. You may either set **DefaultHeartbeatTimeout** (before connecting to the device) or **GevHeartbeatTimeout** (after connecting to the device) to a high enough value. Refer to "Appendix B — Layered Representation of the eBUS SDK" on page 83 for more information.
GEVPlayer	After the eBUS SDK is installed, we recommend that you become familiar with the sample application, GEVPlayer. GEVPlayer demonstrates a wide variety of eBUS SDK functionality, including, but not limited to, image acquisition, serial communication, GenICam control, and event monitoring. GEVPlayer is included with the numerous other sample applications. We suggest that you become familiar with the GenICam browser in GEVPlayer. The browser is the primary tool for controlling the camera and the GigE Vision device functionality. GEVPlayer provides a solid example of how GenICam features are modified and updated in a system. You can customize GEVPlayer to suit your specific needs or create your own application using the eBUS SDK.

Table 4: Design and Development Guidelines (Continued)

Design component	Guideline
Other sample applications	We recommend that you become familiar with the other sample applications that are automatically installed in the following directories: • **Windows:** C:\Program Files\Pleora Technologies Inc\eBUS SDK\Samples and C:\Program Files\Pleora Technologies Inc\eBUS SDK\SamplesDotNet • **Linux:** /opt/pleora/ebus_sdk/share/samples The sample applications are installed in these directory locations if your installation process is using the default installation directory. These sample applications, along with the source code for the GEVPlayer application, demonstrate most of the functionality available in the eBUS SDK. The sample applications are focused on a single concept, which can help you develop a more complete understanding of that concept.

System performance considerations	You can take the following steps to optimize system performance:
	• Adjust Network Interface Card and driver parameters:
	  • Enable jumbo packets especially when using high data rates
	  • Increase the Rx Descriptor for the NIC at high block rates
	• Increase the number of **PvPipeline** buffers, which may help when buffer depth is not large enough (an increase in CPU usage may indicated that buffer depth is not large enough)
	• Optimize **PvStream** parameters (refer to the *Stream Control Application Note* available at the Pleora Technologies Support Center, for more details.).
	• Use the eBUS Universal Driver when using large block size at high block rates. The eBUS Universal Driver is optimized to decrease CPU usage.
	The following application notes, available at the Pleora Technologies Support Center, provide important information you can use to optimize your application with regards to system performance:
	• *Stream Control Application Note*
	• *Configuring Your Computer and Network Adapters for Best Performance Application Note*
	• *Correcting Firewall Issues Application Note*

Using the eBUS SDK API

This section provides information on basic eBUS SDK API services, as well as the more advanced services and Windows GUI components.

Basic Services

You can use the following basic eBUS SDK API services, as outlined in this section:

- Detecting to GigE Vision devices
- Connecting to GigE Vision devices
- Configuring GigE Vision devices
- Receiving data from a GigE Vision transmitter

Detecting GigE Vision Devices

Detecting GigE Vision devices can be accomplished by:

- Using the **PvDeviceFinderWnd** UI class

  - Or -

  Programmatically using **PvSystem**, **PvInterface** and **PvDeviceInfo**.

  You can also use the **PvSystemEventSink** class to perform a callback function. The **PvSystemEventSink** class can be used to notify the application each time a new device is detected and filter devices accordingly.

To detect a GigE Vision device using the PvDeviceFinderWnd GUI function

1. Create a UI-based GigE Vision device finder, for example a **PvDeviceFinderWnd** object.
2. Display the dialog box using **PvWnd::ShowModal**().
3. Retrieve the user's selection (information about the selected GigE Vision device) using **PvDeviceFinderWnd::GetSelected**() to retrieve an instance of **PvDeviceInfo**.

Code Example

C++

```
PvDeviceFinderWnd lFinderWnd;
if (! lFinderWnd.ShowModal().IsOK() )
{
   return;
}

// When dismissed with OK, the device finder dialog keeps a reference
// on the device that was selected by the user. Retrieve this reference.
PvDeviceInfo* lDeviceInfo = lDeviceFinderWnd.GetSelected();

}
```

To programmatically detect a GigE Vision device using PvSystem, PvInterface and PvDeviceInfo

1. Create a **PvSystem** object (This object represents the computer).
2. Set the maximum time to search for GigE Vision devices using **SetDetectionTimeout**().
3. Optionally, use a callback to filter out GigE Vision devices or connect to them more quickly. For more information, refer to **PvSystemEventSink**.
4. Search for all GigE Vision devices through all available NICs on the PC, using **Find**().
5. Get the number of NICs on the PC, using **GetInterfaceCount**().
6. For each NIC on the PC:
   - Get the Interface (**PvInterface** object), using **GetInterface**().
   - Get the number of GigE Vision devices found under the NIC, using **PvInterface::GetDeviceCount**().
   - For each GigE Vision device:
     - Get the GigE Vision device information (**PvDeviceInfo** object). Use **PvInterface::GetDeviceInfo**().
     - Test to see if the GigE Vision device is the one you seek, using **PvDeviceInfo::GetMACAddress**(), **PvDeviceInfo::GetIPAddress**(), and other **PvDeviceInfo** methods.
     - Continue until you find the GigE Vision device you seek.

Code Example

C++

```
PvSystem lSystem;
PvResult lResult;
PvDeviceInfo *lDeviceInfo = NULL;

// Find all GigE Vision devices on the network.
lSystem.SetDetectionTimeout( 200 );

lResult = lSystem.Find();
if( !lResult.IsOK() )
{
    printf( "PvSystem::Find Error: %s", lResult.GetCodeString() );
    return -1;
}

// Get the number of GEV Interfaces that were found using GetInterfaceCount.
PvUInt32 lInterfaceCount = lSystem.GetInterfaceCount();

// Display information about all found interfaces.
// For each interface, display information about all devices.
PvInterface * lInterface= NULL;
for( PvUInt32 x = 0; x < lInterfaceCount; x++ )
{
    // Get pointer to each of interface.
    lInterface = lSystem.GetInterface( x );
    // Get the number of GigE Vision devices that were found using GetDeviceCount.
    PvUInt32 lDeviceCount = lInterface->GetDeviceCount();
    for( PvUInt32 y = 0; y < lDeviceCount; y++ )
    {
        lDeviceInfo = lInterface->GetDeviceInfo( y );
        // Check if the device is the desired one
    }
}
// Connect to the last GigE Vision device found.
PvDevice lDevice;
if( lDeviceInfo != NULL )
{
    printf( "Connecting to %s\n",
        lDeviceInfo->GetMACAddress().GetAscii() );

    lResult = lDevice.Connect( lDeviceInfo );
    if ( !lResult.IsOK() )
    {
        printf( "Unable to connect to %s\n",
            lDeviceInfo->GetMACAddress().GetAscii() );
    }
}
else
{
    printf( "No device found\n" );
}
```

Connecting to GigE Vision devices

Use the following procedures to connect to a GigE Vision device.

To programmatically connect to an IP engine

1. Select a GigE Vision device.

   For more information, see “To programmatically detect a GigE Vision device using PvSystem, PvInterface and PvDeviceInfo” on page 23.

2. Get a reference to the selected GigE Vision device.

   If using **PvDeviceFinderWnd**, use **GetSelected**().

3. Connect to the IP engine using **PvDevice::Connect**().

Code Example

C++

```
PvDeviceFinderWnd lFinderWnd;
if (! lFinderWnd.ShowModal().IsOK() )
{
   return;
}

// When dismissed with OK, the device finder dialog keeps a reference
// on the device that was selected by the user. Retrieve this reference.
PvDeviceInfo* lDeviceInfo = lDeviceFinderWnd.GetSelected();

// Connect to the selected GigE Vision device.
if( lDeviceInfo != NULL )
{
    printf( "Connecting to %s\n",
    lDeviceInfo->GetMACAddress().GetAscii() );

    PvDevice lDevice;
    PvResult lResult = lDevice.Connect( lDeviceInfo );
        if ( !lResult.IsOK() )
        {
            printf( "Unable to connect to %s\n",
            lDeviceInfo->GetMACAddress().GetAscii() );
        }
        else
        {
            printf( "Successfully connected to %s\n",
            lDeviceInfo->GetMACAddress().GetAscii() );
        }
    }
    else
    {
            printf( "No device selected\n" );
```

To connect with known connectivity information using PvDevice

1. Create a GigE Vision device controller using **PvDevice**.
2. Connect to the GigE Vision device using **Connect(const PvString&,…)**.
3. The string can contain either any of; an IP address like “192.168.1.100”, a MAC address such as “00:11:22:33:44:5F” or the device’s name (DeviceUserID) such as “MyDevice”.

Code Example

C++

```
PvDevice *aDevice = new PvDevice();
PvDeviceInfo lDeviceInfo;
aDevice->Connect("192.168.1.100");
if( !lResult.IsOK() )
{
    printf( "PvDevice::Connect Error: %s", lResult.GetCodeString() );
    return -1;
}
```

Connection Heartbeat/Link Status

A heartbeat message is sent periodically from the controller application to the GigE Vision device to ensure that the connection between the application and the device still exists.

GigE Vision devices are controlled through the primary control channel. In order to control a device, an application must lock the primary control channel by writing the required information to request either controlled or shared access to the device using the Control Channel Privilege (CCP) register.

Taking control of a device by locking the CCP register is performed automatically when successfully connecting a PvDevice to a GigE Vision device using the **Connect**() method. By default, **Connect**() assumes **PvAccessControl** where other applications can still access the device in read-only mode without write/control privileges. Other applications can read registers but cannot change the configuration or request the device to start streaming. Only the application locked on the primary control channel can perform these actions. When the application is connected as **PvAccessControl**, the eBUS SDK sends a heartbeat to the device to keep the control channel active. The application is considered as the primary one — no other application may control the device.

The GigE Vision device stays locked for that PvDevice until the CCP is released, for example, when **PvDevice::Disconnect**() is called, or when no heartbeat has been received from the primary application within the **GevHeartbeatTimeout** interval.

If you set a breakpoint in your application, and take time to trace through the code and investigate, you will find that the heartbeat thread is as idle as the rest of the code. You should consider increasing the value of the heartbeat timeout when debugging your code to prevent losing the connection when the debugger stops at a breakpoint. You should set either the **DefaultHeartbeatTimeout** (before connecting to the device), or **GevHeartbeatTimeout** (after connecting to the device), to a high enough value to avoid losing the connection. Pleora suggests a value of 45000, which gives your application 45 seconds to issue each heartbeat. The disadvantage of doing this is that if your application crashes, or you abruptly stop debugging without properly disconnecting the PvDevice (which would have explicitly released the CCP), it can take the GigE Vision device up to 45 seconds to detect that the controller is disconnected and then release the CPP. If this happens, you can either wait the 45 seconds, or reset the device by power cycling.

Configuring a GigE Vision Device

The eBUS SDK facilitates the creation of an array representing the device's GenICam parameters through the use of:

- **PvDevice::GetGenParameters**().
- **PvDevice::GetGenLink**() (can be used to access an array of the PC side parameter settings).

There are also GigE Vision register addressable methods that can be used to control GigE Vision devices through direct address reads and writes, for example, (**PvDevice::ReadMemory**() and **PvDevice::ReadRegister**()).

Camera control can be accomplished by manipulating features in the GenICam node map, assuming those features are exposed by the manufacturer.

The sample applications, GEVPlayer, and PvSimpleUI, provide examples of how to control GigE Vision devices. You can write code similar to the sample code, provided below, to obtain the same results as the sample applications.

Device Parameter and Stream Control

Use the following procedure to control features.

To control features

1. Get the list of features by using one of the following methods:
   - **PvDevice::GetGenParameters**() (GigE Vision device's settings)
   - **PvDevice::GetGenLink**() (PC's communication related settings)
   - **PvStream::GetParameters**() (Image stream settings)
2. Get a reference to the feature using **Get**().
   This returns a **PvGenParameter** instance.
3. If required, get the feature's type using **PvGenParameter::GetType**().

4. Optionally:
   - Get/set the feature's value. Use the **GetValue/SetValue** method for the feature's type (for example, **PvGenInteger::GetValue**(), **PvGenFloat::SetValue**(), and so forth).
   - If the feature is a command, activate it using **PvGenCommand::Execute**().
   - Get/set the feature's value with one method (for example, **GetIntegerValue**, **SetIntegerValue**, **ExecuteCommand**, and so forth).

Code Example

C++

```
// Connect to the GigE Vision device.
PvDevice lDevice;

// Connect to device.
…
…
//

// Get device parameters needed to control streaming.
PvGenParameterArray *lDeviceParams = lDevice.GetGenParameters();

// Negotiate the streaming packet size.
lDevice.NegotiatePacketSize();

// Create the PvStream object.
PvStream lStream;

// Open stream - have the PvDevice do it for us.
printf( "Opening stream to device\n" );
lStream.Open( lDeviceInfo->GetIPAddress() );

// Create the PvPipeline object
PvPipeline lPipeline( &lStream );

// Read the payload size from the device.
PvInt64 lSize = 0;
lDeviceParams->GetIntegerValue("PayloadSize",lSize);

// Set the Buffer size and the Buffer count
lPipeline.SetBufferSize( static_cast<PvUInt32>( lSize ) );
lPipeline.SetBufferCount( 16 );

// Increase for high frame rate without missing block IDs.

// Have to set the Device IP destination to the Stream.
lDevice.SetStreamDestination( lStream.GetLocalIPAddress(), lStream.GetLocalPort() );

// IMPORTANT: the pipeline needs to be "armed", or started before
// we instruct the device to send us images.
printf( "Starting pipeline\n" );
lPipeline.Start();

// Get stream parameters/stats
PvGenParameterArray *lStreamParams = lStream.GetParameters();
```

Continued on next page...

```
// TLParamsLocked is optional but when present, it MUST be set to 1
// before sending the AcquisitionStart command.
if (lDeviceParams->SetIntegerValue("TLParamsLocked",1).IsOK())
{
    printf( "Setting TLParamsLocked to 1\n" );

}

printf( "Resetting timestamp counter...\n" );
lDeviceParams->ExecuteCommand("GevTimestampControlReset");

// The pipeline is already "armed", we just have to tell the device
// to start sending us images.
printf( "Sending StartAcquisition command to device\n" );
PvResult lResult = lDeviceParams->ExecuteCommand("AcquisitionStart");
// Acquire images until the user instructs us to stop
printf( "\n<press the enter key to stop streaming>\n" );

// Tell the device to stop sending images.
printf( "Sending AcquisitionStop command to the device\n" );
lDeviceParams->ExecuteCommand("AcquisitionStop");

// If present reset TLParamsLocked to 0. Must be done AFTER the
// streaming has been stopped.

if (lDeviceParams->SetIntegerValue("TLParamsLocked",0).IsOK())
{
    printf( "Resetting TLParamsLocked to 0\n" );
}

// We stop the pipeline - letting the object lapse out of
// scope would have had the destructor do the same, but we do it anyway.
printf( "Stop pipeline\n" );
lPipeline.Stop();

// Now close the stream. Also optional but nice to have.
printf( "Closing stream\n" );
lStream.Close();

// Finally disconnect the device. Optional, still nice to have.
printf( "Disconnecting device\n" );
lDevice.Disconnect();
```

Programmable Logic Controller (PLC)

Some Pleora devices are equipped with Programmable Logic Controller (PLC) capabilities.

