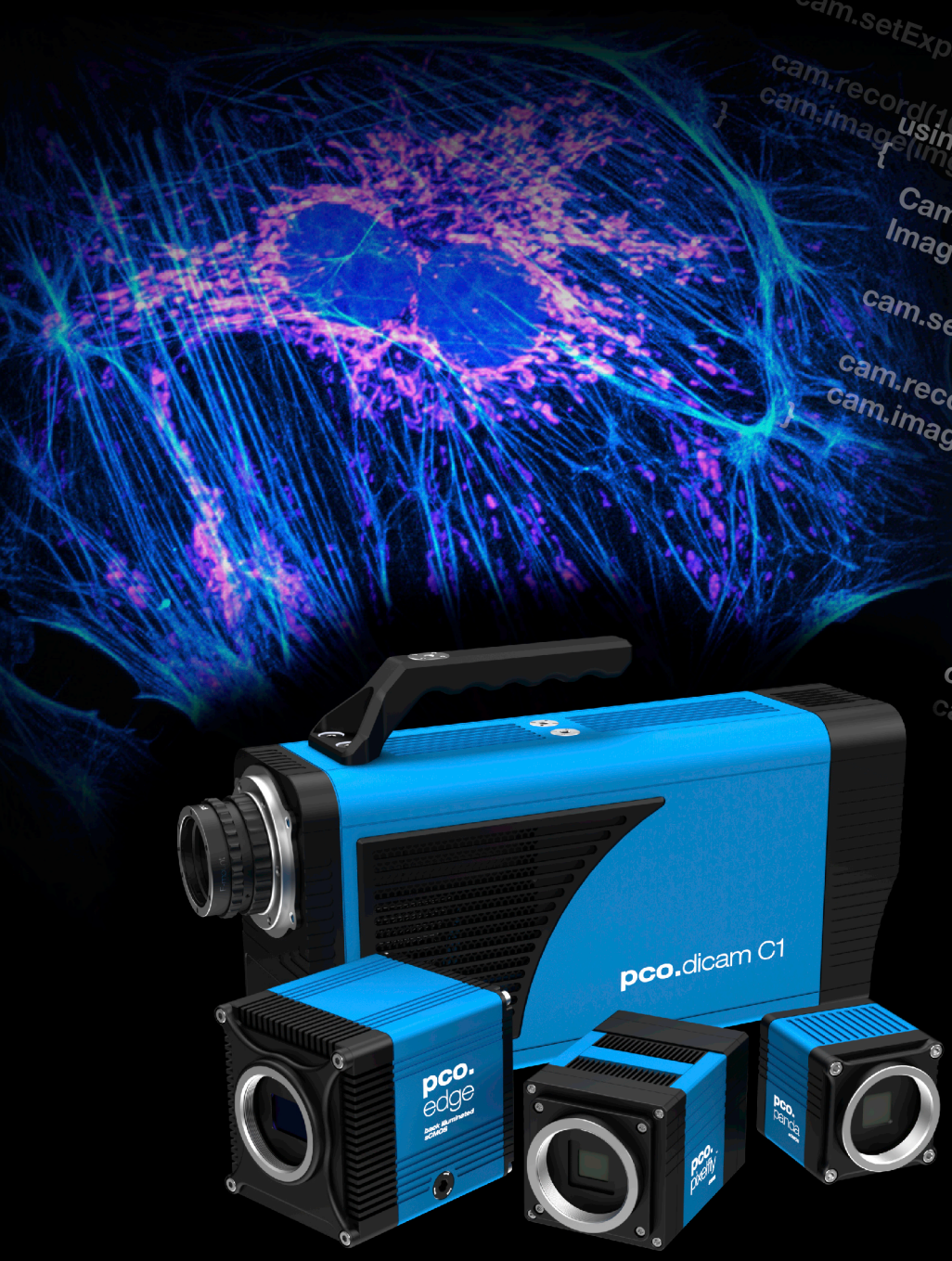
manual

pco.csharp




```
using pco;
{
    Camera cam = new Camera();
    Image img = new Image();
    cam.setExposuretime(0.01);
    cam.record(10, RecordMode.sequence);
    cam.image(img, 1, DataFormat.BGR8);
}
```


```
using pco;
{
    Camera cam = new Camera();
    Image img = new Image();
    cam.setExposuretime(0.01);
    cam.record(10, RecordMode.sequence);
    cam.image(img, 1, DataFormat.BGR8);
}
```

```
using pco;
{
    Camera cam = new Camera();
    Image img = new Image();
    cam.setExposuretime(0.01);
    cam.record(10, RecordMode.sequence);
    cam.image(img, 1, DataFormat.BGR8);
```


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.csharp manual 1.4.0

Released December 2025

Contents

1 General

The **pco.csharp** package is a powerful and easy-to-use high-level C# 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 C# 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

Download the pco.csharp Windows package, unzip it and execute it. Follow the steps in the installer.

In your install directory you will find:

- a visual studio (2019) solution file for all provided examples
- a **samples** folder containing all example projects
- the **pco** folder containing the actual sources of this sdk, i.e the classes and defines described in this document.
  It also contains a ***pco_csharp.csproj*** which generates a library from those sources. Additionally it contains subfolders for the wrapped functions from the underlying SDK's
  - **pco.convert**: export functions, structures and defines from the pco.convert library
  - **pco.sdk**: export functions, structures and defines from the pco.sdk library
  - **pco.recorder**: export functions, structures and defines from the pco.recorder library
- a **bin** folder containing the required library and runtime Dll's.

1.2 Basic Usage

For a simple integration of this SDK, there is a ***pco_csharp.csproj*** inside the **pco** folder. This includes the sources and builds a class library from it.

So for your application you can simply add this project to your Visual Studio solution and refer to it. With this you can simply import the namespaces with the `using` directive as shown in the example code below.
The ***pco_csharp_sample.sln*** shows how this can be done.

```
using pco;
using pco.recorder;
using pco.sdk;
using System;
using System.Threading;

class Program
{
  static void Main(string[] args)
  {
    try
    {
      pco.Camera cam = new pco.Camera();
      pco.Image img = new pco.Image();

      cam.setExposureTime(0.01);

      cam.record(10, pco.RecordMode.sequence);
      cam.image(img, 1, pco.DataFormat.BGR8)
    }
    catch (pco.Camera_Exception ex)
    {
      Console.WriteLine(ex);
      if (ex.error_Code != 0)
        Console.WriteLine("0x{0:X}", ex.error_Code);
    }
    catch (Exception ex)
    {
      Console.WriteLine("Unknown Exception caught.");
      Console.WriteLine(ex);
    }
  }
}
```

This snippet shows the basic usage.

