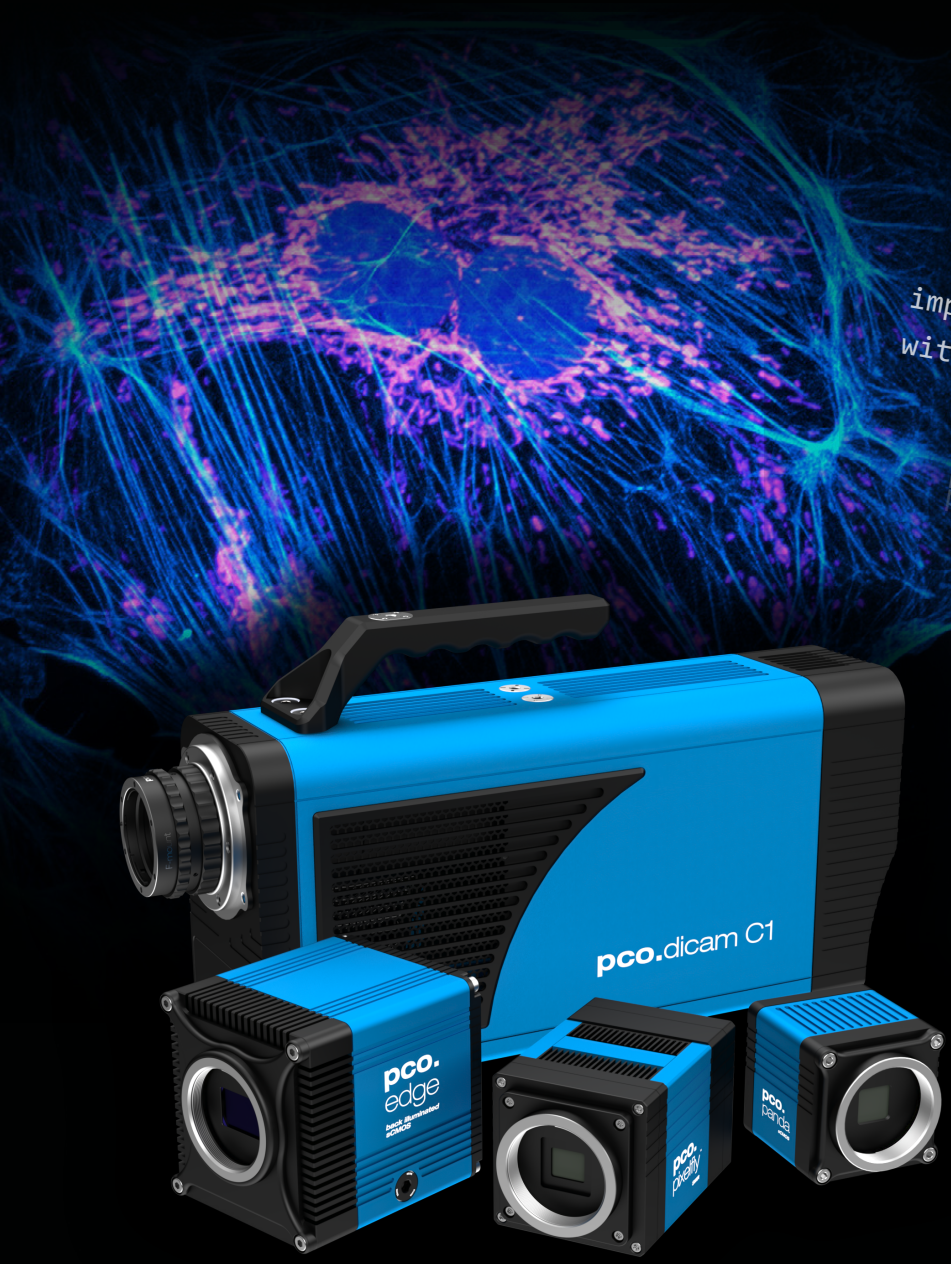
user manual

pco.python




```
import pco
with pco.Camera() as cam:
    cam.record()
    image, meta = cam.image()
    plt.imshow(image, cmap='gray')
    plt.show()
```

python™

EXCELITAS TECHNOLOGIES®

pco.