For a complete working sample of how to control the PLC, refer to the **PvPlcAndGevEvents** sample, located at **C:\Program Files\Pleora Technologies Inc\eBUS SDK\Samples**.

- **PvPlcAndGevEvents**
- **PvPlcDelayerSample**
- **PvRescalerSample**

Device Persistence

To save the device's settings, including its IP configuration and the device name, you can leverage user sets, which save this type of information to the device's flash memory. The IP engine can be configured so that, at the next power cycle, the device settings are automatically set to the previously saved information. You can also restore the default settings at next power cycle, or at any time.

The following parameters are always saved persistently and are independent from the user set operations:

- **GevPersistentIPAddress**
- **GevPersistentSubnetMask**
- **GevPersistentDefaultGateway**
- **GevCurrentIPConfigurationDHCP**
- **GevCurrentIPConfigurationPersistentIP**

The following code example shows how to access the UserSet parameters.

Code Example

C++

```
PvString lValue;
PvGenEnum* lPvGenEnum;
PvGenCommand *lPvGenCommand;

 // ******************************************************************//
 // Save current settings into device flash memory and
 // configure the device to restore those setting upon power up
 // ******************************************************************//
lDevice.GetGenParameters()->SetEnumValue( "UserSetSelector", "UserSet1" );
lDevice.GetGenParameters()->ExecuteCommand( "UserSetSave" );
lDevice.GetGenParameters()->SetEnumValue( "UserSetDefaultSelector", "UserSet1" );

 // ******************************************************************//
 // Reset to the default settings at next power up
 // ******************************************************************//
lDevice.GetGenParameters()->SetEnumValue( "UserSetDefaultSelector", "Default" );
```

Receiving Data from a GigE Vision Transmitter

This section discusses how data is received from a GigE Vision transmitter, such as a GigE Vision camera.

PvStream and PvPipeline

The **PvStream** class is used to receive a stream of image blocks flowing from the GigE Vision transmitter to the computer receiving the images (image receiver).

The image receiver does not have to be the same computer that controls the GigE Vision device.

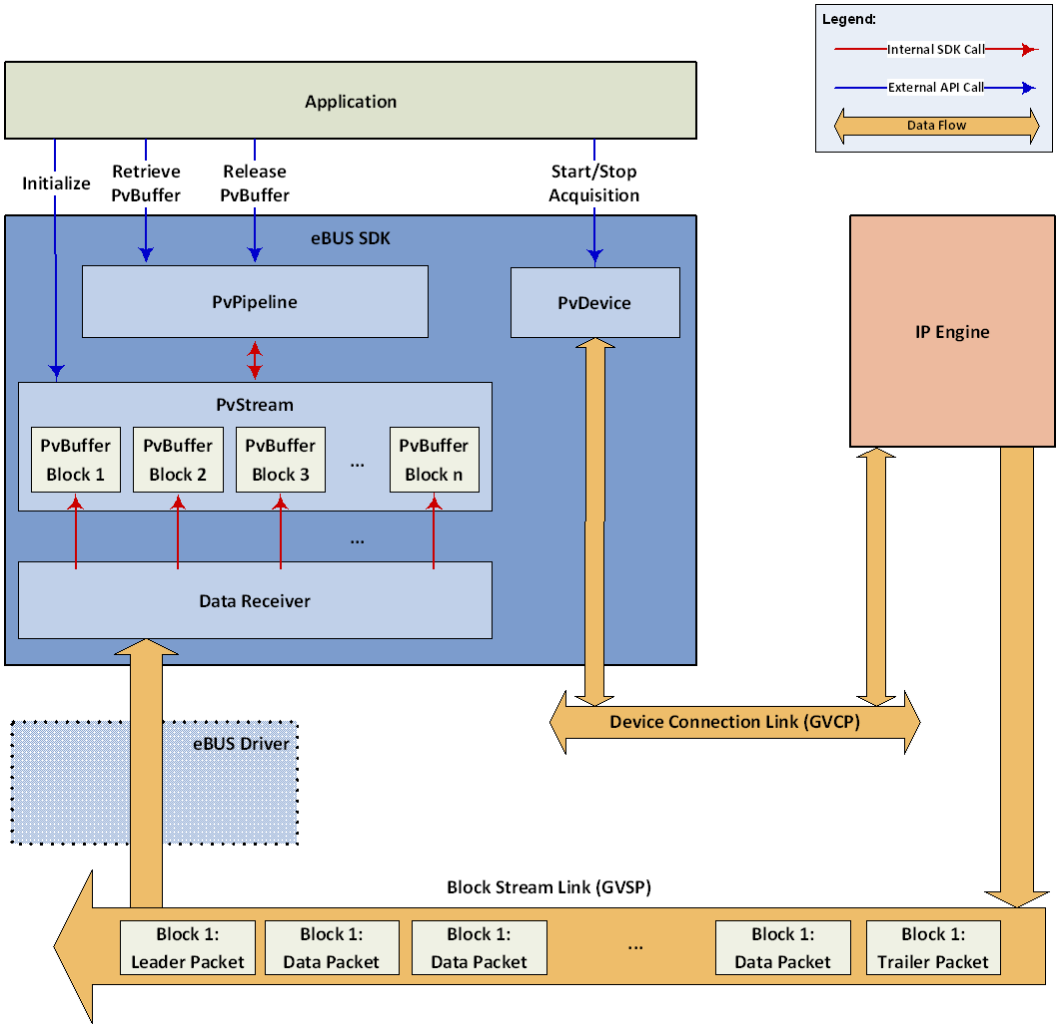
PvStream is usually only responsible for receiving images and attempting to retrieve buffers. Use **PvDevice** to control the device and initiate the image stream acquisition.

PvStream can automatically send packet resend requests to the GigE Vision device, and in some cases read some configuration registers to find out about device capabilities. However, it does not otherwise CONTROL the device in any way.

An image stream can be received directly by the **PvStream** class or by the **PvPipeline** class. **PvPipeline** uses **PvBuffers** that are handled by **PvStream**, which hides some of the complexity of buffer management allowing for simpler code. If you want more control over the system, you can use **PvStream** directly.

The following diagram illustrates how blocks are received from the transmitter.

GigE Vision stream packets are received by the data receiver and assembled to recreate a block (typically, an image) in each **PvBuffer** (illustrated as a block). The **RetrieveBuffer**() method can then be used to retrieve the next available PvBuffer. Once the application is done with the **PvBuffer,** it re-queues the PvBuffer to the PvStream object to be re-used for another block.



The following section describes how to configure a **PvStream** and a **PvBuffer** to receive through the data receiver from the camera. The application must ensure that sufficient free buffers are available in the stream so that the data receiver does not overflow. For optimal performance and to avoid losing images, at least two buffers should be available in PvStream at all times. Add buffers immediately after removing them, (see **PvStreamBase::QueueBuffer**() in the *eBUS SDK API Help File*), using **GetQueuedBufferCount**(). In general, if images are received at 30 FPS, four buffers should be sufficient, but this depends on various factors. The number of buffers varies based on the incoming data rate, data size, processing to be done by the application, and speed of the PC (this includes CPU speed, memory bus speed, and NIC to CPU interface speed). It is a good idea to increase the buffer count when handling high frame rates, especially when missing block IDs are being observed.

To use PvStream to receive images

1. Create a stream object (**PvStream** object) using the **PvStream** constructor.
2. Open the stream using **Open**().
3. Create and configure buffers.

   For more information, see the PvBuffer Class topic in the *eBUS SDK API Help File*.

   Note: The eBUS SDK can allocate the memory, or you can attach to an existing memory pool. If you are using an external memory pool, you can use **PvBuffer::Attach**(). For more information, see **PvBuffer::Attach**() in the *eBUS SDK API Help File*.
4. Create a loop using your own code. In the loop:
   - Queue buffers up to the maximum number of **GetQueuedBufferMaximum**() using **QueueBuffer**.
   - Retrieve a buffer and immediately queue another (to maintain **GetQueuedBufferMaximum**() buffers in the queue). Use **PvStreamBase::RetrieveBuffer**() and **PvStreamBase::QueueBuffer**().
   - Optionally, image stream statistics can be accessed. For more information, see the **PvStatistics** topic in the *eBUS SDK API Help File*.
   - Process the image contained in the **PvBuffer**.
     - Test the success of the image acquisition. Use the **aOperationResult** parameter in **PvStream::RetrieveBuffer**() or **PvBuffer::GetOperationResult**(). **PvBuffer::GetOperationResult**() lets you see the state of the buffer before trying to read the data (image) where you may, for example, be missing packets.
     - Retrieve a **PvImage** interface to the buffer using **GetImage**().
     - Process the image using your own code. You can process the image in place by using:
     - **PvImage::GetAcquiredSize**()
     - **PvImage::GetHeight**() and **PvImage::GetWidth**()
     - **PvImage::GetOffsetX**() and **PvImage::GetOffsetY**()
     - **PvBuffer::GetDataPointer**()
     - Continue queuing, retrieving, and processing buffers.

- For optimal performance and to avoid losing images, at least two buffers should be available in PvStream at all times. Add buffers immediately after removing them, (for more information, see **PvStreamBase::QueueBuffer**() in the *eBUS SDK API Help File*), using **GetQueuedBufferCount**().
- Optionally, get statistics about the image stream performance using **GetParameters**() and **PvGenParameterArray**.

5. Close the stream using **PvStream::Close**().

Sample Code

- To view sample code for **PvStream**, view the **PvStreamSample** file located at **C:\Program Files (x86)\Pleora Technologies Inc\eBUS SDK\Samples**.

Using PvStream and PvPipeline

The following section describes how to use **PvStream** and **PvPipeline** to receive PvBuffers from the data receiver.

PvPipeline provides an intuitive interface on top of a **PvStream** or **PvStreamRaw**. The pipeline allocates buffers, manages buffer size, and runs threads dedicated to pulling buffers out of a **PvStream** and making them available to your application while ensuring sufficient buffers are queued in a **PvStream**.

When the output-queue of the **PvPipeline** is full, buffers are dropped and recycled as available buffers to be queued in **PvStream**. Thus, it is important for the application to ensure that the PvPipeline is serviced regularly.

Similar to **PvStream**, the **PvPipeline** is only responsible for receiving data. Starting the **PvPipeline** only arms it; **PvPipeline** does not interact with the IP engine in any way.

Use **PvDevice** to control the device and initiate image acquisition.

To use PvPipeline to receive images

1. Create a stream object (**PvStream** object) using the **PvStream constructor**.
2. Open the stream using **PvStream::Open**().
3. Create a multi-buffer controller (**PvPipeline** object).
4. Optionally, set the buffer's size using **SetBufferSize**(). The size to set is determined by reading the **GevPayloadSize** on the device after setting Width, Height, and PixelFormat. (For more information, see "Configuring a GigE Vision Device" on page 27.)
5. Optionally, set the number of buffers in the pipeline using **SetBufferCount**().
6. Start the Pipeline using **PvPipeline::Start**().
7. Get one or more image buffers (**PvBuffer** object) using **RetrieveNextBuffer**().

8. If the **PvResult** returned from **RetrieveNextBuffer**() is successful then, process the image in **PvBuffer**.
   - Retrieve a **PvImage** interface to the buffer by using **GetImage**().
   - Test the success of the image acquisition. Use the **aOperationResult** parameter in **PvPipeline::RetrieveNextBuffer**() or **PvBuffer::GetOperationResult**() to test the acquisition operation on the **PvBuffer**.
   - Process the image using your own code. You may process the image in place, if you wish by using:
     - **PvImage::GetAcquiredSize**()
     - **PvImage::GetHeight**() and **PvImage::GetWidth**()
     - **PvImage::GetOffsetX**() and **PvImage::GetOffsetY**()
     - **PvBuffer::GetDataPointer**()
9. Continue retrieving, processing, and returning image buffers. Use **RetrieveNextBuffer**() and **ReleaseBuffer**().

Failure to return the image buffer to the pipeline using **ReleaseBuffer()** can cause the PvPipeline to starve from lack of buffers. In general, (30 FPS), four buffers should be sufficient, but this depends on various factors. The number of buffers varies based on the incoming data rate, data size, processing to be completed by the application, and speed of the computer (this includes CPU speed, memory bus speed, and NIC-to-CPU interface speed). You should increase the buffer count when handling high frame rates, especially when missing block IDs are being observed.

10. Stop the image acquisition using **Stop**().

Sample Code

- To view sample code for **PvPipeline**, view the **PvPipelineSample** file. For information about accessing the sample code, see "Accessing the Sample Applications" on page 15.

Number of Buffers Queued into Data Receiver

In the eBUS SDK with the introduction of the eBUS Universal Pro and a new data receiver, the maximum number of buffers that can be queued into the data receiver has been increased to 64. You should always confirm this number by calling **GetBufferQueuedMaximum**().

If you are using **PvPipeline,** this is automatically handled for you, where "n" is the number of buffers used by the **PvPipeline**. It is configured with the **PvPipeline::SetBufferCount**() method and the default buffer count is 16.

