PLEORA TECHNOLOGIES INC.

eBUS SDK C++ API

eBUS SDK Version 6.4.0

Quick Start Guide

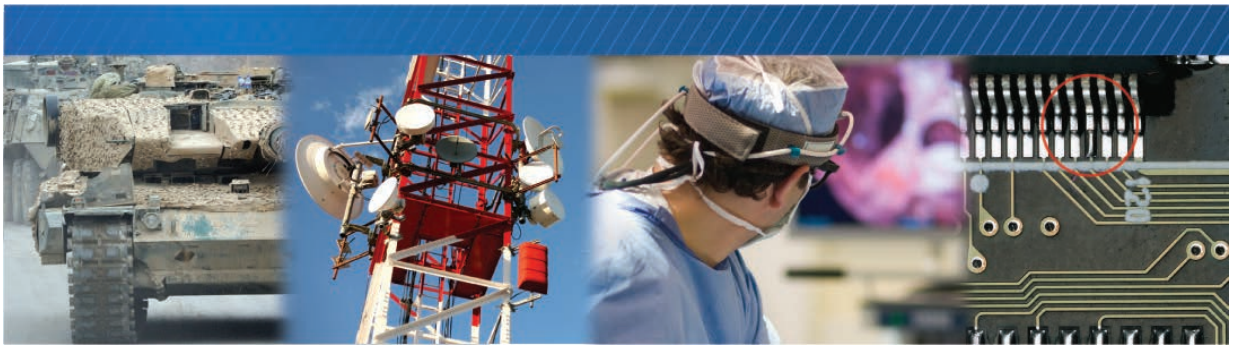
Pleora Technologies

Document Number

QSG-0012 Revision 15.0 - 0 2/7/24

Table of Contents

Chapter 1



About this Guide

This chapter describes the purpose and scope of this guide, and provides a list of complementary guides.

The following topics are covered in this chapter:

What this Guide Provides

This guide provides you with the information you need to install the eBUS SDK (which lets you use the eBUS SDK C++ API) and an overview of the system requirements.

You can use the sample applications to see how the API classes and methods work together for device configuration and control, unicast and multicast communication, image and data acquisition, image display, and diagnostics. You can also use the sample code to verify that your system is working properly (that is, determine whether there is a problem with your code or your equipment).

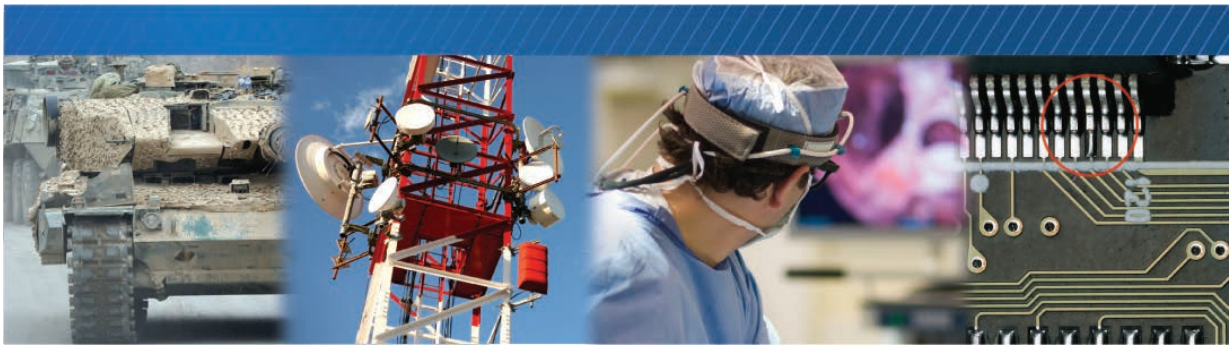
For troubleshooting information and technical support contact information for Pleora Technologies, see the last few chapters of this guide.

Related Documents

The *eBUS SDK .NET API Quick Start Guide* is complemented by the following Pleora Technologies documents, which are available on the Pleora Technologies Support Center (supportcenter.pleora.com):

- *eBUS Player Quick Start Guide*
- *eBUS Player User Guide*
- *eBUS SDK C++ API Help File*
- *Getting Started with eBUS Edge*
- *eBUS SDK for Linux Quick Start Guide*
- *eBUS SDK Licensing Overview Knowledge Base Article*
- *Configuring Your Computer and Network Adapters for Best Performance Knowledge Base Article*

Chapter 2



Introducing the eBUS SDK C++ API

This chapter describes the eBUS SDK C++ API, which is a feature of the eBUS SDK that allows you to develop custom vision systems to acquire and transmit images and data using C++.

The following topics are covered in this chapter:

About the eBUS SDK C++ API

eBUS SDK is built on a single API to receive video over GigE, 10 GigE, and USB 3.0 that is portable across Windows, Linux, and macOS operating systems. With an eBUS SDK Seat License, designers can develop production-ready software applications in the same environment as their end-users, and quickly and easily modify applications for different media, while avoiding supporting multiple APIs from various vendors. Compared to camera vendor provided SDKs, eBUS frees developers from being tied to a specific camera, and instead they can choose the device that is best for the application.

eBUS Edge for Sensor Devices

eBUS Edge is a software implementation of a full device level GigE Vision transmitter, without requiring any additional hardware. Adding eBUS Edge to a CPU's software stack turns it into a fully compliant GigE Vision device that supports image transmission and enables the device to respond to control requests from a host controller. eBUS Edge is GigE Vision and GenICam compliant, meaning end-users can use any standards compliant third-party image processing system. eBUS Edge currently supports the GigE Vision standard

eBUS Receive for Host Applications

eBUS Receive manages high-speed reception of images or data into buffers for hand-off to the end application for further analysis. Developers can write applications that run on a hostcomputer to seamlessly control and configure an unlimited number of GigE Vision or USB3 Vision and GenICam compliant sensors.

The eBUS Universal Pro driver reduces CPU usage when receiving images or data, leaving more processing power for analysis and inspection applications while helping meet latency and throughput requirements for real-time applications. The eBUS Universal Pro driver is easily integrated into third-party processing software to bring performance advantages to end-user applications.

eBUS SDK Licenses

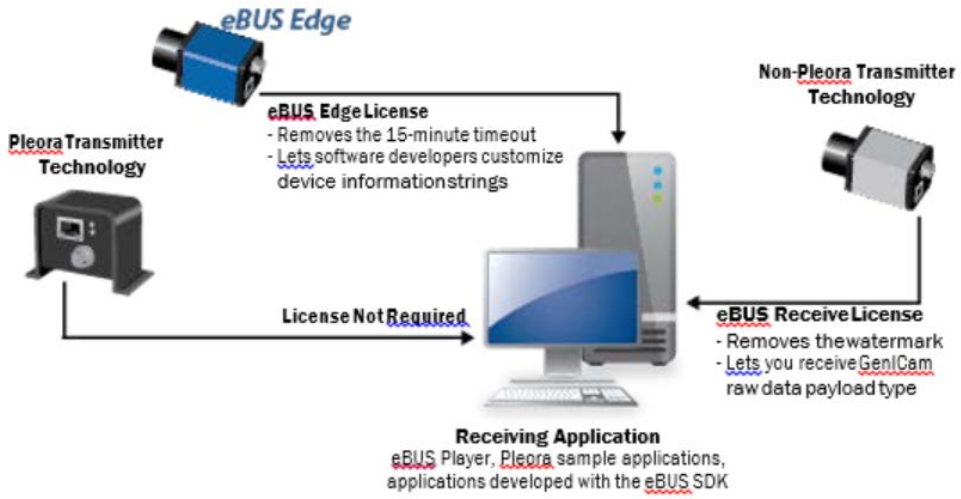
Some components of the eBUS SDK require a Pleora license to remove transmit and receive limitations.

eBUS Player, Sample Applications, and Software Applications Created with the eBUS SDK

When a license is not present and you are receiving images from third-party transmitter technology, an embossed Pleora watermark will appear on the images that you receive. In addition, you will not be able to receive the GenICam raw data payload type from the transmitting device.