As soon as a `Camera` object is created, a camera is searched, opened and initialized. There are several functions to adjust the camera settings. Here we set the exposure time to 10 ms using `cam.setExposureTime`. Calling `record()` will start the recording. Depending on the recorder mode, the function either waits until record is finished (like for sequence mode which is selected here) or directly returns (see 1.3 for the full list of available modes).

The `Image` class handles the image data, i.e. it enables you to easily get the data either as 16 bit raw image or in various color and monochrome formats (see 1.4 for the full list of available formats).

With the `image` / `images` / `imageAverage` functions you can get the recorded images in several different formats. [1]

Here we want to have the image with **index** 1 in the *BGR8* format.

1.3 Recorder Modes

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

Some modes are blocking, i.e. the `record` function waits until recording is finished, some are non-blocking.

Some modes store images in memory, other save images directly to file(s) on the disk and some are recording and reading directliy into and from camera internal memory. However, for all modes, the recorded images can be accessed in the same way, just as they would be in memory.

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
`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

Continued on next page

[1] Depending on the camera

Continued from previous page

Mode	Storage	Blocking	Description
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 the recorder mode is represented as an enum type:

```
public enum RecordMode : ushort
{
    sequence, sequence_non_blocking, ring_buffer, fifo,
    sequence_dpcore, sequence_non_blocking_dpcore,
    ring_buffer_dpcore, fifo_dpcore,
    tif, multitif, pcoraw, b16, dicom, multidicom,
    camram_ring, camram_segment
};
```

Additionally a flag can be optionally set for the recorder mode. In the code those flags are represented as an enum type:

```
public enum RecordFlags : ushort
{
    memory_use_dpcore = 0x0100, //PCO_RECORDER_DEFINES.↩
        PCO_RECORDER_MODE_USE_DPCORE
    file_split_doubleimg = 0x8000, //PCO_RECORDER_DEFINES.↩
        PCO_RECORDER_DOUBLEIMG_SPLIT
    file_split_doubleimg_sequence = 0xC000 //PCO_RECORDER_DEFINES.↩
        PCO_RECORDER_DOUBLEIMG_SPLIT_SEQUENCE
};
```

Note Setting ROI and using PCO_RECORDER_DOUBLEIMG_SPLIT or PCO_RECORDER_DOUBLEIMG_-SPLIT_SEQUENCE is applied to each split image. For more information on the DotPhoton preparation and image compression, please visit DotPhoton or feel free to contact us.

1.4 Image Formats

In addition to the standard 16-bit raw image data you can also get images in different formats, shown in the table below.

The format is selected when calling the `image` / `images` / `imageAverage` functions (see 2.1.30, 2.1.31, 2.1.32) of the `Camera` class. The image data is stored in an `Image` object, which enables you to access both the the raw data and the image data in the selected format.

Format	Description
`Mono8`	Get image as 8-bit grayscale data
`Mono16`	Get image as 16-bit grayscale/raw data
`BGR8`	Get image as 24-bit color data in bgr format
`BGRA8`	Get image as 32-bit color data (with alpha channel) in bgra format
`BGR16`	Get image as 48-bit color data in bgr format (only possible for color cameras)

In the code the data format is represented as an enum type:

```
public enum DataFormat : ushort
{
    Undefined,
    Mono8, // 8-bit camera, compressed images
    Mono16,
    BGR8,
    BGRA8,
    BGR16,
    CompressedMono8
};
```

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 2.1.27) or `loadlut` (see 2.1.28) from the camera class.

1.5 Error Handling

In the example in 1.2, the code is surrounded by a try-catch block.

Error handling works this way:

- the underlying SDKs (**pco.sdk, pco.recorder, pco.convert**) have a C-API which provides error codes as return values of the exported functions
- the `Camera` and `Image` classes in this package use the `Camera_Exception` class to transform those error codes into an exception
- this exception is then thrown by the class in case something goes wrong.

For robust programs we recommend to always surround code, where `Camera` and `Image` class functions are used, with a try-catch and react on the error in the catch block.

The ILogger interface is supported and can be passed in the constructor of the `Camera` and `Image` class.

Supported logging levels are: `Error`, `Warnings` and `Information`

Additionally you can also enable the logging of the underlying SDK's. For more information on that please visit our pco.logging page.

2 API Documentation

The pco.csharp package consists of 3 different classes:

- `pco.Camera` is the main class for controlling the camera, acquiring and reading images
- `pco.Image` is the class for handling the image data. Images can have various formats, but the raw data is also available
- `pco.Camera_Exception` is an exception class for mapping PCO error codes to `Exception` objects.

2.1 pco.Camera

This section describes the functions of the `Camera` class. The following list provides a short overview of the most important functions:

- **Constructor** open and initialize a camera with its default configuration
- **Destructor** close the camera and clean up everything
- **defaultConfiguration()** set default configuration to the camera
- **getConfiguration()** get current camera configuration
- **setConfiguration()** set a new configuration to the camera
- **configureHWIO_*_***()** configure the HWIO channels (1-4)
- **autoExposureOn(), autoExposureOff()** switch auto exposure on/off
- **configureAutoExposure()** set the parameters for auto exposure calculations
- **getExposureTime()** get current exposure time
- **setExposureTime()** set new exposure time to the camera
- **record()** initialize and start the recording of images
- **stop()** stop the current recording
- **waitForFirstImage()** wait until the first image has been recorded
- **waitForNewImage()** wait until a new image has been recorded
- **getConvertControl()** get current color convert settings
- **setConvertControl()** set new color convert settings
- **image()** read a recorded image
- **images()** read a series of recorded images
- **imageAverage()** read an averaged image (averaged over all recorded images)
- **hasRam()** check if camera has internal memory for recording with camram
- **switchToCamRam()** set the camram segment where the images should be written to/read from
- **getCamRamSegment()** get segment number of the active segment
- **getCamRamMaxImages()** get number of images that can be stored in the active segment
- **getCamRamNumImages()** get number of images that are available in the active segment
- **setCamRamAllocation()** set allocation distribution of camram segments.

