PLEORA TECHNOLOGIES INC.

eBUS SDK Python API
eBUS SDK Version 6.3

Quick Start Guide

Pleora Technologies

These products are not intended for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Pleora Technologies Inc. (Pleora) customers using or selling these products for use in such applications do so at their own risk and agree to indemnify Pleora for any damages resulting from such improper use or sale.

Trademarks

CoreGEV, PureGEV, eBUS, iPORT, vDisplay, AutoGEV, AutoGen, AI Gateway, eBUS Studio, and all product logos are trademarks of Pleora Technologies. Third party copyrights and trademarks are the property of their respective owners.

Notice of Rights

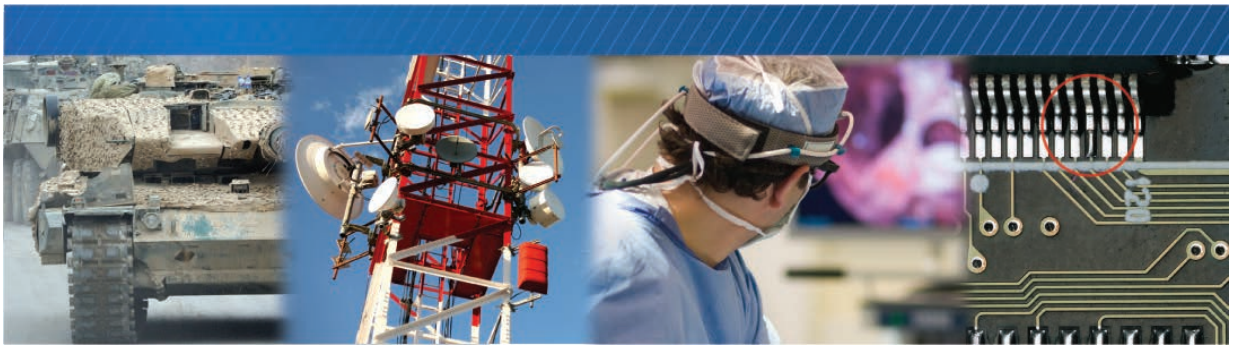
All information provided in this manual is believed to be accurate and reliable. No responsibility is assumed by Pleora for its use. Pleora reserves the right to make changes to this information without notice. Redistribution of this manual in whole or in part, by any means, is prohibited without obtaining prior permission from Pleora.

Document Number

EX001-017-0025 Version 1.0 03/24/23

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 Python API) and an overview of the system requirements.

You can use the Python sample applications to see how the Python 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 Python sample code to verify that your system is working properly (that is, determine whether there is a problem with your code or your equipment).

You can also consult the *eBUS SDK Python API* Help files for further information regarding the Python API itself. These help files are HTML based. On Windows, they can be found under the Python install directory:

PYTHON_INSTALLATION_PATH\Lib\site-packages\ebus-python\docs\index.html

On Linux, these files are co-located with the standard eBUS SDK files:

/opt/pleora/ebus_sdk/<distribution targeted>/share/doc/python/index.html

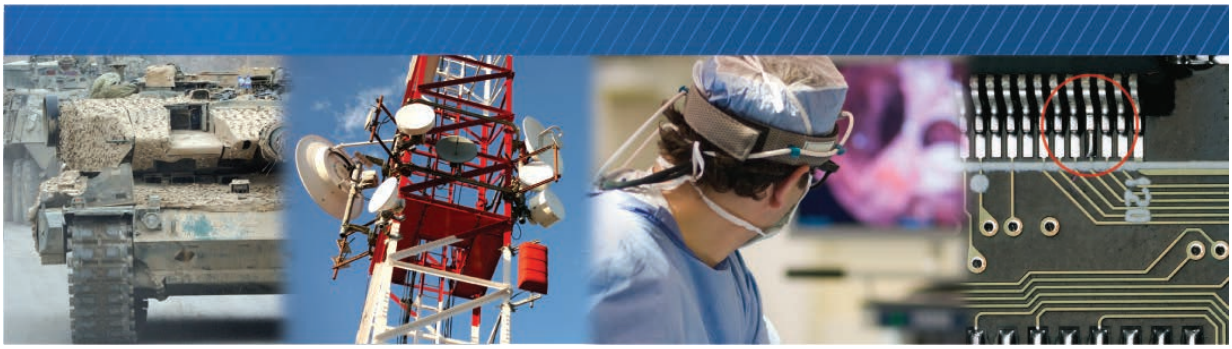
For troubleshooting information and technical support contact information for Pleora Technologies, see the last few chapters of this guide.