An Excelitas Technologies Brand

Excelitas PCO GmbH asks you to carefully read and follow the instructions in this document.
For any questions or comments, please feel free to contact us at any time.

pco.
An Excelitas Technologies Brand

telephone:	+49 (0) 9441 2005 50
fax:	+49 (0) 9441 2005 20
postal address:	Excelitas PCO GmbH Donaupark 11 93309 Kelheim, Germany
email:	pco@excelitas.com
web:	www.pco.de

pco.python user manual 2.1.2

Released February 2024

Contents

1 General

The Python package **pco** is a powerful and easy to use high level Software Development Kit (SDK) for working with PCO cameras. It contains everything needed for camera setup, image acquistion, readout and color conversion.

The high-level class architecture makes it very easy to integrate PCO cameras into your own software, while still having access to the underlying pco.sdk and pco.recorder interface for a detailed control of all possible functionalities.

1.1 Installation

Install from pypi (recommended):

```
$ pip install pco
```

Besides the Python Standard Libary the package `numpy` is required and installed automatically. For image display, the following modules can be used:

- `opencv-python`
- `matplotlib`
- `Pillow`

The pco module is supported for python versions greater 3.8.

Note: For cameras with USB interface on linux you will need to add usb rules to the system. This can be done with executing the following shell script as **sudo**:

```
echo "# links for pco usb cameras" >> ./pco_usb.rules
echo "# " >> ./pco_usb.rules
echo 'SUBSYSTEM=="usb" , ATTR{idVendor}=="1cb2" , GROUP="video" , ↩
    MODE="0666" , SYMLINK+="pco_usb_camera%n"' >> ./pco_usb.rules

mkdir -p "/etc/udev/rules.d"

# copy usb rules if not existing
FILE=/etc/udev/rules.d/pco_usb.rules
if ! [ -f "$FILE" ]; then
    cp ./pco_usb.rules "/etc/udev/rules.d"
    # update udev rules
    udevadm trigger || true
fi

rm "./pco_usb.rules"
```

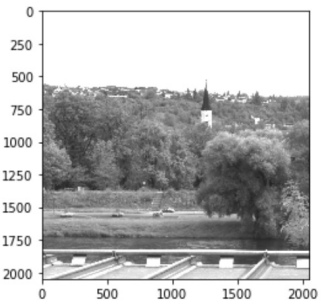
1.2 Basic Usage

```
import matplotlib.pyplot as plt
import pco

with pco.Camera() as cam:

    cam.record(mode="sequence")
    image, meta = cam.image()

    plt.imshow(image, cmap='gray')
    plt.show()
```



1.3 Recorder Modes

Depending on your workflow you can choose between different recording modes.

In blocking modes the `record` function waits until the specified number of images is reached. In non-blocking modes the caller must ensure that either recording is finished or the process is waiting for the next acquired image (`wait_for_first_image` / `wait_for_new_image`), e.g. for live view.

Memory modes are holding image data in RAM, while file modes save images directly to file(s) on the disk. However, images acquired with file mode can also be accessed from memory via `image` functions after recording is done.

CamRam modes are using the camera's internal RAM memory for high-speed acquisition. Images can be queried by reading from a segment or on the fly.

Mode	Storage	Blocking	Description
`sequence`	Memory	yes	Record a sequence of images.
`sequence non blocking`	Memory	no	Record a sequence of images, do not wait until record is finished.
`ring buffer`	Memory	no	Continuously record images in a ringbuffer, once the buffer is full, old images are overwritten.

