Python API Manual

Introduction to the API

The Python API (VmbPy) is a Python wrapper around the VmbC API. It provides all functions from VmbC, but enables you to program with less lines of code.

We recommend using VmbPy for:

- Quick prototyping
- Getting started with programming machine vision or embedded vision applications
- Easy interfacing with deep learning frameworks and libraries such as OpenCV via NumPy arrays

Is this the best API for you?

Vimba X provides three APIs:

- The **Python API** is ideal for quick prototyping. We also recommend this API for an easy start with machine vision or embedded vision applications. For best performance and deterministic behavior, the C and C++ APIs are a better choice. To ease migration, the structure of VmbPy is very similar to VmbCPP.
- The **C API** is easy-to-use, but requires more lines of code than the Python API. It can also be used as API for C++ applications.
- The **C++ API** has an elaborate class architecture. It is designed as a highly efficient and sophisticated API for advanced object-oriented programming including the STL (standard template library), shared pointers, and interface classes. If you prefer an API with less design patterns, we recommend the C API.

All Vimba X APIs cover the following functions:

- Listing currently connected cameras
- Controlling camera features
- Receiving images from the camera
- Getting notifications about camera connections and disconnections

Compatibility

Supported operating systems and cameras are listed in the Release Notes for your operating system.

Compatible Python version: Python 3.7.x or higher. For 64-bit operating systems, we recommend using a 64-bit Python interpreter.

Installation

Prerequisites

Note

Python is not provided with the Vimba X SDK. To use the Python API, install Python version 3.7 or higher as described below.

Note

Updating Vimba X to a higher version does not automatically update any installed Vimba X Python site packages. Please install the .whl file of VmbPy as described below again manually, especially if the error “Invalid VmbC Version” occurs.

Installing Python - Windows

Tip

If your system requires multiple, coexisting Python versions, consider using pyenv-win, available at https://github.com/pyenv-win/pyenv-win to install and maintain multiple Python installations.

1. Download the latest Python release from python.org, available at https://www.python.org/downloads/windows/.
2. Execute the downloaded installer and ensure that the latest pip version is installed.

If you don’t have admin privileges for all directories, read the instructions for all operating systems below.

To verify the installation, open the command prompt and enter:

```
python --version
python -m pip --version
```

Please ensure that the Python version is 3.7 or higher and pip uses this Python version. Optionally, install NumPy and OpenCV as extras. NumPy enables conversion of `VmbPy.Frame` objects to numpy arrays. Opencv ensures that the numPy arrays are valid OpenCV images.

```
# Install Python with NumPy and OpenCV export
python -m pip install '/path/to/vmbpy-X.Y.Z-py-none-any.whl[numpy,opencv]'
```

Installing Python - Linux

On Linux systems, the Python installation process depends heavily on the distribution. If python3.7 is not available for your distribution or your system requires multiple python versions to coexist, use pyenv, available at https://realpython.com/intro-to-pyenv/ instead.

If you don't have admin privileges for all directories, read the instructions for all operating systems below.

1. Install or update python3.7 with the packet manager of your distribution.
2. Install or update pip with the packet manager of your distribution.

To verify the installation, open a console and enter:

```
python3 --version
python3 -m pip --version
```

Optionally, install NumPy and OpenCV as extras. NumPy enables conversion of `VmbPy.Frame` objects to numpy arrays. Opencv ensures that the numPy arrays are valid OpenCV images.

```
# Install Python with NumPy and OpenCV export
python3 -m pip install '/path/to/vmbpy-X.Y.Z-py-none-any.whl[numpy,opencv]'
```

Yocto on NXP i.MX8 and OpenCV

The GPU in the i.MX8 systems requires using the system-wide opencv-python package. When you create the Python environment, please use the `--system-site-packages` flag to include the system-wide OpenCV package.

If you don't set up a separate environment, a warning is shown during the VmbPy installation. You can ignore this warning.

Installing Python - macOS

1. Download the latest Python release from python.org, available at https://www.python.org/downloads/macos/.
2. Execute the downloaded installer.