Driver and Stream Settings

PvStream makes use of the IP stack to retrieve IP (GVSP) packets which make up a particular block (typically an image). Depending on the system requirements, it is important to note that it may be necessary to tune the system for optimal performance.

Table 5: Driver and Stream Settings

Recommended action	Benefits
For Windows OS	
Install Universal Pro Filter Driver	Decreases CPU usage
Enable Jumbo Packet option on NIC	Decreases network overhead, and CPU usage (lowers the number of interrupts).
Set large Rx Descriptor on NIC	Decreases likelihood of losing packets
Interrupt Moderation	Allows the adapter to moderate interrupts.
For Linux OS	
Set Jumbo Packets on NIC	Important when large blocks are used > 1500 bytes
Set large Rx Descriptor on NIC	Decreases likelihood of losing packets
Set large Rx Socket size on IP stack	Decreases likelihood of losing packets (use set_socket_buffer_size.sh)
Set root as running level (optional)	Running as root switches the schedulers you are running on Linux.; the application runs in a more deterministic and stable fashion.

Some communication settings are available in order to modify the behavior of the image stream.

GEVPlayer's **Image Stream Control** window is a good place to become familiar with the features and their descriptions.

All parameters and statistics are accessed through **PvStream::GetParameters**(). The stream control configuration settings differ depending on the driver being used.

Stream Statistics

The eBUS SDK stream statistics provides the application information such as image count, average bandwidth, and missing block IDs, for example. The statistics for a PvStream object are provided through the GenICam interface.

Code Example

C++

```
//****************************************************************************//
// Retrieve the up-to-date statistics values at any time you like             //
//****************************************************************************//
PvInt64 lV1, lV2, lV3;
lStream.GetParameters()->GetIntegerValue( "ImagesCount", lV1 );
lStream.GetParameters()->GetIntegerValue( "BytesCount", lV2 );
lStream.GetParameters()->GetIntegerValue( "BlockIDsMissing", lV3 );

printf( "Images count: %lld bytes count %lld missing block IDs: %lld",lV1, lV2, lV3 );
```

- The stream statistics are cumulative values which are reset by executing the **PvStatistics::Reset** command.
- Even though the example above shows how to retrieve integers, some statistics are expressed as floating point values. To handle these statistic types, use **PvGenFloat** and **PvGenParameterArray::GetFloatValue**().

Block Acquisition

Blocks (image) may be received by one or multiple receivers. In the case of one receiver, a unicast connection is sufficient (point to point). In the case of multiple receivers, an IGMP-compliant switch can be used to multicast the same block (image) to multiple receivers.

Three types of acquisition are possible using the eBUS SDK,

- Unicast Acquisition
- Remote Unicast Acquisition
- Multicast Acquisition

Unicast Acquisition

1. Once connected, get the GigE Vision device's GenICam parameters using **PvDevice::GetGenParameters**().
2. Configure how the GigE Vision device grabs images. Relevant GenICam features include:
   - Width
   - Height
   - PixelFormat
3. Configure your PC(s) to receive blocks through **PvStream**.
4. Configure your **PvDevice's** streaming destination using **PvDevice::SetStreamDestination**().

   The parameters passed identify the location to which the image data should be transmitted. The destination consists of an IP address (unicast or multicast) and a port number.
5. Lock the interface for streaming. Set **TLParamsLocked** feature of the GigE Vision device's GenICam interface to "1".

All Pleora devices use **TLParamsLocked** as an Integer, some third-party GigE Vision devices define it as a Boolean.

GenICam has the concept of locking features using the **TLParamsLocked** node. Any feature can link to this node. When **TLParamsLocked** is set to **1**, all features linked to this feature cannot be changed. As an example of a typical use case, you can set **TLParamsLocked** to **1** before **AcquisitionStart** and **0** after **AcquisitionStop**. This protects the streaming related parameters from being changed during streaming. The features linked to this node are determined by the GenICam XML supplied by the device manufacturer.

6. Use the **AcquisitionStart** function of the GigE Vision device's GenICam interface to start the image stream from the device.
7. When required, use the **AcquisitionStop** feature of the GigE Vision device's GenICam interface to stop image acquisition.
8. Set **TLParamsLocked** feature of the GigE Vision device's GenICam interface to **0**.

Code Example

C++

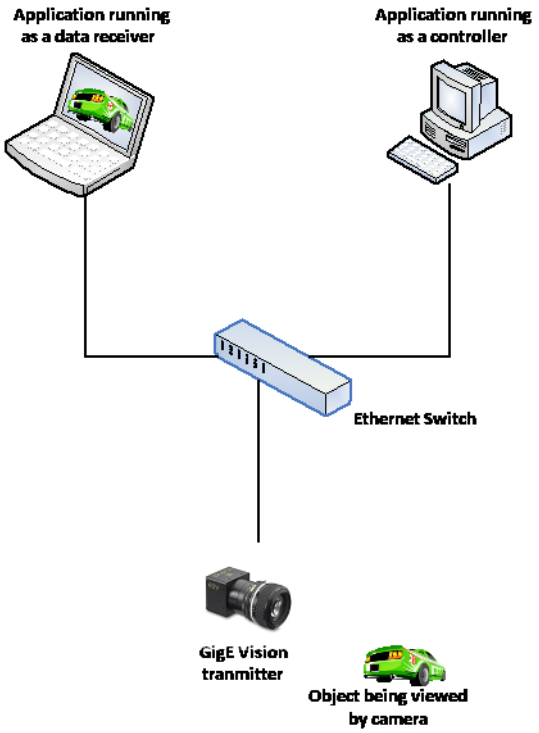
```
// Connect PvDevice as above
// Open PvStream as above
// ...

// Instruct device to stream to our PvStream object
lDevice.SetStreamDestination(
   lStream.GetLocalIPAddress(),
   lStream.GetLocalPort() );

// Read payload size from the devices GenICam interface. Optional, could also be
// computed locally or automatically handled by the PvPipeline after missing a frame
// with BUFFER_TOO_SMALL
lDevice.GetIntegerValue("PayloadSize", lPayloadSizeValue);

// Instantiate, start pipeline
PvPipeline lPipeline;
lPipeline.SetBufferSize( lPayloadSize );
lPipeline.Start();

// Lock device parameters
lDevice.SetIntegerValue("TLParamsLocked", 1);

// Instruct device to start acquisition
mDevice.ExecuteCommand("AcquisitionStart");

// Acquisition loop
while ( ... )
{
   PvBuffer *lBuffer = NULL;

   // Retrieve buffer from pipeline
   PvResult lResult =
      lPipeline.RetrieveNextBuffer( &lBuffer );

   // If retrieve succeeded
   if ( lResult.IsOK() )
   {
      if ( lBuffer->GetOperationResult.IsOK() )
      {
         //
         // Do something with the buffer…
         //
      }

         // IMPORTANT!!
      lPipeline.ReleaseBuffer( lBuffer );
   }
}

// Instruct device to stop acquisition
mDevice.ExecuteCommand("AcquisitionStop");

// Unlock device parameters
lDevice.SetIntegerValue("TLParamsLocked", 0);

// Cleanup
lPipeline.Stop();
lStream.Close();
lDevice.Disconnect();
```

Sample Code

- To view sample code for **PvPipeline**, view the **PvPipelineSample** file. For information about accessing the sample code, see “Accessing the Sample Applications” on page 15.

Remote Unicast

A remote unicast setup is made up a management entity (PC) configuring a GigE Vision transmitter. A receiver then connects with a unicast address to the unit and receives the blocks.



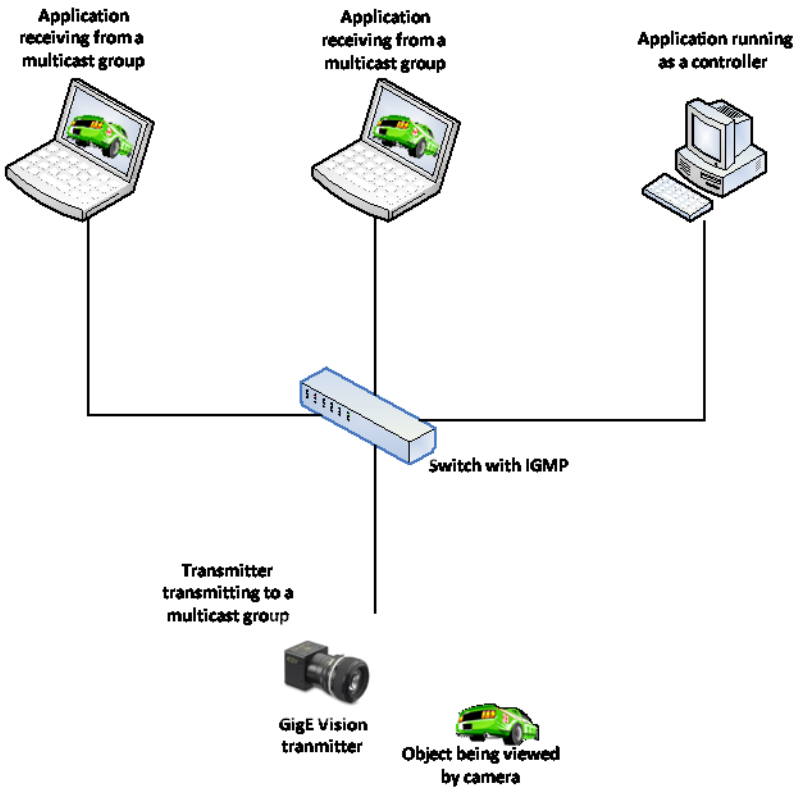
Code Example

C++

```
//*************************************************************************//
// Step 1: Set the Communication parameters before connecting //
//*************************************************************************//
// For an example for setting DefaultHeartTimeout, see the
// “Access Communication Settings” section
//*************************************************************************//
// Step 2: Connect to device
//*************************************************************************//
PvDevice lDevice;

// method 1: Connect using the device info lDeviceInfo obtained in the
// "Detect and Select Device" section
lDevice.Connect(lDeviceInfo, aAccessType );
// method 2: Connect using the IP Engine's IP address; in this example it is
// "192.168.1.128"
lDevice.Connect("192.168.1.128", aAccessType );
// method 3: Connect using IP Engine's MAC address; in this example it is
// “00:11:1C::00:76:0A” The format "00-11-1C-00-76-0A also works.

lDevice.Connect("00:11:1C:00:76:0A", aAccessType );
//*************************************************************************//
// Step 3: Set up the stream packet size. Only the device controller can do
// this. If you skip this step, the current value of GevSCPPacketSize on
// the device is used.
//*************************************************************************//
// Option A: Perform auto negotiation to find largest packet size that can be
// used on the network link. At the completion of this call, GevSCPPacketSize
// is set to the largest possible packet size; if NegotiatePacketSize() fails,
// it is set to 1476 lDevice.NegotiatePacketSize( 0, 1476 );

// Option B: Set a value that you know works well. In the case of multicasting
// or remote unicast, this is good practice, as auto negotiation does not test
// all stream receivers to determine the largest packet size that can be used
// lDevice.GetGenParameters()->SetIntegerValue( “GevSCPSPacketSize”,1476 );

//*************************************************************************//
// Step 4: Set up proper streaming networking topology
//*************************************************************************//

// Scenario 2: Remote unicast controller -- this application controls the
// device and does not receive the image data. In this example the stream goes
// to destination IP address 192.168.1.89 and port 2001. As this application
// does not receive image data, there is no need to create a stream object
// lDevice.SetStreamDestination( "192.168.1.89" , 2001 );

// Scenario 3: For Remote unicast data receiver -- this application does not
// control the device and only receives the image data. It expects image data
// coming in from the NIC with IP address: 192.168.1.89 on port 2001
// This application does not need to connect to the device, but still needs to
// know the device's IP address to open the stream
lStream.Open(lDeviceInfo ->GetIPAddress(), 2001, 0, "192.168.1.89" );
```

Multicast Acquisition

A multicast setup is made up of a management entity (PC or vDisplay) configuring a GigE Vision transmitter to send block data to a multicast group granted by an IGMP Ethernet switch. GigE Vision receivers may then connect to the multicast group and receive the blocks.



On Controller (master) PC

1. Connect to the IP engine using **PvDevice::Connect**().
2. Set the streaming destination to the multicast groups IP and port.
3. Lock device parameters.
4. Execute the **AcquisitionStart** command from the device's GenICam interface.

   The application receiver (slave) should receive images from the GigE device transmitter.
5. Execute the **AcquisitionStop** command from the devices GenICam interface.
6. Unlock the devices parameters.
7. Cleanup.

Code Example

C++

```
//*************************************************************************//
// Step 1: Set the Communication parameters before connecting //
//*************************************************************************//
// For an example for setting DefaultHeartTimeout, see the
// "Access Communication Settings" section //
**************************************************************************//
// Step 2: Connect to device
//*************************************************************************//
PvDevice lDevice;

// method 1: Connect using the device info lDeviceInfo obtained in the
// "Detect and Select Device" section
lDevice.Connect(lDeviceInfo, aAccessType );
// method 2: Connect using the IP Engine's IP address; in this example it is
// "192.168.1.128"
lDevice.Connect("192.168.1.128", aAccessType );
// method 3: Connect using IP Engine's MAC address; in this example it is
// “00:11:1C::00:76:0A” The format “00-11-1C-00-76-0A also works.

lDevice.Connect("00:11:1C:00:76:0A", aAccessType );
//*************************************************************************//
// Step 3: Set up the stream packet size. Only the device controller can do
// this. If you skip this step, the current value of GevSCPPacketSize on
// the device is used.
//*************************************************************************//
// Option A: Perform auto negotiation to find largest packet size that can be
// used on the network link. At the completion of this call, GevSCPPacketSize
// is set to the largest possible packet size; if NegotiatePacketSize() fails,
// it is set to 1476 lDevice.NegotiatePacketSize( 0, 1476 );

// Option B: Set a value that you know works well. In the case of multicasting
// or remote unicast, this is good practice, as auto negotiation does not test
// all stream receivers to determine the largest packet size that can be used
 lDevice.GetGenParameters()->SetIntegerValue( "GevSCPSPacketSize",1476 );

//*************************************************************************//
// Step 4: Set up proper streaming networking topology
//*************************************************************************//

// Scenario 4: Multicast master -- this application controls the device.
lDevice.SetStreamDestination( "239.192.1.1" , 1042 );
// In case this application also receives image data it opens the stream
// otherwise it does not need to create a stream object
lStream.Open(lDeviceInfo ->GetIPAddress(), "239.192.1.1" , 1042)
```

Sample Code

- To view sample code for **PvMulticastMaster**, view the **PvMulticastMasterSample** file. For information about accessing the sample code, see "Accessing the Sample Applications" on page 15.