Continued on next page

Continued from previous page

Mode	Storage	Blocking	Description
`fifo`	Memory	no	Record images in fifo mode, i.e. you will always read images sequentially and once the buffer is full, recording will pause until older images have been read.
`sequence dpcore`	Memory	yes	Same as `sequence`, but with DotPhoton preparation enabled.
`sequence non blocking dpcore`	Memory	no	Same as `sequence_non_blocking`, but with DotPhoton preparation enabled.
`ring buffer dpcore`	Memory	no	Same as `ring_buffer`, but with DotPhoton preparation enabled.
`fifo dpcore`	Memory	no	Same as `fifo`, but with DotPhoton preparation enabled.
`tif`	File	no	Record images directly as tif files.
`multitif`	File	no	Record images directly as one or more multitiff file(s).
`pcoraw`	File	no	Record images directly as one pcoraw file.
`dicom`	File	no	Record images directly as dicom files.
`multidicom`	File	no	Record images directly as one or more multi-dicom file(s).
`camram_segement`	Camera RAM	no	Record images to camera memory. Stops when segment is full.
`camram_ring`	Camera RAM	no	Record images to camera memory. Ram segment is used as ring buffer.

In the code this is represented as string, transferred to the record function (default is `sequence`):

Note For more information on the DotPhoton preparation and image compression, please visit DotPhoton or feel free to contact us.

1.4 Image Formats

All image data is always transferred as 2D or 3D numpy array. Besides the standard 16 bit raw image data you also have the possibility to get your images in different formats, shown in the table below.

The format is selected when calling the `image` / `images` / `image_average` functions (see 2.1.13, 2.1.14, 2.1.15) of the `Camera` class. The image data is stored as numpy array, which enables you to work with it in the most pythonic way.

Format	Description
`Mono8,mono8`	Get image as 8 bit grayscale data.
`Mono16,mono16,raw16,bw16`	Get image as 16 bit grayscale/raw data.
`BGR8,bgr`	Get image as 24 bit color data in bgr format.
`RGB8,rgb`	Get image as 24 bit color data in rgb format.
`BGRA8,bgra8,bgra`	Get image as 32 bit color data (with alpha channel) in bgra format.
`RGBA8,rgba8,rgba`	Get image as 32 bit color data (with alpha channel) in rgba format.
`BGR16,bgr16`	Get image as 48 bit color data in bgr format (only possible for color cameras).
`RGB16,rgb16`	Get image as 48 bit color data in rgb format (only possible for color cameras).

Note For monochrome cameras, the `BGR16` format is not available and the colors in the `BGR8`/ `BGRA8` depend on the selected lut, which is a standard grayscale mapping by default. For selecting different lut files you can use the functions `setConvertControl` (see 2.1.10) or `loadlut` (see 2.1.11) from the camera class.

1.5 Event and Error Logging

The pco package supports the python `logging` library, to enable logging output of the pco package. Therefore, the predefined `StreamHandler` from the pco package can be used:

```
logger = logging.getLogger("pco")
logger.setLevel(logging.INFO)
logger.addHandler(pco.stream_handler)
```

Supported logging levels are: `ERROR`, `WARNING`, `INFO`, `DEBUG`.

The logging output has following format and is written to sys.stderr:

```
...
[2023-03-07 10:39:21,270] [0.016 s] [sdk] get_camera_type: OK
...
```

2 API Documentation

This section describes the methods, variables and objects of the Camera class. The following list provides a short overview of the most important functions:

The **pco.Camera** class offers the following methods:

- **__init__()** Opens and initializes a camera with its default configuration.
- **__exit__()** Closes the camera and cleans up everything (e.g. end of with-statement).
- **close()** Closes the camera and cleans up everything.
- **default_configuration()** Set default configuration to the camera.
- **record()** Initialize and start the recording of images.
- **stop()** Stop the current recording.
- **wait_for_first_image()** Wait until the first image has been recorded.
- **wait_for_new_image()** Wait until a new image has been recorded.
- **get_convert_control()** Get current color convert settings.
- **set_convert_control()** Set new color convert settings.
- **load_lut()** Set the lut file for the convert control setting.
- **adapt_white_balance()** Do a white-balance according to a transferred image.
- **image()** Read a recorded image as numpy array.
- **images()** Read a series of recorded images as a list of numpy arrays.
- **image_average()** Read an averaged image (averaged over all recorded images) as numpy array.
- **switch_to_camram()** Set camram segment for read via image functions.
- **set_camram_allocation()** Set allocation distribution of camram segments.