Check if Python and pip are installed:

```
python3 --version
pip3 --version
```

Install VmbPy

VmbPy is provided as .whl file in the Vimba X installation directory. We recommed using this fle. You can install it with `pip install` (Windows) or `pip3 install` (some Linux distros and macOS).

pip documentation: https://pip.pypa.io/en/stable/cli/pip_install/

The sources of VmbPy are available on GitHub:

https://github.com/alliedvision/VmbPy

Note

Please note that Allied Vision can offer only limited support if an application uses a modified version of the API.

Additionally, you can install VmbPy from PyPI. Note that this does not include other essential Vimba X components such as the transport layers and drivers.

General aspects of the API

Entry point

The entry point of VmbPy is the Vimba X singleton representing the underlying Vimba X System.

Entity documentation

All entities of the Python API are documented via docstring.

Context manager

The Vimba X singleton implements a context manager. The context entry initializes:

- System features discovery
- Interface and transport layer (TL) detection
- Camera detection

The context entry always handles:

- API startup (including an optional method for advanced TL configuration) and shutdown
- Opening and closing cameras, interfaces, and TLs
- Feature discovery for the opened entity

Always call all methods for Camera, Feature, and Interface within the scope of a `with` statement:

```
from vmbpy import *
with VmbSystem.get_instance() as vmb:
    cams = vmb.get_all_cameras()
```

See also

For details about the optional method for advanced TL configuration, see the SDK Manual, chapter TL activation and deactivation and the *ListCameras* example.

Classes

The Camera class implements a context manager. On entering the Camera's context, all camera features are detected and can be accessed only within the `with` statement. Additionally to getting and setting camera features, the Camera class handles the camera access mode (default: Full Access).

Note

For changing the pixel format, always use the convenience functions instead of the camera feature, see section Changing the pixel format.

The Frame class stores raw image data and metadata of a single frame. The Frame class implements deepcopy semantics. Additionally, it provides methods for pixel format conversion and ancillary data access. Like all objects containing Features, AncillaryData implements a

context manager that must be entered before features can be accessed. The Frame class offers methods for NumPy and OpenCV export.

The following code snippet shows how to:

- Acquire a single frame
- Convert the pixel format to Mono8
- Store it using opencv-python

```
import cv2
from vmbpy import *

with VmbSystem.get_instance() as vmb:
    cams = vmb.get_all_cameras()
    with cams[0] as cam:
        frame = cam.get_frame()
        frame.convert_pixel_format(PixelFormat.Mono8)
        cv2.imwrite('frame.jpg', frame.as_opencv_image())
```

Tip

Optionally, this transformation can use a pre allocated *destination_buffer* to reduce possible overhead from memory allocations and garbage collection (see the *convert_pixel_format.py* example).

The Interface class contains all data of detected hardware interfaces that cameras are connected to. An Interface has associated features. The Interfaces can be queried from the VmbSystem and Interface features can be accessed within the `with` scope of VmbSystem. The Interface class does not implement a context manager on its own. The following code snippet prints all features of the first detected Interface.

```
from vmbpy import *

with VmbSystem.get_instance() as vmb:
    interface = vmb.get_all_interfaces()[0]
    for feat in interface.get_all_features():
        print(feat)
```

API usage

See also

For a quick start, we recommend using the examples.

Listing cameras

See also

To list available cameras, see the *list_ cameras.py* example

Cameras are detected automatically on context entry of the Vimba X instance. The order in which detected cameras are listed is determined by the order of camera discovery and therefore not deterministic. The discovery of GigE cameras may take several seconds. Before opening cameras, camera objects contain all static details of a physical camera that do not change throughout the object's lifetime such as the camera ID and the camera model.

Cameras and hardware interfaces such as USB can be detected at runtime by registering a callback at the Vimba X instance. The following code snippet registers a callable, creating a log message as soon as a camera or an interface is connected or disconnected. It runs for 10 seconds waiting for changes of the connected hardware.

```
from time import sleep
from vmbpy import *

@ScopedLogEnable(LOG_CONFIG_INFO_CONSOLE_ONLY)
def print_device_id(dev , state ):
    msg = 'Device: {}, State: {}'.format(str(dev), str(state ))
    Log.get_instance(). info(msg)

with VmbSystem.get_instance() as vmb:
    vmb.register_camera_change_handler(print_device_id)
    vmb.register_interface_change_handler(print_device_id)
    sleep(10)
```

Listing features

See also

To list the features of a camera and its physical interface, see the *list_ features.py* example.