Related Documents

The *eBUS SDK Python 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 Python API Help File*
- *eBUS SDK for Linux Quick Start Guide*
- *Getting Started with eBUS Edge*
- *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 Python API

This chapter describes the eBUS SDK Python 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 Python.

The following topics are covered in this chapter:

About the eBUS SDK Python 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, 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 host computer 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

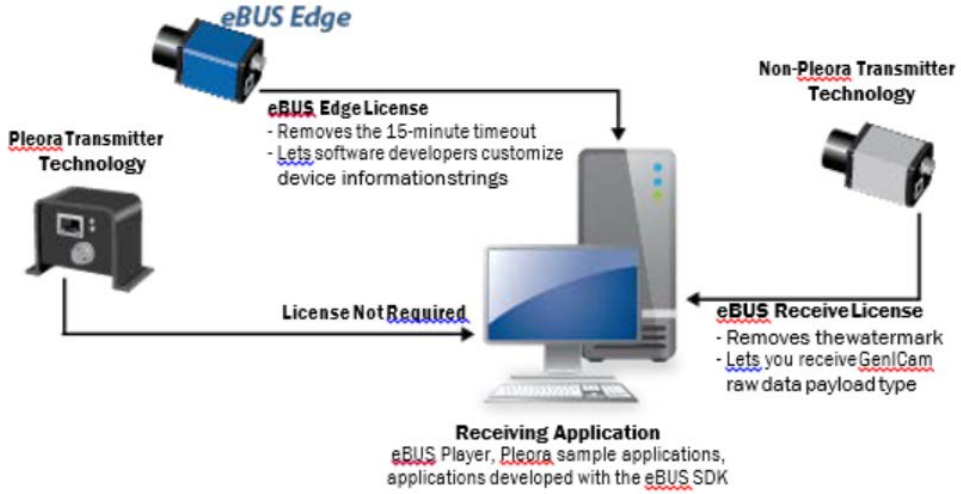
Some components of the eBUS SDK require a Pleora license to remove transmit and receive limitations.

eBUS Receiver License

When a license is not present and you are receiving images from third-party GigE Vision or USB3 Vision transmitter technology, an embossed Pleora watermark will appear on the images that you receive. In addition, you will not be able to receive the GigE Vision or USB3 Vision raw data payload type from the transmitting device.

GigE Vision Devices Created with the eBUS Edge API

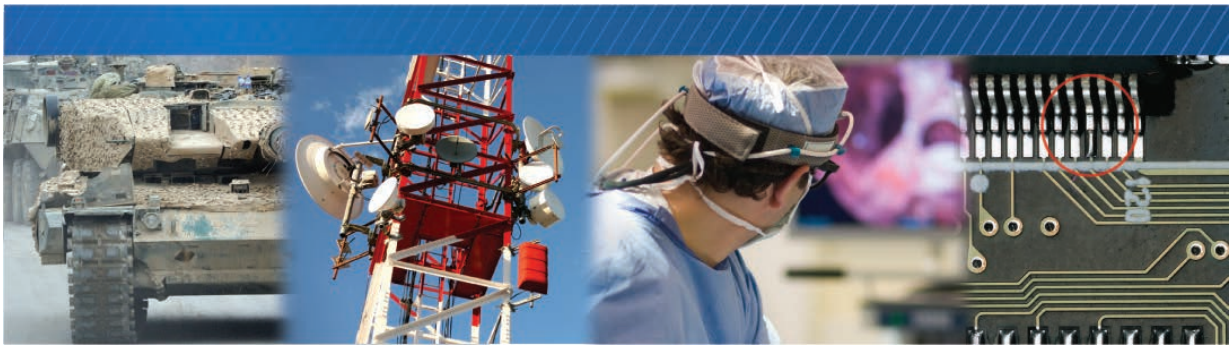
When an eBUS Edge license is not present on the eBUS Edge application side, the eBUS receiver application will automatically disconnect from the eBUS Edge device after 15 minutes. In addition, the eBUS Edge device will report hard-coded device information (such as the device model name), instead of your 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

eBUS SDK Python is packaged differently for Windows compared with Linux based systems. On Linux systems, eBUS SDK Python is incorporated into the traditional eBUS SDK install package and is therefore installed at the same time. On Windows, eBUS SDK Python is a stand alone package that needs to be installed separately after eBUS SDK has been installed.

The instructions in this chapter are based on the Windows 10 or Windows 11 operating system. The steps may vary depending on your computer's operating system.