The **pco.Camera** class has the following properties:

- **camera_name** get the camera name.
- **camera_serial** get the serial number of the camera.
- **is_recording** get a flag to indicate if the camera is currently recording.
- **is_color** get a flag to indicate if the camera is a color camera.
- **recorded_image_count** get the number of currently recorded images.
- **configuration** get/set the camera configuration.
- **description** get the (static) camera description parameters.
- **exposure_time** get/set the exposure time (in seconds).
- **delay_time** get/set the delay time (in seconds).
- **has_ram** get flag that indicate camram support of the camera.
- **camram_segment** get segment number of active segment.
- **camram_max_images** get number of images that can be stored in the active segment.

- **camram_num_images** get number of images that are available in the active segment.

The **pco.Camera** class holds the following objects:

- **sdk** offers direct access to all underlying functions of the **pco.sdk**.
- **rec** offers direct access to all underlying functions of the **pco.recorder**.
- **conv** offers direct access to all underlying functions of the **pco.convert** according to the selected data_format.

2.1 Methods

This section describes all methods offered by the **pco.Camera** class.

2.1.1 __init__

Description Opens and initializes the camera.

Do not call this explicitly, this function is called automatically when a camera object is created. Either directly `cam = pco.Camera()` or by the `with` statement.

```
with pco.Camera() as cam:
    # do some stuff
```

Prototype

```
def __init__(self,
    interface=None):
```

Parameter

Name	Description
interface	Specific interface or list of interfaces to search for cameras. If None, search on all interfaces. Available parameters: ”FireWire”, ”Camera Link MTX”, ”GenICam”, ”Camera Link NAT”, ”GigE”, ”USB 2.0”, ”Camera Link ME4”, ”USB 3.0”, ”CLHS”

2.1.2 __exit__

Description Closes the activated camera and releases the blocked ressources.

Do not call this explicitly, this function is called automatically when a camera object is destroyed. Either directly `cam.close()` or by the `with` statement.

```
with pco.Camera() as cam:
    # do some stuff
```

Prototype

```
def __exit__(self, exc_type, exc_value, exc_traceback):
```

2.1.3 close

Description Closes the activated camera and releases the blocked ressources. This function must be called before the application is terminated. Otherwise, the resources remain occupied.

This function is called automatically if the camera object was released by the `with` statement. An explicit call to `close()` is no longer necessary.

```
with pco.Camera() as cam:
    # do some stuff
```

Prototype

```
def close(self):
```

2.1.4 default_configuration

Description (Re)set the camera to its default configuration.

Prototype

```
def default_configuration(self):
```

2.1.5 record

Description Creates, configures, and starts a new recorder instance. The entire camera configuration must be set before calling `record()`. The properties `exposure_time` and `delay_time` are the only exception. These properties have no effect on the recorder object and can be called up during the recording.

Prototype

```
def record(self,
    number_of_images=1,
    mode="sequence",
    file_path=None):
```

Parameter

Name	Description
number_of_images	Sets the number of images allocated in the driver. The RAM or disk (depending on the mode) of the PC limits the maximum value.
mode	Defines the recording mode for this record (see 1.3)
file_path	Path where the image file(s) should be stored (only for modes who directly save to file, see 1.3).

2.1.6 stop

Description Stops the current recording.

In `'ring␣buffer'` and `'fifo'` mode, this function must be called by the user. In `'sequence'` and `'sequence␣non␣blocking'` mode, this function is automatically called up when the `number_of_images` is reached.