GigE Vision Devices Created with the eBUS Edge API

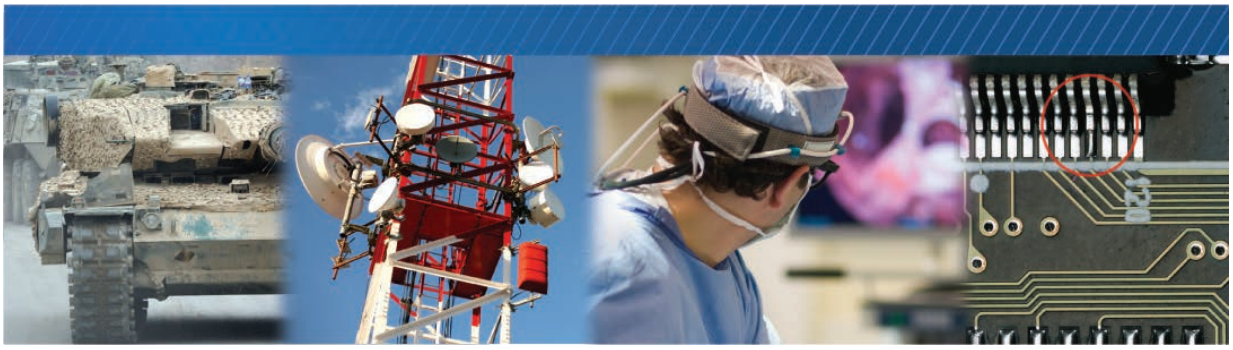
When a license is not present, GigE Vision receivers will be disconnected from the GigE Vision device after 15 minutes. In addition, your device will report hard-coded device information (such as the device model name), instead of your organization's customized information.



Activating an eBUS SDK License

For detailed information about licensing, including details on activating a license, see the eBUS SDK *Licensing Overview Knowledge Base Article* available on the Pleora Technologies Support Center at supportcenter.pleora.com.

Chapter 3



Installing the eBUS SDK

The eBUS SDK C++ API is installed on our computer during the installation of the eBUS SDK.

For installation instructions on Linux x86 and Linux ARM systems, please refer to the *eBUS SDK for Linux Quick Start Guide*.

The following topics are covered in this chapter:

System Requirements

Ensure the computer on which you install the eBUS SDK meets the following recommended requirements:

- At least one Gigabit Ethernet NIC (if you are using GigE Vision devices) or at least one USB 3.0 port (if you are using USB3 Vision devices).
- An appropriate compiler or integrated development environment (IDE):
  - Visual Studio 2019, 2017, 2015, 2013, 2012, and 2010.
  - Sample project files (.vcxproj) are compatible with Visual Studio 2012 (and later). When you open them with Visual Studio for the first time, you have the option of upgrading them.
- One of the following operating systems:
  - Microsoft Windows 11 64-bit
  - Microsoft Windows 10, 32-bit or 64-bit
  - Microsoft Windows 8.1, 32-bit or 64-bit
  - Microsoft Windows 7 with Service Pack 1 (or later), 32-bit or 64-bit
  - For the x86 Linux platform:
    - Ubuntu 22.04 LTS 64-bit
    - Ubuntu 20.04 LTS 64-bit
    - Ubuntu 18.04 LTS 64-bit
    - Red Hat Enterprise Linux 8, 64-bit
    - CentOS Stream 8, 64-bit
  - For the Linux for ARM platform:
    - NVIDIA Jetson TX2, Jetson Nano, Jetson Xavier NX, Jetson AGX Xavier, Jetson TX2 NX (Ubuntu 18.04 with Jetpack 4.6)
    - NVIDIA Jetson AGX Xavier, Jetson Xavier NX, Jetson AGX Orin, Jetson Orin NX (Ubuntu 20.04 with Jetpack 5.1)
  - For the x86 Linux and Linux ARM platforms, Qt us required to compile GUI-based samples:
    - **For Ubuntu 22.04 Desktop**: 64-bit Qt 5.15.3
    - **For Ubuntu 20.04 Desktop**: 64-bit Qt 5.12.8
    - **For Ubuntu 18.04 Desktop**: 64-bit Qt 5.9.5
    - **For Ubuntu 20.04 for ARM:** Qt 5.12.8
    - **For Ubuntu 18.04 for ARM**: Qt 5.9.5
    - **For RHEL 8,** 64-bit: Qt 5.15.3
    - **CentOS Stream 8,** 64-bit: Qt 5.15.3

For supported USB 3.0 host controller chipsets, consult the eBUS SDK Release Notes, available on the Pleora Support Center.

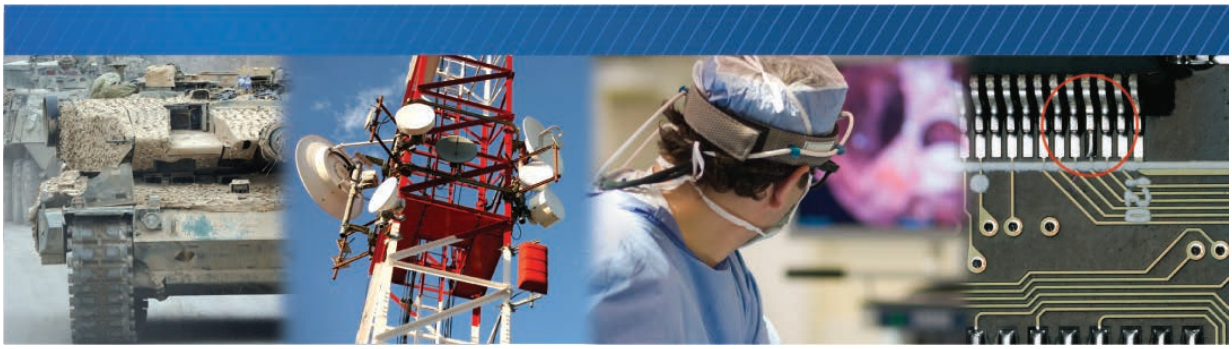
Depending on the incoming and outgoing bandwidth requirements, as well as the performance of each NIC, you may require multiple NICs. For example, even though Gigabit Ethernet is full duplex (that is, it manage 1 Gbps incoming and 1 Gbps outgoing), the computer's PCIe bus may not have enough bandwidth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

Installing the eBUS SDK

Because the eBUS SDK C++ API is part of the eBUS SDK on Windows, it is included in the eBUS SDK for Windows installation package. You can download the eBUS SDK from the Pleora Support Center at supportcenter.pleora.com.

If you use the Linux operating system, you must install the eBUS SDK as superuser. For full details about installing the eBUS SDK for Linux, see the eBUS SDK for Linux Quick Start Guide, available at the Pleora Support Center (supportcenter.pleora.com).