The following topics are covered in this chapter:

System Requirements

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

- At least one Gigabit Ethernet NIC (if you are using GigE Vision devices).
- An appropriate compiler or integrated development environment (IDE):
  - Visual Studio Code.

One of the following operating systems:

- Microsoft Windows 11 64-bit
- Microsoft Windows 10, 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 running JetPack 4.6 (Ubuntu 18.04)
  - NVIDIA Jetson AGX Xavier, Jetson Xavier NX, Jetson AGX Orin, Jetson Orin NX running JetPack 5.1 (Ubuntu 20.04)
- For the x86 Linux and Linux ARM platforms, Qt is required to compile C++ 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 20.04 for ARM: Qt 5.12.8
  - For Ubuntu 18.04 Desktop: 64-bit Qt 5.9.5
  - For Ubuntu 18.04 for ARM: Qt 5.9.5
  - For RHEL 8.7, 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.

- For Linux x86/ARM platforms, we provide eBUS Python for the stock Python version of the OS:
  - For Ubuntu 22.04 Desktop (64-bit), eBUS Python for Python 3.10 is installed with the SDK
  - For Ubuntu 20.04 Desktop (64-bit), eBUS Python for Python 3.8 is installed with the SDK

- For Ubuntu 20.04 for ARM (64-bit), eBUS Python for Python 3.8 is installed with the SDK
- For Ubuntu 18.04 Desktop (64-bit), eBUS Python for Python 3.6 is installed with the SDK
- For Ubuntu 18.04 for ARM (64-bit), eBUS Python for Python 3.6 is installed with the SDK
- For RHEL 8 (64-bit), eBUS Python for Python 3.6 is installed with the SDK
- For CentOS 8 Stream (64-bit), eBUS Python for Python 3.6 is installed with the SDK

For non-default Python versions for different Linux ARM/x86 platforms, consult your Pleora support representative.

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.

Python Releases for Windows

For more information see https://www.python.org/downloads/windows/

Only the latest of each minor version is officially supported. The following Windows versions are supported:

- Windows 8.1, 10, and 11: Python 3.6, 3.7, 3.8, 3.9, and 3.10.
- Windows 7: Python 3.6, 3.7, and 3.8.

Python version 3.11 is not currently supported.

The following table indicates the Python releases for Windows applications:

Table 1: Supported Python releases for Windows Versions

Windows installer (64-bit)	Windows 8.1 and later	Windows 7
Python 3.10.9	Available	Not Available
Python 3.11.0	Available	Not Available
Python 3.9.13	Available	Not Available
Python 3.8.10	Available	Available
Python 3.7.9	Available	Available
Python 3.6.8	Available	Available

Installing the eBUS Python package on Windows

Since the eBUS SDK Python API is not part of the eBUS SDK on Windows, we must first install the base eBUS SDK using the Windows installation package, then install the appropriate eBUS Python package for the specific version of Python you are using. You can download the eBUS SDK and the associated eBUS Python packages 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).

Installation of eBUS-Python dependency packages

The following dependency packages are required for eBUS-Python:

1. Python (3.6, 3.7, 3.8, 3.9, and /or 3.10)
2. pip
3. numpy
4. opencv-python (optional for some samples)
5. eBUS-Python

1. How to install Python on Windows

Installing and using Python on Windows is very simple. The installation procedure involves just three steps:

1. Download the binaries (Python Releases for Windows | Python.org)
2. Run the executable installer.
3. Add Python to PATH environmental variables.

2. How to upgrade pip on Windows

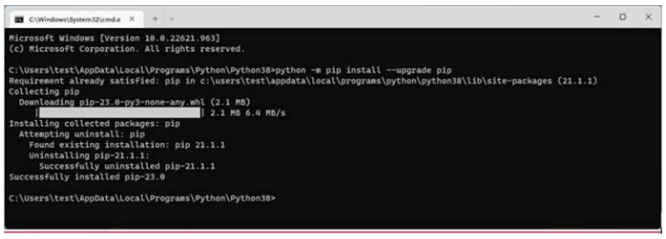
From a terminal, run the following command:

```
python -m pip install --upgrade pip
```

If you have only one Python package installed and you have added the Python in your PATH. You can call python everywhere.

If you have several Python packages installed, you should launch python from the installed location.

