user manual

pco.python

excelitas.com

excelitas®

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.

address:	Excelitas PCO GmbH Donaupark 11 93309 Kelheim, Germany
phone:	(+49) 9441-2005-0 (+1) 866-662-6653 (+86) 0512-6763-4643
mail:	pco@excelitas.com
web:	www.excelitas.com/pco

pco.python user manual 2.4.1

Released June 2025

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"
```

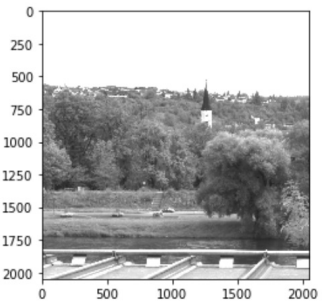
1.2 Basic Usage

```
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 2.1.20, 2.1.21, 2.1.22) 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 2.1.17) or `loadlut` (see 2.1.18) 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.
- **configureHWIO_*_***()** Configure the HWIO channels (1-4)
- **auto_exposure_on(), auto_exposure_off()** Switch auto exposure on/off
- **configure_auto_exposure()** Set the parameters for auto exposure calculations
- **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.
- **interface** get the interface 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.

Optionally you can specify either which interface you want to look at or the serial number of the camera you want to open or both.

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
```

Note for windows If you specify a serial number to be opened, we recommend to also specify the interface as this reduces the time for the function call.

Prototype

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

Parameter

Name	Description
`interface`	Specific interface to search for cameras. If `None`, search on all interfaces.
`serial`	Search for the camera with this specific serial number. If `None`, search for any camera.

Note Available interfaces are:

```
"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 configureHWIO_1_exposureTrigger

Description Configure the HWIO connector 1.

This connector is used for the exposure trigger signal input.

Prototype

```
def configureHWIO_1_exposureTrigger(self
  on,
  edgePolarity
);
```

Parameter

Name	Description
on	Flag if the HWIO connector should be enabled or disabled
edgePolarity	Polarity the connector should react on (valid are "rising edge" and "falling edge")

2.1.6 configureHWIO_2_acquireEnable

Description Configure the HWIO connector 2.

This connector is used for the acquire enable signal input.

Prototype

```
def configureHWIO_2_acquireEnable(self
  on,
  polarity
);
```

Parameter

Name	Description
on	Flag if the HWIO connector should be enabled or disabled
polarity	Polarity the connector should have (valid are "high level" and "low level")

2.1.7 configureHWIO_3_statusBusy

Description Configure the HWIO connector 3.

This connector is typically used for the status busy output of the camera. Depending on the camera it can also be configured to output different kind of signals, which can be selected by the `signal_type` parameter.

Prototype

```
def configureHWIO_3_statusBusy(self,
  on,
  polarity,
  signal_type
);
```

Parameter

Name	Description
`on`	Flag if the HWIO connector should be enabled or disabled
`polarity`	Polarity the connector should have (valid are "`high level`" and "`low level`")
`signal_type`	Type of the signal the connector should have (valid are "`status busy`", "`status line`", "`status armed`")

Return value

Name	Description
`signal_type_valid`	Boolean flag if the `signal_type` that was selected is valid for the camera.

Note Even if you select a `signal_type` that is not valid, i.e. the function returns false, the `on` and `polarity` parameters are set anyway.

2.1.8 configureHWIO_4_statusExpos

Description Configure the HWIO connector 4.

This connector is typically used for the status exposure output of the camera. Depending on the camera it can also be configured to output different kind of signals, selected by the `signal_type` parameter. In some cases, different timing modes for the exposure output signal can be selected by the `signal_timing` parameter.

Prototype

```
def configureHWIO_4_statusExpos(self,
  on,
  polarity,
  signal_type,
  signal_timing = None
);
```

Parameter

Name	Description
`on`	Flag if the HWIO connector should be enabled or disabled
`polarity`	Polarity the connector should have (valid are "`high level`" and "`low level`")
`signal_type`	Type of the signal the connector should have (valid are "`status expos`", "`status line`", "`status armed`")
`signal_timing`	Timing of exposure output signal (valid are "`first line`", "`global`", "`last line`", "`all lines`") Only valid for Rolling Shutter cameras and `signal_type` "`status expos`" (default is `None`, i.e. will not be set)

Return value

Name	Description
`signal_type_valid`	Boolean flag if the `signal_type` that was selected is valid for the camera.