For blocking recorder modes (see 1.3), the recording is automatically stopped when the required number of images is reached. In this case `stop()` is not needed.

Prototype

```
def stop(self):
```

2.1.7 wait_for_first_image

Description Wait until the first image has been recorded and is available.

In recorder mode `'sequence␣non␣blocking'`, `'ring␣buffer'`. and `'fifo'`, the function `record()` returns immediately. Therefore, this function can be used to wait for images from the camera before calling
`image()`, `images()`, or `image_average()`.

Prototype

```
def wait_for_first_image(self,
    delay=True,
    timeout=None):
```

Parameter

Name	Description
delay	Flag if a small delay should be used in the waiting loop (typically recommended to reduce CPU load).
timeout	If not `None`, the waiting loop will be aborted if no image was recorded during `timeout` seconds.

2.1.8 wait_for_new_image

Description Wait until a new image has been recorded and is available (i.e. an image that has not been read yet).

Prototype

```
def wait_for_new_image(self,
    delay=True,
    timeout=None):
```

Parameter

Name	Description
delay	Flag if a small delay should be used in the waiting loop (typically recommended to reduce CPU load).
timeout	If not `None`, the waiting loop will be aborted if no image was recorded during `timeout` seconds.

2.1.9 get_convert_control

Description Get the current convert control settings for the specified data format.

Prototype

```
def get_convert_control(self,
    data_format):
```

Parameter

Name	Description
data_format	Data format for which the convert settings should be queried.

Return value

Datatype	Description
dict	dictionary containing the current convert settings for the specified data format.

2.1.10 set_convert_control

Description Set convert control settings for the specified data format.

Prototype

```
def set_convert_control(self,
    data_format,
    convert_ctrl):
```

Parameter

Name	Description
data_format	Data format for which the convert settings should be set.
convert_ctrl	Dictionary of convert control settings that should be set.

Dict Keys The available keys for `convert_ctrl` vary according to camera properties and image format. Cameras with color sensor support conversion control for its Bayer pattern, non-colored must provide a LUT file for assigning colors to the monochromic image data.

Key	Supported data formats
”sharpen”: <bool>	"Mono8", "BGR8", "BGR16"
”adaptive_sharpen”: <bool>	"Mono8", "BGR8", "BGR16"
”flip_vertical”: <bool>	"Mono8", "BGR8", "BGR16"
”auto_minmax”: <bool>	"Mono8", "BGR8", "BGR16"
”min_limit”: <int>	"Mono8", "BGR8", "BGR16"
”max_limit”: <int>	"Mono8", "BGR8", "BGR16"
”gamma”: <double>	"Mono8", "BGR8", "BGR16"
”contrast”: <int>	"Mono8", "BGR8", "BGR16"
”color_temperature”: <int>	"BGR8", "BGR16"
”color_saturation”: <int>	"BGR8", "BGR16"
”color_vibrance”: <int>	"BGR8", "BGR16"

Continued on next page

Continued from previous page

Key	Supported data formats
”color_tint”: <int>	`"BGR8"`, `"BGR16"`
”lut_file”: <file_path>	`"BGR8"`, for non-colored cameras

2.1.11 load_lut

Description Set the lut file for the convert control settings.

This is just a convenience function, the lut file could also be set using `set_convert_control` (see: 2.1.10).

Prototype

```
def load_lut(self,
    data_format,
    lut_file):
```

Parameter

Name	Description
`data_format`	Data format for which the lut file should be set.
`lut_file`	Actual lut file path to be set.

2.1.12 adapt_white_balance

Description Do a white-balance according to a transferred image.

Prototype

```
def adapt_white_balance(self,
    image,
    data_format,
    roi);
```

Parameter

Datatype	Description
`image`	Image that should be used for white-balance computation.
`data_format`	Data format for which the white balance values should be set.
`roi`	If not `None`, use only the specified ROI for white-balance computation.