- From a terminal, run the following command if you use python 3.8
    - `C:\Users\<username>\AppData\Local\Programs\Python\Python38>python -m pip install --upgrade pip`



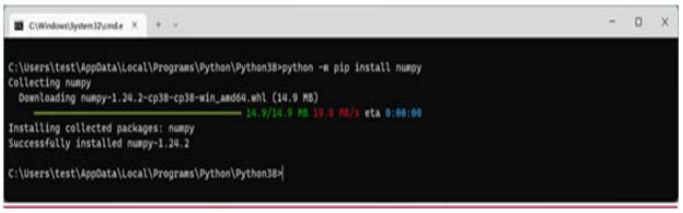
3. How to install numpy on Windows

From a terminal, run the following command:

```
python -m pip install numpy
```

If you have several Python packages installed, you need to specify the python executable.
From a terminal run the following command with the default Python path, if you use python **3.8**:

```
C:\Users\<username>\AppData\Local\Programs\Python\Python38>python -m pip install numpy
```



4. (optional) How to install opencv-python on Windows

From a terminal, run the following command:

```
python -m pip install opencv-python
```

If you have several Python packages installed, you need to specify the python executable. From a terminal, run the following command with the default Python installation path, if you use python **3.8**:

```
C:\Users\<username>\AppData\Local\Programs\Python\Python38>python -m pip install opencv-python
```

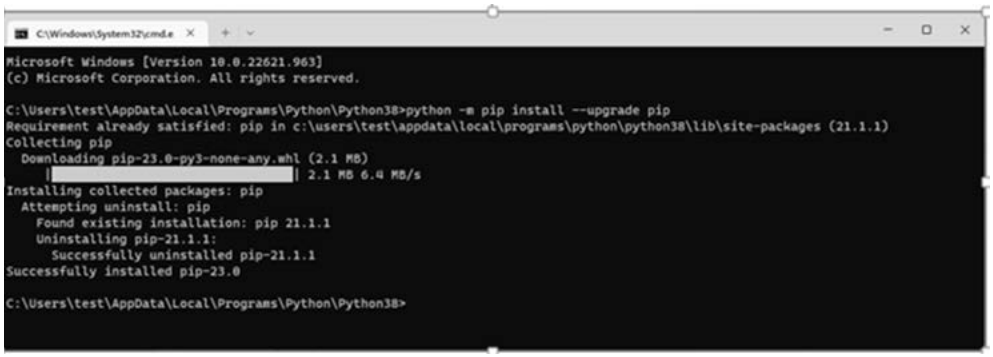
5. How to install eBUS-Python on Windows

From a terminal, run the following command:

```
python -m pip install <path of the package>\ebus_python-6.3.0-<build number>-py<python version>-none-win_amd64.whl
```

If you have several Python packages installed, you need to specify the python executable. From a terminal, run the following command with the default Python installation path, if you use python **3.8**:

C:\Users**<username>**\AppData\Local\Programs\Python**Python38**\python -m pip install **<path of the package>**\ebus_python-6.3.0-<build number>-**py38**-none-win_amd64.whl



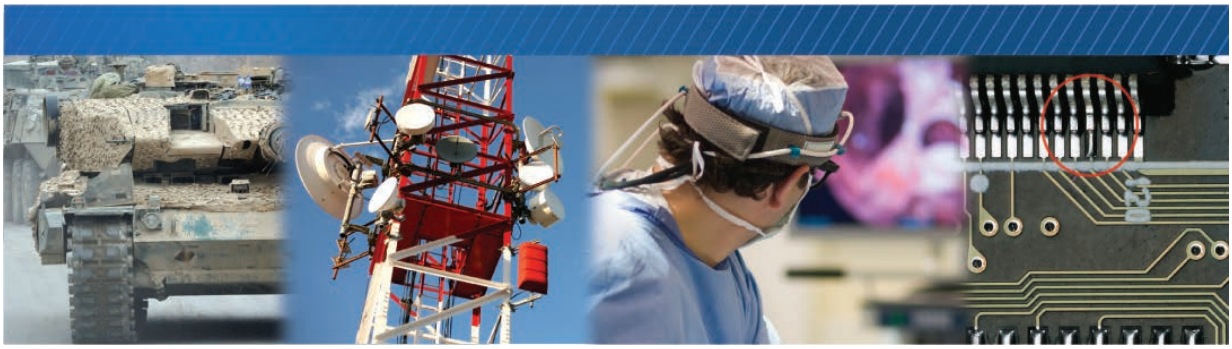
Location of installed eBUS-Python on Windows

C:\Users**<username>**\AppData\Local\Programs\Python**Python<Python version>**\Lib\site-packages\ebus-python

With the default installation path if you use python 3.8:

C:\Users**<username>**\AppData\Local\Programs\Python**Python38**\Lib\site-packages\ebus-python