Chapter 4



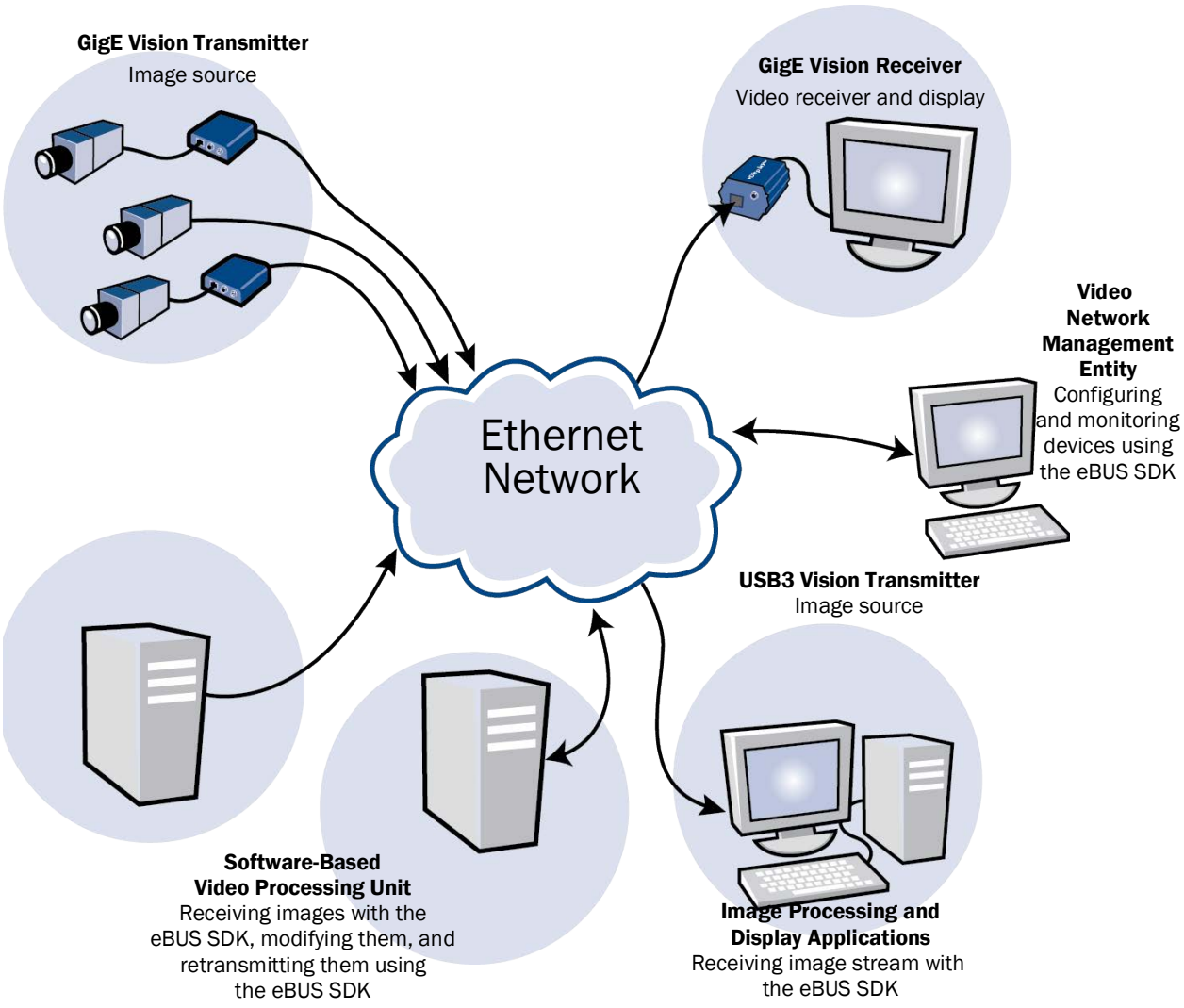
Using the Sample Code

To illustrate how you can use the eBUS SDK C++ API to acquire and transmit images, the SDK includes sample code that you can use. This chapter provides a description of the sample code and provides general information about accessing the code to create sample applications.

The following topics are covered in this chapter:

Overview: System Components

The following illustration shows the components that are used, illustrating the relationship between the eBUS SDK, GigE Vision receivers, GigE Vision transmitters, and USB3 Vision transmitters.



Description of Samples

The following table lists the C++ sample applications that are available. For information about the C# and VB.NET samples that are also available, see the *eBUS SDK .NET API Quick Start Guide*.

For information about using the eBUS SDK with macOS, see the *eBUS SDK for macOS Quick Start Guide*.

Please visit the Pleora Support Center at supportcenter.pleora.com for copies of these publications.

PLCAndGEVEventsSample, **SerialBridge**, **SimpleGUIApplication**, and **TransmitTiledImage** are not available for Linux.

Table 1: Sample Code

Sample code	Function	Type of application that is created
eBUS Demo Application		
eBUSPlayer	eBUS Player is a full-featured application that showcases the functionality of the eBUS SDK, which is used to detect, connect, and configure GigE Vision and USB3 Vision devices, and display and stream images. This sample provides advanced examples of the eBUS SDK classes and functions that are available and provides full source code. **Note:** If you want to use H.264 encoding with the eBUSPlayer sample application, see the *Encoding Images in the H.264 Video Format and Saving them in an MP4 File Knowledge Base Article*, available on the Pleora Support Center (supportcenter.pleora.com).	UI-based, Cocoa. • Windows: MFC • Linux: Qt
Getting Started		
PvStreamSample	This "Hello World" sample shows you how to connect to a GigE Vision or USB3 Vision device, receive an image stream, stop streaming, and disconnect from the device.	Command line. • All platforms
SimpleGUIApplication, SimpleGUIApplicationVB	This UI sample provides a basic user interface using C++ and MFC to detect, connect, and configure GigE Vision and USB3 Vision devices, and display and stream images. This sample is a good starting point to show how to create a GUI project to receive images from a GigE Vision or USB3 Vision camera.	UI-based, Cocoa. • Windows: MFC
Image Streaming		

Table 1: Sample Code (Continued)

Sample code	Function	Type of application that is created
PvPipelineSample	This sample extends the "Hello World" PvStreamSample by showing how buffers are managed internally by the PvPipeline class. This removes some of the complexity of buffer management from the application when compared to the PvStream sample.	Command line. • All platforms
MultiSource for GigE Vision devices	This command line sample for GigE Vision devices shows you how to receive images from a GigE Vision device that has multiple streaming sources.	Command line. • All platforms
ImageProcessing	This sample illustrates how to acquire an image and process it using an external buffer to interface with a non-Pleora library. This is useful when you want to interface the eBUS SDK to popular third-party SDKs for image processing or machine learning, such as OpenCV.	Command line. • Windows
MulticastMaster for GigE Vision devices	This sample shows how to connect to a GigE Vision device and initiate a multicast stream to allow multiple slave devices to receive and process the image stream simultaneously. This sample is used in conjunction with the **MulticastSlave** sample, which listens to the multicast stream.	Command line. • All platforms
MulticastSlave for GigE Vision devices	This sample shows how to configure the eBUS SDK to receive an image stream from a GigE Vision device that is configured for multicast mode. This sample is used in conjunction with the **MulticastMaster** sample, which initiates the multicast stream.	Command line. • All platforms
TapReconstruction	This sample generates synthetic images for each supported tap geometry*, applies a filter to reconstruct each image, and then saves the images as TIFF files for display. Tap reconstruction is used with multi-tap cameras, which may not send the pixels in order, depending on the camera's tap geometry. *Tap geometries are defined by the *GenICam Standard Features Naming Convention (SFNC)*.	Command line. • Windows
DirectShowDisplay	This sample shows how to integrate the eBUS SDK DirectShow source filter and how to build a filter graph that receives images from a GigE Vision or USB3 Vision device. This functionality is recommended for existing DirectShow applications to enable quick integration of GigE Vision or USB3 Vision devices. Note: This sample assumes good knowledge of DirectShow and COM.	UI based. • Windows: MFC, C# .NET

Table 1: Sample Code (Continued)