On streaming (slave) PCs

1. Open **PvStream** to listen for devices on the same multicast groups IP and port.
2. Create and start a **PvPipeline**.
3. Retrieve **PvBuffer** pointers from **PvPipeline**, and re-queue the pointer when done.

   It is important to do re-queue the pointer, otherwise there will not be enough buffers in the PvPipeline.
4. Disconnect from the GigE Vision device using **PvDevice::Disconnect.**

Code Example

C++

```
//*************************************************************************//
// Step 1: Set the Communication parameters before connecting
//
//*************************************************************************//
// For an example for setting DefaultHeartTimeout, see the
// "Access Communication Settings" section
//*************************************************************************//
// Step 2: Connect to device
//*************************************************************************//
PvDevice lDevice;
// method 1: Connect using the device info lDeviceInfo obtained in the
// "Detect and Select Device" section
lDevice.Connect(lDeviceInfo, aAccessType );
// method 2: Connect using the IP Engine's IP address; in this example it is
// "192.168.1.128"
lDevice.Connect("192.168.1.128", aAccessType );
// method 3: Connect using IP Engine's MAC address; in this example it is
// “00:11:1C::00:76:0A” The format “00-11-1C-00-76-0A also works.

lDevice.Connect("00:11:1C:00:76:0A", aAccessType );
//*************************************************************************//
// Step 3: Set up the stream packet size. Only the device controller can do
// this. If you skip this step, the current value of GevSCPPacketSize on
// the device is used.

//*************************************************************************//
// Option A: Perform auto negotiation to find largest packet size that can be
// used on the network link. At the completion of this call, GevSCPPacketSize
// is set to the largest possible packet size; if NegotiatePacketSize() fails,
// it is set to 1476 lDevice.NegotiatePacketSize( 0, 1476 );

// Option B: Set a value that you know works well. In the case of multicasting
// or remote unicast, this is good practice, as auto negotiation does not test
// all stream receivers to determine the largest packet size that can be used
 lDevice.GetGenParameters()->SetIntegerValue( "GevSCPSPacketSize",1476 );

//*************************************************************************//
// Step 4: Set up proper streaming networking topology
//*************************************************************************//

// Scenario 5: Multicast slave -- this application does not control the device
// and only receives image data. This application does not need to connect to
// the device, but still needs to know the device's IP Address
lStream.Open(lDeviceInfo ->GetIPAddress(), "239.192.1.1" , 1042)
```

Sample Code

- To view sample code for **PvMulticastSlave**, view the **PvMulticastSlaveSample** file. For information about accessing the sample code, see "Accessing the Sample Applications" on page 15.

Using a GigE Vision Transmitter to Send GigE Vision Data

The eBUS SDK provides the capability to send GigE Vision data to GigE Vision receivers, using the Video Server API. The Video Server API is a component of the eBUS SDK that transmits video over the network from a computer to one or more alternate destinations, in a GigE Vision® compliant manner.

The Video Server API does not currently support the GigE Vision packet resend capability.

To configure a GigE Vision transmitter to send GigE Vision data

1. Create a **PvBuffer** pool.

   Note: You can use the eBUS SDK to allocate the memory for a buffer, or you can attach buffers to an existing memory pool (allocated by another method). If you are using an existing memory pool, you can use **PvBuffer::Attach**(). For more information, see **PvBuffer::Attach**() in the *eBUS SDK API Help File*.
2. Optionally, initialize the transmitter with the **PvBuffer** pool just created by calling **PvTransmitterRaw::LoadBufferPool**().
3. Create an instance of **PvTransmitterRaw**.
4. Bind the **PvBuffer** pool to the **PvTransmitter**.

   Note: LoadBufferPool() is an optional method that provides the transmitter with a set of buffers that can be retrieved immediately by calling **RetrieveFreeBuffer**().
5. Optionally create a **PvVirtualDevice** instance so the application can respond to discovery requests by the receivers.
   - Call **PvVirtualDevice::StartListening**() to listen on a specific network interface.

     Note: Only one **PvVirtualDevice**() instance can exist on a NIC, however multiple **PvTransmitterRaw**() instances can exist on a NIC. The number of **PvTransmitterRaw**() instances is based on the amount of network bandwidth that is available.
6. Use **PvTransmitterRaw::Open**() to open a connection to the destination GigE Vision receiver.
7. Call **PvTransmitter::ResetStats**() to reset the transmitter statistics.
8. Call **PvTransmitter::SetMaxPayloadThroughput**() to set the desired effective throughput.

9. In a loop:
   - Call **PvTransmitterRaw::RetrieveBuffer**() to retrieve a free buffer from the list of available buffers.
   - Copy the data to the newly retrieved buffer.
   - Call **PvTransmitterRaw::QueueBuffer**() to queue the buffer to the transmitter.

     Note: PvTransmitterRaw::QueueBuffer() is an asynchronous method call (with its own thread) that can potentially return before the PvBuffer is sent out of the system.

Do not access the PvBuffer after it is queued to the transmitter (because the SDK no longer owns the buffer). Doing so may result in corrupt data.

If you are not using the buffer pool, you can use **PvBuffer.GetOperationalResult**() to determine if the buffer was sent out properly.

10. To gracefully stop transmission.
    - Call **PvTransmitterRaw::AbortQueuedBuffer**() to abort transmission of currently queued PvBuffers.
    - In a loop call **PvTransmitterRaw::RetrieveFreeBuffer** and delete all reclaimed PvBuffers.
    - Call **PvVirtualDevice::StopListening**() to stop the device from listening.
    - Call **PvTransmitter::Close**() to close the transmitter instance.

Code Example

C++

```
// Allocate transmit buffers
    PvBufferList lBuffers;
    PvBufferList lFreeBuffers;
    for ( PvUInt32 i = 0; i < lConfig.GetBufferCount(); i++ )
    {
        // Alloc new buffer
        PvBuffer *lBuffer = new PvBuffer();
        lBuffer->GetImage()->Alloc( lWidth, lHeight, lPixelFormat );

        // Set to 0
        memset( lBuffer->GetDataPointer(), 0x00, lSize );

        // Add to both buffer list and free buffer list
        lBuffers.push_back( lBuffer );
        lFreeBuffers.push_back( lBuffer );
    }

    // Create transmitter, set packet size
    PvTransmitterRaw lTransmitter;
    lTransmitter.SetPacketSize( lConfig.GetPacketSize() );
```

Continued on next page...

```
// Create virtual device (used for discovery)
   PvVirtualDevice lDevice;
   lDevice.StartListening( lConfig.GetSourceAddress() );

   cout << "Listening for device discovery requests on " << lConfig.GetSourceAddress() << endl;

   // Open transmitter - sets destination and source
   PvResult lResult = lTransmitter.Open(
       lConfig.GetDestinationAddress(), lConfig.GetDestinationPort(),
       lConfig.GetSourceAddress(), lConfig.GetSourcePort() );
   if ( !lResult.IsOK() )
   {
       cout << "Failed to open a connection to the transmitter." << endl;
       return 1;
   }

   // Used to transmit at a steady frame rate
   PvFPSStabilizer lStabilizer;

   // Acquisition/transmission loop
   while(IsTransmitting)
   {

       // Step 1: Copy image data to a free buffer
       // Are there buffers available for transmission?
       if ( lFreeBuffers.size() > 0 )
       {
          // Retrieve buffer from list
          PvBuffer *lBuffer = lFreeBuffers.front();
          lFreeBuffers.pop_front();

          // Copy the image to the free buffer, user defined CopyToPvBuffer method
          CopyToPvBuffer( lBuffer );

          // Queue the buffer for transmission
          lTransmitter.QueueBuffer( lBuffer );
        }

       // Step 2: Retrieve free buffer(s) and add to free buffer pool
       PvBuffer *lBuffer = NULL;
       while ( lTransmitter.RetrieveFreeBuffer( &lBuffer, 0 ).IsOK() )
       {
           // Queue buffers back in available buffer list
           lFreeBuffers.push_back( lBuffer );
       }
   }

   // Close transmitter (will also abort buffers)
   lTransmitter.Close();
```

Advanced SDK Functionality

The following section provides you with steps to configure advanced SDK functionality.

Using the PvAcquisitionStateManager to Control the Image Stream and Lock the GenICam Node Map

The PvAcquisitionStateManager class controls the starting and stopping of the stream, and locks the GenICam node map features while video is streaming. For detailed information about how the PvAcquisitionStateManager class works with the AcquisitionStart, AcquisitionStop, and TLParamsLocked GenICam features, see the *eBUS SDK API Help File*.

To use the PvAcquisitionStateManager to control the image stream and lock the GenICam node map

1. The container class declaration (in this case PvSimpleUISampleDlg) needs to inherit from PvAcquisitionStateEventSink.
2. The container class implementation (in the code example we are using PvSimpleUISampleDlg):
   - Creates the acquisition state manager.
   - Registers the callback.
   - Implements the callback.
   - Acts upon the state change.

Code Example

C++

```
class PvSimpleUISampleDlg : public CDialog, PvGenEventSink, PvAcquisitionStateEventSink
{
:
:
    // PvAcquisitionStateEventSink implementation
    void OnAcquisitionStateChanged( PvDevice* aDevice, PvStreamBase* aStream, PvUInt32
    aSource, PvAcquisitionState aState );
:
:
    PvAcquisitionStateManager *mAcquisitionStateManager;
:
:
}
```

```
    // Create acquisition state manager
    mAcquisitionStateManager = new PvAcquisitionStateManager( &mDevice, &mStream );
    // Register callback
    mAcquisitionStateManager->RegisterEventSink( this );

// Callback implementation
void PvSimpleUISampleDlg::OnAcquisitionStateChanged( PvDevice* aDevice, PvStreamBase* aStream,
    PvUInt32 aSource, PvAcquisitionState aState )
{
// Add some code to act upon the state change.
}
```

Using the PvFPSStabilizer Class to Specify the Frame Rate that is Displayed

The PvFPSStabilizer class allows you to display video at a desired frame rate, even though the frame rate of the video received by the SDK may be faster, or be subject to jitter. This class is not part of the core of the eBUS SDK. It is provided as a utility class shared by some Pleora samples.

To use this class, instantiate an object and call the **IsTimeToDisplay**() method each time a new buffer is received and specify the desired frame rate.

This class does not directly influence the behavior of the display. It assumes that a frame was displayed each time **IsTimeToDisplay**() returns **true**. **IsTimeToDisplay**() returns **true** when — dependent on the specified frame rate target and the previous times **IsTimeToDisplay**() returned **true** — it is time to display a frame.

To use the PvFPSStablizer class to specify the frame rate that is displayed

1. Start acquisition.
2. Reset the history using the **Reset**() method of the **PvFPSStablizer** class.
3. When a new buffer is received, call the **IsTimeToDisplay**() method to determine if **PvDisplayWnd::Display** should be called.

   The **IsTimeToDisplay**() method is used to determine if displaying a frame now would bring the stream closer to the desired display frame rate (the **aTargetFPS** property).

   If the **IsTimeToDisplay**() method returns **true**, call **PvDisplayWnd::Display**.

Code Example

C++

```
if ( mStabilizer.IsTimeToDisplay( mTargetFPS ) )
    mDisplayWnd->Display( *aBuffer, mVSync );
```

Persisting Configuration Settings

PvConfigurationWriter and **PvConfigurationReader** let you save and load settings and state information. You can save:

- The state of the GigE Vision device and stream states, for example, all parameters.
- Your own custom strings.

To save your state information

1. Create a configuration writer, using **PvConfigurationWriter** constructor.
2. Store the state information in the configuration writer.
   - For GigE Vision device settings (**PvDevice** objects), using **Store**(**PvDevice**,**PvString**).
   - For stream settings (**PvStream** objects), using **Store**(**PvStream**,**PvString**).
3. Optionally, store your own state information, using **Store**(**PvString**,**PvString**).
4. Save the stored information to disk, using **Save**().

To load your GigE Vision device settings (PvDevice object) or stream receiver (PvStream object) state from disk

1. Create a configuration reader using **PvConfigurationReader**.
2. Load the file from disk, using **Load**().
3. Retrieve all devices/streams or access them by name, that is:
   - Get the number of configurations by using the following methods:
     - **GetDeviceCount**()
     - **GetStreamCount**()
     - **GetStringCount**()

You can either retrieve all devices/streams or access them by name, if you know which name to look for. It depends what the application needs.

   - Get the name of a configuration by using the following methods:
     - **GetDeviceName**()
     - **GetStreamName**()
     - **GetStringName**()
4. Apply a configuration to an object by using the following methods:
   - **Restore(PvUInt32,PvDevice*)**
   - **Restore(PvUInt32,PvStream &)**
   - **Restore(PvUInt32,PvString &)**

Code Example

C++

```
PvDevice lDevice;
PvStream lStream;
// Connect and detect to device and stream
…
//*****************************************************************************//
//   Save all settings into the persistent file (XML format) on PC             //
//*****************************************************************************//
PvConfigurationWriter lWriter;
lWriter.Store( &lDevice, "MyDevice" );
lWriter.Store( &lStream, "MyStream" );
lWriter.Store( "Some custom string content", "MyString" );
// Optional save a custom string
lWriter.Save( "MyConfigFile.pvcfg" );

//*****************************************************************************//
//       Load the persistent file and apply the settings                       //
//*****************************************************************************//

PvConfigurationReader lReader;
lReader.Load( "MyConfigFile.pvcfg" );
lReader.Restore( "MyDevice", &lDevice );
lReader.Restore( "MyStream", lStream );
// Load custom string from configuration file
PvString lString;
lReader.Restore( lString, "MyString" );
```

- Communication settings are saved and loaded at the same time the device's settings are saved and loaded.
- Devices are identified by their MAC addresses in the device and stream configuration containers. Their IP addresses are not saved.
- If the **PvDevice** object used in the restore operation is already connected or opened, only the device control settings are applied (the SDK does not disconnect and reconnect to the original device). If the PvDevice object is not connected, the configuration reader object tries to find the original device on the network and connects to it before restoring the device's control settings.
- If the PvStream object used in the restore operation is already opened, only the stream's control parameters are applied. If the **PvStream** object is not opened, the configuration reader object tries to open the **PvDevice** object before restoring the stream's control parameters.
- It is possible to save more than one device, stream and/or string in a single configuration file. Individual device/stream/string instances can either be referred to by index or name.
- As the container supports more than one device/stream/string, they can be retrieved based on their position in the container or they can be named.