Chapter 4



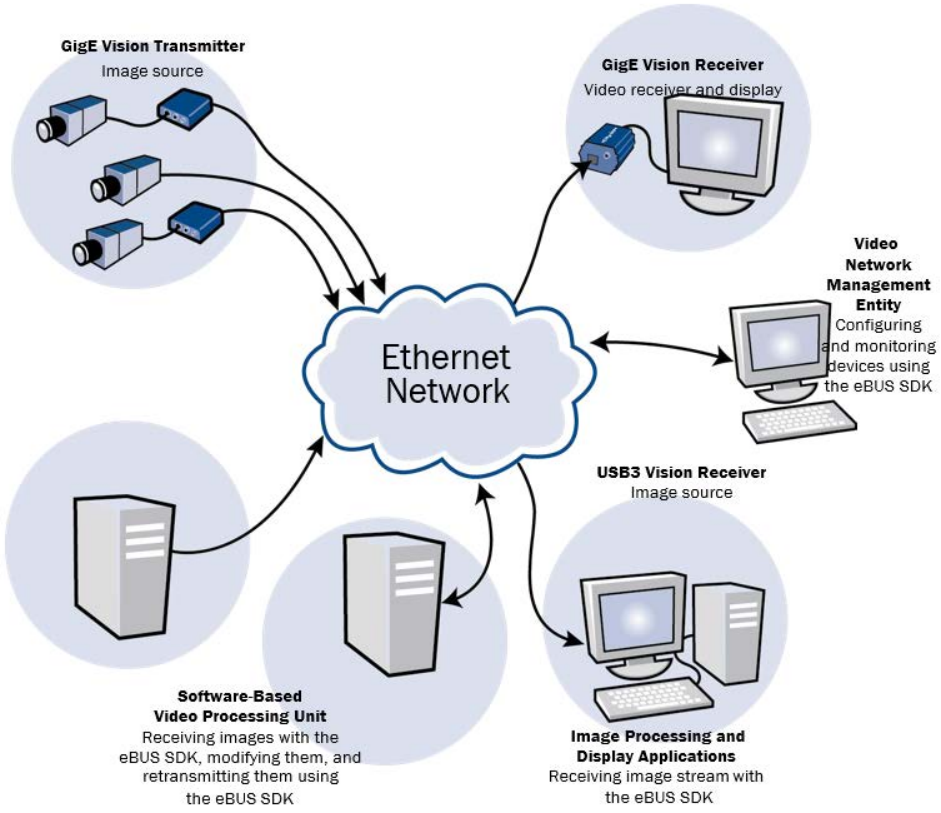
Using the Sample Code

To illustrate how you can use the eBUS SDK Python API to acquire and transmit images, the SDK includes sample code that you can use. This chapter provides a description of the sample code.

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

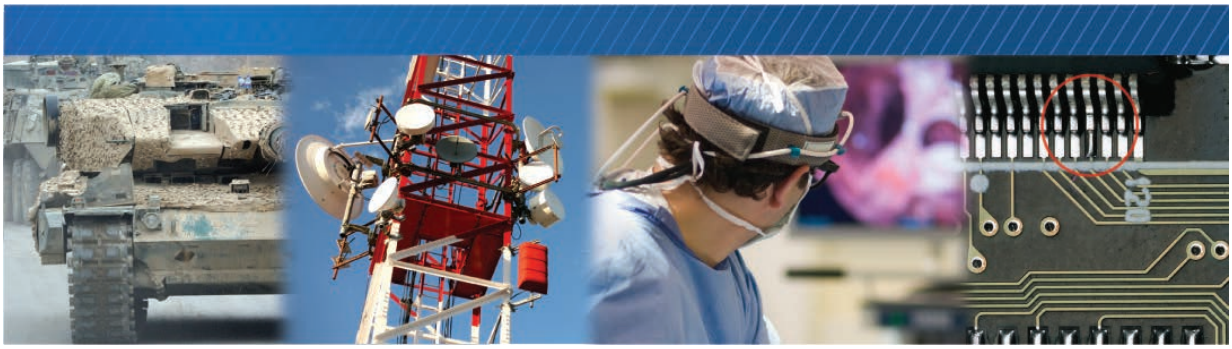
Table 2: Sample Code

Sample code	Function	Type of application that is created
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
Image Streaming		
ReceiveMultiPartSample	This sample shows how to connect to a GigE Vision device and receive a stream of multi-part payload type. The images that are acquired are displayed on screen using different windows. The SoftDeviceGEVMultiPart sample can be used to instantiate a software GigE Vision Device with a multi-part interface	Command line. • All platforms
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	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 • For GigE Vision devices only
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. • All platforms
Discovery and Connection		
DeviceFinder	This sample shows how to detect and enumerate GigE Vision and USB3 Vision devices on the network.	Command line. • All platforms

