PLEORA TECHNOLOGIES INC.

eBUS SDK C++ API

eBUS SDK Version 4.0

Quick Start Guide

Pleora Technologies

Document Number

EX001-017-0012, Version 2.0 6/24/14

Table of Contents

Chapter 1

About this Guide

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

The following topics are covered in this chapter:

- "What this Guide Provides" on page 2
- "Related Documents" on page 2

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 C++ API Quick Start Guide* is complemented by the following guides:

- *eBUS Player Quick Start Guide*
- *eBUS Player User Guide*
- *eBUS SDK Programmer's Guide*
- *eBUS SDK C++ API Help File*
- *eBUS SDK for Linux Quick Start Guide*
- *eBUS SDK Licensing Application Note*
- *Configuring Your Computer and Network Adapters for Best Performance Application Note*

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" on page 4
- "eBUS SDK Licenses" on page 4

About the eBUS SDK C++ API

The eBUS SDK C++ API provides classes that allow you to build custom vision applications. The API is interoperable with both Pleora and third-party devices supporting the GigE Vision®, USB3 Vision™, and GenICam™ standards.

Sample code is provided that demonstrates how to use the API for device configuration and control, multicast communication, image and data acquisition, image display, and diagnostics.

This API can be used with the Windows® and Linux operating systems. For system requirements, see "System Requirements" on page 6.

eBUS SDK Licenses

While the eBUS SDK C++ API is a licensed product, you can trial the API without purchasing a license. However, the following limitations apply:

- Received images (received from third-party devices, such as non-Pleora enabled cameras) have an embossed watermark.
- Raw data is only received from Pleora-enabled cameras.

Understanding Licensing

Each license is associated to the MAC address of a NIC.

When you purchase a transmit license or a receive license, your Pleora representative will request the MAC address of a network interface card (NIC) in the workstation. Pleora includes the MAC address in the license file that your representative provides you with, which allows the eBUS SDK to accept the license. The MAC address is used to identify the workstation, and is required regardless of the protocol (GigE Vision or USB3 Vision).

Activating an eBUS SDK License

For detailed information about licensing, including details on activating a license, see the *eBUS SDK Licensing Application Note* available on the Pleora Technologies Support Center.

Chapter 3

Installing the eBUS SDK with the eBUS SDK C++ API

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

The instructions in this chapter are based on the Windows 7 operating system. The steps may vary depending on your computer's operating system.

The following topics are covered in this chapter:

- "System Requirements" on page 6
- "Installing the eBUS SDK" on page 7

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 2005, Visual Studio 2008, Visual Studio 2010, or Visual Studio 2012

    Sample project files (.vcxproj) are compatible with Visual Studio 2010 and Visual Studio 2012.
  - For the Linux operating system:
    - gcc 4.4.7 (or later) to compile non-GUI samples
    - QMake version 2.01a, with Qt version 4.6.2 to compile GUI-based samples
    - Kernel-devel package for your specific kernel version to compile the eBUS Universal Pro for Ethernet filter driver
- One of the following operating systems:
  - Microsoft® Windows 8, 32-bit or 64-bit
  - Microsoft Windows 7 with Service Pack 1 (or later), 32-bit or 64-bit
  - Microsoft Windows XP, 32-bit with Service Pack 3 (or later)*
  - Windows 2008 Server with Service Pack 3 (or later), 32-bit or 64-bit*
  - Red Hat Enterprise Linux 6, 32-bit or 64-bit
  - CentOS 6, 32-bit or 64-bit
  - Ubuntu 12.04 LTS, 32-bit or 64-bit

* This operating system is only supported for GigE Vision devices (USB3 Vision is not supported on Windows XP).

For supported USB 3.0 host controller chipsets, consult the *eBUS SDK 4.0 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 can simultaneously 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, it is included in the eBUS SDK installation package.

To install the eBUS SDK

- Follow the standard installation instructions to install the eBUS SDK on your computer.

  If you do not have the CD or USB stick, you can access installation files from the Pleora Support Center at www.pleora.com.

  Please note that runtime packages are also available, for easy integration into applications developed with the eBUS SDK.

  The runtime packages install the items that are required to run an application created with the eBUS SDK (including GigE Vision and USB3 Vision drivers). They exclude the items that are used to create applications with the eBUS SDK (such as header files, libraries, sample code, and API Help files).