Sample Code

- To view sample code for **PvConfigurationReader**, view the **PvConfigurationReaderSample** file located at **C:\Program Files (x86)\Pleora Technologies Inc\eBUS SDK\Samples**.

Link Status and Device Recovery

The following section describes how the eBUS SDK manages its connection with devices.

eBUS SDK Connection Lost and Recovery Mechanism

The eBUS SDK keeps the link with the device alive by sending heartbeat packets. When the device does not receive a heartbeat, the connection is declared lost and the recovery process starts.

The eBUS SDK provides a very flexible recovery mechanism. It is based on callbacks to notify the application of connection status changes so that the application can take appropriate action.

Device Recovery

Device Callbacks are achieved by implementing methods from the **PvDeviceEventSink** class. After connecting a PvDevice, use the **RegisterEventSink**() method to register the callback **PvDeviceEventSink** instance.

Table 6: Device Recovery Callbacks

Notification	Description
OnLinkDisconnected	Notification that the connection to the device has been lost. This happens when the heartbeat thread of PvDevice fails to read the CCP register of the device. This notification is sent to the application even if automatic device recovery is disabled.
OnLinkReconnected	Notification that the connection to the device has been automatically restored. The notification states that device control was regained; The notification does not specify whether the cable was unplugged, or that the device was power cycled, for example. This notification is only received if the connection was restored. Connection is only automatically restored if device recovery is enabled.

To enable automatic device recovery, set the **LinkRecoveryEnabled** GenICam Boolean of **PvDevice::GetGenLink**() to **True**. It is set to **False** by default.

If device recovery is enabled, you receive the following notifications when a device is disconnected, and then reconnected:

- **OnLinkDisconnected**. The device connection has been lost.
- **OnLinkReconnected**. The device has been reconnected.

If recovery is disabled and the connection to the device is lost, you receive only an **OnLinkDisconnected** event.

PC to Device Communication Link Settings

Some communication settings, described in the following table, are available in order to modify the behavior of the communication link.

Table 7: Communication Settings

Communication setting	Description
HeartbeatTimeout	The application, for example GEVPlayer, sends periodic heartbeats to the GigE Vision device. The length of time between heartbeats is based on the HeartbeatInterval value set. If the device does not send an acknowledgement before the heartbeat expires, then the link between the application and the device is deemed to be disconnected.
HeartbeatInterval	Time in milliseconds between each heartbeat sent to the GigE Vision device.
MCTT	The amount of time (in milliseconds) that the GigE Vision device must wait before either timing out or receiving acknowledgment for a message (interrupt).
MCRC	The number of retransmissions allowed when the GigE Vision device times out waiting for acknowledgement of a message (interrupt).
AnswerTimeout	The amount of time the GigE Vision Device can take to respond to a command from the application.
CommandRetryCount	The number of times a command is attempted before it is considered to be a failed command.
LinkRecoveryEnabled	Used to attempt to automatically reconnect to a device when the connection is lost.

GEVPlayer's **Communication Control** window is a good place to get familiar with all the features and their descriptions.

The communication parameters are accessed through **PvDevice::GetGenLink**() (not **PvDevice::GetGenParameters**(), which is for device control parameter access).

The following communication parameters are available either before or after the device connects.

Table 8: Communication Parameters

Communication parameters	Available	Description
DefaultHeartbeatTimeout	At connection	At connection, the **DefaultHeartbeatTimeout** value is used to set the devices **GevHeartbeatTimeout**.
GevHeartbeatTimeout	After connection	If you would like to change the heartbeat timeout after connecting to the device, change the **GevHeartbeatTimeout**.
DefaultMCTT	At connection	At connection, the **DefaultMCTT** value is used to set the device's **GevMCTT**.
GevMCTT	After connection	If you would like to change the messaging timeout value after connecting to the device, change the **GevMCTT**.

Table 8: Communication Parameters

Communication parameters	Available	Description
DefaultMCRC	At connection	At connection, the **DefaultMCRC** value is used to set the device's **GevMCRC**.
GevMCRC	After connection	If you would like to change the count retransmissions after connecting to the device, change the **GevMCRC**.

AutoNegotiation and **DefaultPacketSize** are not automatically used at connection time. The application needs to call **PvDevice::NegotiatePacketSize**() to let the eBUS SDK negotiate the maximum packet size or set the device feature **GevSCPSPacketSize** as the **DefaultPacketSize**.

The **AnswerTimeout**, **CommandRetryCount**, **HeartbeatInterval** and **LinkRecoveryEnabled** parameters can be changed before and after connecting to the device.

You may also override the command and messaging ports as shown below.

Table 9: Command and Messaging Ports — Before and After Connection

Before connection	After connection
When **ForcedCommandPortEnabled** is set to **True**, *ForcedCommandPort* can be set	**CommandPort** is set to the value of **ForcedCommandPort**
When ForcedMessagingPortEnabled is set to **True**, **ForcedMessagingPort** can be set	MessagingPort is set to the value of **ForcedMessagingPort**

Event Sinks

PvSystemEventSink: Auto-find Controller

PvSystemEventSink contains a callback that lets you define what happens when a **PvSystem** object finds a GigE Vision device. The actual behavior is for you to define, but you could use this class to:

- Automatically filter GigE Vision devices that you (programmatically) determine are inappropriate.
- Begin displaying available GigE Vision devices before the search timeout expires, in the same way that PvDeviceFinderWnd does, for example.

To comply with the auto-find controller (PvSystemEventSink object) interface

1. Create a class (**MySink**) that inherits from **PvSystemEventSink** (you can also extend an existing class).
2. Declare the **OnDeviceFound**() method.
3. Define the code for the **OnDeviceFound**() method.

To use the auto-find controller (PvSystemEventSink object)

1. Create an instance of your sink class using **MySink** (for example).
2. Find GigE Vision devices as you normally would (see **PvSystem** in the *eBUS SDK API Help File*), but before using **PvSystem::Find**(), register your **MySink** class using **PvSystem::RegisterEventSink**().
3. Once you're done finding GigE Vision devices, unregister the event sink, using **PvSystem::UnregisterEventSink**().

Code Example

C++

```
class MySink: public PvSystemEventSink
{
    virtual void MySink::OnDeviceFound(
        PvInterface *aInterface, PvDeviceInfo *aDeviceInfo,
        bool &aIgnore );
};


void MySink::OnDeviceFound(
        PvInterface *aInterface, PvDeviceInfo *lDeviceInfo,
        bool &aIgnore )
 {
        if( lDeviceInfo != NULL )
        {
            //Handle the device found accordingly
            //…
        }
}


int deviceFind()
{
    MySink mySink;
    PvSystem aSystem;
    aSystem.RegisterEventSink(&mySink);
    PvResult lResult;
    PvDeviceInfo *lDeviceInfo = 0;

    // Find all GigE Vision devices on the network.
    aSystem.SetDetectionTimeout( 100 );
    lResult = aSystem.Find();
    if( !lResult.IsOK() )
    {
        printf( "PvSystem::Find Error: %s", lResult.GetCodeString() );
        return -1;
    }
    return 0;
}
```

Parameter callbacks - PvGenEventSink

To create a callback that runs when the feature's value changes:

1. Get the feature through the following steps:
   - Use **PvGenParameterArray::Get**() to get specific feature object.
   - Get the feature's type using **PvGenParameter::GetType**().
   - Optionally:
     - Get/Set the feature's value. Use the **GetValue/SetValue** method for the feature's type (for example, **PvGenInteger::GetValue**(), **PvGenFloat::SetValue**(), etc.)
     - If the feature is a command, activate it using **PvGenCommand::Execute**().
2. Create a subclass of **PvGenEventSink**.
3. In your new class, override **PvGenEventSink::OnParameterUpdate**().

   To use PvGenEventSink, PvGenParameter instances must register to the event sink using **PvGenParameter::RegisterEventSink**().

Sample Code

- To view sample code for **PvGenEventSink,** view the **GEVPlayerSample** file located at **C:\Program Files (x86)\Pleora Technologies Inc\eBUS SDK\Samples**.

Force GigE Device IP address

Use **PvDevice::SetIPConfiguration**() method when you wish to change the GigE Vision device's IP address. This is especially necessary when the device and the NIC are not on the same subnet.

As **PvDevice::SetIPConfiguration**() method is performed through a broadcast command, it works on devices that are not reachable because of their IP configuration, but are on the same physical network.

The gateway parameter is optional and set by default to "0.0.0.0".

You do not need to detect and select the device to set its IP address.

Code Example

C++

```
// Change the IP configuration of a device
PvDevice::SetIPConfiguration(
     "00-11-22-33-44-55", // Device MAC address
     "192.168.138.62", // New IP address of the device
     "255.255.255.0", // New subnet mask of the device
     "192.168.138.1" ); // New gateway of the device
```

Serial Device Control

The GigE Vision device is connected to a camera head and may also be connected to other peripherals.

The eBUS SDK provides the following API classes which allow you to perform serial communication with the camera head or other peripherals.

- **PvSerialPortIPEngine**
- **PvIPEngineI2CBus** for an I2C addressable peripheral
- **PvSerialBridge**

PvSerialPortIPEngine

Provides access to the serial ports on a GigE Vision device.

PvSerialPortIPEngine allows you to perform serial communications through Pleora's GigE Vision devices. **PvSerialPortIPEngine** is not supported by non-Pleora devices.

In some cases, use of the **PvSerialPortIPEngine** class to directly access the camera head interferes with the synchronization between the GigE Vision device and the camera head. As an example, the SB-Pro IP engine uses the serial port to communicate with the camera head of a Sony Block camera. If you use **PvSerialPortIPEngine** at the same time, you may interfere with the communication exchanges.

To use a serial port

1. Connect to a device using any of the **PvDevice::Connect**() methods.
2. Open a serial port to the device using **PvSerialPortIPEngine::Open**().
3. Write to the serial port using **PvSerialPortIPEngine::Write**().
4. Read from the serial port using **PvSerialPortIPEngine::Read**().
5. Finally close the serial port using **PvSerialPortIPEngine::Close**().

To check if a GigE Vision device supports a specific serial port

1. Connect to a device using any of the **PvDevice::Connect** methods.
2. Use the **PvSerialPortIPEngine::IsSupported**() static method.

Code Example

C++

```
// Get device parameters need to control streaming
PvGenParameterArray *lParams = lDevice.GetGenParameters();

// Configure serial port - this is done directly on the device GenICam interface,
    not on the serial port object! We use Uart0, 9600 N81
lParams->SetEnumValue( "Uart0BaudRate", "Baud9600" );
lParams->SetEnumValue( "Uart0NumOfStopBits", "One" );
lParams->SetEnumValue( "Uart0Parity", "None" );

// For this test to work without attached serial hardware we enable the port
    loopback
lParams->SetBooleanValue( "Uart0Loopback", true );


// Open serial port
PvSerialPortIPEngine lPort;
PvResult lResult = lPort.Open( &lDevice, PvIPEngineSerial0 );
if ( !lResult.IsOK() )
{
        printf( "Unable to open serial port on device: %s %s\n",
            lResult.GetCodeString().GetAscii(), lResult.GetDescription().GetAscii() );
        return false;
}
printf( "Serial port opened\n" );

// Make sure the PvSerialPortIPEngine receive queue is big enough
lPort.SetRxBufferSize( 2 << TEST_COUNT );

PvUInt32 lSize = 1;
for ( int lCount = 0; lCount < TEST_COUNT; lCount++ )
{
        // Allocate test buffers
        unsigned char *lInBuffer = new unsigned char[ lSize ];
        unsigned char *lOutBuffer = new unsigned char[ lSize ];

        // Fill input buffer with random data
        for ( PvUInt32 i = 0; i < lSize; i++ )
        {
            lInBuffer[ i ] = ::rand() % 256;
        }

        // Send the buffer content on the serial port
        PvUInt32 lBytesWritten = 0;
        lResult = lPort.Write( lInBuffer, lSize, lBytesWritten );
        if ( !lResult.IsOK() )
        {
            // Unable to send data over serial port!
            printf( "Error sending data over the serial port: %s %s\n",
                lResult.GetCodeString().GetAscii(),
                lResult.GetDescription().GetAscii() );
               return false;
        }

        printf( "Sent %i bytes through the serial port\n", lBytesWritten );
```

Continued on next page...

```
        // Wait until we have all the bytes or we timeout. The Read method only
        // times out if not data is available when the function call occurs,
        // otherwise it returns all the currently available data. It is possible
        // we have to call Read multiple times to retrieve all the data if all the
        // expected data hasn't been received yet.


        // Your own code driving a serial protocol should check for a message
        // being complete, whether it is on some sort of EOF or length. You should
        // pump out data until you have what you are waiting for or reach some
        //timeout.
        PvUInt32 lTotalBytesRead = 0;
        while ( lTotalBytesRead < lSize )
        {
            PvUInt32 lBytesRead = 0;
            lResult = lPort.Read( lOutBuffer + lTotalBytesRead, lSize - lTotalBytesRead,
            lBytesRead, 100 );
            if ( lResult.GetCode() == PvResult::Code::TIMEOUT )
            {
                printf( "Timeout\n" );
                break;
            }

            // Increment read head
            lTotalBytesRead += lBytesRead;
            }

            // Validate answer
            if ( lTotalBytesRead != lBytesWritten )
            {
                // Did not receive all expected bytes
                printf( "Only received %i out of %i bytes\n", lTotalBytesRead,
                lBytesWritten );
            }
            else
            {
                // Compare input and output buffers
                PvUInt32 lErrorCount = 0;
                for ( PvUInt32 i = 0; i < lBytesWritten; i++ )
                {
                if ( lInBuffer[ i ] != lOutBuffer[ i ] )
                {
                lErrorCount++;
                }
            }

                // Display error count
               printf( "Error count: %i out of %i (%.3f%%)\n", lErrorCount, lBytesWritten,
            (double)lErrorCount / (double)lBytesWritten * 100.0 );
        }

        // Free test buffers
        delete []lInBuffer;
        delete []lOutBuffer;

        // Grow test case
        lSize *= 2;
}

// Close serial port
lPort.Close();
printf( "Serial port closed\n" );
```

PvIPEngineI2CBus

Some Pleora cameras, sensors, or peripherals can be controlled using I2C. The **PvIPEngineI2CBus** class is not supported by non-Pleora devices.