Table 2: Sample Code (Continued)

Sample code	Function	Type of application that is created
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. • All platforms
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
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
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. • All platforms
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. • All platforms
SoftDeviceGEVMultiPart	This sample shows how to use PvSoftDeviceGEV to create a software GigE Vision multi-part transmitter device.	Command line. • All platforms

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 Python Sample Code

The Python sample code is available in the following locations:

- **Windows.**

  You can access the Python sample code here:
  `PYTHON_INSTALLATION_PATH\Lib\site-packages\ebus-python\samples`

  For example, you can use the following formatted text when accessing the Python sample code. This information applies to the default installation options available for Python 3.10 on Windows 11:

  `C:\Users\<username>\AppData\Local\Programs\Python\Python310\Lib\site-packages\ebus-python\samples`

- **Linux.**

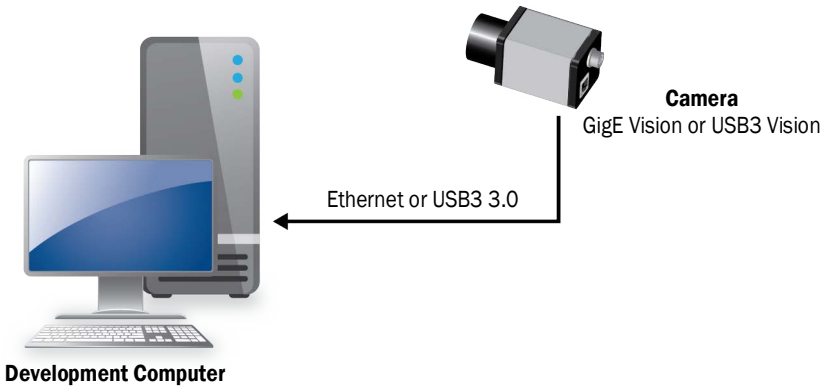
  You can access the Python sample code here:

  `opt/pleora/ebus_sdk/<distribution targeted>/share/samples/python/ebus`

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 Visual Studio Code. 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.



Windows and Linux Support

PvStreamSample can be used on the Windows and Linux operating systems. Platform-specific code is abstracted by **`PvSampleUtils.py`**, 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 3: 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.

Module Imports

The following modules are required by the sample. Please note that the `PvSampleUtils.py` module provides some basic helper and multi-platform routines:

Python

```
import numpy as np
import eBUS as eb
import lib.PvSampleUtils as psu
```

PvStreamSample

The **PvStreamSample** allows the user to select a device, connect to a device (**connect_to_device**), start the image stream (**Open_stream**, **configure_stream**, **configure_stream_buffers**), and process the image stream (**acquire_images**). 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.

kb.start(), kb.getch() and **kb.kbhit()** are platform-independent helper functions. This function is provided in the **PvSampleUtils** module.

Python

```
print("PvStreamSample:")

connection_ID = psu.PvSelectDevice()
if connection_ID:
    device = connect_to_device(connection_ID)
    if device:
        stream = open_stream(connection_ID)
        if stream:
            configure_stream(device, stream)
            buffer_list = configure_stream_buffers(device, stream)
           acquire_images(device, stream)
            buffer_list.clear()

            # Close the stream
            print("Closing stream")
            stream.Close()
            eb.PvStream.Free(stream);

        # Disconnect the device
        print("Disconnecting device")
        device.Disconnect()
        eb.PvDevice.Free(device)

print("<press a key to exit>")
kb.start()
kb.getch()
kb.stop()
```

The connect_to_device Function

connect_to_device 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 **PvDevice** object is returned and can now be used to control the device and initiate streaming.

Python

```
def connect_to_device(connection_ID):
    # Connect to the GigE Vision or USB3 Vision device
    print("Connecting to device.")
    result, device = eb.PvDevice.CreateAndConnect(connection_ID)
    if device == None:
        print(f"Unable to connect to device: {result.GetCodeString()} ({result.GetDescription()})")
    return device
```

The open_stream Function

open_stream 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 and can now be used to receive images as **PvBuffer** objects.