2.1.1 Constructor

Description Initialize the camera.

Prototype

```
Camera(
    CameraInterface cameraInterface = CameraInterface.Any,
    UInt32 serial = PCO_CAMERA_DEFINES.UNDEF_DW,
    ILogger log = null
);
```

Parameter

Name	Type	Description
cameraInterface	CameraInterface	Specific interface to search for cameras. If undefined, search on all interfaces.
serial	UInt32	Search for the camera with this specific serial number. If undefined, search for any camera.

Note

```
public enum CameraInterface : ushort
{
    FireWire        = PCO_SDK_DEFINES.PCO_INTERFACE_FW,
    CameraLinkMTX   = PCO_SDK_DEFINES.PCO_INTERFACE_CL_MTX,
    GenICam         = PCO_SDK_DEFINES.PCO_INTERFACE_GENICAM,
    CameraLinkNAT   = PCO_SDK_DEFINES.PCO_INTERFACE_CL_NAT,
    GigE            = PCO_SDK_DEFINES.PCO_INTERFACE_GIGE,
    USB             = PCO_SDK_DEFINES.PCO_INTERFACE_USB,
    CameraLinkME4   = PCO_SDK_DEFINES.PCO_INTERFACE_CL_ME4,
    USB3            = PCO_SDK_DEFINES.PCO_INTERFACE_USB3,
    WLAN            = PCO_SDK_DEFINES.PCO_INTERFACE_WLAN,
    CLHS            = PCO_SDK_DEFINES.PCO_INTERFACE_CLHS,
    Any             = PCO_CAMERA_DEFINES.UNDEF_W
};
```

2.1.2 Destructor

Description Close the activated camera and release the blocked resources.

Prototype

```
Camera.Dispose();

// or as an alternative directly calling
Camera.close();
```

2.1.3 isRecording

Description Return the flag if a recording is currently active.

Prototype

```
bool isRecording();
```

Return value

Name	Type	Description
recording	bool	Flag if a recording is currently active

2.1.4 isColored

Description Return the flag if camera is a color camera.

Prototype

```
bool isColored();
```

Return value

Name	Type	Description
colored	bool	Flag if camera is colored

2.1.5 getDescription

Description Return the description parameters of the camera.

Prototype

```
Description getDescription();
```

Return value

Name	Type	Description
description	Description	Structure containing the camera description (see 2.4.5)

2.1.6 defaultConfiguration

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

Prototype

```
void defaultConfiguration();
```

2.1.7 getRawFormat

Description Get the current raw pixel format

Prototype

```
RawFormat getRawFormat();
```

Return value

Name	Type	Description
getRawFormat	RawFormat	Current raw format

2.1.8 getConfiguration

Description Get the current camera configuration.

Prototype

```
Configuration getConfiguration();
```

Return value

Name	Type	Description
configuration	Configuration	Structure containing the current configuration of the camera (see 2.4.4)

2.1.9 setConfiguration

Description Set a configuration to the camera.

Prototype

```
void setConfiguration(Configuration config);
```

Parameter

Name	Type	Description
config	Configuration	Configuration that should be set (see 2.4.4).

2.1.10 configureHWIO_1_exposureTrigger

Description Configure the HWIO connector 1.

This connector is used for the exposure trigger signal input.

Prototype

```
void configureHWIO_1_exposureTrigger(
  bool on,
  HWIO_EdgePolarity polarity
);
```

Parameter

Name	Type	Description
on	bool	Flag if the HWIO connector should be enabled or disabled
polarity	HWIO_EdgePolarity	Polarity the connector should react on (see 2.1.13.1)

2.1.11 configureHWIO_2_acquireEnable

Description Configure the HWIO connector 2.

This connector is used for the acquire enable signal input.

Prototype

```
void configureHWIO_2_acquireEnable(
  bool on,
  HWIO_Polarity polarity
);
```

Parameter

Name	Type	Description
on	bool	Flag if the HWIO connector should be enabled or disabled
polarity	HWIO_Polarity	Polarity the connector should have (see 2.1.13.1)

2.1.12 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

```
bool configureHWIO_3_statusBusy(
  bool on,
  HWIO_Polarity polarity,
  HWIO_3_SignalType signal_type
);
```

Parameter

Name	Type	Description
`on`	`bool`	Flag if the HWIO connector should be enabled or disabled
`polarity`	`HWIO_Polarity`	Polarity the connector should have (see 2.1.13.1)
`signal_type`	`HWIO_3_SignalType`	Type of the signal the connector should have (see 2.1.13.1)

Return value

Name	Type	Description
`signal_type_valid`	`bool`	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.

2.1.13 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

```
bool configureHWIO_4_statusExpos(
  bool on,
  HWIO_Polarity polarity,
  HWIO_4_SignalType signal_type,
  HWIO_StatusExpos_Timing signal_timing = HWIO_StatusExpos_Timing.↩
     undefined
);
```

Parameter

Name	Type	Description
`on`	`bool`	Flag if the HWIO connector should be enabled or disabled
`polarity`	`HWIO_Polarity`	Polarity the connector should have (see 2.1.13.1)
`signal_type`	`HWIO_4_SignalType`	Type of the signal the connector should have (see 2.1.13.1)
`signal_timing`	`HWIO_StatusExpos_Timing`	Timing of exposure output signal (see 2.1.13.1). Only valid for Rolling Shutter cameras and `signal_type status_expos` (default is `undefined`, i.e. will not be set)

Return value

Name	Type	Description
`signal_type_valid`	`bool`	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.