In this case, the **PvIPEngineI2CBus** lets you send commands to your camera and receive the camera's replies.

I2C uses the concept of a master and slave, where the master controls the clock and initiates reads and writes. For **PvIPEngineI2CBus**, the GigE Vision device is the I2C master. The I2C slave is typically a camera, but could be any I2C-capable device connected to the GigE Vision device.

The **PvIPEngineI2CBus** methods manage the GigE Vision device and camera communication required to send a message to the camera and to read back replies. For example, the **BurstWrite**() method involves the following transaction:

1. Your application calls **BurstWrite**().
2. The eBUS SDK passes the entire message to the GigE Vision device using Ethernet.
3. **BurstWrite**() blocks.
4. GigE Vision device to camera: [Start condition] aSlaveAddress [Write - SDA low]
5. Camera to GigE Vision device: Acknowledge
6. GigE Vision device to camera: [First 8 bits of aBuffer]
7. Camera to GigE Vision device: Acknowledge
8. GigE Vision device to camera: [Next 8 bits of aBuffer]
9. Camera to GigE Vision device: Acknowledge

   ...
10. GigE Vision device to camera: [Last 8 bits of aBuffer]
11. Camera to GigE Vision device: Acknowledge
12. GigE Vision device to camera: [Stop condition]
13. GigE Vision device reports the successful transmission of the message to the PC (via Ethernet).
14. **BurstWrite**() method unblocks (returns).

To send I2C-protocol serial commands to your camera

1. Create an I2C serial controller using **PvIPEngineI2CBus.**
2. Send a command to the camera using **BurstWrite**().
3. Test to ensure the message sent succeeded by checking the **PvResult** returned by **BurstWrite**().
4. Retrieve the camera's reply using **BurstRead**().
5. Test to ensure the message retrieval succeeded by checking the **PvResult** returned by **BurstRead**().
6. Test the content of the message from the camera by using **aBuffer** and **aBufferSize** parameters and your own code.

Sample Code

- To view sample code for **PvI2CSample**, view the **PvI2CSample** file located at **C:\Program Files (x86)\Pleora Technologies Inc\eBUS SDK\Samples**.

PvSerialBridge

Bridges a computer serial port or Camera Link DLL to an IP engine serial port.

This class is only available for the Windows operating system.

PvSerialBridge is bidirectional, it can either:

- Take data from a computer's serial port or eBUS SDK Camera Link DLL (clserpte.dll) and send it to an IP engine serial port.
- Or, take data from an IP engine's serial port and send it a computer serial port or eBUS SDK Camera Link DLL.

Depending on the Start method that is used, the bridge is either started as a serial bridge or Camera Link DLL bridge.

Statistics can be retrieved about a bridge: count of transmitted and received bytes.

The serial port can be used with a stand-alone application, usually a camera configuration application relying on a serial port (ie: COM1, COM2, etc.) to interact with the camera. The port selected by the application must be connected with a NULL modem to the port used by the bridge. It can be two real serial ports connected together or two virtual ports with a serial port emulation software/driver.

The Camera Link DLL can be used with a stand-alone application, usually a camera configuration application, relying on the API described in the Camera Link standard used to interface a serial port. Instead of being the serial port of a frame grabber here it is the serial port of a Pleora IP engine.

For more information, see the *Establishing a Serial Bridge Application Note*, available at the Pleora Technologies Support Center.

Image Saving

PvBufferWriter lets you save image buffers (PvBuffer objects) to the disk. The image buffer can be saved as a Windows Bitmap file (RGB 32) or in raw format, regardless of the original image pixel type.

The raw format has no header (width, height, pixel format, etc.) it just contains the raw image data as in the PvBuffer/PvImage.

To save an image buffer to disk

1. Acquire an image.

   For more information, see "To use PvStream to receive images" on page 33.
2. Create an instance of **PvBufferWriter::PvBufferWriter**().
3. Save the image to disk using **Store**().

Code Example

C++

```
PvBufferWriter lWriter;
// Save as WindowsBitmap
lWriter.Store( lBuffer, "image.bmp", PvBufferFormatBMP );

// Save as raw data
lWriter.Store( lBuffer, "image.raw", PvBufferFormatRaw );
```

Sample Code

- To view sample code for **PvBufferWriterSample**, view the **PvBufferWriterSample** file located at **C:\Program Files (x86)\Pleora Technologies Inc\eBUS SDK\Samples**.

Pixel Format Conversion

The **PvBufferConverter** class converts an image buffer with one pixel type to another pixel type. With the eBUS SDK, you simply use a PvBuffer- Converter to which you pass an input and output buffer. This converts the pixel type from one of the defined GigE Vision pixel types to RGB. It automatically converts the content of the input buffer to the output buffer, using the pixel type of the output buffer. You don't have to select a specific pixel converter; it is automatically managed for you.

The pixel converter does not perform tap reconstruction, nor Bayer pattern offset adjustment. Deinterlacing is managed by the **PvFilterDeinterlace** class. Not all pixel type conversion are supported, use **IsConversionSupported()**.

To change an image's format

1. Acquire an image.

   For more information, see "To use PvStream to receive images" on page 33.
2. Create an image-format converter (PvBufferConverter object) using **PvBufferConverter::PvBufferConverter**().
3. Test to see if you can convert your image to your preferred pixel type using **IsConversionSupported**().

   For your source pixel type, use **PvBuffer::GetPixelType**(); for your destination, select an enumeration from **PvPixelType.h**.
4. Create a destination buffer.

   For more information, see the **PvBuffer Class Reference** topic in the *eBUS SDK API Help File*. (you define the destination image's pixel type during this step).
5. Optionally, configure options such as Bayer filtering, using **SetBayerFilter**().
6. Convert the image to the new pixel type using **Convert**().

Code Example

C++

```
//*****************************************************************************//
//    Create converter object and a destination buffer of RGB32bit pixel format
//
//*****************************************************************************//
PvBufferConverter lConverter;

PvBuffer lOutput;
lOutput.Alloc( lSizeX, lSizeY, PvPixelWinRGB32 );

lConverter.Convert( lBuffer, &lOutput );
```

Sample Code

- To view sample code for PvRGBFilter, view the **PvRGBFilterSample** file located at **C:\Program Files (x86)\Pleora Technologies Inc\eBUS SDK\Samples**.

Window UI Components

The following list includes predefined display UI component classes that can be used to display and interactively control the device's attributes:

- **PvDeviceFinderWnd**: can be used to find all GigE Vision devices on a network and lets the user select one.
- **PvGenBrowserWnd**: can be used to monitor and control GenICam features.
- **PvDisplayWnd**: can be used to display the contents of an image.
- **PvTerminalIPEngineWnd**: can be used to directly communicate from the PC to a serial based component that is part of the GigE Vision device.
- **PvSerialBridgeManagerWnd**: A user interface for configuring PvSerialBridge instances of a PvDevice.

Display Images - PvDisplayWnd

The **PvDisplayWnd** class is used to display a PvBuffer.

To display images

1. Decide if you want the display in a standalone window or embedded as an element of a window you control.
2. Create a display control (**PvDisplayWnd** object).
3. Optionally set the window's position by using **SetPosition**().
4. Optionally, set the background color using **SetBackgroundColor**().
5. Make the display controller appear.
   - To embed the display in your own application (a window you control yourself), use **Create**().
     - In Windows, the second parameter of **PvDisplayWnd.Create**() must have a unique resource ID, i.e. it is not used by any other controls of the parent window.
     - In Linux, it is required to pass a **QWidget** pointer to the first parameter of **PvDisplayWnd.Create**(). You do not need any input for the second parameter.
   - To open the display in a modal window, use **ShowModal**().
   - To open the display in a modeless window, use **ShowModeless**().

6. Make an image appear in the display area using **Display**().
7. Optionally, update the image (show a movie by displaying many images sequentially, for example) using additional calls to **Display**().
8. Optionally move the window.
   - For standalone windows use **SetPosition**().
   - For embedded windows, move the parent window, then force the display control to update to its new location using **UpdateBounds**().
9. Close the window using **Close**().

- The eBUS SDK does not provide zoom or pan functionality programmatically. However, it is possible for the you to zoom and pan using your keyboard and mouse.
- It is not necessary to convert the buffer in order to display it. The **PvDisplayWnd** class takes in the raw buffer and converts it to the native video adapter pixel format to display. It is the most efficient way of displaying a buffer. The buffer conversion is performed by the display using the video adapter memory as the destination. If you use your own converter, you have to convert to a memory buffer and then have the display copy that buffer to the video adapter memory.
- If you need to apply a **PvFilterRGB** class to your image for display purposes, set the converter of the **PvDisplayWnd**' class o use the **PvFilterRGB** object you have configured. Then, the **PvDisplayWnd** class applies the filter automatically on every image being displayed.
- If the standalone display window is created by a Windows console application, call the **PvDisplayWnd::DoEvents**() method from the main thread of a console application in a loop.
- See the **GEVPlayerSample** and **PvSimpleUISample** files for complete, working examples of how to embed a display into your own dialog.
- In Linux the Qt library is used for eBUS SDK GUI functions including display. As of eBUS SDK 2.0, the Qt display does not yet support zooming and panning.

Code Example

C++

```
//*****************************************************************//
// Option 1: Display in the embedded window of application
//*****************************************************************//
lDisplayWnd.Create( GetSafeHwnd(), 10000 );
lDisplayWnd.SetPosition( 20, 20, 400, 300 );

lDisplayWnd.SetPosition( ;
lResult =lDisplayWnd.SetBackgroundColor( 0x00, 0x00, 0x20);
//
// ********************************************************************//
// Option 2: Display in a stand alone window //
// ********************************************************************* //
PvDisplayWnd lDisplay;
lDisplay.SetPosition( 0, 0, 640, 480 );
lDisplayWnd.SetBackgroundColor( 0x00, 0x00, 0x20);
lDisplayWnd.SetTitle("My Display");
lDisplay.ShowModeless();


// Setup the PvFilterRGB for PvDisplayWnd's converter (optional)
PvFilterRGB lFilterRGB;
PvBufferConverter &aConverter = lDisplayWnd.GetConverter();
aConverter.SetRGBFilter( lFilterRGB );

//  Display a retrieved image
lDisplayWnd.Display( *lBuffer );


//  To close the display window
lDisplayWnd.Close( );
```

Sample Code

- To view sample code for window UI components, see the **GEVPlayerSample** and the **PvSimpleUISample** files. For information about accessing the sample code, see "Accessing the Sample Applications" on page 15.

Chapter 7

System Level Information

This section provides you with important information you need to know when configuring your system to ensure efficient video data transmission.

The following topics are covered in this chapter:

Bandwidth Overhead Calculation

This section provides you with information to assist you with calculating Ethernet bandwidth requirements. To understand the overall bandwidth requirements, it is helpful to understand the contents of a single Ethernet Frame.

Single Ethernet Frame Size

The following table shows you sizes of the components contained in an Ethernet frame transferred on an Ethernet link or cable.

The open source network protocol analyzer software, Wireshark, is often used to capture Ethernet frame information for debugging purposes. Information captured by Wireshark is shown below in blue.

Table 10: Ethernet Frame Information

Preamble	Start-of-frame-delimiter	MAC destination	MAC source	Ethertype /length	Payload	CRC32
7 octets of 10101010	1 octet of 10101011	6 octets	6 octets	2 octets	46-1500/ 9000 octets	4 octets
		Minimum size: 64 bytes Maximum size, small packets: 1518 bytes Maximum size, jumbo packets: > 9018 bytes				
Minimum size: 72 bytes Maximum size, small packets: 1526 bytes Maximum size, jumbo packets: 9026 bytes						

The following table shows you the individual sizes of the GigE Vision Stream Protocol (GVSP) and Pleora protocol UDP components.

Table 11: UDP Information

Items	Size (bytes)
IP header	20
UDP header	8
Protocol header	8 for GigE Vision Stream Protocol (GVSP)
User data payload	Could be data of image, leader, trailer, or dummy padding
Ethernet payload padding	Could be non-zero to ensure the Payload size is equal to or larger than 46.

Based on the information shown in the tables above, we know that the Ethernet frame size is the size of the GVSP payload, plus 62 bytes.

- From the Wireshark frame information, shown in blue in the UDP Information table, the User Datagram part of the image frame contains the Protocol header + User data payload + Ethernet payload padding.
- In eBUS SDK, the GevSCPSPacketSize is the payload size including IP header, UDP header, GVSP header, and User data payload. For easy calculation, GevSCPSPacketSize = 36 bytes + User data payload.
- In GigE Vision protocol, for each image frame, there is always one leader and trailer packet. The last image data packet only contains image data and no padding. So the last image data packet could be smaller than the rest of the full size image data packets.

Calculating Ethernet Bandwidth

This section provides an example of how to calculate the Ethernet bandwidth required to transfer images that are 640 pixels x 480 pixels, in mono 8 format, at 200fps, along with the UDP overhead. In this example, the image data payload size is set to 1440 bytes per packet which corresponds to:

GevSCPSPacketSize =36 bytes +1440 bytes =1476 bytes per packet

One image frame size is 640x480 =307200 bytes. (307200 bytes / (1440 bytes per packet)) = (213 packets) x (1440 bytes per packet) + 480 bytes means that for each image frame there are 213 full image data packets and one partial image data packet. The Ethernet frame size is shown in the following table.