2.1.13 image

Description Get a recorded image in the given format. The type of the image is a `numpy.ndarray`. This array is shaped depending on the resolution and ROI of the image.

Prototype

```
def image(self,
    image_index=0,
    roi=None,
    data_format="Mono16",
    comp_params=None):
```

Parameter

Name	Description
`image_index`	Index of the image that should be queried, use PCO_-RECORDER_LATEST_IMAGE for latest image (for recorder modes fifo/fifo_dpcore always use 0 (see 1.3)).
`roi`	Soft ROI to be applied, i.e. get only the ROI portion of the image.
`data_format`	Data format the image should have (see 1.4).
`comp_params`	Dictionary containing the compression parameters, not implemented yet.

Return value

Datatype	Description
`(numpy.ndarray, dict)`	Tuple of image data as `numpy.ndarray` and metadata as dictionary.

Dict Keys The available keys for `meta` can vary according to camera configuration. However, `"data␣format"` and `"recorder␣image␣number"` are always available.

Key	Meta data
`"data␣format"`: <str>	`"Mono8", "Mono16", "BGR8", "BGR16", "CompressedMono8"`
`"recorder␣image␣number`: <int>	from pco.recorder
`"timestamp"`: <dict>	{`"image␣counter": <int>, "year": <int>, "month": <int>, "day": <int>, "hour": <int>, "minute": <int>, "second": <float>, "status": <int>`}
”version”: metadata: <int>	from PCO_METADATA_STRUCT
”exposure time”: <int>	from PCO_METADATA_STRUCT
”framerate”: metadata: <float>	in Hz
”sensor temperature”: <int>	from PCO_METADATA_STRUCT
”pixel clock”: <int>	from PCO_METADATA_STRUCT
”conversion factor”: <int>	from PCO_METADATA_STRUCT
”serial number”: <int>	from PCO_METADATA_STRUCT
”camera type”: <int>	from PCO_METADATA_STRUCT
”bit resolution”: <int>	from PCO_METADATA_STRUCT

Continued on next page

Continued from previous page

Key	Meta data
”sync status”: <int>	from PCO_METADATA_STRUCT
”dark offset”: <int>	from PCO_METADATA_STRUCT
”trigger mode”: <int>	from PCO_METADATA_STRUCT
”double image mode: <int>	from PCO_METADATA_STRUCT
”camera sync mode: <int>	from PCO_METADATA_STRUCT
”image type”: <int>	from PCO_METADATA_STRUCT
”color pattern”: <int>	from PCO_METADATA_STRUCT
”image size”: <int>	from PCO_METADATA_STRUCT
”binning”: <int>	from PCO_METADATA_STRUCT
”camera subtype”: <int>	from PCO_METADATA_STRUCT
”event number”: <int>	from PCO_METADATA_STRUCT
”image size offset: <int>	from PCO_METADATA_STRUCT
”timestamp bcd”: <dict>	{”image counter”: <int>, ”year”: <int>, ”month”: <int>, ”day”: <int>, ”hour”: <int>, ”minute”: <int>, ”second”: <float>, ”status”: <int> }

Example

```
>>> cam.record(number_of_images=1, mode='sequence')

>>> image, meta = cam.image()

>>> type(image)
numpy.ndarray

>>> image.shape
(2160, 2560)

>>> image, metadata = cam.image(roi=(1, 1, 300, 300))

>>> image.shape
(300, 300)
```

2.1.14 images

Description Get a series of images in the given format as list of numpy arrays.

The positions of the images to query are defined by a start index and a block size. If this block size is `None`, all images, beginning with the given start index, are read

Prototype

```
def images(self,
  roi=None,
  start_idx=0,
  blocksize=None,
  data_format="Mono16",
  comp_params=None):
```

Parameter

Name	Description
`roi`	Soft ROI to be applied, i.e. get only the ROI portion of the images.
`start_idx`	Index of the first image that should be queried.
`blocksize`	Number of images that should be copied (if `None`, all recorded images, beginning at `start_idx`, are copied).
`data_format`	Data format the images should have (see 1.4).
`comp_params`	Dictionary containing the compression parameters, not implemented yet.