2.1.13.1 HWIO Types

For the **configureHWIO_****** functions we have the following enum definitions:

HWIO_Polarity

```
public enum HWIO_Polarity : UInt16
{
  high_level      = 0x0001,
  low_level       = 0x0002
};
```

HWIO_-EdgePolarity

```
public enum HWIO_EdgePolarity : UInt16
{
  high_level      = 0x0004,
  low_level       = 0x0008
};
```

HWIO_3_-SignalType

```
public enum HWIO_3_SignalType : UInt32
{
  status_busy = 0,
  status_line = 2,
  status_armed = 3
};
```

HWIO_4_-SignalType

```
public enum HWIO_4_SignalType : UInt32
{
  status_expos = 1,
  status_line = 2,
  status_armed = 3
};
```

HWIO_-StatusExpos_-Timing

```
public enum HWIO_StatusExpos_Timing : UInt32
{
  undefined     = 0x00000000,
  first_line    = 0x00000001,
  global        = 0x00000002,
  last_line     = 0x00000003,
  all_lines     = 0x00000004
};
```

2.1.14 configureAutoExposure

Description Set the auto exposure parameters.

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

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

Prototype

```
void configureAutoExposure(
  AutoExposureRegion region_type,
  double min_exposure_s,
  double max_exposure_s);
```

Parameter

Name	Type	Description
`region_type`	`AutoExposureRegion`	Image region type that should be used for auto exposure computation (see 2.4.1).
`min_exposure_s`	`double`	Minimum exposure value that can be used for auto exposure
`max_exposure_s`	`double`	Maximum exposure value that can be used for auto exposure

2.1.15 autoExposureOn

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 `configureAutoExposure()`.

Prototype

```
void autoExposureOn();
```

2.1.16 autoExposureOff

Description Deactivate the auto exposure feature.

Prototype

```
void autoExposureOff();
```

2.1.17 getExposureTime

Description Get the current exposure time of the camera.

Prototype

```
double getExposureTime();
```

Return value

Name	Type	Description
exposure_time_s	double	Exposure time of the camera [s]

2.1.18 setExposureTime

Description Set a new exposure time to the camera.

Prototype

```
void setExposureTime(double exposure_time_s);
```

Parameter

Name	Type	Description
exposure_time_s	double	Exposure time [s] that should be set

2.1.19 getDelayTime

Description Get the current delay time of the camera.

Prototype

```
double getDelayTime();
```

Return value

Name	Type	Description
delay_time_s	double	Delay time of the camera [s]

2.1.20 setDelayTime

Description Set a new delay time to the camera.

Prototype

```
void setDelayTime(double delay_time_s);
```

Parameter

Name	Type	Description
delay_time_s	double	Delay time [s] that should be set

2.1.21 record

Description Create, configure, and start a new recorder instance. The entire camera configuration must be set before calling `record()`. The commands for getting and setting delay/exposure time are the only exception. These can be called up during the recording.

Prototype

```
void record(
    int num_images = 1,
    RecordMode record_mode = RecordMode.sequence,
    string file_path = null,
    RecordFlags record_flags = 0
);
```

Parameter

Name	Type	Description
num_images	int	Sets the number of images allocated in the driver. The RAM, disk (of the PC) or camera RAM (depending on the mode) limits the maximum value.
record_mode	RecordMode	Defines the recording mode for this record (see 1.3).
file_path	string	Path where the image file(s) should be stored (only for modes who directly save to file, see 1.3).
record_flags	RecordFlags	Additional recorder mode flags (see 1.3).

2.1.22 stop

Description Stop the current recording.

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

```
void stop();
```

2.1.23 waitForFirstImage

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

Prototype

```
void waitForFirstImage(
    bool delay = true,
    double timeout_s = default
);
```

Parameter

Name	Type	Description
delay	bool	Flag if a small delay should be used in the waiting loop (typically recommended to reduce CPU load)
timeout_s	double	If defined, the waiting loop will be aborted if no image was recorded during `timeout_s` seconds.

2.1.24 waitForNewImage

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

Prototype

```
void waitForNewImage(
    bool delay = true,
    double timeout_s = default
);
```

Parameter

Name	Type	Description
delay	bool	Flag if a small delay should be used in the waiting loop (typically recommended to reduce CPU load)
timeout_s	double	If defined, the waiting loop will be aborted if no new image was recorded during timeout_s seconds.

2.1.25 getRecordedImageCount

Description Get the number of currently recorded images.

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.
In these cases, check for if (cam.getRecordedImageCount()> 0) to see if a new image is available.

Prototype

```
uint getRecordedImageCount();
```

Return value

Name	Type	Description
recorded_image_count	UInt32	Number of currently recorded images

2.1.26 getConvertControl

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

Prototype

```
ConvertControl getConvertControl(DataFormat data_format);
```

Parameter

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

Return value

Name	Type	Description
convert_control	ConvertControl	Structure containing the current convert settings for the specified data format(see 2.4.6)

2.1.27 setConvertControl

Description Set convert control settings for the specified data format.

Prototype

```
void setConvertControl(
    DataFormat data_format,
    ConvertControl convert_ctrl
);
```

Parameter

Name	Type	Description
data_format	DataFormat	Data format for which the convert settings should be set.
convert_ctrl	ConvertControl	Convert control settings that should be set.

Example

```
pco.ConvertControl conv_ctrl = getConvertControl(pco.DataFormat.BGR8)↩
    ;
if (conv_ctrl is ConvertControlPseudoColor)
{
    ConvertControlPseudoColor cc = (ConvertControlPseudoColor)(↩
        conv_ctrl);
    cc.lut_file = lut_file;
    cam.setConvertControl(pco.DataFormat.BGR8, cc);
}
```

2.1.28 loadLut

Description Set the lut file for the convert control settings.

This is just a convenience function, the lut file could also be set using `setConvertControl` (see: 2.1.27).

Prototype

```
void loadLut(
    DataFormat data_format,
    string lut_file);
```

Parameter

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

2.1.29 adaptWhiteBalance

Description Do a white-balance using a transferred image.

Prototype

```
void adaptWhiteBalance(Image image, Roi crop = null);
```

Parameter

Name	Type	Description
`image`	`Image`	Image that should be used for white-balance computation
`crop`	`Roi`	Use only the specified ROI for white-balance computation

2.1.30 image

Description

Get a recorded image in the given format. The type of the image is an `Image` object (see 2.2).

The `Image` object has to be created by the caller and transferred to the function. Internally, it automatically checks the allocated buffer size and adapts it according to the format and ROI. There is no special pre-allocation needed.

Performance can be increased through the definition of roi and data format or reusing the `Image` object.