Sample code	Function	Type of application that is created
Discovery and Connection		
DeviceFinder	This sample shows how to detect and enumerate GigE Vision and USB3 Vision devices on the network.	UI based. • All platforms
ConnectionRecovery	This sample shows how to automatically recover from connectivity issues, such as accidental disconnects and power interruptions, to build more robustness into your eBUS SDK application.	Command line. • Windows
eBUSDriverInstallationAPI	This sample shows how to display information about network configuration and apply eBUS drivers to network adapters on a given PC.	Command line. • Windows
Configuration and Event Monitoring		
DeviceSerialPort	This sample shows how to send commands to a camera or other device that accepts serial input commands through a compatible Pleora iPORT video interface using the Pleora device's General Purpose Input/Output (GPIO) signals, including UART or BULK.	Command line. • All platforms
CameraBridge	This sample shows how to control a Camera Link camera through a compatible Pleora iPORT CL Series External Frame Grabber using the following Camera Link protocols: CLProtocol and GenCP.	Command line. • All platforms
SerialBridge	This sample shows how to control a serial device connected to a compatible Pleora iPORT video interface using the Pleora Camera Link DLL interface or a null modem COM port on your computer. You should use this sample if your camera manufacturer has provided you with a camera configuration software application that relies on a Camera Link DLL to facilitate camera control or if you want to connect to the camera using a null modem connection.	Command line • Windows
GenICamParameters	This sample shows how to enumerate and display the GenICam features and settings of a GenICam-compatible device by discovering and accessing the features of the device's node map. The node map is built programmatically from the device's GenICam XML file.	Command line. • All platforms
EventSample	This sample shows how to receive and handle events sent from USB3 Vision and GigE Vision devices including Pleora eBUS Edge through EVENT_CMD and EVENTDATA_CMD.	Command line. • All platforms

Table 1: Sample Code (Continued)

Sample code	Function	Type of application that is created
PlcAndGevEvents for GigE Vision devices	Shows how to control and handle Programmable Logic Controller (PLC) states and handle GigE Vision events. **Note:** This sample is compatible with the PLC found in first generation Pleora products, such as the iPORT NTx-Pro Embedded Video Interface, the iPORT NTx-Mini Embedded Video Interface, and the iPORT PT1000-CL External Frame Grabber.	UI-based. • Windows: MFC
ConfigurationReader	This sample shows how to save the configuration state of your GigE Vision or USB3 Vision device, your eBUS SDK application preferences, and any custom strings and property lists to a file. It also illustrates how to open and restore this information.	Command line. • All platforms
eBUS Edge Code Samples		
SoftDeviceGEVSimple	This sample shows how to create a basic software GigE Vision device with one streaming source and a single pixel type. A sample test pattern is generated as a streaming source.	Command line. • Windows and Linux
SoftDeviceGEV	This sample shows how to create a fully functioning software GigE Vision device with multiple streaming sources and fixed width and height pixel types. A sample test pattern is generated as a streaming source. This sample also illustrates how to implement custom GenApi features and device registers, as well as how to access the GVCP messaging channel to send events and chunk data.	Command line. • Windows and Linux
SoftDeviceGEVTrigger	This sample shows how to use PvSoftDeviceGEV to create different types of triggers in a software GigE Vision transmitter device.	Command line. Windows and Linux
SoftDeviceGEVMultiPart	This sample shows how to use PvSoftDeviceGEV to create a software GigE Vision multi-part transmitter device.	Command line. • Windows and Linux

For developers who have worked with earlier releases of the eBUS SDK, please note that the Video Server API and the following sample applications are not recommended for new designs (NRND): **TransmitChunkData**, **TransmitProcessedImage**, **TransmitTestPattern**, and **TransmitTiledImages**. The eBUS Edge code samples are recommended replacements.

Using H.264 Encoding with the eBUSPlayer Sample Application on the Windows Operating System

You can use eBUS Player to encode images in H.264 video format and save them on your computer in an MP4 file.

If you are interested in saving MP4 video in your application, you can refer to the `Mp4WriterWin32` class in the eBUS Player C++ sample application for an example.

Linux-based versions of eBUS SDK do not have an API to save video in MP4 format. The user must use third party libraries to do this. Please refer to eBUS SDK for Linux Quick Start Guide for an example showing how to enable MP4 saving on eBUS Player (Chapter 2 “Installing the eBUS SDK for Linux” step 7).

Enabling H.264 Encoding on the Windows Operating System

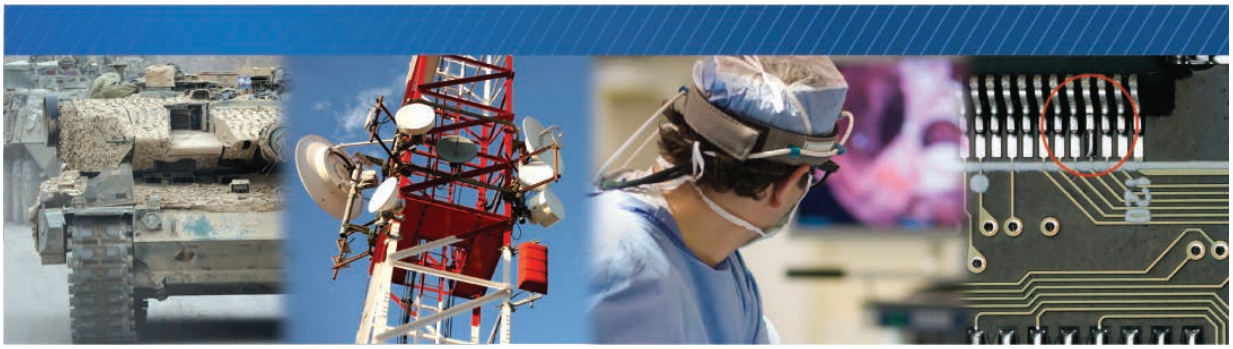
On the Windows operating system, this feature can be used when you start eBUS Player from the Windows Start menu. However, it is important to note that if you are compiling the eBUS Player sample application using Visual Studio, this feature is not enabled by default.

To enable H.264 encoding in the eBUS Player sample, define the `PV_ENABLE_MP4` preprocessor macro in the project properties of the eBUS Player sample application.

Video Encoder Information

When saving H.264 video in an MP4 container, the eBUS SDK uses the Microsoft Media Foundation H.264 video encoder. The default video encoder settings are used, with the exception of the frame rate and average bit rate. The video is encoded at 30 frames per second*. And the average bit rate is the value set by **`PvMp4Writer::SetAvgBitrate`**.

Chapter 5



Code Walkthrough: Acquiring Images with the eBUS SDK

This section walks you through the code contained in **PvStreamSample**. This sample illustrates how to detect available devices, connect to a device, and start an image stream.

The following topics are covered in this chapter:

Accessing the Sample Code

The sample code is available in the following locations:

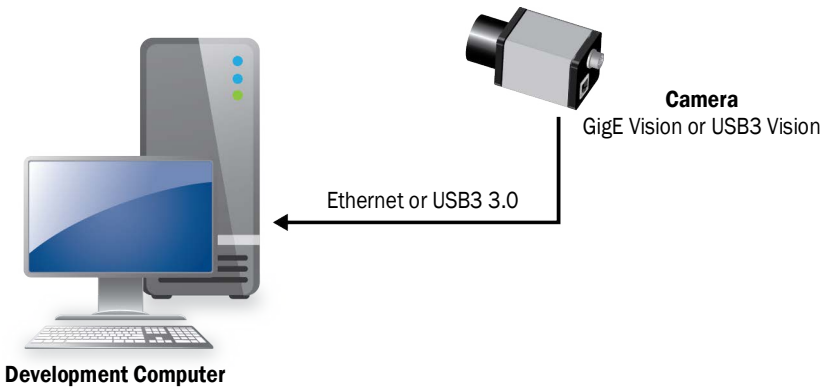
- **Windows.** C:\Program Files\Pleora Technologies Inc\eBUS SDK\Samples
- **Linux.** /opt/pleora/ebus_sdk/*<distribution targeted>*/share/samples

For Windows installations, you must copy the sample code to a location on your computer (such as your C: drive) before you open the sample code in your IDE. On the Windows operating system, access to these directories is restricted.

