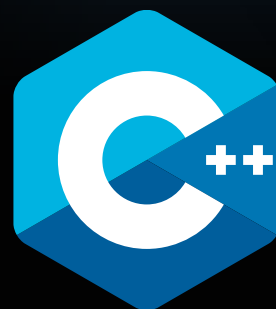


# manual

## pco.cpp

```
pco::Camera cam;  
pco::Image img;  
  
cam.setExposureTime(0.01);  
cam.record(10, pco::RecordMode::sequence);  
cam.image(img, 1, pco::DataFormat::BGR8);  
pco::Camera cam;  
pco::Image img;  
  
cam.setExposureTime(0.01);  
cam.record(10, pco::RecordMode::sequence);  
cam.image(img, 1, pco::DataFormat::BGR8);  
pco::Camera cam;  
pco::Image img;  
  
cam.setExposureTime(0.01);  
cam.record(10, pco::RecordMode::sequence);  
cam.image(img, 1, pco::DataFormat::BGR8);
```



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

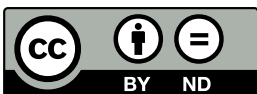
**pco.**<sup>®</sup>

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

pco.cpp manual 1.6.0

Released December 2025

©Copyright Excelitas PCO GmbH



This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

# Contents

<b>1</b>	<b>General</b>	<b>5</b>
1.1	Installation . . . . .	5
1.2	Basic Usage . . . . .	6
1.3	Recorder Modes . . . . .	6
1.4	Image Formats . . . . .	8
1.5	Error Handling . . . . .	10
<b>2</b>	<b>API Documentation</b>	<b>11</b>
2.1	pco::Camera . . . . .	11
2.1.1	Constructor . . . . .	12
2.1.2	Destructor . . . . .	12
2.1.3	getName . . . . .	13
2.1.4	getSerial . . . . .	13
2.1.5	getRawFormat . . . . .	13
2.1.6	getInterface . . . . .	13
2.1.7	isRecording . . . . .	14
2.1.8	isColored . . . . .	14
2.1.9	getDescription . . . . .	14
2.1.10	defaultConfiguration . . . . .	14
2.1.11	getConfiguration . . . . .	15
2.1.12	setConfiguration . . . . .	15
2.1.13	configureHWIO_1_exposureTrigger . . . . .	15
2.1.14	configureHWIO_2_acquireEnable . . . . .	16
2.1.15	configureHWIO_3_statusBusy . . . . .	16
2.1.16	configureHWIO_4_statusExpos . . . . .	17
2.1.16.1	HWIO Types . . . . .	18
2.1.17	configureAutoExposure . . . . .	19
2.1.18	autoExposureOn . . . . .	19
2.1.19	autoExposureOff . . . . .	19
2.1.20	getExposureTime . . . . .	20
2.1.21	setExposureTime . . . . .	20
2.1.22	getDelayTime . . . . .	20
2.1.23	setDelayTime . . . . .	20
2.1.24	record . . . . .	21
2.1.25	stop . . . . .	21
2.1.26	waitForFirstImage . . . . .	21
2.1.27	waitForNewImage . . . . .	22
2.1.28	getRecordedImageCount . . . . .	22
2.1.29	getConvertControl . . . . .	23
2.1.30	setConvertControl . . . . .	23
2.1.31	loadLut . . . . .	24
2.1.32	adaptWhiteBalance . . . . .	24
2.1.33	image . . . . .	25
2.1.34	images . . . . .	26
2.1.35	imageAverage . . . . .	27
2.1.36	hasRam . . . . .	28
2.1.37	switchToCamRam . . . . .	28
2.1.38	setCamRamAllocation . . . . .	28
2.1.39	getCamRamSegment . . . . .	29
2.1.40	getCamRamMaxImages . . . . .	29
2.1.41	getCamRamNumImages . . . . .	29
2.1.42	sdk . . . . .	30

2.1.43	rec	30
2.1.44	conv	30
2.2	pco::Image	31
2.3	pco::CameraException	32
2.4	Structs	33
2.4.1	AutoExposure	33
2.4.2	Binning	35
2.4.3	Roi	35
2.4.4	Configuration	35
2.4.5	Description	36
2.4.6	ConvertControl	37
2.5	etc::XCite	41
2.5.1	Constructor	41
2.5.2	Destructor	41
2.5.3	xcite	41
2.5.4	getComPort	41
2.5.5	getType	42
2.5.6	getDescription	42
2.5.7	getConfiguration	42
2.5.8	setConfiguration	42
2.5.9	defaultConfiguration	43
2.5.10	SwitchOn	43
2.5.11	SwitchOff	43
2.5.12	ExecuteCommand	43
2.5.13	Structs	44
2.5.13.1	XCITE_Configuration	44
2.5.13.2	XCITE_Description	44
2.5.13.3	XCiteType	44
2.5.13.4	XCite_TypeName	45
2.5.13.5	XCiteException	45

### 3 About Excelitas PCO 46

# 1 General

The **pco.cpp** package is 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 acquisition, 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

**Windows** [Download](#) the pco.cpp Windows package, unzip it and execute it. Follow the steps in the installer.

By default, **pco.cpp** will be installed into **C:\Program Files\PCO Digital Camera Toolbox\pco.cpp**.

