



Vimba

Vimba Python Manual - Beta Release

0.2.1

Legal Notice

Trademarks

Unless stated otherwise, all trademarks appearing in this document are brands protected by law.

Warranty

The information provided by Allied Vision is supplied without any guarantees or warranty whatsoever, be it specific or implicit. Also excluded are all implicit warranties concerning the negotiability, the suitability for specific applications or the non-breaking of laws and patents. Even if we assume that the information supplied to us is accurate, errors and inaccuracy may still occur.

Copyright

All texts, pictures and graphics are protected by copyright and other laws protecting intellectual property.

All rights reserved.

Headquarters:
Allied Vision Technologies GmbH
Taschenweg 2a
D-07646 Stadtroda, Germany
Tel.: +49 (0)36428 6770
Fax: +49 (0)36428 677-28
e-mail: info@alliedvision.com

Contents

1	Contacting Allied Vision	4
2	Document history and conventions	5
2.1	Document history	6
2.2	Conventions used in this manual	6
2.2.1	Styles	6
2.2.2	Symbols	6
3	Purpose and scope of the API	8
3.1	Compatibility	9
3.2	Prerequisites	9
4	Introduction to the API	10
4.1	General aspects of the API	10
4.2	Classes	10
5	API usage	12
5.1	Listing cameras	12
5.2	Listing features	12
5.3	Accessing features	13
5.4	Acquiring images	13
5.5	Changing the pixel format	14
5.6	Listing ancillary data	15
5.7	Loading and saving user sets	16
5.8	Loading and saving settings	16
5.9	Handling events	16
5.10	Software trigger	16
5.11	Trigger over Ethernet - Action Commands	17
5.12	Multithreading	18
6	Migrating to the Vimba C or C++ API	19
7	Troubleshooting	20
7.1	Camera settings	20
7.1.1	GigE cameras	20
7.1.2	USB cameras	20
7.1.3	Goldeye CL cameras	20
7.2	Logging	21
7.2.1	Logging levels	21
7.3	Other issues	22

1 Contacting Allied Vision

Contact information on our website

<https://www.alliedvision.com/en/meta-header/contact-us>

Find an Allied Vision office or distributor

<https://www.alliedvision.com/en/about-us/where-we-are>

Email

info@alliedvision.com

support@alliedvision.com

Sales Offices

EMEA: +49 36428-677-230

North and South America: +1 978 225 2030

California: +1 408 721 1965

Asia-Pacific: +65 6634-9027

China: +86 (21) 64861133

Headquarters

Allied Vision Technologies GmbH

Taschenweg 2a

07646 Stadtroda

Germany

Tel: +49 (0)36428 677-0

Fax: +49 (0)36428 677-28

Managing Directors (Geschäftsführer): Andreas Gerke, Peter Tix

2 Document history and conventions



This chapter includes:

2.1	Document history	6
2.2	Conventions used in this manual	6
2.2.1	Styles	6
2.2.2	Symbols	6

2.1 Document history

Version	Date	Changes
0.0.1	November 2019	First draft for internal use
0.1.0	November 2019	Beta release version for customer feedback
0.2.1	January 2020	Beta release. Changed feature naming, changed <code>get_frame_iter</code> to <code>get_frame_generator</code> , added software trigger, several minor improvements

2.2 Conventions used in this manual

To give this manual an easily understood layout and to emphasize important information, the following typographical styles and symbols are used:

2.2.1 Styles

Style	Function	Example
Emphasis	Programs, or highlighting important things	Emphasis
Publication title	Publication titles	<i>Title</i>
Web reference	Links to web pages	Link
Document reference	Links to other documents	Document
Output	Outputs from software GUI	Output
Input	Input commands, modes	<i>Input</i>
Feature	Feature names	Feature

2.2.2 Symbols



Practical Tip

**Safety-related instructions to avoid malfunctions**

Instructions to avoid malfunctions

**Further information available online**

3 Purpose and scope of the API

Vimba's Python API is a Python Wrapper around the Vimba C API. It provides all functions from the Vimba C API, but enables you to program Allied Vision cameras with less lines of code.