For Linux systems, it is recommended to make a copy of the "samples" directory and save it somewhere else.

Required Items

The sample code requires that you have a GigE Vision device connected to a NIC on your computer or a USB3 Vision device connected to a USB 3.0 port on your computer.



Windows and Linux Support

PvStreamSample can be used on the Windows and Linux operating systems. Platform-specific code is abstracted by **`PvSampleUtils.h`**, which is installed on your computer as part of the eBUS SDK.

Classes Used in the PvStreamSample

PvStreamSample uses the classes listed in the following table.

Table 2: Classes Used in the Sample

Class	Description
PvDeviceInfo	Used to access information about a device, such as its manufacturer information, protocol (either GigE Vision or USB3 Vision), serial number, ID, and version.
PvDevice	Used to connect to and control a device and initiate the image stream. Protocol and interface-specific functionality is available in two subclasses, **PvDeviceGEV** and **PvDeviceU3V**.
PvStream	Provides access to the image stream. Like PvDevice, there are protocol and interface-specific subclasses, **PvStreamGEV** and **PvStreamU3V**.
PvBuffer	Represents a block of data from the device, such as an image.
PvResult	A simple class that represents the result of various eBUS SDK functions methods.

Header Files

The following header files are required by the sample. Please note that `PvSamplesUtils.h` provides some basic helper and multi-platform routines.

C++

```
#include <PvSampleUtils.h>
#include <PvDevice.h>
#include <PvDeviceGEV.h>
#include <PvDeviceU3V.h>
#include <PvStream.h>
#include<PvStreamGEV.h>
#include <PvStreamU3V.h>
#include <PvBuffer.h>

#ifdef PV_GUI_NOT_AVAILABLE
#include <PvSystem.h>
#else
#include <PvDeviceFinderWnd.h>
#endif // PV_GUI_NOT_AVAILABLE
```

`PV_GUI_NOT_AVAILABLE` is defined at compile time. It allows the sample to support both GUI and non-GUI systems. In your applications, this is helpful in cases where you may not want to use a GUI or the user's system does not have GUI support. It also allows you to use Visual Studio Express, which does not support the Microsoft Foundation Class Library (MFC), which is used later in the sample (by **PV_SAMPLE_INIT**). If defined, the sample will present the user with a command line prompt instead of a graphical dialog box when selecting a device for connection. This is described in more detail in "The SelectDevice Function" on page 24.

Function Prototypes

Next, we define the required function prototypes, including **SelectDevice**, **ConnectToDevice**, **OpenStream**, **ConfigureStream**, **CreateStreamBuffers**, and **AcquireImages**, which are called by the **main**() function.

C++

```
///
/// Function Prototypes
///
const PvDeviceInfo *SelectDevice( PvDeviceFinderWnd *aDeviceFinderWnd );
const PvDeviceInfo *SelectDevice( PvSystem *aPvSystem );
PvDevice *ConnectToDevice( const PvDeviceInfo *aDeviceInfo );
PvStream *OpenStream( const PvDeviceInfo *aDeviceInfo );
void ConfigureStream( PvDevice *aDevice, PvStream *aStream );
void CreateStreamBuffers( PvDevice *aDevice, PvStream *aStream );
void AcquireImages( PvDevice *aDevice, PvStream *aStream );
```

The main() Function

main() calls functions that allow the user to select a device (**SelectDevice**), connect to a device (ConnectToDevice), start the image stream (**OpenStream**, **ConfigureStream**, **CreateStreamBuffers**), and process the image stream (**AcquireImages**). To perform these tasks, the following objects are required:

- A PvDeviceInfo object, which indicates the device that the user has selected for streaming.
- A PvDevice object, which allows the user to control the selected device.
- A PvStream object, which is used to receive the image stream for the selected device.

SelectDevice is overloaded in two functions to support both GUI and non-GUI device selection.

PV_SAMPLE_INIT and **PV_SAMPLE_TERMINATE** are macros that are expanded at compile time to create and delete a context that allows a UI-based device finder to be opened from within the command line application on Windows systems. **Note:** For the Linux operating system, these macros are empty.

PvWaitForKeyPress is a platform-independent helper function. This function is provided in the **PvSampleUtils.h** include file.

C++

```
//
// Main function
//
int main()
{
    const PvDeviceInfo *lDeviceInfo = NULL;
    PvDevice *lDevice = NULL;
    PvStream *lStream = NULL;

    PV_SAMPLE_INIT();

#ifdef PV_GUI_NOT_AVAILABLE
    PvSystem *lPvSystem = new PvSystem;
    lDeviceInfo = SelectDevice( lPvSystem );
#else
    PvDeviceFinderWnd *lDeviceFinderWnd = new PvDeviceFinderWnd();
    lDeviceInfo = SelectDevice( lDeviceFinderWnd );
#endif // PV_GUI_NOT_AVAILABLE

    cout << "PvStreamSample:" << endl << endl;

    if ( NULL != lDeviceInfo )
    {
        if ( lDevice = ConnectToDevice( lDeviceInfo ) )
        {
            if ( lStream = OpenStream( lDeviceInfo ) )
            {
                ConfigureStream( lDevice, lStream );
                CreateStreamBuffers( lDevice, lStream );
                AcquireImages( lDevice, lStream );
            }

            // Close the stream
            cout << "Closing stream" << endl;
            lStream->Close();
            PvStream::Free( lStream );
        }

        // Disconnect the device
        cout << "Disconnecting device" << endl;
        lDevice->Disconnect();
        PvDevice::Free( lDevice );
    }
```

Continued on next page...

```
    // Disconnect the device
            cout << "Disconnecting device" << endl;
            lDevice->Disconnect();
            PvDevice::Free( lDevice );
        }

        cout << endl;
        cout << "<press a key to exit>" << endl;
        PvWaitForKeyPress();

    #ifdef PV_GUI_NOT_AVAILABLE
        delete lPvSystem;
        lPvSystem = NULL;
    #else
        delete lDeviceFinderWnd;
        lDeviceFinderWnd = NULL;
    #endif // PV_GUI_NOT_AVAILABLE

        PV_SAMPLE_TERMINATE();

        return 0;
    }
```

The SelectDevice Function

SelectDevice provides the user with a way to see all of the available GigE Vision and USB3 Vision devices, and select one. In the GUI implementation, a device finder dialog box presents all of the devices that are available to the application. The non-GUI implementation presents a list of devices in the command line window, and is intended for use on systems that do not have GUI support, such as embedded systems.

When the user selects a device, an associated **PvDeviceInfo** object is returned to **main()**. **PvDevice** and **PvStream** use the **PvDeviceInfo** object to connect to the device.

The **`DeviceFinder`** object is allocated in `PvDeviceFinderWnd`, which must stay in scope to ensure that the **`PvDeviceInfo`** object remains valid.

C++

```
const PvDeviceInfo *SelectDevice( PvDeviceFinderWnd *aDeviceFinderWnd )
{
    const PvDeviceInfo *lDeviceInfo = NULL;
    PvResult lResult;

    if (NULL != aDeviceFinderWnd)
    {
        // Display list of GigE Vision and USB3 Vision devices
        lResult = aDeviceFinderWnd->ShowModal();
        if ( !lResult.IsOK() )
        {
            // User hit cancel
            cout << "No device selected." << endl;
            return NULL;
        }

        // Get the selected device information.
        lDeviceInfo = aDeviceFinderWnd->GetSelected();
    }

    return lDeviceInfo;
}
const PvDeviceInfo *SelectDevice( PvSystem *aPvSystem )
{
    const PvDeviceInfo *lDeviceInfo = NULL;
    PvResult lResult;

    if (NULL != aPvSystem)
    {
        // Get the selected device information.
        lDeviceInfo = PvSelectDevice( *aPvSystem );
    }

    return lDeviceInfo;
}
```

The ConnectToDevice Function

ConnectToDevice establishes a connection with the device.