Accessing features

As an example for reading and writing a feature, the following code snippet reads the current exposure time and increases it. Depending on your camera model and camera firmware, feature naming may be different.

```
from vmbpy import *

with VmbSystem.get_instance() as vmb:
    cams = vmb.get_all_cameras()
    with cams[0] as cam:
        exposure_time = cam.ExposureTime

        time = exposure_time.get()
        inc = exposure_time.get_increment()

        exposure_time.set(time + inc)
```

Acquiring images

The Camera class supports synchronous and asynchronous image acquisition. For high performance, acquire frames asynchronously and keep the registered callable as short as possible.

See also

The SDK Manual, section Synchronous and asynchronous image acquisition, provides background knowledge. The C API Manual, section Image Capture vs. Image Acquisition, provides detailed information about functions of the underlying C API.

To activate “alloc and announce” (optional): Use the optional parameter /x to overwrite `allocation_ mode`, see the *AsynchronousGrab* example.

```
# Synchronous grab
from vmbpy import *

with VmbSystem.get_instance() as vmb:
    cams = vmb.get_all_cameras()
    with cams[0] as cam:
        # Aquire single frame synchronously
        frame = cam.get_frame()

        # Aquire 10 frames synchronously
        for frame in cam.get_frame_generator(limit=10):
            pass
```

Acquire frames asychronously by registering a callable being executed with each incoming frame:

```python
# Asynchronous grab
import time
from vmbpy import *

def frame_handler(cam: Camera, stream: Stream, frame: Frame):
    cam.queue_frame(frame)

with VmbSystem.get_instance() as vmb:
    cams = vmb.get_all_cameras()
    with cams[0] as cam:
        cam.start_streaming(frame_handler)
        time.sleep(5)
        cam.stop_streaming()
```

The *asynchronous_ grab.py* example shows how to grab images and prints information about the acquired frames to the console.

The *asynchronous_ grab_ opencv.py* example shows how to grab images. It runs for 5 seconds and displays the images via OpenCV.

Changing the pixel format

Note

Always use the convenience functions instead of the PixelFormat feature of the Camera.

To easily change the pixel format, always use the convenience functions instead of the PixelFormat feature of the Camera. The convenience function `set_pixel_format(fmt)` changes the Camera pixel format by passing the desired member of the `PixelFormat` enum. When using the PixelFormat feature (not recommended), a correctly pre-formatted string has to be used instead.

Before image acquisition is started, you can get and set pixel formats within the Camera class:

```python
# Camera class methods for getting and setting pixel formats
# Apply these methods before starting image acquisition

get_pixel_formats()   # returns a tuple of all pixel formats supported by the camera
get_pixel_format()    # returns the current pixel format
set_pixel_format(fmt) # enables you to set a new pixel format
```

Note

The pixel format cannot be changed while the camera is acquiring images.

After image acquisition in the camera, the Frame contains the pixel format of the camera. Now you can convert the pixel format with the `convert_ pixel_ format()` method.

See also

See the *AsynchronousGrab* example, it contains a pixel format conversion.

Listing chunk data

Note

To use the *chunk* feature, make sure your camera supports it.

Chunk data are image metadata such as the exposure time that are available in the Frame. To activate chunk, see the user documentation of your camera.

```
# Before using chunk, open your camera as usual
def chunk_callback(features: FeatureContainer):
    chunk_timestamp = features.ChunkTimestamp.get()
    print(f'Chunk callback executed, ChunkTimestamp={chunk_timestamp}')

def frame_callback(cam: Camera, stream: Stream, frame: Frame):
    print(f'Frame callback executed for {frame}')

    # Calling this method only works if chunk mode is activated!
    frame.access_chunk_data(chunk_callback)
    stream.queue_frame(frame)

try:
    cam.start_streaming(frame_callback),
    time.sleep(1)
finally:
    cam.stop_streaming()
```

See also

The *list_ chunk_ data.py* example shows in detail how to list chunk data such as the frame count or feature values such as the exposure time. See List Chunk data.

Loading and saving user sets

See also

To save the camera settings as a user set in the camera and load it, use the *user_ set.py* example.

Loading and saving settings

Additionally to the user sets stored in the camera, you can save the feature values as an XML file to your host PC. For example, you can configure your camera with Vimba X Viewer, save the settings, and load them with any Vimba X API.