Prototype

```
void image(
  Image image,
  uint image_index = 0,
  Roi crop = null,
  DataFormat data_format = DataFormat.Undefined,
  PCO_Recorder_CompressionParams comp_params = default
);
```

Parameter

Name	Type	Description
`image`	`Image`	`Image` object for storing the image
`image_index`	`uint`	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))
`crop`	`Roi`	Soft ROI to be applied, i.e. get only the ROI portion of the image (see 2.4.3 for the `Roi` structure)
`data_format`	`DataFormat`	Data format the image should have (see 1.4)
`comp_params`	`PCO_Recorder_CompressionParams`	Compression parameters, not implemented yet

2.1.31 images

Description

Get a series of images in the given format as `List`. The type of the images is an `Image` object (see 2.2).

The position of the images in the recorder to query are defined by a start index and the length of the transferred List that should hold the images (i.e. there is no additional length parameter)

The `Image` List has to be created by the caller and transferred to the function. Internally, the function automatically checks if `Image` Objects already exist or not. When the List is empty, it is filled with `Image` Objects, otherwise the existing `Image` objects are updated. There is no special pre-allocation needed. Performance can be increased through the definition of ROI and data format of the List's `Image` objects.

Prototype

```
void images(
  List<Image> images,
  Roi crop = default,
  uint start_index = 0,
  DataFormat data_format = DataFormat.Undefined,
  PCO_Recorder_CompressionParams comp_params = default
);
```

Parameter

Name	Type	Description
images	List<Image>	A List of `Image` objects for storing the images
crop	Roi	Soft ROI to be applied, i.e. get only the ROI portion of the images (see 2.4.3 for the `Roi` structure)
start_index	uint	Index of the first image that should be queried (the number of images is defined by the length of the image vector)
data_format	DataFormat	Data format the images should have (see 1.4)
comp_params	PCO_Recorder_CompressionParams	Compression parameters, not implemented yet

2.1.32 imageAverage

Description Get an averaged image, averaged over all recorded images in the given format. The type of the image is a `Image` object (see 2.2).

The `Image` object has to be created by the caller and transferred to the function. Internally it automatically checks the allocated buffer size and adapts it according to the format and ROI. There is no special pre-allocation needed.

Note We recommend that you not use this function while recording is active, as it may give unexpected results (especially in `ring_buffer` mode, see 1.3).
Record the number of images you want to average as a sequence, then after all images have been recorded, use this function to calculate the average.

Prototype

```
void imageAverage(
    Image image,
    Roi crop = default,
    DataFormat data_format = DataFormat.Undefined
);
```

Parameter

Name	Type	Description
`image`	`Image`	`Image` object for storing the averaged image
`crop`	`Roi`	Soft ROI to be applied, i.e. get only the ROI portion of the image (see 2.4.3 for the `Roi` structure).
`data_format`	`DataFormat`	Data format the averaged image should have (see 1.4)

2.1.33 hasRam

Description Flag indicating whether camera-internal memory for recording with camram is available

Prototype

```
bool hasRam();
```

Return value

Name	Type	Description
has_camram	bool	Boolean indicating whether cam ram is available

2.1.34 switchToCamRam

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

Prototype

```
void switchToCamRam();

void switchToCamRam(ushort segment);
```

Parameter

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

2.1.35 setCamRamAllocation

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

```
void setCamRamAllocation(ArrayList percents);
```

Parameter

Name	Type	Description
percents	ArrayList	Array that holds percentages of segment distribution. Length: 1 <= size( ) <= 4

2.1.36 getCamRamSegment

Description Get segment number of active camram segment.

Prototype

```
ushort getCamRamSegment();
```

Return value

Name	Type	Description
segment_num	ushort	Number of active camram segment

2.1.37 getCamRamMaxImages

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

Prototype

```
uint getCamRamMaxImages();
```

Return value

Name	Type	Description
max_image_count	uint	Maximal images for recording to active segment

2.1.38 getCamRamNumImages

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

Prototype

```
uint getCamRamNumImages();
```

Return value

Name	Type	Description
image_count	uint	Number of images available for readout from active segment

2.1.39 getConv

Description Get the internal handle to the pco.convert API for a specific image format. This is needed whenever you need to call special pco.convert functions directly.

Prototype

```
IntPtr getConv(DataFormat data_format);
```

Parameter

Name	Type	Description
data_format	DataFormat	Data format for which the convert handle should be queried.

Return value

Name	Type	Description
conv	IntPtr	Handle to the pco.convert library functions

2.1.40 Accessors

Accessors are function-like possibilities to get some properties of a `Camera` object, which shouldn't be overwritten.

2.1.40.1 cameraName

Description Get the name of the camera

Prototype

```
string cameraName;
```

Return value

Name	Type	Description
name	string	Camera name

2.1.40.2 cameraSerial

Description Get the serial number of the camera.

Prototype

```
uint cameraSerial;
```

Return value

Name	Type	Description
serial_number	uint	Camera serial number

2.1.40.3 sdk

Description Get the internal handle to the pco.sdk API. This is needed whenever you need to call special pco.sdk functions directly.

Prototype

```
IntPtr sdk;
```

Return value

Name	Type	Description
sdk	IntPtr	Handle to the pco.sdk library functions

2.1.40.4 rec

Description Get the internal handle to the pco.recorder API. This is needed whenever you need to call special pco.recorder functions directly.

Prototype

```
IntPtr rec;
```

Return value

Name	Type	Description
rec	IntPtr	Handle to the pco.recorder library functions

2.2 pco.Image

The `Image` class stores the data of an image. With convenient methods you can access the raw image data, and if available, additional information such as metadata and timestamp.