Return value

Datatype	Description
`(list(numpy.ndarray), list(dict))`	Tuple of list of images as `numpy.ndarray` and list of metadata as dictionary.

Example

```
>>> cam.record(number_of_images=20, mode='sequence')

>>> images, metadatas = cam.images()

>>> len(images)
20

>>> for image in images:
  ...     print('Mean: {:7.2f} DN'.format(image.mean()))
  ...
  Mean: 2147.64 DN
  Mean: 2144.61 DN
  ...

>>> images = cam.images(roi=(1, 1, 300, 300))

>>> images[0].shape
(300, 300)
```

2.1.15 image_average

Description Get an averaged image, averaged over all recorded images in the given format. The type of the image is a `numpy.ndarray`.

Prototype

```
def image_average(self,
    roi=None,
    data_format="Mono16"):
```

Parameter

Name	Description
`roi`	Soft ROI to be applied, i.e. get only the ROI portion of the image.
`data_format`	Data format the image should have (see 1.4).

Return value

Datatype	Description
`numpy.ndarray`	Image data as `numpy.ndarray`.

Example

```
>>> cam.record(number_of_images=100, mode='sequence')

>>> avg = cam.image_average()

>>> avg = cam.image_average(roi=(1, 1, 300, 300))
```

2.1.16 switch_to_camram

Description Sets camram segment and prepare internal recorder for reading images from camera-internal memory.

Prototype

```
def switch_to_camram(self,
  segment=None):
```

Parameter

Name	Description
segment	Segment number for image readout. Optional parameter.

Example

```
>>> cam.switch_to_camram(1)

>>> if camram_num_images > 0:
>>>     img, meta = image(0)
```

2.1.17 set_camram_allocation

Description Set allocation distribution of camram segments.

Prototype Maximum number of segments is 4. Accumulated sum of parameter values must not be greater than 100.

```
def set_camram_allocation(self,
  percents):
```

Parameter

Name	Description
percents	List of numbers that represent percentages for segment size distribution. Length: 1 <= len( ) <= 4

Example

```
>>> cam.set_camram_allocation([70, 20])
# or
>>> cam.set_camram_allocation([0.25, 0.25, 0.25, 0.25])
```

2.2 Properties

This section describes all variables offered by the **pco.Camera** class.

2.2.1 camera_name

The camera_name property gets the name of the camera as string.
This is a **readonly** property.

2.2.2 camera_serial

The camera_serial property gets the serial number of the camera as number.
This is a **readonly** property.

2.2.3 is_recording

The is_recording property is flag to check if the camera is currently recording.
This is a **readonly** property.

2.2.4 is_color

The is_color property is a flag to check if the camera is a color camera.
This is a **readonly** property.

2.2.5 recorded_image_count

The recorded_image_count property gets the count of currently recorded images.
This is a **readonly** property.

NOTE For recorder modes fifo and fifo_dpcore (see 1.3) this represents the current fill level of the fifo buffer, not the overall number of recorded images. So here it would be enough to check for `if cam.recorded_image_count > 0 :` to see if a new image is available.

2.2.6 description

The description property gets the (static) camera description parameters as dictionary with the following keys:

- ”serial”: <integer>
- ”type”: <string>
- ”sub type”: <integer>
- ”interface type”: <string>
- ”min exposure time”: <float>
- ”max exposure time”: <float>
- ”min exposure step”: <float>

- ”min delay time”: <float>
- ”max delay time”: <float>
- ”min delay step”: <float>

This is a **readonly** property.