We recommend using the Vimba Python API 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 provides four APIs:

- The **Vimba 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.
- The **Vimba 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 **Vimba 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 Vimba C API.
- The **Vimba .NET API** supports all .NET languages. Its general concept is similar to the C++ API.

All Vimba APIs cover the following functions:

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

3.1 Compatibility

Supported cameras

All Allied Vision cameras

Compatible Python version

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

Tested operating systems

We have tested the Beta version with Windows 10 and Ubuntu 18.04 LTS.

Compatible Vimba version

Vimba 3.1 (contains C API version 1.8.1).

3.2 Prerequisites

To use the Vimba Python API beta version, install:

- Python 3.7 or higher
- Vimba 3.1 (contains C API version 1.8.1). If you choose Custom Installation, make sure the Vimba C API, Vimba Image Transform, and the transport layers for your cameras are installed.
- NumPy and OpenCV (optional)



Detailed installation instructions are available in the README, especially if you have installed multiple Python versions or use an ARM board.

4 Introduction to the API

4.1 General aspects of the API

Entry point

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

Entity documentation

All entities of the Vimba Python API are documented via docstring.

Context manager

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

- System features discovery
- Interface detection
- Camera detection

The context entry always handles:

- Vimba API startup and shutdown
- Opening and closing cameras, interfaces, and ancillary data
- Feature discovery for the opened entity

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

```
from vimba import *
with Vimba.get_instance() as vimba:
    cams = vimba.get_all_cameras()
```

4.2 Classes

Camera

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

Frame

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 vimba import *

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

Interface

The Interface class contains all data of detected hardware interfaces cameras are connected to. An Interface has associated features and implements a context manager as well. On context entry, all features are detected and can be accessed within the `with` statement scope. The following code snippet prints all features of the first detected Interface.

```
from vimba import *

with Vimba.get_instance() as vimba:
    inters = vimba.get_all_interfaces()
    with inters[0] as interface:
        for feat in interface.get_all_features():
            print(feat)
```

5 API usage

For a quick start, we recommend using the code examples.

5.1 Listing cameras

To list available cameras, see the `list_cameras.py` example. Cameras are detected automatically on context entry of the Vimba 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.

Plug and play

Cameras and hardware interfaces such as USB can be detected at runtime by registering a callback at the Vimba 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.



The Camera Link specification doesn't support plug and play. In this case, changes to the camera list cannot be detected while Vimba is running.

```
from time import sleep
from vimba 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)

vimba = Vimba.get_instance()
vimba.register_camera_change_handler(print_device_id)
vimba.register_interface_change_handler(print_device_id)

with vimba:
    sleep(10)
```

5.2 Listing features

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

5.3 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 vimba import *

with Vimba.get_instance() as vimba:
    cams = vimba.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)
```

5.4 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.



The Vimba Manual, section *Synchronous and asynchronous image acquisition*, provides background knowledge. The Vimba C API Manual, chapter Image Capture (API) and Acquisition, provides detailed information about functions of the underlying C API.

```
# Synchronous grab
from vimba import *

with Vimba.get_instance() as vimba:
    cams = vimba.get_all_cameras()
    with cams[0] as cam:
        # Acquire single frame synchronously
        frame = cam.get_frame()

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

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

```
# Asynchronous grab
import time
from vimba import*

def frame_handler(cam, frame):
    cam.queue_frame(frame)

with Vimba.get_instance() as vimba:
    cams = vimba.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.

5.5 Changing the pixel format

Convenience functions

To easily change the pixel format, the Vimba Python API offers convenience functions. The Camera pixel format is changed via the `PixelFormat` enum. The Frame is changed via the `VimbaPixelFormat` enum. In both cases, the Vimba Python API uses `VimbaPixelFormat`. Always use the convenience functions instead of the `PixelFormat` camera feature.

Getting and setting pixel formats

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

```
# 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
```



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

Converting a pixel format

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.

The following code snippet shows how to query a pixel format and apply it to the camera:

```
from vimba import *

with Vimba.get_instance() as vimba:
    cams = vimba.get_all_cameras()
    with cams[0] as cam:

        # Get pixel formats available in the camera
        fmts = cam.get_pixel_formats()

        # In this case, we want a format that supports colors
        fmts = intersect_pixel_formats(fmts, COLOR_PIXEL_FORMATS)

        # In this case, we want a format that is compatible with OpenCV
        fmts = intersect_pixel_formats(fmts, OPENCV_PIXEL_FORMATS)

        if fmts:
            cam.set_pixel_format(fmts[0])

        else:
            print('Abort. No valid pixel format found.')
```

The following code snippet shows how to:

- Acquire a single frame
- Convert the pixel format to Mono with 8-bit depth
- Save the frame as JPG using opencv-python

```
import cv2
from vimba import *

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

5.6 Listing ancillary data

The `list_ancillary_data.py` example shows how to list ancillary data such as the frame count or feature values such as the exposure time.

5.7 Loading and saving user sets

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

5.8 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 Viewer, save the settings, and load them with any Vimba API. To do this, use the `load_save_settings.py` example.

5.9 Handling events

To get notifications about feature changes, use the `event_handling.py` example (for GigE cameras only).

5.10 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 Viewer. To program a software trigger application, use the following code snippet.


```
# Software trigger for continuous image acquisition

import time
from vimba import *

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

def main():
    with Vimba.get_instance() as vimba:

        cam = vimba.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()
```

5.11 Trigger over Ethernet - Action Commands

Selected GigE cameras with the latest firmware support Action Commands. With 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). To learn more about Action Commands, see the `action_commands.py` example and read the application note [Trigger over Ethernet - Action Commands](#).

5.12 Multithreading

To get started with multithreading, use the `multithreading_opencv.py` example. 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.

6 Migrating to the Vimba C or C++ API

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

7 Troubleshooting

7.1 Camera settings

7.1.1 GigE cameras

Make sure to set the *PacketSize* feature of GigE cameras to a value supported by your network card. If you use more than one camera on one interface, the available bandwidth has to be shared between the cameras.

- *GVSPAdjustPacketSize* configures GigE cameras to use the largest possible packets.
- *DeviceThroughputLimit* (or *StreamBytesPerSecond*) configure the individual bandwidth if multiple cameras are used.
- The maximum packet size might not be available on all connected cameras. Try to reduce the packet size.

More information:

The Technical Manual of your camera provides detailed information on how to configure your system.

<https://www.alliedvision.com/en/support/technical-documentation.html>

7.1.2 USB cameras

Under Windows, make sure the correct driver is applied. For more details, see Vimba Manual, chapter Vimba Driver Installer.

To achieve best performance, see the technical manual of your USB camera, chapter Troubleshooting:

<https://www.alliedvision.com/en/support/technical-documentation.html>

7.1.3 Goldeye CL cameras

- The pixel format, all features affecting the image size, and *DeviceTapGeometry* must be identical in Vimba and the frame grabber software.
- Make sure to select an image size supported by the frame grabber.
- The baud rate of the camera and the frame grabber must be identical.
- Vimba doesn't support image streaming with Goldeye CL cameras, but you can use Vimba to configure the camera settings.

7.2 Logging

You can enable and configure logging to:

- Create error reports
- Prepare the migration to the Vimba C API or the Vimba C++ API



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

7.2.1 Logging levels

The Vimba 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 vimba import *

vimba = Vimba.get_instance()
vimba.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')

vimba.disable_log()
```

Tracing

The logging level *Trace* enables the most detailed reports. Additionally, you can use it to prepare the migration to the Vimba C API or the Vimba 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.

To create a trace log file, use the `create_trace_log.py` example.

Avoiding large log files

All previous examples enable and disable logging globally via the Vimba 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 vimba 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()
```

7.3 Other issues

- To use the Vimba Python API, the installation of a compatible Vimba C version and Vimba Image Transform version is required. To check the versions, use `vimba.get_version()`.
- For changing the pixel format, always use the convenience functions instead of the camera feature, see section Changing the pixel format.