Table 12: Ethernet Frame Size

Ethernet frame size (bytes)	GigE Vision protocol
Leader	92
Payload packets	1502x213
Last payload packet	542
Trailer*	72
Total	320632

* There are 2 bytes of Ethernet Payload padding for the trailer.

Table 13: Data Rate and Bandwidth

	Bytes/s	MB/s	Bandwidth over head (%)
Image data rate	(307200 bytes / frame) x (200 frame / second) = 61440000 bytes / second	58.59	
Ethernet bandwidth required for GigE Vision protocol	(320632 bytes / frame) x (200 frame / second) = 64126400 bytes / second	61.16	4.39 **Note**: Bandwidth overhead decreases to approximately 0.5% when jumbo frames are used.

List of UDP Ports on the GigE Vision Device

The following table identifies ports available on Pleora GigE Vision devices.

Table 14: UDP Ports

Name in GEV protocol	Pleora device using GigE Vision protocol
GVCP	3956*
Message	4
Stream channel #0	20202. Note that this could be changed in future.
Stream channel #1	20203

* Port 3956 is for any GigE Vision device as define in the GigE Vision Specification.

Host-Side Port Control

When your GigE Vision compliant application, for example the Pleora sample application GEVPlayer, connects to a GigE Vision device using the method **PvDevice::Connect**(), or when the application opens a **PvStream**, the eBUS SDK uses, by default, any host-side UDP ports assigned by the operating system. When the application takes control of the GigE Vision device, there are three host-side UDP ports the application can control. They are the command link (used to send commands to the IP engine), messaging channel (used to send messages from the IP engine to the application), and stream destination UDP ports.

The command link and messaging channel host-side UDP port can be configured through the command link parameters.

The stream destination port can be configured by passing the port number as the aDataPort of PvStream::Open().

Chapter 8

Building and Distributing an eBUS SDK Application

To develop an application on a Windows-based operating system, please see Creating a new C/C++ project in the *eBUS SDK API Help File*.

To develop an application on Linux, please refer to the *eBUS Linux Software Guide*.

The environment variable $(PUREGEV_ROOT) can be used to reference the base directory location.

To distribute the application, refer to the *Custom Install Guide*.

eBUS Driver Installer API

This sample shows how to display or install eBUS drivers to the network adapters on a given computer from your own application (through the eBUS driver installation API) instead of using the Driver Installation Tool. The Driver Installation Tool is built around the same SDK. Some customers have been developing their own Driver Installation Tool.

Chapter 9

Appendix A — eBUS SDK Primitives and Classes

This chapter provides you with descriptions of eBUS SDK primitives and classes, and provides the following information in table format:

Table 15: eBUS SDK Primitives

Primitive	Description
PvUInt8	Unsigned integer, 8 bits
PvInt8	Signed integer, 8 bits
PvUInt16	Unsigned integer, 16 bits
PvInt16	Signed integer, 16 bits
PvUInt32	Unsigned integer, 32 bits
PvInt32	Signed integer, 32 bits
PvUInt64	Unsigned integer, 64 bits
PvInt64	Signed integer, 64 bits

Table 16: eBUS SDK Classes

Class	Description
PvBuffer	Used to hold data received from the device through a PvStream. Can manage its own internal memory buffer (Alloc/Free) or be attached to an external memory buffer (Attach/Detach).
PvBufferConverter	Used for image buffer conversion from any GigE Vision pixel type to a pixel type that can be displayed.
PvBufferWriter	Used to save image buffers to the hard drive. Supports saving raw data (as contained in the buffer) and can also save to RGB32 bitmaps. When saving to RGB32 bitmaps, on-the-fly pixel conversion is performed if required.
PvConfigurationReader	Used to read pvcfg configuration files and restore saved PvDevice, PvStream, PvString and raw PvGenParameterArray content. A pvcfg configuration file contains XML data.
PvConfigurationWriter	Used to write pvcfg configuration files holding complete PvDevice, PvStream, PvString and raw PvGenParameterArray state information. A pvcfg configuration file contains XML data.
PvDevice (Host-side representation of a GigE Vision device)	Used to control a GigE Vision device. Use the Connect/Disconnect methods to connect a PvDevice to your GigE Vision device. The GenICam interface of the device can be retrieved using the GetGenParameter method. It returns a PvGenParameter- Array pointer. This GenICam interface is built at runtime from an XML file that is downloaded automatically from the device. The host-side communication link parameters are also configured through a PvGenParameterArray that can be retrieved using the GetGenLink method. The register map of the device can be directly accessed using the ReadRegister, WriteRegister, ReadMemory and WriteMemory methods, although this is reserved for more advanced applications and requires knowledge of the register map specific to each GigE Vision camera or device. See **PvDeviceEventSink** (in the *eBUS SDK API Help File*) for information on how to receive raw messaging channel events from the device.
PvDeviceEventSink	Provides an event sink interface. Classes that need to receive event notifications should inherit PvDeviceEventSink, and should be registered through **PvDevice::RegisterEventSink**() in order to receive PvDevice notifications. Use these notifications to be informed of raw messaging channel events or unexpected disconnection events.
PvDeviceFinderWnd (GUI component)	Used to display a modal dialog displaying all GigE Vision devices on the network and prompting the user to select one. Displayed using ShowModal. Returns **OK** if the user selected a device and dismissed the dialog using the **OK** button. The selected device can be retrieved in the form a PvDeviceInfo pointer using GetSelectedDevice. Available on both Qt and MFC versions of PvGUI library.

Table 16: eBUS SDK Classes (Continued)

Class	Description
PvDeviceInfo	Contains information about a device detected on the network. Can be retrieved from either a PvInterface of PvDeviceFinderWnd. Can be used to connect a PvDevice to a GigE Vision device through the Connect method.
PvDisplayWnd (GUI component)	Used to display the contents of an image buffer. Typically used as a stand-alone modeless window or embedded in a dialog. Available in the PvGUI library as either a Qt-based (OpenGL accelerated) or MFC-based (DirectX accelerated) control.
PvFilter	Used for image processing.
PvFilterDeinterlace	Used to de-interlace image buffer content. Can either work on a buffer containing the Even and Odd fields or from a pair of buffers containing the Even and then Odd fields.
PvFilterRGB	Used to apply a RGB independent gain and offset to any RGB buffer,
PvGenBoolean (IBoolean GenICam parameter)	Can be true or false. Inherits from PvGenParameter. Displayed in the GenICam browser as a combo box having true and false as possible values.
PvGenBrowserWnd (GenICam browser GUI component)	Attach a PvGenParameterArray() using the SetGenParameterArray() method. Available on Qt and MFC PvGUI versions of the PvGUI library.
PvGenCommand (ICommand GenICam parameter)	Cannot be read or written, but can be triggered using the Execute() method. Inherits from PvGenParameter. Displayed in the GenICam browser as a button that can be activated.
PvGenEnum (IEnumeration GenICam parameter)	Holds a group of possible values each having unique string and integer values. Inherits from PvGenParameter. Displayed in the GenICam browser as a combo box containing all the possible enumeration entry strings.
PvGenEnumEntry (One possible value of a PvGenEnum)	Represented by both a unique string and integer value.
PvGenEventSink	Interface you inherit one of your classes from in order to receive GenICam parameter invalidation events. In order to receive these notifications you need to register your class using the RegisterEventSink() method of PvGenParameter().
PvGenFile (GenICam file model implementation)	Allows the reading/writing of a file to a GenICam node map (typically the device's) as long as it follows the standard GenICam model.
PvGenFloat (IFloat GenICam parameter)	Represents a floating point value. Inherits from PvGenParameter. Displayed in the GenICam browser as a text edit with a spin button.

Table 16: eBUS SDK Classes (Continued)

Class	Description
PvGenInteger (IInteger GenICam parameter)	Represents an integer value. Inherits from PvGenParameter. Displayed in the GenICam browser as a text edit with a spin button.
PvGenParameter (Base class for all PvGen specialized types)	Contains common properties like **GetName**() and **GetDescription**().
PvGenParameterArray (An eBUS SDK abstraction of a GenApi node map)	Contains a group of parameters that can be referred to by name or index. Provides helper methods that can be used to query and cast PvGen parameters in the right type and get/set values on PvGen parameters using a single line of code. Can be attached to a **PvGenTreeBrowserWnd** GUI object to allow dynamic interaction with the content of the GenApi node map. The main PvGenParameterArray objects in the eBUS SDK are: • **PvDevice::GetGenParameters**() for the main device GenICam interface, built from the GenICam XML downloaded from the device • **PvDevice::GetGenLink**() for the communication channel parameters (host-side) of a PvDevice, built from a static GenICam XML bundled in the PvDevice library • **PvStream::GetParameters**() for the stream parameters and statistics, built from a static GenICam XML bundled in the PvStream library
PvGenRegister (IRegister GenICam parameter)	Represents a contiguous section of a register map. While the parameter can be visible in the GenICam browser, it is not possible to interact with the content of a PvGenRegister.
PvGenString (IString GenICam parameter — represents a string.)	Displayed in the GenICam browser as a plain text edit control.
PvInterface (Component of device discovery)	Represents one network interface card on the system. PvSystem enumerates all network interface cards represented by PvInterface objects. PvInterface objects can in turn enumerate all GigE Vision devices accessible through their related network interface cards as PvDeviceInfo objects.
PvIPEngineI2CBus	Used to control an I2C bus attached to an GigE Vision device.

Table 16: eBUS SDK Classes (Continued)

Class	Description
PvPipeline	Provides buffer management and an acquisition thread on top of a PvStream object. Buffers are retrieved from the pipeline using RetrieveNextBuffer(). Once no longer needed (processing completed) the buffers need to be returned to the pipeline using ReleaseBuffer(). We recommend allocating the buffers to the device's PayloadSize (GenICam parameter) value when initiating streaming using the SetBufferSize() method. By default 16 buffers are used by PvPipeline. Using more buffers reduces the odds of dropping frames at the pipeline output but at the expense of using more memory and increasing potential latency. Use SetBufferCount() to change the number of buffers used by the pipeline. Even though PvPipeline can be used in most applications, it is recommended to use a PvStream directly if you are working at high frame rates or when you need to customize the pipeline behavior.
PvResult	Class representing function return values.
PvSerialPort (Base class for PvSerialPortIPEngine)	Provides a unified interface for serial port transmit and receive operations that does not change for different serial port implementations.
PvSerialPortIPEngine	Used to interact with the serial ports on Pleora-based GigE Vision devices. Supports UART, as well as Bulk interfaces (which can be configured as either UART or USRT). For Bulk interface mode over I2C, use PvIPEngineI2CBus instead. The serial port is not configured through the PvSerialPortIPEngine object. Use the PvDevice GenICam interface instead. IsSupported() can be used to query if a device supports a specific serial interface. Open() is used to open a serial port through a PvDevice. Write() and Read() are used to send and receive data through the serial port.
PvStatistics	Streaming statistics interface of a PvStreamRaw object. A pointer to the PvStreamRaw statistics can be retrieved using the GetStatistics() method. When using PvStream, the statistics are only provided through the **PvStream::GetParameters**().
PvStream	Stream object that can be configured and that can provide its statistics through a GenICam interface. Use GetGenParameters() to retrieve the GenICam interface. Unlike PvStreamRaw, PvStream requires GenApi to be deployed in the system.
PvStreamBase	Base class of both PvStream (GenICam) and PvStreamRaw (pure C++) streaming classes. Provides core capabilities like queuing and retrieving buffers.
PvStreamRaw	Stream object that can be configured and is providing its statistics directly through a C++ interface, bypassing the GenICam interface. It is recommended to use PvStream instead whenever possible.

Table 16: eBUS SDK Classes (Continued)

Class	Description
PvString	String class used to push and retrieve strings in and out of the eBUS SDK. Its main goal is to provide a unified interface for both multi-byte and unicode and to prevent string type mismatches between an application and the eBUS SDK. PvString is not meant as a complete string class for eBUS SDK users. It provides only basic capabilities to achieve its eBUS SDK input and output goals.
PvSystem	Represents the host system in the PvSystem, PvInterface, and PvDeviceInfo device finder architecture. Use the Find() method to discover devices. PvSystem enumerates each network interface card. Use GetInterfaceCount() and GetInterface() to retrieve pointers to PvInterface objects. Use GetDeviceCount() and GetDeviceInfo() on retrieved PvInterface pointers to get discovered devices in the form of PvDeviceInfo pointers.
PvSystemEventSink	Used for device found callbacks. To use it, inherit your class from PvSystemEventSink and override the **OnDeviceFound**() method. Your method is called immediately after a device is found, before **PvSystem::Find**() completes. It can be used to report devices before the discovery timeout is elapsed (like PvDeviceFinderWnd does) but can also be used to filter out devices by setting the aIgnore parameter of the callback to true.
PvTerminalIPEngineWnd	Used to interact with the serial port on a GigE Vision device GigE Vision device. Consists of a pair of transmit and receive text controls that can be used to interact live with a serial port. It does not require a serial port on the PC; just a connected PvDevice. All available serial ports on the PvDevice are displayed in a combo box that the user can dynamically select.
PvTransmitterRaw	Class for transmitting blocks using the GigE Vision streaming protocol.
PvVirtualDevice	Virtual GigE Vision device. Used to provide basic GigE Vision device capabilities such as device discovery. Can complement the PvTransmitterRaw class by allowing you to open a data receiver connection using PvStream or PvStreamRaw.
PvWnd (Base class for all GUI components)	Provides common methods to control the title, move, open and close dialogs and controls. Use ShowModal() to create a modal dialog. A modal dialog prevents the user from interacting with the other windows and dialogs of the application. The device finder as used by GEVPlayer is a modal dialog. Use ShowModeless() to create a modeless dialog. A modeless dialog lets the user interact with other dialogs and windows of the application. The GenICam tree browser of GEVPlayer is a modeless dialog. Use Create() to create the window as a child control of a specific window. GEVPlayer creates its PvDisplayWnd in such a way, nesting it in its main dialog.