Note Even if you select a `signal_type` that is not valid, i.e. the function returns false, the `on` and `polarity` parameters are set anyway.

2.1.9 configure_auto_exposure

Description Set the auto exposure parameters.

This does not activate or deactivate the auto exposure functionality.
For this please use `auto_exposure_on()` and `auto_exposure_off()`.

Note While `auto_exposure_on()` and `auto_exposure_off()` can be called also during record, this function can only be called when recording is off.

Prototype

```
def configure_auto_exposure(self,
  region_type,
  min_exposure_s,
  max_exposure_s);
```

Parameter

Name	Description
`region_type`	Image region type that should be used for auto exposure computation (see below).
`min_exposure_s`	Minimum exposure value that can be used for auto exposure
`max_exposure_s`	Maximum exposure value that can be used for auto exposure

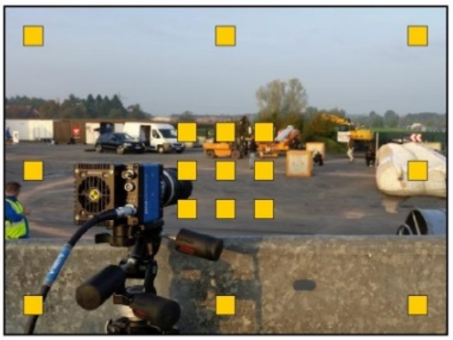
Note There are 4 different types of regions available (default is `'balanced'`)

```
'balanced'
'center_based'
'corner_based'
'full'
```

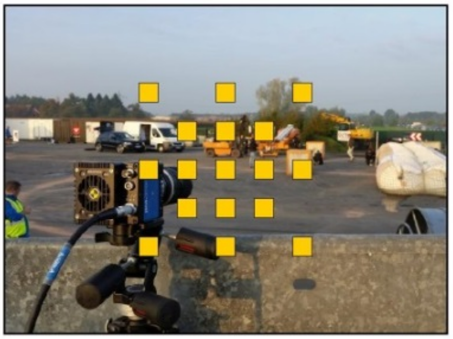
The size of the pixel clusters is fixed, but depends on the overall image size and is treated separately for width and height:

- For width/height >= 1300 the cluster size is 100
- For 1300 > width/height >= 650 the cluster size is 50
- For 650 > width/height >= 325 the cluster size is 25
- For width/height < 325 the cluster size equal to width/height

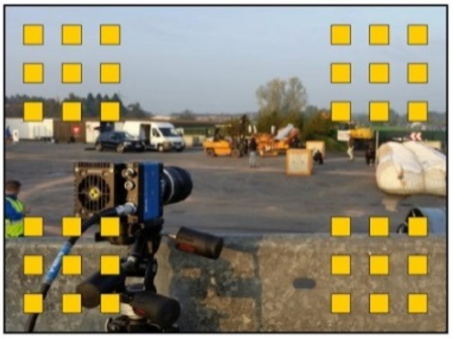
balanced Measurement fields positioned centrally and in all corners



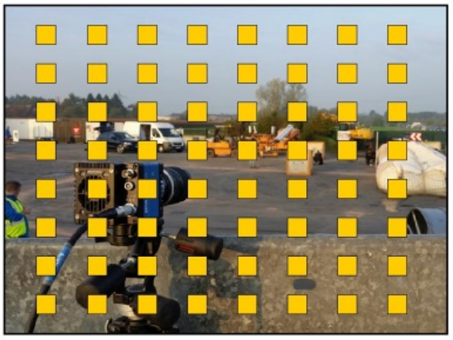
center_based Measurement fields positioned centrally.



corner_based Measurement fields positioned in all four corners.



full Measurement fields across the image.



2.1.10 auto_exposure_on

Description Activate the auto exposure feature.

This will use the currently set configuration for auto exposure.
To set the auto exposure mode parameters please use `configure_auto_exposure()`.