Python

```
def open_stream(connection_ID):
    # Open stream to the GigE Vision or USB3 Vision device
    print("Opening stream from device.")
    result, stream = eb.PvStream.CreateAndOpen(connection_ID)
    if stream == None:
        print(f"Unable to stream from device. {result.GetCodeString()} ({result.GetDescription()})")
 return stream
```

The configure_stream 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 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.

Python

```
def configure_stream(device, stream):
    # If this is a GigE Vision device, configure GigE Vision specific streaming parameters
    if isinstance(device, eb.PvDeviceGEV):
        # Negotiate packet size
        device.NegotiatePacketSize()
        # Configure device streaming destination
        device.SetStreamDestination(stream.GetLocalIPAddress(), stream.GetLocalPort())
```

The configure_stream_buffers Function

configure_stream_buffers 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.

Python

```
def configure_stream_buffers(device, stream):
    buffer_list = []
    # Reading payload size from device
    size = device.GetPayloadSize()

    # Use BUFFER_COUNT or the maximum number of buffers, whichever is smaller
    buffer_count = stream.GetQueuedBufferMaximum()
    if buffer_count > BUFFER_COUNT:
        buffer_count = BUFFER_COUNT

    # Allocate buffers
    for i in range(buffer_count):
        # Create new pvbuffer object
        pvbuffer = eb.PvBuffer()
        # Have the new pvbuffer object allocate payload memory
        pvbuffer.Alloc(size)
        # Add to external list - used to eventually release the buffers
        buffer_list.append(pvbuffer)

    # Queue all buffers in the stream
    for pvbuffer in buffer_list:
        stream.QueueBuffer(pvbuffer)
    print(f"Created {buffer_count} buffers")
    return buffer_list
```

The acquire_images Function

In the **acquire_images** 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.

Python

```
def acquire_images(device, stream):
    # Get device parameters need to control streaming
    device_params = device.GetParameters()

    # Map the GenICam AcquisitionStart and AcquisitionStop commands
    start = device_params.Get("AcquisitionStart")
    stop = device_params.Get("AcquisitionStop")

    # Get stream parameters
    stream_params = stream.GetParameters()

    # Map a few GenICam stream stats counters
    frame_rate = stream_params.Get("AcquisitionRate")
    bandwidth = stream_params[ "Bandwidth" ]
…
```

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

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.

Python

```
# Enable streaming and send the AcquisitionStart command
    print("Enabling streaming and sending AcquisitionStart command.")
    device.StreamEnable()
    start.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 (**result**) to determine that a buffer has been retrieved. If a buffer has been retrieved, then it checks the value of the **PvResult** object (**operational_result**) to verify the acquisition operation (for example, it checks if the operation timed out, had too many resends, or was aborted.)

Python

```
print("\n<press a key to stop streaming>")
kb.start()
while not kb.is_stopping():
    # Retrieve next pvbuffer
    result, pvbuffer, operational_result = stream.RetrieveBuffer(1000)
    if result.IsOK():
        if operational_result.IsOK():
            #
            # We now have a valid pvbuffer. This is where you would typically process the pvbuffer.
               # ----------------------------------------------------------------------------------
                        -------
               # ...

             result, frame_rate_val = frame_rate.GetValue()
             result, bandwidth_val = bandwidth.GetValue()

              print(f"{doodle[doodle_index]} BlockID: {pvbuffer.GetBlockID()}", end='')

             payload_type = pvbuffer.GetPayloadType()
```

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 **PvPayloadTypeChunk, PvPayloadTypeRawData,** or **PvPayloadTypeMultiPart.**

Python

```
if payload_type == eb.PvPayloadTypeImage:
    image = pvbuffer.GetImage
    image_data = image.GetDataPointer()
    print(f" W: {image.GetWidth()} H: {image.GetHeight()} ", end='')

    if opencv_is_available:
        if image.GetPixelType() == eb.PvPixelMono8:
           display_image = True
        if image.GetPixelType() == eb.PvPixelRGB8:
           image_data = cv2.cvtColor(image_data, cv2.COLOR_RGB2BGR)
           display_image = True

        if display_image:
        cv2.imshow("stream",image_data)
        else:
            if not warning_issued:
                # display a message that video only display for Mono8 / RGB8 images
                print(f" ")
        print(f" Currently only Mono8 / RGB8 images are displayed", end='\r')
                print(f"")
                warning_issued = True

        if cv2.waitKey(1) & 0xFF != 0xFF:
            break

  elif payload_type == eb.PvPayloadTypeChunkData:
      print(f" Chunk Data payload type with {pvbuffer.GetChunkCount()} chunks", end='')

  elif payload_type == eb.PvPayloadTypeRawData:
      print(f" Raw Data with {pvbuffer.GetRawData().GetPayloadLength()} bytes", end='')

  elif payload_type == eb.PvPayloadTypeMultiPart:
       print(f" Multi Part with {pvbuffer.GetMultiPartContainer().GetPartCount()} parts", end='')

  else:
      print(" Payload type not supported by this sample", end='')
```

If **operational_result** 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.

Python