The following list provides an overview of the functions:

- **Constructor** can be called with and without camera or image-size information. If called with image-size and data format information, the image buffer is pre-allocated according to data format and ROI
- **is8Bit()** get flag if the stored image is 8-bit
- **is16Bit()** get flag if the stored image is 16-bit
- **isColored()** get flag if the stored image is a color image
- **getDataFormat()** get the format of the stored image
- **width()** get width of the stored image
- **height()** get height of the stored image
- **validAllocation()** check pre-allocation of image buffer according the parameter data format and ROI
- **resize()** adapt allocation of the image buffer according to the parameter data format and ROI
- **setRecorderImageNumber()** set number of the stored image (used in `Camera` class internally)
- **getRecorderImageNumber()** get number of the stored image
- **setMetaData()** set metadata of the stored image (used in `Camera` class internally)
- **clearMetaData()** clear metadata of the stored image
- **getMetaDataRef()** get reference to the metadata of the stored image
- **getMetaData()** get metadata of the stored image
- **setTimestamp()** set timestamp of the stored image (used in `Camera` class internally)
- **clearTimestamp()** clear timestamp of the stored image
- **getTimestamp()** get timestamp of the stored image
- **getTimestampRef()** get reference to the timestamp of the stored image
- **size()** get image size in pixel
- **vector_8bit()** get image data as `byte[]` array of 8-bit values (for 8-bit image formats)
- **vector_16bit()** get image data as `ushort[]` array of 16-bit values (for 16-bit image formats)
- **raw_vector_8bit()** get raw image data as `ushort[]` array of 8-bit values
- **raw_vector_16bit()** get raw image data as `ushort[]` array of 16-bit values
- **raw_is8Bit()** get flag if the raw image data is 8 bit
- **raw_is16Bit()** get flag if the raw image data is 16 bit.

2.3 pco.Camera_Exception

The `Camera_Exception` class is derived from `Exception` and transforms PCO error codes into exception objects which are thrown by the `Camera` class in case of an error. With this workflow you can catch camera errors with a try-catch block just like any other `Exception`.

This class only introduce additional Constructors, thus it has the same set of functions as the regular `System.Exception`.

The following list provides an overview of these Constructors:

- **Camera_Exception(string message)** creates `Exception` with this message
- **Camera_Exception(uint err_code)** transforms the PCO error code and creates `Exception` with this error code message
- **Camera_Exception(string message, uint err_code)** transforms the PCO error code, creates `Exception` with this error code message and appends it to the message
- **Camera_Exception(string message, Exception inner)** appends any `Exception` Error message to this message.

2.4 Structs

In the following sections you will find all structures used in the `Camera` class.

2.4.1 AutoExposure

Description Structure holding the auto exposure information.

Name	Type	Description
`region`	`AutoExposureRegion`	Region type that should be used for auto exposure calculation (see below for explanation)
`horz`	`double`	Minimum exposure value that can be used for auto exposure
`mode`	`double`	Maximum exposure value that can be used for auto exposure

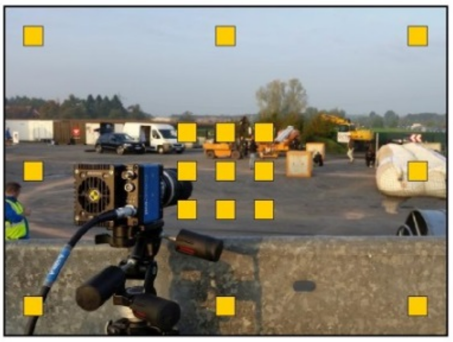
Note

```
public enum AutoExposureRegion
{
  balanced,
  center_based,
  corner_based,
  full
};
```

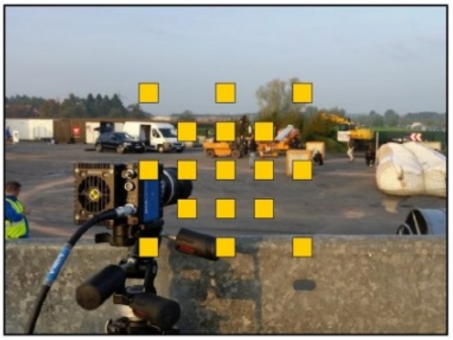
The size of the pixel clusters is fixed, but depends on the overall image size and is treated seperately for width and height:

- 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.

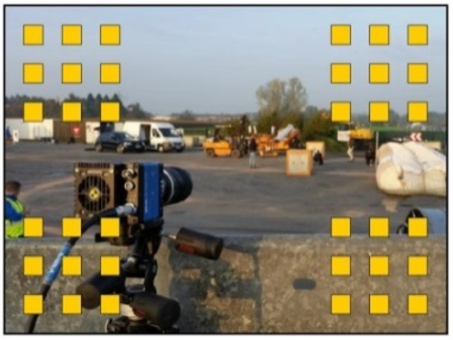
balanced Measurement fields positioned centrally and in all corners



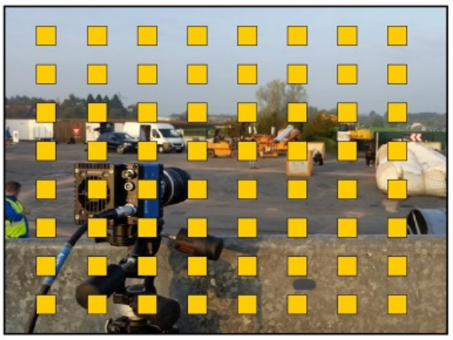
center_based Measurement fields positioned centrally.



corner_based Measurement fields positioned in all four corners.



full Measurement fields across the image.



2.4.2 Binning

Description Structure holding the binning information.

Name	Type	Description
vert	UInt16	Vertical binning
horz	UInt16	Horizontal binning
mode	BinningMode	Binning mode (default is BinningMode.sum)

Note

```
public enum BinningMode
{
  sum,
  average
};
```

2.4.3 Roi

Description Structure holding the ROI information

Name	Type	Description
x0	UInt64	Left position of ROI (starting from 1)
y0	UInt64	Top position of ROI (starting from 1)
x1	UInt64	Right position of ROI (up to full width)
y1	UInt64	Bottom position of ROI (up to full height)

Additionally the following convenience function are available:

- **size_t width()** get width of the ROI
- **size_t height()** get height of the ROI
- **size_t size()** get overall size in pixel
- **size_t evenPaddedWidth()** get padded width
- **size_t paddedSize()** get padded overall size.

roi vs crop `roi` is the hardware ROI. What the camera records and sends to the memory/computer. If it is changed new images have to be recorded to be applied. `crop` is the software ROI. The `crop` can never be greater than the `roi` and can be changed without recording new images.

2.4.4 Configuration

Description Structure holding a camera configuration.