Prototype

```
def auto_exposure_on(self);
```

2.1.11 auto_exposure_off

Description Deactivate the auto exposure feature.

Prototype

```
def auto_exposure_off(self);
```

2.1.12 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).

2.1.13 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):
```

2.1.14 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.

2.1.15 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.

2.1.16 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.

2.1.17 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"
”add_conv_flags”: <int>	"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"

Continued on next page

Continued from previous page

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

2.1.18 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: 2.1.17).

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.

2.1.19 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.

2.1.20 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="Undefined",
    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", "BGRA8", "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
”readout mode: <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)
```

2.1.21 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="Undefined",
  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)
```

2.1.22 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="Undefined"):
```

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))
```

2.1.23 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)
```

2.1.24 set_camram_allocation

Description Set allocation distribution of camram segments.

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

Prototype

```
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 interface

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

2.2.4 raw_format

The raw_format property gets the current raw format of the camera as string.
This is a **readonly** property.

2.2.5 is_recording

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

2.2.6 is_color

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

2.2.7 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.

2.2.8 has_ram

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

2.2.9 camram_segment

Get segment number of active camram segment

2.2.10 camram_max_images

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

2.2.11 camram_num_images

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

2.2.12 exposure_time

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

2.2.13 delay_time

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

2.2.14 description

The description property gets the (static) camera description parameters as dictionary with the following keys:
This is a **readonly** property.

Datatype	Name	Description
<integer>	serial	Serial number of the camera
<string>	type	Sensor type
<integer>	sub type	Sensor sub type
<string>	interface type	Interface type
<float>	min exposure time	Minimal possible exposure time [s]
<float>	max exposure time	Maximal possible exposure time [s]
<float>	min exposure step	Minimal possible exposure step [s]
<float>	min delay time	Minimal possible delay time [s]
<float>	max delay time	Maximal possible delay time [s]
<float>	min delay step	Minimal possible delay step [s]

Continued on next page

Continued from previous page

Datatype	Name	Description
<integer>	min width	Minimal possible image width (hardware ROI)
<integer>	min height	Minimal possible image height (hardware ROI)
<integer>	max width	Maximal possible image width (hardware ROI)
<integer>	max height	Maximal possible image height (hardware ROI)
<tuple[int, int]>	roi steps	Hardware ROI stepping as tuple of (horz, vert)
<bool>	roi is horz symmetric	Flag if hardware ROI has to be horizontally symmetric (i.e. if x0 is increased, x1 has to be decreased by the same value)
<bool>	roi is vert symmetric	Flag if hardware ROI has to be vertically symmetric (i.e. if y0 is increased, y1 has to be decreased by the same value)
<integer>	bit resolution	Bit-resolution of the sensor
<bool>	has timestamp	Flag if camera supports the timestamp setting
<bool>	has ascii-only timestamp	Flag if camera supports setting the timestamp to ascii-only
<bool>	has trigger extexpctrl	Flag if camera supports external exposure controls
<list[integer]>	pixelrates	List containing all possible pixelrate frequencies (index 0 is default)
<bool>	has trigger extexpctrl	Flag if camera supports trigger mode external exposure control
<bool>	has acquire	Flag if camera supports the acquire mode setting
<bool>	has extern acquire	Flag if camera supports the external acquire setting
<bool>	has metadata	Flag if metadata can be activated for the camera
<bool>	has ram	Flag if camera has internal memory
<list[integer]>	binning horz vec	List containing all possible horizontal binning values
<list[integer]>	binning vert vec	List containing all possible vertical binning values
<bool>	has average binning	Flag if camera supports average binning
<list[string]>	supported pixel formats	List containing all possible pixel formats

2.2.15 configuration

Get/Set the current configuration of the camera. The parameters are stored in a dictionary with the following keys:

Datatype	Key	Description
<float>	exposure time	Exposure time [s]
<float>	delay time	Delay time [s]
<tuple[int, int, int, int]>	roi	Hardware ROI as tuple of (x0, y0, x1, y1)
<string>	timestamp	Timestamp mode
<integer>	pixel rate	Pixelrate
<string>	trigger	Trigger mode
<string>	acquire	Acquire mode
<string>	metadata	Metadata mode
<string>	noise filter	Noise filter mode
<tuple[int, int, str]>	binning	Binning setting as tuple of (horz, vert, mode)
<tuple[str, int, int]>	auto exposure	Auto-Exposure setting as tuple of (region-type, min exposure, max exposure), see 2.1.9 for detailed explanation of the parameters
<string>	pixel format	Pixel format

The values of the default configuration is 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, "sum"),
                     'auto exposure': ("balanced", 0.001, 0.1),
                     'pixel format': '16'}
```

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()
    ...
```

2.3 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** library.