GigE Vision and USB3 Vision devices are represented by different classes (**PvDeviceGEV** and **PvDeviceU3V**) and they share a parent class (**PvDevice**) that abstracts most of the differences. When possible, you should use a **PvDevice** object instead of a protocol-specific object to reduce code duplication. To create a **PvDevice** object from a **PvDeviceInfo** object without explicitly checking the protocol of the device, use the **CreateAndConnect** static factory method from the PvDevice class, which abstracts the device type.

It is important that objects allocated with **CreateAndConnect** be freed with **PvDevice::Free**, as shown later in the sample.

If it is not important for your application to support both GigE Vision and USB3 Vision devices (for example, your organization only uses devices of a particular type), you can call the **GetType** method of the **PvDeviceInfo** object (**aDeviceInfo**) to determine whether the device is GigE Vision or USB3 Vision. Then you could create a new **PvDeviceGEV** or **PvDeviceU3V** object and call the **Connect** method directly.

A pointer to the **PvDevice** object is returned to **main**() and can now be used to control the device and initiate streaming.

C++

```
PvDevice *ConnectToDevice( const PvDeviceInfo *aDeviceInfo )
{
    PvDevice *lDevice;
    PvResult lResult;

    // Connect to the GigE Vision or USB3 Vision device
    cout << "Connecting to " << aDeviceInfo->GetDisplayID().GetAscii() << "." << endl;
    lDevice = PvDevice::CreateAndConnect( aDeviceInfo, &lResult );
    if ( lDevice == NULL )
    {
        cout << "Unable to connect to " << aDeviceInfo->GetDisplayID().GetAscii() << "." << endl;
    }

    return lDevice;
}
```

The OpenStream Function

OpenStream initiates the image stream. Again, the sample uses a static factory method (**CreateAndOpen**) to create and open the **PvStream** object, which allows your application to support both GigE Vision and USB3 Vision devices.

A pointer to the **PvStream** object is returned to main() and can now be used to receive images as **PvBuffer** objects.

C++

```
PvStream *OpenStream( const PvDeviceInfo *aDeviceInfo )
{
    PvStream *lStream;
    PvResult lResult;

    // Open stream to the GigE Vision or USB3 Vision device
    cout << "Opening stream to device." << endl;
    lStream = PvStream::CreateAndOpen( aDeviceInfo->GetConnectionID(), &lResult );
    if ( lStream == NULL )
    {
       cout << "Unable to stream from " << aDeviceInfo->GetDisplayID().GetAscii() << "." << endl;
    }

    return lStream;
}
```

The ConfigureStream Function

For most of this sample, there is no need to distinguish between GigE Vision or USB3 Vision devices, as the eBUS SDK classes abstract the device type. However, when using a GigE Vision device, you must set a destination IP address for the image stream. In this sample, the destination is automatically set to be the IP address of the network interface card on the PC used to interface with the device (which is the most common configuration).

Also, for optimal performance over Gigabit Ethernet, it is necessary to determine the largest possible packet size for the connection (ideally the link would use jumbo frames — typically about 9000 bytes). This is the only place in the application where we check the device type.

Jumbo frames are configured on your computer's network interface card (NIC). For more information, see the operating system documentation or the *Configuring Your Computer and Network Adapters for Best Performance Knowledge Base Article*, available on the Pleora Support Center at supportcenter.pleora.com.

When developing your application, you may prefer to hard-code the packet size based on your target system, instead of using **`PvDeviceGEV::NegotiatePacketSize.`**

First, we use a dynamic cast to determine if the **`PvDevice`** object represents a GigE Vision device. If it is a GigE Vision device, we do the required configuration. If it is a USB3 Vision device, no stream configuration is required for this sample. When we create a pointer to the **`PvStream`** object, we use a static cast (because we already know that the **`PvStream`** object represents a stream from a GigE Vision device (**`PvStreamGEV`**), and no checking is required).

C++

```
void ConfigureStream( PvDevice *aDevice, PvStream *aStream )
{
   // If this is a GigE Vision device, configure GigE Vision specific streaming parameters
   PvDeviceGEV* lDeviceGEV = dynamic_cast<PvDeviceGEV *>( aDevice );
   if ( lDeviceGEV != NULL )
   {
      PvStreamGEV *lStreamGEV = static_cast<PvStreamGEV *>( aStream );

      // Negotiate packet size
      lDeviceGEV->NegotiatePacketSize();

      // Configure device streaming destination
      lDeviceGEV->SetStreamDestination( lStreamGEV->GetLocalIPAddress(), lStreamGEV->GetLocalPort() );
   }
}
```

The CreateStreamBuffers Function

CreateStreamBuffers allocates memory for the received images.

PvStream contains two buffer queues: an "input" queue and an "output" queue. First, we add **PvBuffer** objects to the input queue of the **PvStream** object by calling **PvStream::QueueBuffer** once per buffer. As images are received, **PvStream** populates the **PvBuffers** with images and moves them from the input queue to the output queue. The populated **PvBuffers** are removed from the output queue by the application (using **PvStream::RetrieveBuffer**), processed, and returned to the input queue (using **PvStream::QueueBuffer**).

The memory allocated for **PvBuffer** objects is based on the resolution of the image and the bit depth of the pixels (the payload) retrieved from the device using **PvDevice::GetPayloadSize**. The device returns the number of bytes required to hold one buffer, based on the configuration of the device.

When designing applications that deal with higher frame rate streams or that run on slower platforms, it may be necessary to increase the **BUFFER_COUNT** (to give you some margin for performance dips when you cannot process buffers fast enough for a short period). This allows the application to avoid a scenario where all buffers are in the output queue awaiting retrieval, and none are available in the input queue to store newly-received images.

C++

```
void CreateStreamBuffers( PvDevice *aDevice, PvStream *aStream )
{
    PvBuffer *lBuffers = NULL;

    // Reading payload size from device
    uint32_t lSize = aDevice->GetPayloadSize();

    // Use BUFFER_COUNT or the maximum number of buffers, whichever is smaller
    uint32_t lBufferCount = ( aStream->GetQueuedBufferMaximum() < BUFFER_COUNT ) ?
        aStream->GetQueuedBufferMaximum() :
        BUFFER_COUNT;

    // Allocate buffers
    lBuffers = new PvBuffer[ lBufferCount ];
    for ( uint32_t i = 0; i < lBufferCount; i++ )
    {
        ( lBuffers + i )->Alloc( static_cast<uint32_t>( lSize ) );
    }

    // Queue all buffers in the stream
    for ( uint32_t i = 0; i < lBufferCount; i++ )
    {
        aStream->QueueBuffer( lBuffers + i );
    }
}
```

The AcquireImages Function

In this function method, we acquire images from the device.

First the sample retrieves an array of GenICam features that will be used to control the device. These features are defined in the GenICam XML file that is present on all GigE Vision and USB3 Vision devices. Then, it maps two GenICam commands from the array to local variables that will be used later to start and stop the stream.

Next, it retrieves an array of GenICam features that represent the stream parameters. It maps two GenICam floating point values that represent stream statistics, which will later be used to display the data rate and bandwidth during image acquisition.

C++

```
void AcquireImages( PvDevice *aDevice, PvStream *aStream )
{
   // Get device parameters need to control streaming
   PvGenParameterArray *lDeviceParams = aDevice->GetParameters();

   // Map the GenICam AcquisitionStart and AcquisitionStop commands
   PvGenCommand *lStart = dynamic_cast<PvGenCommand *>( lDeviceParams->Get( "AcquisitionStart" ) );
   PvGenCommand *lStop = dynamic_cast<PvGenCommand *>( lDeviceParams->Get( "AcquisitionStop" ) );

   // Get stream parameters
   PvGenParameterArray *lStreamParams = aStream->GetParameters();

   // Map a few GenICam stream stats counters
   PvGenFloat *lFrameRate = dynamic_cast<PvGenFloat *>( lStreamParams->Get( "AcquisitionRate" ) );
   PvGenFloat *lBandwidth = dynamic_cast<PvGenFloat *>( lStreamParams->Get( "Bandwidth" ) );
   ...
```

To start the image stream, we enable streaming on the device (**PvDevice::StreamEnable**) and execute the GenICam **AcquisitionStart** command (**lStart**).