Chapter 10

Appendix B — Layered Representation of the eBUS SDK

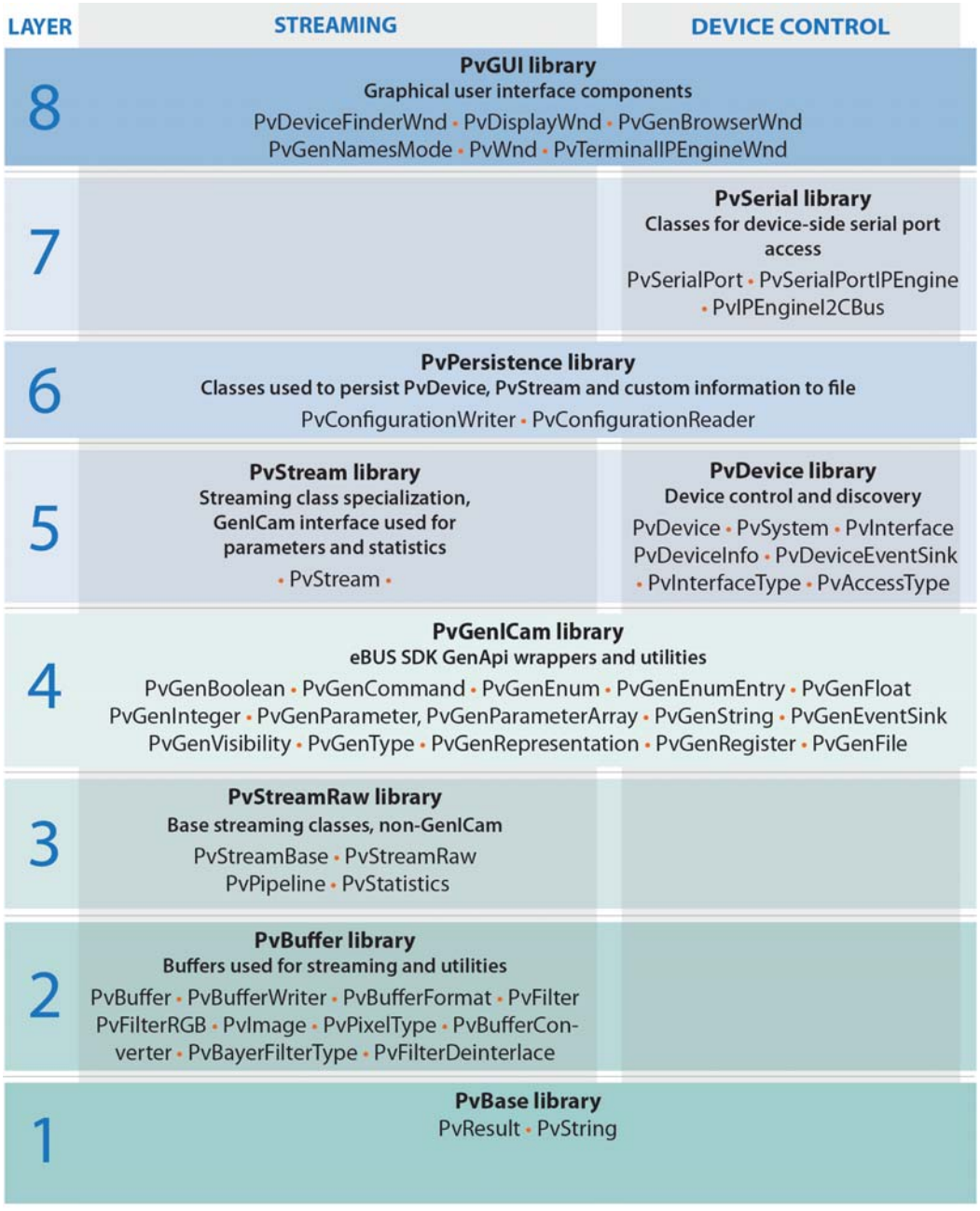
This section provides an overview of the eBUS SDK in the form of a table. Each SDK layer depends on the layer below.

An application that only requires streaming data and that does not require GenICam functionality from a device is really only required to use the PvStreamRaw library. In this case the application would need Layers 1 to 3.

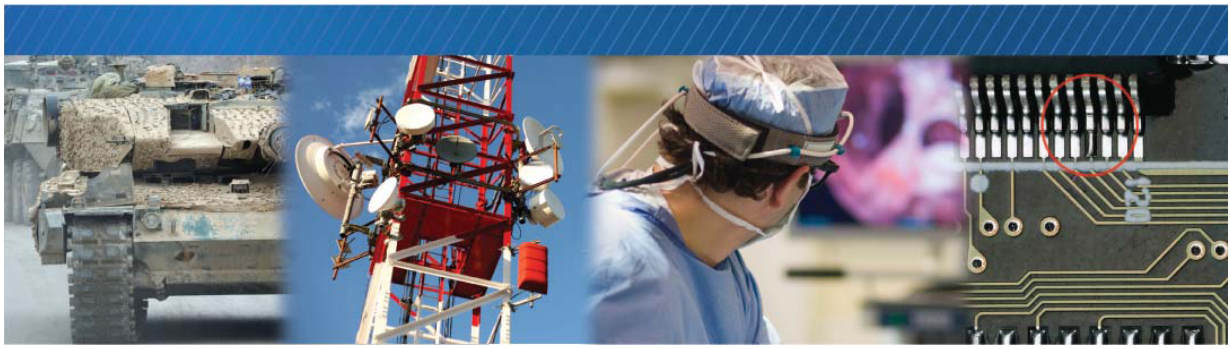
An application that needs to control and stream data from a GenICam capable GEV device typically uses the core layers 1 to 6.

Layers 7 and 8 are helper libraries, which can be developed by the customer if they choose not to use the ones supplied by the eBUS SDK.

Figure 1: Layered Representation of the eBUS SDK



Chapter 11



Appendix C — log4cxx Facility

The eBUS SDK makes use of the log4cxx facility to log events.

The logcfg file can be used to control the event log's destination and level.

For more information about log4cxx, go to http://logging.apache.org/log4cxx/index.html.

If you are using Windows, the file is located in **C:\Program Files\Common Files\Pleora\log4cxx\default.logcfg**

If you are using Linux, the file is located in /opt/pleora/ebus_sdk/lib/log4cxx/default.logcfg

You can also use the $PT_LOG_CONFIG environment variable to force the specific path of the logcfg file. To do this, modify the following line, in the Example.log file, by adding "R":

```
log4j.rootLogger=info, stdout, R
```

RootLogger is the parent category of all log4cxx categories. It is also possible to control logging through categories. Pleora provides all categories in the log4cxx configuration files, but they are commented out. Removing the comment enables logging for specific categories.

You can configure the following logging levels:

- info
- warning
- error
- fatal

Commonly used destinations (appenders) include:

- R - RollingFileAppender
- stdout- ConsoleAppender

The Rolling file appender (R) attributes can also be modified.

Other loggers can also be added.

```
log4j.appender.R.File=${APPDATA} /example.log
```

${APPDATA} is C:\Users\<user name>\AppData\Roaming on Windows 7 and C:\Documents and Settings\<user name>\Application Data on Windows XP. Note that this folder is hidden by default in Windows explorer. You can reveal this folder by changing your Windows settings to reveal hidden folders and files.

- Ensure that your process has permissions to write to the directory you point to.
- In Linux, the home folder shortcut(~) is not supported.
- Enabling the log degrades the overall performance of the system and it should only be done for debugging purposes. You should enable only the datapoints that are absolutely necessary for debugging purposes.
- For Windows 7 and Vista, it may be necessary to open the configuration file as a member of the Administrators group.

Chapter 12

Reference: C++ eBUS SDK and .NET SDK Comparison

This chapter outlines the differences between the C++ version of the eBUS SDK and the C# version.

The following topics are covered in this chapter:

The *eBUS SDK .NET API Help File* provides detailed information about the .NET classes available in the Chapter. The Help file is available from the Windows **Start** menu (**All Programs** > **Pleora Technologies Inc.** > **eBUS SDK** > **eBUS SDK .NET API**).

Types

The eBUS SDK types used in C++ are replaced by native .NET types. For example, PvString (which is used to input and output strings to/from the eBUS SDK) is replaced with a string.

Table 17: .NET Equivalent of eBUS SDK C++ Types

C++ type	Replaced by the following .NET type
PvString	String
PvInt64	Int64
PvUInt64	UInt64
PvInt32	Int32
PvUInt32	UInt32
PvInt16	Int16
PvUInt16	UInt16
PvUInt8	UInt8

Properties

The Get and Set methods in C++ are replaced by properties in the .NET version of the eBUS SDK. A property internally defines Get and Set methods but it is only accessed through its property name. This is because Get and Set methods without properties are not commonly used in .NET.

Exceptions to this rule are the Get and Set methods that require additional parameters (for example, accessing values in the GenICam node tree). Because these methods cannot be replaced with properties, they remain the same as their C++ counterparts.

C++

```
PvSerialPort lPort;
PvUInt32 lSize = lPort.GetRxBufferSize();
lPort.SetRxBufferSize( lSize );
```

C#

```
PvSerialPort lPort = new PvSerialPort();
UInt32 lSize = lPort.RxBufferSize;
lPort.RxBufferSize = lSize;
```

Enumeration Types

Typed native .NET enumerations (not to be confused with GenICam enumerations) are used to wrap C++ enumerations. The same underlying C++ values are used when defining the .NET enumerations.

.NET enumerations are different from C++ enumerations in the following ways:

- They throw an exception when you attempt to set them to an unsupported integer value.
- They can be converted to strings and the strings can be parsed and turned into an enumeration value.
- They can be enumerated in statements, such as foreach statements.

Error Management

The .NET version of the eBUS SDK uses exceptions for error management, whereas the C++ version uses methods that return error codes. For example, the eBUS SDK uses a void return type and throws a PvException object when an error occurs, whereas the C++ version returns a PvResult object.

A PvException object contains a Result property, which is the PvResult that the method would have returned. The ToString and Description members of PvException are adapted to return a string-based representation of the PvResult containing the error code and description.

The only exception to this rule is streaming methods that manage buffers in the PvStreamBase, PvPipeline, and PvTransmitterRaw classes. For performance reasons try/catch and exception management are not used to handle these errors. For these methods, there is no change from the C++ approach — these methods return a PvResult object.

C++ eBUS SDK methods that have a PvResult return value and take a reference to a parameter to return the true return value, now simply return the return value (or are implemented as properties). A good example is the GetValue method of the PvGenParameter class.

C++

```
PvInt64 lValue = 0;
PvResult lResult = lInteger.GetValue( lValue );
if (!lResult.IsOK())
{
   // ...
}
```

C#

```
try
{
   Int64 lValue = lInteger.Value;
}
catch (PvException lEx)
{
   // ...
}
```

C++

```
PvResult lResult = lDevice.Connect( "192.168.138.164" );
if ( !lResult.IsOK() )
{
    // ...
}

lResult = lStream.Open( "192.168.138.164" );
if ( !lResult.IsOK() )
{
    // ...
}

lResult = lPipeline.Start();
if ( !lResult.IsOK() )
{
    // ...
}
```

C#

```
try
{
   lDevice.Connect( "192.168.138.164" );
   lStream.Open( "192.168.138.164" );
   lPipeline.Start();
}
catch (PvException lEx)
{
    // ...
}
```

Enumerators

Some eBUS SDK classes are containers for other objects that can be enumerated. .NET interfaces are implemented for these classes to allow the program to go through all of the objects with a foreach statement:

- The PvInterface class enumerates PvDeviceInfo objects.
- The PvSystem class enumerates PvInterface objects.
- The PvGenEnum class enumerates PvEnumEntry objects.
- The PvGenParameterArray enumerates PvGenParameter objects.

C# (Without an Enumerator)

```
UInt32 lCount = lArray.Count;
for (UInt32 i = 0; i < lCount; i++)
{
   PvGenParameter lP =
      lParameterArray.Get(i);
   // …
}
```

C# (With an Enumerator)

```
foreach (PvGenParameter lP in lArray)
{
   // ...
}
```

Collections

The C++ version of the eBUS SDK has collection classes such as PvStringList, PvPropertyList, and PvParameterList. In .NET, native List templates are used instead, such as List<string>, List<PvProperty>, and List<PvParameter>.

Callbacks

All C++ eBUS SDK callbacks or virtual methods used as callbacks are implemented in PvDotNet as .NET events of typed delegates.

Registering to an event is done in .NET using the += operator next to an event. The IntelliSense® feature of Visual Studio then automatically generates an event handler in the .NET code.

This rule ensures that the callback mechanism is as expected by .NET users. It also leverages the automated code generation performed by Visual Studio to help developers quickly and effortlessly hook up events to their own code.

C++

```
class MyClass : PvDeviceEventSink
{
   void OnLinkDisconnected( PvDevice *aDevice )
   {
      // ...
   }
}

MyClass lMyObject;
lDevice.RegisterEvenSink( &lMyObject );
// ...
lDevice.UnregisterEventSink( &lMyObject );
```

C#

```
private void Device_OnLinkDisconnected( PvDevice aDevice)
{
   // ...
}

lDevice.OnLinkDisconnected += new OnLinkDisconnectedEvent(Device_OnLinkDisconnected);
   // ...
lDevice.OnLinkDisconnected -= new OnLinkDisconnectedEvent(Device_OnLinkDisconnected);
```

Chapter 13

Technical Support

At the Pleora Support Center, you can:

- Download the latest software.
- Log a support issue.
- View documentation for current and past releases.
- Browse for solutions to problems other customers have encountered.
- Get presentations and application notes.
- Get the latest news and information about our products.
- Decide which of Pleora's products work best for you.

To visit the Pleora Support Center

- Go to www.pleora.com and click **Support Center**.

  If you have not registered yet, you are prompted to register.

  Accounts are usually validated within one business day.

If you have difficulty finding an existing solution in the knowledge base, post a question by clicking **Log a Case**. Provide as many specific details about your system and the nature of the issue as possible.