Chapter 4

Using the Sample Code

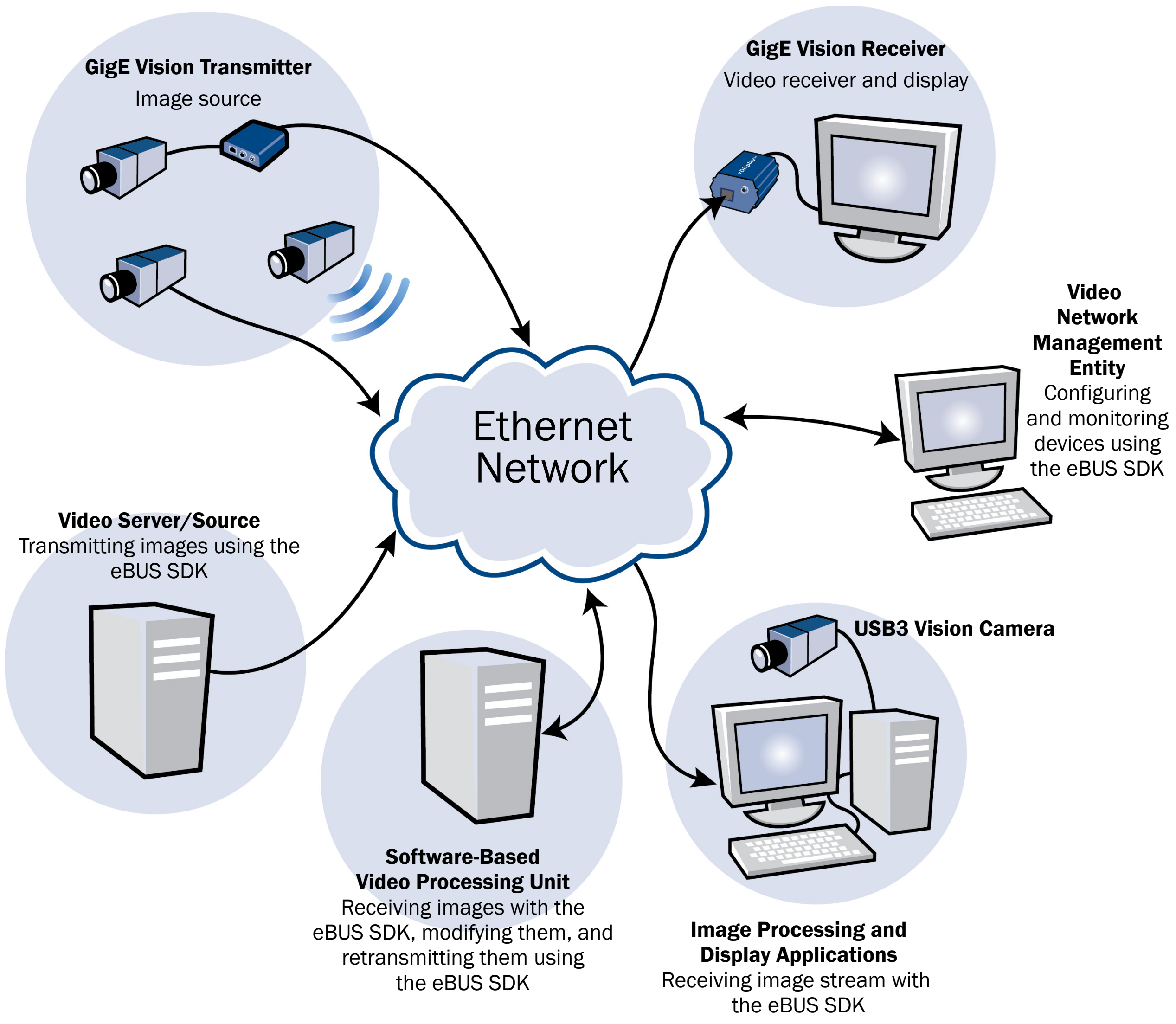
To illustrate how you can use the eBUS SDK C++ API to acquire and transmit images, the SDK includes sample code that you can use. This chapter provides a description of the sample code and provides general information about accessing the code to create sample applications.

The following topics are covered in this chapter:

- "Overview: System Components" on page 10
- "Description of Samples" on page 11

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 cameras.