```
>>> 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** library.

It is not 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** library.

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.

2.4 XCite

2.4.1 __init__

Description Open and initializes the XCite connection. Optionally you can specify either which interface you want to look at or the name of the XCite device you want to open or both.

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

```
with pco.XCite() as xcite:
    # do some stuff
```

Prototype

```
    def __init__(self, xcite_type="Any", port=""):
```

Parameter

Name	Description
xcite_type	Specify which XCite device to find.
port	Specific interface to search the XCite device.

2.4.2 __exit__

Description Close any active XCite connection and releases the blocked ressources.

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

```
with pco.XCite() as xcite:
    # do some stuff
```

Prototype

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

2.4.3 close

Description Close any active XCite connection 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 xcite object was released by the `with` statement. An explicit call to `close()` is no longer necessary.

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

Prototype

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

2.4.4 default_configuration

Description Reset the configuration of the xcite device to the default values, turns all lights off.

Prototype

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

2.4.5 switchOn

Description Switch the configured lights on

Prototype

```
def switchOn(self):
```

2.4.6 switchOff

Description Switch all lights off

Prototype

```
def switchOff(self):
```

2.4.7 Properties

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

2.4.7.1 configuration

Get/Set the current configuration of the xcite. The parameters are stored in a dictionary with the following keys:

Datatype	Name	Description
<dict[integer, integer]>	<wavelength as integer>	”intensity”: <int>, ”led state”: bool

2.4.7.2 description

The description property gets the (static) xcite description parameters as dictionary with the following keys:
This is a **readonly** property.

Datatype	Name	Description
<integer>	serial	Serial number of the xcite
<string>	name	XCite name
<string>	com port	Com port as a string
<dict>	wavelength exclusivity	<wavelength as integer>: <int>
<dict>	wavelength intensity limits	<wavelength as integer>: (<int>, <int>) Minimal/maximum possible intensities[%]

2.4.8 Objects

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

2.4.8.1 xcite

The object `xcite` allows direct access to all underlying functions of the **etc.xcite** library. It is normally not necessary to call a xcite method directly. All functions are covered by the methods of the `XCite` class.

3 About Excelitas PCO

Pioneering in Cameras and Optoelectronics (PCO) has been our shared philosophy since our establishment in 1987. Starting with image-intensified cameras, followed by the co-invention of the groundbreaking sCMOS sensor technology, PCO greatly surpassed the imaging performance standards of the day. Acquired by Excelitas in 2021, our PCO camera portfolio continues to forge ahead as a leader in digital imaging innovation across diverse applications such as scientific and industrial research, automotive testing, quality control, and metrology.

With sophisticated mechanical design, extensive software support, and a broad range of accessories, we deliver adaptable solutions for all demands. This adaptability extends to tailor-made firmware and custom image sensors, which allow us to develop highly specialized solutions for all our customers. PCO represents a world-renowned brand of high-performance camera systems that complement Excelitas' expansive range of illumination, optical, and sensor technologies and extend the bounds of our end-to-end photonic solutions capabilities.

Our comprehensive camera portfolio covers the entire spectrum - from deep ultraviolet (DUV) to shortwave infrared (SWIR), from long exposure to high-speed, from line scan to high-resolution area scan. Our camera systems are controlled and processed through an intuitive and powerful software suite addressing an extensive range of platforms and architectures.

pco.®

address:	Excelitas PCO GmbH Donaupark 11 93309 Kelheim, Germany
phone:	(+49) 9441-2005-0 (+1) 866-662-6653 (+86) 0512-6763-4643
mail:	pco@excelitas.com
web:	www.excelitas.com/pco

excelitas®

excelitas.com