See also

See the *load_ save_ settings.py* example, see Load and save settings.

Software trigger

Software trigger commands are supported by all Allied Vision cameras. To get started with triggering and explore the possibilities, you can use Vimba X Viewer. To program a software trigger application, use the following code snippet.

```
# Software trigger for continuous image acquisition
import time
from vmbpy import *

def handler(cam: Camera, stream: Stream, frame: Frame):
    print('Frame acquired: {}'.format(frame), flush=True)
    cam.queue_frame(frame)

def main():
    with VmbSystem.get_instance() as vmb:
        cam = vmb.get_all_cameras()[0]

        with cam:
            cam.TriggerSource.set('Software')
            cam.TriggerSelector.set('FrameStart')
            cam.TriggerMode.set('On')
            cam.AcquisitionMode.set('Continuous')

            try:
                cam.start_streaming(handler)
                time.sleep(1)
                cam.TriggerSoftware.run()
                time.sleep(1)
                cam.TriggerSoftware.run()
                time.sleep(1)
                cam.TriggerSoftware.run()
            finally:
                cam.stop_streaming()

if __name__ == '__main__':
    main()
```

Trigger over Ethernet - Action Commands

You can broadcast a trigger signal simultaneously to multiple GigE cameras via GigE cable. Action Commands must be set first to the camera(s) and then to the API, which sends the Action Commands to the camera(s).

See also

Find more details in the application note: Trigger over Ethernet (ToE) - Action Commands

Multithreading

To get started with multithreading, use the *multithreading_opencv.py* example, see Multithreading OpenCV. You can use the example with one or multiple cameras. The FrameConsumer thread displays images of the first detected camera via OpenCV in a window of 480 x 480 pixels, independent of the camera's image size. The example automatically constructs, starts, and stops FrameProducer threads for each connected or disconnected camera.

Migrating to the C or C++ API

The Python API is optimized for quick and easy prototyping. To migrate to the C API, we recommend using VmbPy's extensive logging capabilities. In the log file, the order of operations is the same as in the C API. Migrating to the C++ API is eased by similar names of the functions and by a similar API structure.

Troubleshooting

Frequent questions:

- To use the VmbPy API, the installation of a compatible C API version and Image Transform version is required. To check the versions, use `VmbSystem.get_ version()`.
- Error: "Invalid VmbC Version" although the correct C API version is installed: Updating Vimba X does not automatically update any installed VmbPy site packages. Please perform the installation again manually.
- For changing the pixel format, always use the convenience functions instead of the camera feature, see section Changing the pixel format.
- For general issues, see Troubleshooting.

Logging

You can enable and configure logging to:

- Create error reports

- Prepare the migration to the C API or the C++ API

Tip

If you want to send a log file to our Technical Support team, always use logging level *Trace*.

Logging levels

The Python API offers several logging levels. The following code snippet shows how to enable logging with level Warning. All messages are printed to the console.

```
from vmbpy import *

with VmbSystem.get_instance() as vmb:
    vmb.enable_log(LOG_CONFIG_WARNING_CONSOLE_ONLY)

    log = Log.get_instance()
    log.critical('Critical, visible')
    log.error('Error, visible')
    log.warning('Warning , visible')
    log.info('Info, invisible')
    log.trace('Trace, invisible')

    vmb.disable_log()
```

The logging level *Trace* enables the most detailed reports. Additionally, you can use it to prepare the migration to the C API or the C++ API. *Trace* is always used with the `TraceEnable()` decorator. The decorator adds a log entry of level *Trace* as soon as the decorated function is called. In addition, a log message is added on function exit. This log message shows if the function exit occurred as expected or with an exception.

See also

To create a trace log file, use the *create_ trace_ log.py* example, see Create Trace Log.

All previous examples enable and disable logging globally via the VmbSystem object. For more complex applications, this may cause large log files. The `ScopedLogEnable()` decorator allows enabling and disabling logging on function entry and exit. The following code snippet shows how to use `TraceEnable()` and `ScopedLogEnable()`.

```
from vmbpy import *

@TraceEnable()
def traced_function():
    Log.get_instance(). info('Within Traced Function')

@ScopedLogEnable(LOG_CONFIG_TRACE_CONSOLE_ONLY)
def logged_function():
    traced_function()

logged_function()
```