GigE Vision Transmitter
Image source
GigE Vision Receiver
Video receiver and display
Video Network Management Entity
Configuring and monitoring devices using the eBUS SDK
Ethernet Network
Video Server/Source
Transmitting images using the eBUS SDK
USB3 Vision Camera
Software-Based Video Processing Unit
Receiving images with the eBUS SDK, modifying them, and retransmitting them using the eBUS SDK
Image Processing and Display Applications
Receiving image stream with the eBUS SDK

Description of Samples

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

Table 1: Sample Code

Sample code	Function	Type of application that is created
User interface components		
SimpleGUIApplication	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. This sample is only available for the Windows operating system.	UI-based.
Discovery and connection		
DeviceFinder for GigE Vision devices	Illustrates how to use the PvSystem, PvInterface, PvDeviceInfo, and PvDeviceFinderWnd classes to detect and enumerate devices.	Command line
ConnectionRecovery	Performs image acquisition with automatic recovery support from problems such as accidental disconnects, power interrupts, and so on. This sample is only available for the Windows operating system.	Command line
Configuration and event monitoring		
DeviceSerialPort	Provides serial communication through a Pleora device.	Command line
SerialBridge	Controls a serial device (usually a camera head) connected to a device from either a camera configuration application or a CLProtocol GenICam interface. This sample is only available for the Windows operating system.	Command line
GenICamParameters	Shows how to discover and access the features of a GenApi node map built from a GenICam XML file programmatically, using the PvGenParameterArray class.	Command line

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. This sample is only available for the Windows operating system.	UI-based
ConfigurationReader	Illustrates how to use the PvConfigurationReader and PvConfigurationWriter classes. It illustrates how to persist the state of your applications, devices, stream configuration, custom strings, and so on.	Command line
Image streaming		
PvPipelineSample	In step-by-step order, connects to a device, displays an image stream, stops streaming, and disconnects from the device. Illustrates how to use the PvPipeline class for image acquisition.	Command line
PvStreamSample	In step-by-step order, connects to a device, displays an image stream, stops streaming, and disconnects from the device. Illustrates how to use the PvStream class for image acquisition.	Command line
DualSource	Provides a user interface that lets you view image streams from two sources simultaneously. This sample is only available as a C++ sample for the Linux operating system.	UI-based
MultiSource for GigE Vision devices	Acquires images from a single or multi-source device, on all available sources.	Command line
DirectShowDisplay	Shows how to use the eBUS SDK DirectShow source filter to receive images from a Pleora device, which is also being controlled. **Note:** This sample assumes good knowledge of DirectShow and COM.	User interface (UI)-based
ImageProcessing	Demonstrates how to acquire an image and process it using an external buffer.	Command line

Table 1: Sample Code (Continued)

Sample code	Function	Type of application that is created
MulticastMaster for GigE Vision devices	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 MulticastSlave sample, which listens to the multicast stream.	Command line
MulticastSlave for GigE Vision devices	Receives an image stream as a multicast slave.* This sample is used in conjunction with the MulticastMaster sample, which initiates the multicast stream.	Command line
Integrated applications		
eBUS Player	Provides the source code for the eBUS Player application, which is used to detect, connect, and configure devices, and display and stream images.	UI-based
NetCommand for GigE Vision devices	Provides the source code for the NetCommand application, that is used to detect, connect, and configure multiple GigE Vision devices. A precompiled version of this sample is available in Program Files\Pleora Technologies Inc\eBUS SDK\Binaries. This sample is only available for the Windows operating system.	UI-based
Image transmission		
TransmitProcessedImage for GigE Vision devices	Receives images from a GigE Vision device, prints text on them, and transmits them to a given destination.	Command line. Only available for the C++ programming language.
TransmitTestPattern for GigE Vision devices	Transmits a test pattern to a given destination.	Command line
TransmitTiledImages for GigE Vision devices	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. A precompiled version of this sample is available in Program Files\Pleora Technologies Inc\eBUS SDK\Binaries. This sample is only available for the Windows operating system.	UI-based

* This sample requires an Ethernet switch and NIC that support Internet Group Management Protocol (IGMP) v2.

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 32-bit.** C:\Program Files\Pleora Technologies Inc\eBUS SDK\Samples
- **Windows 64-bit.** C:\Program Files (x86)\Pleora Technologies Inc\eBUS SDK\Samples
- **Linux.** /share/samples

We recommend that you 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.

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.

If you are using a USB3 Vision device on the Windows operating system, you must use Windows 7 (or later). USB 3.0 is not supported in earlier releases of Windows.

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 these Samples

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.

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 19.

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 Application Note*, available on the Pleora Support Center at www.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++

```cpp
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, 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.

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, GigE Vision, and USB3 Vision devices.

Not all scenarios and solutions are listed here. You can refer to the Pleora Technologies Support Center at www.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 the following table 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 network, 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.

Table 3: Troubleshooting Tips (Continued)

Symptom	Possible cause	Resolution
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.
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, NetCommand, 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.
- 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.