Name	Type	Description
exposure_time_s	double	Exposure time [s]
delay_time_s	double	Delay time [s]

Continued on next page

Continued from previous page

Name	Type	Description
roi	Roi	Hardware ROI structure (see 2.4.3)
timestamp_mode	UInt16	Timestamp mode
pixelrate	UInt32	Pixelrate
trigger_mode	UInt16	Trigger mode
acquire_mode	UInt16	Acquire mode
metadata_mode	UInt16	Metadata mode
noise_filter_mode	UInt16	Noise filter mode
binning	Binning	Binning structure (see 2.4.2)
auto_exposure	AutoExposure	Auto-Exposure structure (see 2.4.1)

2.4.5 Description

Description Structure holding the camera description information.

Name	Type	Description
serial	UInt32	Serial number of the camera
type	UInt16	Sensor type
sub_type	UInt16	Sensor sub type
interface_type	UInt16	Interface type
min_exposure_time_s	double	Minimal possible exposure time
max_exposure_time_s	double	Maximal possible exposure time
min_exposure_step_s	double	Minimal possible exposure step
min_delay_time_s	double	Minimal possible delay time
max_delay_time_s	double	Maximal possible delay time
min_delay_step_s	double	Minimal possible delay step
min_width	UInt64	Minimal possible image width (hardware ROI)
min_height	UInt64	Minimal possible image height (hardware ROI)
max_width	UInt64	Maximal possible image width (hardware ROI)
max_height	UInt64	Maximal possible image height (hardware ROI)
roi_step_horz	UInt64	Horizontal ROI stepping (hardware ROI)
roi_step_vert	UInt64	Vertical ROI stepping (hardware ROI)
roi_symmetric_horz	bool	Flag if hardware ROI has to be horizontally symmetric (i.e. if x0 is increased, x1 has to be decreased by the same value)

Continued on next page

Continued from previous page

Name	Type	Description
roi_symmetric_vert	bool	Flag if hardware ROI has to be vertically symmetric (i.e. if y0 is increased, y1 has to be decreased by the same value)
bit_resolution	UInt16	Bit-resolution of the sensor
has_timestamp_mode	bool	Flag if camera supports the timestamp setting
has_timestamp_mode_ascii_only	bool	Flag if camera supports setting the timestamp to ascii-only
pixelrate_vec	List<UInt32>	Vector containing all possible pixelrate frequencies (index 0 is default)
has_trigger_mode_extexpctrl	bool	Flag if camera supports trigger mode external exposure control
has_acquire_mode	bool	Flag if camera supports the acquire mode setting
has_ext_acquire_mode	bool	Flag if camera supports the external acquire setting
has_metadata_mode	bool	Flag if metadata can be activated for the camera
has_ram	bool	Flag if camera has internal memory
binning_horz_vec	List<UInt16>	Vector containing all possible horizontal binning values
binning_vert_vec	List<UInt16>	Vector containing all possible vertical binning values
has_average_binning	bool	Flag if camera supports average binning

2.4.6 ConvertControl

Description Structure containing (color) convert information.

Depending on the image format (see 1.4) a different structure will be used.

Mono8 format **ConvertControlMono**

Name	Type	Description
sharpen	bool	Flag if the image should be sharpened
adaptive_sharpen	bool	Flag if adaptive sharpening should be enabled
flip_vertical	bool	Flag if the image should be vertically flipped
auto_minmax	bool	Flag if auto scale should be enabled
add_conv_flags	int	Variable to set additional flags for image/color conversion (default is 0)
min_limit	int	Minimum scaling value (will be ignored if auto scale is enabled)

Continued on next page

Continued from previous page

Name	Type	Description
max_limit	int	Maximum scaling value (will be ignored if auto scale is enabled)
gamma	double	Gamma of the image (default is 1.0)
contrast	int	Contrast of the image (default is 0)

Color camera and color format

ConvertControlColor

Name	Type	Description
sharpen	bool	Flag if the image should be sharpened
adaptive_sharpen	bool	Flag if adaptive sharpening should be enabled
flip_vertical	bool	Flag if the image should be vertically flipped
auto_minmax	bool	Flag if auto scale should be enabled
add_conv_flags	int	Variable to set additional flags for image/color conversion (default is 0)
min_limit	int	Minimum scaling value (will be ignored if auto scale is enabled)
max_limit	int	Maximum scaling value (will be ignored if auto scale is enabled)
gamma	double	Gamma of the image (default is 1.0)
contrast	int	Contrast of the image (default is 0)
pco_debayer_algorithm	bool	Flag if PCO debayering should be used
color_temperature	int	Color temperature of the image
color_saturation	int	Color saturation of the image
color_vibrance	int	Color vibrance of the image
color_tint	int	Color tint of the image

BW camera and color format

ConvertControlPseudoColor

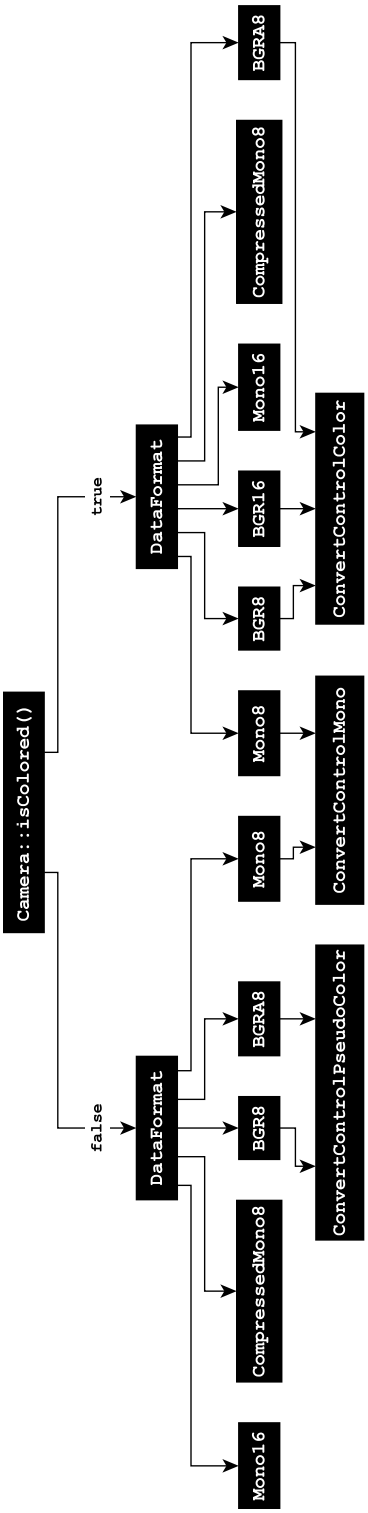
Name	Type	Description
sharpen	bool	Flag if the image should be sharpened
adaptive_sharpen	bool	Flag if adaptive sharpening should be enabled
flip_vertical	bool	Flag if the image should be vertically flipped
auto_minmax	bool	Flag if auto scale should be enabled
add_conv_flags	int	Variable to set additional flags for image/color conversion (default is 0)
min_limit	int	Minimum scaling value (will be ignored if auto scale is enabled)
max_limit	int	Maximum scaling value (will be ignored if auto scale is enabled)
gamma	double	Gamma of the image (default is 1.0)
contrast	int	Contrast of the image (default is 0)
color_temperature	int	Color temperature of the image
color_saturation	int	Color saturation of the image