For GigE Vision devices, **StreamEnable** sets the **TLParamsLocked** feature, which prevents changes to the streaming related parameters during image acquisition.

For USB3 Vision devices, it sets the **TLParamsLocked** feature, configures the USB driver for streaming, and sets the stream enable bit on the device.

C++

```
         // Enable streaming and send the AcquisitionStart command
         cout << "Enabling streaming and sending AcquisitionStart command." << endl;
         aDevice->StreamEnable();
         lStart->Execute();
```

In the next section, we set up a doodle that will indicate to the user that images are being acquired. The doodle will animate every time a buffer is returned using **RetrieveBuffer** (regardless of whether we got an image or a timeout) until the user presses a key. We also initialize variables to access GenICam statistics (block count, acquisition rate, and bandwidth) that were retrieved earlier.

Next, we start the loop, retrieve the first **PvBuffer**, and check the results. When we retrieve the **PvBuffer** object, we remove it temporarily from the **PvStream** output buffer queue and process it. When processing is complete, we add the **PvBuffer** object back into the input buffer queue.

To verify that a buffer has been retrieved successfully from the stream object and to verify the acquisition of an image, we examine the two values supplied by **RetrieveBuffer**. First, we check the value of a **PvResult** object (**lResult**) to determine that a buffer has been retrieved. If a buffer has been retrieved, then it checks the value of the **PvResult** object (**lOperationResult**) to verify the acquisition operation (for example, it checks if the operation timed out, had too many resends, or was aborted).

C++

```
char lDoodle[] = "|\\-|-/";
int lDoodleIndex = 0;
double lFrameRateVal = 0.0;
double lBandwidthVal = 0.0;

// Acquire images until the user instructs us to stop.
cout << endl << "<press a key to stop streaming>" << endl;
while ( !PvKbHit() )
{
    PvBuffer *lBuffer = NULL;
    PvResult lOperationResult;

    // Retrieve next buffer
    PvResult lResult = aStream->RetrieveBuffer( &lBuffer, &lOperationResult, 1000 );
    if ( lResult.IsOK() )
    {
        if ( lOperationResult.IsOK() )
        {
            PvPayloadType lType;
```

Now that we have obtained a **PvBuffer** with an image, we display some general statistics retrieved from the device, including block ID, width, height, and bandwidth. This is the point at which your application would typically process the buffer. Then, we discard the image and requeue the **PvBuffer** object in the input queue by calling **PvStream::QueueBuffer**.

It is important to note that the stream may not contain an image, so we use the **PvPayloadType** enumeration to check that an image is included. For example, **PvPayloadType** can be **PvPayloadTypeImage**, **PvPayloadTypeUndefined** (an undefined or non-initialized payload type), or **PvPayloadTypeRawData**.

C++

```
    //
    // We now have a valid buffer. This is where you would typically process the buffer.
    // -----------------------------------------------------------------------------------------
    // ...

    lFrameRate->GetValue( lFrameRateVal );
    lBandwidth->GetValue( lBandwidthVal );

    // If the buffer contains an image, display width and height.
    uint32_t lWidth = 0, lHeight = 0;
    lType = lBuffer->GetPayloadType();

    cout << fixed << setprecision( 1 );
    cout << lDoodle[ lDoodleIndex ];
    cout << " BlockID: " << uppercase << hex << setfill( '0' ) << setw( 16 ) << lBuffer->GetBlockID();
    if ( lType == PvPayloadTypeImage )
    {
        // Get image specific buffer interface.
        PvImage *lImage = lBuffer->GetImage();

        // Read width, height.
        lWidth = lBuffer->GetImage()->GetWidth();
        lHeight = lBuffer->GetImage()->GetHeight();
        cout << "  W: " << dec << lWidth << " H: " << lHeight;
    }
    else {
        cout << " (buffer does not contain image)";
    }
    cout << "  " << lFrameRateVal << " FPS  " << ( lBandwidthVal / 1000000.0 ) << " Mb/s   \r";
}
...
```

If **lOperationalResult** returns something other than **OK**, a **PvBuffer** object has been retrieved, but it is not valid (for example, only part of the image could be retrieved or a timeout occurred). In this case, an error message is presented and we also re-queue the **PvBuffer** object back to the **PvStream** object so it can be used again.

C++

```
    else
    {
      // Non OK operation result
      cout << lDoodle[ lDoodleIndex ] << " " << lOperationResult.GetCodeString().GetAscii() << "\r";
    }

    // Re-queue the buffer in the stream object.
    aStream->QueueBuffer( lBuffer );
}

...
```

If **lResult** returns something other than **OK**, a **PvBuffer** object was not retrieved and therefore there is no **PvBuffer** to requeue. In this case the error message is also presented to the user.

C++

```
        else
        {
            // Retrieve buffer failure
            cout << lDoodle[ lDoodleIndex ] << " " << lResult.GetCodeString().GetAscii() << "\r";
        }

        ++lDoodleIndex %= 6;
    }
```

The remainder of the sample is used to stop acquisition and clean up resources when the user presses a key. First, we execute the GenICam **AcquisitionStop** command (**lStop**). Then we disable the stream.

For GigE Vision devices, **StreamDisable** resets the **TLParamsLocked** feature, which allows changes to the streaming related parameters to occur.

For USB3 Vision devices, **StreamDisable** resets the TLParamsLocked feature and sets the stream enable bit on the device.

C++

```
        PvGetChar(); // Flush key buffer for next stop.
        cout << endl << endl;

        // Tell the device to stop sending images.
        cout << "Sending AcquisitionStop command to the device" << endl;
        lStop->Execute();

        // Disable streaming on the device
        cout << "Disable streaming on the controller." << endl;
        aDevice->StreamDisable();
```

Now that streaming has stopped, we mark all of the buffers in the input queue as aborted (using **PvStream::AbortQueuedBuffers**), which moves the buffers to the output queue.

For **PvStreamGEV** objects, before resuming streaming after a pause, you should flush the queue using **PvStreamGEV::FlushPacketQueue**, which removes all unprocessed UDP packets from the data receiver.

C++

```
        // Abort all buffers from the stream and dequeue
        cout << "Aborting buffers still in stream" << endl;
        aStream->AbortQueuedBuffers();
        while ( aStream->GetQueuedBufferCount() > 0 )
        {
            PvBuffer *lBuffer = NULL;
            PvResult lOperationResult;

            aStream->RetrieveBuffer( &lBuffer, &lOperationResult );
        }
    }
```

If your application does not abort queued buffers, your application will receive timeout errors when you restart **PvStream**, since the buffers in the input queue will have exceeded the timeout value.