```
            else:
                # Non OK operational result
                print(f"{doodle[ doodle_index ]} {operational_result.GetCodeString()}       ", end='\r')
        # Re-queue the pvbuffer in the stream object
        stream.QueueBuffer(pvbuffer)
…
```

If **result** 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.

Python

```
            else:
                # Retrieve pvbuffer failure
                print(f"{doodle[ doodle_index ]} {result.GetCodeString()}      ", end='\r')

        doodle_index = (doodle_index + 1) % 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 (**stop**). 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.

Python

```
kb.stop()
 if opencv_is_available:
    cv2.destroyAllWindows()

    # Tell the device to stop sending images.
    print("\nSending AcquisitionStop command to the device")
    stop.Execute()

    # Disable streaming on the device
    print("Disable streaming on the controller.")
    device.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.

Python

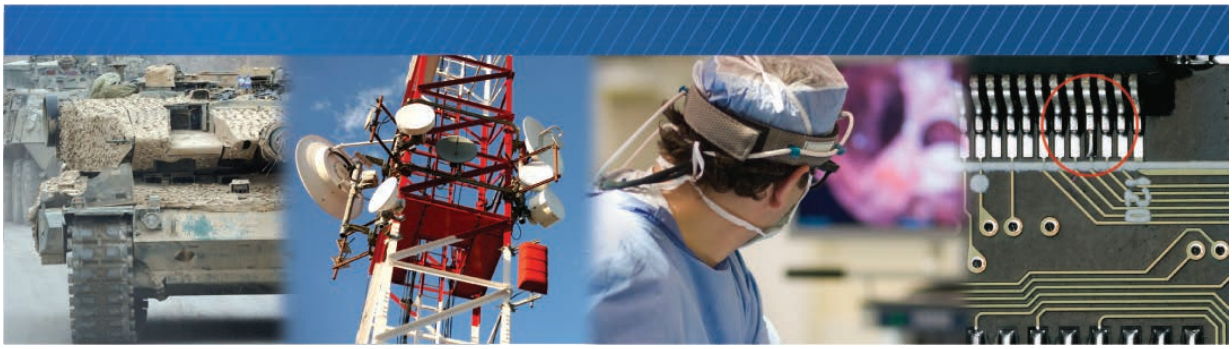
```
# Abort all buffers from the stream and dequeue
    print("Aborting buffers still in stream")
    stream.AbortQueuedBuffers()
    while stream.GetQueuedBufferCount() > 0:
       result, pvbuffer, lOperationalResult = stream.RetrieveBuffer()
```

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 Python 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 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 4: 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.

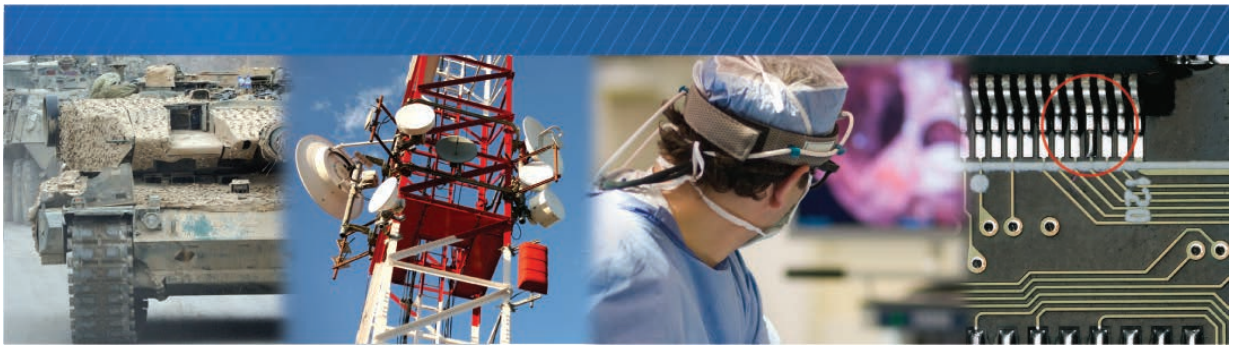
Table 4: Troubleshooting Tips (Continued)

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 4: 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.