Continued on next page

Continued from previous page

Name	Type	Description
color_vibrance	int	Color vibrance of the image
color_tint	int	Color tint of the image
lut_file	string	Path of the lut file that should be used

Overview **Assignment of ConvertControl structs to DataFormat and BW/colored camera**



2.5 etc.XCite

2.5.1 Constructor

Description Initialize the connection to an X-Cite® light source. Optionally one can specify either the device or the com Port.

Prototype

```
public Xcite(
    XCiteType type = XCiteType.Any,
    string comPort = ""
)
```

Parameter

Name	Type	Description
type	XCiteType	X-Cite® device type (see 2.5.13.3)
comPort	string	Com Port as a string

2.5.2 Dispose

Description Close the activated connection and release blocked resources.

Prototype

```
    public void Dispose()
```

2.5.3 xcite

Description Return the handler for the xcite connection for the xcite library.

Prototype

```
public IntPtr xcite { get => xcite_; }
```

Return value

Name	Type	Description
xcite_	IntPtr	IntPtr for the current xcite connection

2.5.4 getComPort

Description Return the Com Port of the current connection.

Prototype

```
public string getComPort()
```

Return value

Name	Type	Description
com_port	string	Com Port

2.5.5 getType

Description Return the X-Cite® device type.

Prototype

```
public XCiteType getType()

public string getTypeStr()
```

Return value

Name	Type	Description
type	XCiteType	X-Cite® device type (see 2.5.13.3)
type	string	X-Cite® device type as string (see 2.5.13.3)

2.5.6 getDescription

Description Return the description parameters of the X-Cite® device.

Prototype

```
public XCITE_Description getDescription()
```

Return value

Name	Type	Description
desc	XCITE_Description	Description structure of the X-Cite® device (see 2.5.13.2)

2.5.7 getConfiguration

Description Return the configuration parameters of the X-Cite® device.

Prototype

```
public XCITE_Configuration getConfiguration()
```

Return value

Name	Type	Description
conf	XCITE_Configuration	Configuration structure of the X-Cite® device (see 2.5.13.1)

2.5.8 setConfiguration

Description Write a configuration to the X-Cite® device.

Prototype

```
public void setConfiguration(
    XCITE_Configuration config
)
```

Parameter

Name	Type	Description
config	XCITE_Configuration	Configuration structure for the X-Cite® device (see 2.5.13.1)

2.5.9 defaultConfiguration

Description Reset the configuration of the X-Cite® device to the default values, turns all lights off.

Prototype

```
    public void defaultConfiguration()
```

2.5.10 SwitchOn

Description Switch the configured lights on.

Prototype

```
public void switchOn();
```

2.5.11 SwitchOff

Description Switch all lights off.

Prototype

```
public void switchOff();
```

2.5.12 ExecuteCommand

Description The command list for X-Cite® is available by request. To obtain the latest update, please contact Excelitas Technologies support.

Prototype

```
public string executeCommand(
    string cmd,
    string in_value
)
```

Parameter

Name	Type	Description
cmd	string	Command to be sent to the X-Cite® device
in_value	string	Parameters if necessary for the command

Return value

Name	Type	Description
ret	string	Response string

2.5.13 Structs

In the following sections you will find all structures and enums used in the `XCite` class.

2.5.13.1 XCITE_Configuration

Description Structure holding a X-Cite® configuration.

Name	Type	Description
`intensities`	`UInt32[]`	Vector of available intensities
`on_states`	`Byte[]`	Vector of which lights are on

2.5.13.2 XCITE_Description

Description Structure holding the X-Cite® description information.

Name	Type	Description
`serial`	`UInt32`	Serial number
`type`	`XCiteType`	XCite type (see 2.5.13.3)
`name`	`string`	Name of the X-Cite® device
`wavelengths_vec`	`UInt32[]`	Array of available wavelengths
`exclusivity_vec`	`UInt32[]`	Array of value indicating which wavelengths can be set exclusively (matching wheel number). Wheel number 0: independant activation possible
`intensity_max_vec`	`UInt32[]`	Array of available maximum intensities
`intensity_min_vec`	`UInt32[]`	Array of available minimum intensities

2.5.13.3 XCiteType

Description Enumeration of all XCiteTypes.

```
public enum XCiteType
{
  [Description("120PC")]
  XC_120PC = 0,
  [Description("exacte")]
  XC_exacte,
  [Description("120LED")]
  XC_120LED, // USB 04D8 F615
  [Description("110LED")]
  XC_110LED,
  [Description("mini")]
  XC_mini,
  [Description("XYLIS")]
  XC_XYLIS,
  [Description("XR210")]
  XC_XR210,
  [Description("XLED1")]
```

```
    XC_XLED1,
    [Description("XT600")]
    XC_XT600, // USB 04D8 F53D
    [Description("XT900")]
    XC_XT900,
    [Description("<Undefined Type>")]
    Any = 0xFFFF,
  }
```

2.5.13.4 XCiteException

The `etc.xcite_Exception` class is derived from `exception` and transforms PCO error codes into exception objects which are thrown by the `XCite` class in case of an error. With this workflow you can catch camera errors with a try-catch block just like any other `exception`.

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