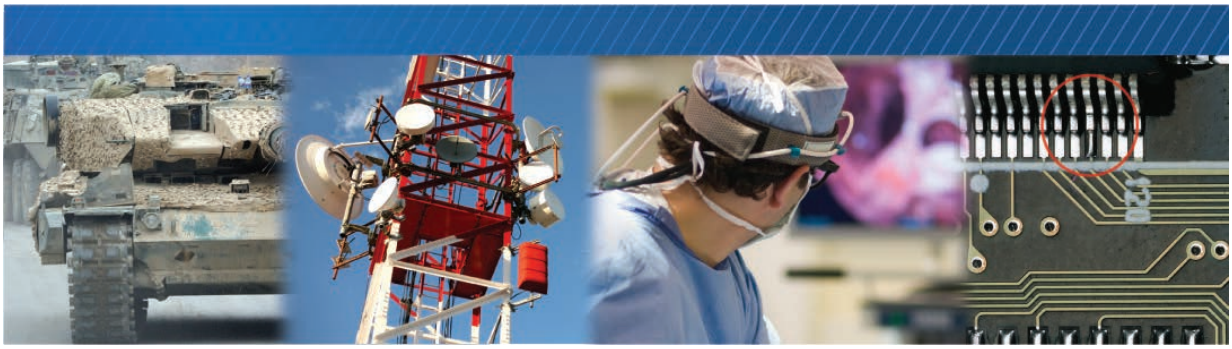
While our sample does not necessarily require that we abort and remove the buffers from the queue (because we do not restart **PvStream** in this sample), it is included in this sample to illustrate the concept of clearing buffers.

Finally, we remove all of the buffers from the queue (using **PvStream::RetrieveBuffer**) so they can be requeued the next time the stream is enabled.

C++

```
                // Abort all buffers in the stream and dequeue
                mStream.AbortQueuedBuffers();
                for (int i = 0; i < mStream.QueuedBufferCount; i++)
                {
                    lResult = mStream.RetrieveBuffer(ref lBuffer, ref lOperationResult);
                    if (lResult.IsOK)
                    {
                        lBuffer = null;
                    }
                }
            }
            catch (PvException lExc)
            {
                MessageBox.Show(lExc.Message, Text);
            }
        }
```

Chapter 6



Troubleshooting

This chapter provides you with troubleshooting tips and recommended solutions for issues that can occur when using the eBUS SDK C++ API.

Not all scenarios and solutions are listed here. You can refer to the Pleora Technologies Support Center at supportcenter.pleora.com for additional support and assistance. Details for creating a customer account are available on the Pleora Technologies Support Center.

Refer to the product release notes that are available on the Pleora Technologies Support Center for known issues and other product features.

Troubleshooting Tips

The scenarios and known issues listed in this chapter are those that you might encounter during the setup and operation of your device. Not all possible scenarios and errors are presented. The symptoms, possible causes, and resolutions depend upon your particular setup and operation.

If you perform the resolution for your issue and the issue is not corrected, we recommend you review the other resolutions listed in this table. Some symptoms may be interrelated.

Table 3: Troubleshooting Tips

Symptom	Possible cause	Resolution
SDK cannot detect or connect to the Pleora device	Power not supplied to the device, or inadequate power supplied	Both the detection and connection to the device will fail if adequate power is not supplied to the device. Verify that the Network LED is active. For information about the LEDs, see the documentation accompanying the device. Re-try the connection to the device with your application.
	The GigE Vision device is not connected to the network	Verify that the network LED is active. If this LED is illuminated, check the LEDs on your network switch to ensure the switch is functioning properly. If the problem continues, connect the device directly to the computer to verify its operation. For information about the LEDs, see the documentation accompanying the device.
	The GigE Vision device and computer are not on the same subnet	Images might not appear in your application if the GigE Vision device and the computer running your application are not on the same subnet. Ensure that these devices are on the same subnet. In addition, ensure that these devices are connected using valid gateway and subnet mask information. You can view the IP address information in the **Available Devices** list in your application. A red icon appears beside the device if there is an invalid IP configuration.
SDK cannot detect the API or transmitter	NIC that is receiving and NIC that is transmitting are on different subnets	Ensure the transmitting and receiving NICs are on the same subnet.
Errors appear	For GigE Vision devices, the drivers for your NIC may not be the latest version	Ensure you have installed the latest drivers from the manufacturer of your NIC.

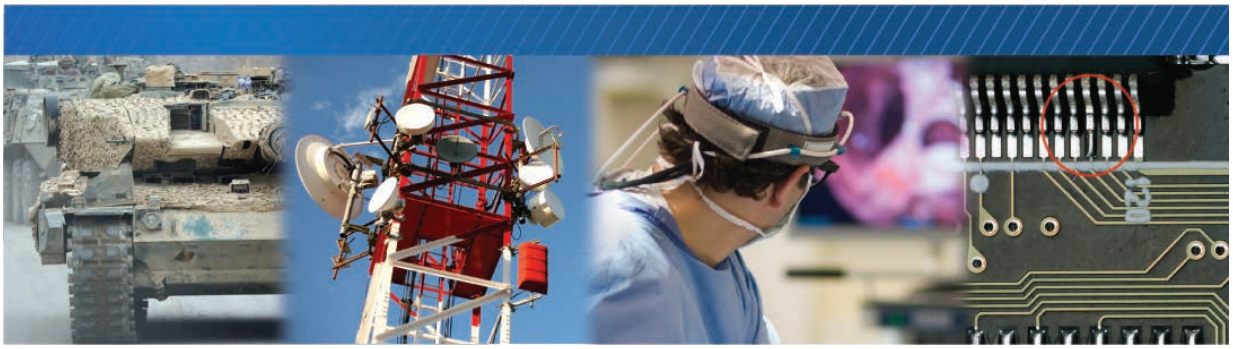
Table 3: Troubleshooting Tips (Continued)

Symptom	Possible cause	Resolution
SDK is able to connect, but no images appear in your application. In a multicast GigE Vision configuration, images appear on a display monitor connected to a vDisplay HDI-Pro External Frame Grabber but do not appear in your application.	In a multicast configuration, the device may not be configured correctly	Images only appear on the display if you have configured the device for a multicast network configuration. The device and all multicast receivers must have identical values for both the **GevSCDA** and **GevSCPHostPort** features in the **TransportLayerControl** section. For more information, see the documentation accompanying the device.
	In a multicast configuration, your computer's firewall may be blocking your application	Ensure that your application is allowed to communicate through the firewall.
	Anti-virus software or firewalls blocking transmission	Images might not appear in your application because of anti-virus software or firewalls on your network. Disable all virus scanning software and firewalls, and re-attempt a connection to the device with your application.
	Ensure jumbo packets are properly configured for the NIC	Enable jumbo packet support for the NIC and network switch (as required). If the NIC or network switch does not support jumbo packets, disable jumbo packets for the transmitter.

Table 3: Troubleshooting Tips (Continued)

Symptom	Possible cause	Resolution
Dropped packets: eBUS Player, or applications created using the eBUS SDK	Insufficient computer performance	The computer being used to receive images from the device may not perform well enough to handle the data rate of the image stream. The GigE Vision driver reduces the amount of computer resources required to receive images and is recommended for applications that require high throughput. Should the application continue to drop packets even after the installation of the GigE Vision driver, a computer with better performance may be required.
	Insufficient NIC performance	The NIC being used to receive images from the GigE Vision device may not perform well enough to handle the data rate of the image stream. For example, the bus connecting the NIC to the CPU may not be fast enough, or certain default settings on the NIC may not be appropriate for reception of a high-throughput image stream. Examples of NIC settings that may need to be reconfigured include the number of Rx Descriptors and the maximum size of Ethernet packets (jumbo packets). Additionally, some NICs are known to not work well in high-throughput applications. For information about maximizing the performance of your system, see the *Configuring Your Computer and Network Adapters for Best Performance Application Note*, available on the Pleora Support Center.

Chapter 7



Technical Support

On the Pleora Support Center, you can:

- Download the latest software and firmware.
- Log a support issue.
- View documentation for current and past releases.
- Browse for solutions to problems other customers have encountered.
- Read knowledge base articles for information about common tasks.

To visit the Pleora Support Center

- Go to supportcenter.pleora.com.

  Most material is available without logging in to a Support Center account. To access software and firmware downloads, in addition to other content, log in to the Support Center. If you do not have an account, click **Request Account**.

  Accounts are usually validated within one business day.