2.2.7 exposure_time

Get/Set the exposure time [s] of the camera

2.2.8 delay_time

Get/Set the delay time [s] of the camera

2.2.9 configuration

Get/Set the current configuration of the camera. The parameters are stored in a dictionary as shown in the following example.

```
config = cam.configuration

...

cam.configuration = {'exposure time': 10e-3,
                     'delay time': 0,
                     'roi': (1, 1, 512, 512),
                     'timestamp': 'ascii',
                     'pixel rate': 100_000_000,
                     'trigger': 'auto sequence',
                     'acquire': 'auto',
                     'noise filter': 'on,
                     'metadata': 'on',
                     'binning': (1, 1)}
```

The property can only be changed before the `record()` function is called. It is a dictionary with a certain number of entries. Not all possible elements need to be specified. The following sample code only changes the `'pixel␣rate'` and does not affect any other elements of the configuration.

```
with pco.Camera() as cam:

    cam.configuration = {'pixel rate': 286_000_000}

    cam.record()
    ...
```

2.2.10 has_ram

Get flag indicating whether camera-internal memory for recording with camram is available

2.2.11 camram_segment

Get segment number of active camram segment

2.2.12 camram_max_images

Get number of images that can be stored in the active camram segment

2.2.13 camram_num_images

Get number of images that are available in the active camram segment

2.3 Objects

This section describes all objects offered by the **pco.Camera** class.

2.3.1 sdk

The object `sdk` allows direct access to all underlying functions of the **pco.sdk**.

```
>>> cam.sdk.get_temperature()
{'sensor temperature': 7.0, 'camera temperature': 38.2, 'power ↩
    temperature': 36.7}
```

All return values from `sdk` functions are dictionaries. Not all camera settings are covered by the `Camera` class. Special settings have to be set directly by calling the respective `sdk` function.

2.3.2 rec

The object `rec` offers direct access to all underlying functions of the **pco.recorder**.

It is not be necessary to call a recorder class method directly. All functions are fully covered by the methods of the `Camera` class.

2.3.3 conv

The object `conv` is a dictionary of convert objects to offer direct access to all underlying functions of the **pco.convert**.

Valid dictionary keys are:

- `Mono8`: To access the pco.convert object for monochrome color conversion
- `BGR8`: To access the pco.convert object for color conversion
- `BGR16`: To access the pco.convert object for 48bit color conversion (color cameras only)

It is not be necessary to call a conv class method directly. All functions are fully covered by the methods of the `Camera` class.

3 About Excelitas PCO

PCO, an Excelitas Technologies® Corp. brand, is a leading specialist and Pioneer in Cameras and Optoelectronics with more than 30 years of expert knowledge and experience of developing and manufacturing high-end imaging systems. The company's cutting edge sCMOS and high-speed cameras are used in scientific and industrial research, automotive testing, quality control, metrology and a large variety of other applications all over the world.

The PCO® advanced imaging concept was conceived in the early 1980s by imaging pioneer, Dr. Emil Ott, who was conducting research at the Technical University of Munich for the Chair of Technical Electrophysics. His work there led to the establishment of PCO AG in 1987 with the introduction of the first image-intensified camera followed by the development of its proprietary Advanced Core technologies which greatly surpassed the imaging performance standards of the day.

Today, PCO continues to innovate, offering a wide range of high-performance camera technologies covering scientific, high-speed, intensified and FLIM imaging applications across the scientific research, industrial and automotive sectors.

Acquired by Excelitas Technologies in 2021, PCO represents a world renowned brand of high-performance scientific CMOS, sCMOS, CCD and high-speed cameras that complement Excelitas' expansive range of illumination, optical and sensor technologies and extend the bounds of our end-to-end photonic solutions capabilities.

pco.

An Excelitas Technologies Brand

pco.

An Excelitas Technologies Brand

postal address:	Excelitas PCO GmbH Donaupark 11 93309 Kelheim, Germany
telephone:	+49 (0) 9441 2005 0
e-mail:	pco@excelitas.com
web:	www.excelitas.com/pco

EXCELITAS TECHNOLOGIES®