In your install directory you will find:

- a Visual Studio (2019) solution file with all provided examples
- a **samples** directory containing all example projects
- the **pco.camera** directory containing the actual source files of this class-interface sdk (projects need to cover these source files for compilation)
- the **include** directory containing all necessary headers of pco.sdk and pco.recorder
- the **bin** directory with the compiled example programs and runtime dll
- the **lib** directory with pco.sdk libraries for implicit linking.

**Linux** [Download](#) the pco.cpp Linux package.

Use `dpkg`<sup>1</sup> to install the \*.deb file:

```
sudo dpkg -i pco.cpp_*.*.*_amd64.deb
```

By default, **pco.cpp** will be installed into **/opt/pco/pco.cpp**.

In your install directory you will find:

- a Makefile for compiling all provided examples
- a **samples** directory containing all example projects
- the **pco.camera** directory containing the actual source files of this sdk (projects need to cover these source files for compilation)
- the **include** directory containing all necessary headers of pco.sdk and pco.recorder
- the **bin** directory with the compiled example programs
- the **lib** directory with linux pco.sdk libraries.

To work with the code, you might need to copy the content to a directory where you have full read/write access rights.

---

<sup>1</sup>The dpkg package needs to be installed for this, this can be done by `sudo apt-get install dpkg`.

## 1.2 Basic Usage

```
#include "../pco.camera/stdafx.h"
#include "../pco.camera/camera.h"
#include "../pco.camera/cameraexception.h"

int main()
{
    try
    {
        pco::Camera cam;
        pco::Image img;

        cam.setExposureTime(0.01);

        cam.record(10, pco::RecordMode::sequence);
        cam.image(img, 1, pco::DataFormat::BGR8)
    }
    catch (pco::CameraException& err)
    {
        std::cout << "Error Code: " << err.error_code() << std::endl;
        std::cout << err.what() << std::endl;
    }
    return 0;
}
```

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.<sup>2</sup>

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.

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 (`waitForFirstImage` / `waitForNewImage`), 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.

---

<sup>2</sup>Depending on the camera

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

Mode	Storage	Blocking	Description
<code>sequence</code>	Memory	yes	Record a sequence of images
<code>sequence_non_blocking</code>	Memory	no	Record a sequence of images, do not wait until record is finished
<code>ring_buffer</code>	Memory	no	Continuously record images in a ringbuffer, once the buffer is full, old images are overwritten
<code>fifo</code>	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
<code>sequence_dpcore</code>	Memory	yes	Same as <code>sequence</code> , but with DotPhoton preparation enabled
<code>sequence_non_blocking_dpcore</code>	Memory	no	Same as <code>sequence_non_blocking</code> , but with DotPhoton preparation enabled
<code>ring_buffer_dpcore</code>	Memory	no	Same as <code>ring_buffer</code> , but with DotPhoton preparation enabled
<code>fifo_dpcore</code>	Memory	no	Same as <code>fifo</code> , but with DotPhoton preparation enabled
<code>tif</code>	File	no	Record images directly as tif files
<code>multitiff</code>	File	no	Record images directly as one or more multitiff file(s)
<code>pcoraw</code>	File	no	Record images directly as one pcoraw file
<code>dicom</code>	File	no	Record images directly as dicom files
<code>multidicom</code>	File	no	Record images directly as one or more multidicom file(s)
<code>camram_segement</code>	Camera RAM	no	Record images to camera memory. Stops when segment is full

Continued on next page

Continued from previous page

Mode	Storage	Blocking	Description
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:

```
enum class RecordMode {
    sequence, sequence_non_blocking, ring_buffer, fifo,
    sequence_dpcore, sequence_non_blocking_dpcore,
    ring_buffer_dpcore, fifo_dpcore,
    tif, multitif, pcoraw, bl16, dicom, multidicom,
    camram_segment, camram_ring
};
```

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

```
enum class RecordFlags : WORD {
    memory_use_dpcore          = PCO_RECORDER_MODE_USE_DPCORE,
    file_split_doubleimg       = PCO_RECORDER_DOUBLEIMG_SPLIT,
    file_split_doubleimg_sequence = ↔
    PCO_RECORDER_DOUBLEIMG_SPLIT_SEQUENCE
};
```

Setting ROI and using PCO\_RECORDER\_DOUBLEIMG\_SPLIT or PCO\_RECORDER\_DOUBLEIMG\_SPLIT\_SEQUENCE is applied to each split image.

**Note** 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 2.1.33, 2.1.34, 2.1.35) of the `Camera` class. The image data is stored in an `Image` object, which enables you to access both 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)
RGB8	Get image as 24-bit color data in rgb format
RGBA8	Get image as 32-bit color data (with alpha channel) in rgba format
RGB16	Get image as 48-bit color data in rgb format (only possible for color cameras)

---

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

```
enum class DataFormat {  
    Undefined,  
    Mono8, // 8 bit camera, compressed images  
    Mono16,  
    BGR8,  
    BGRA8,  
    RGB8,  
    RGBA8,  
    RGB16,  
    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 2.1.30) or `loadLut` (see 2.1.31) 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 `CameraException` 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.

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.cpp 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::CameraException` is an exception class for mapping PCO error codes to `std::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.

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.

**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**

```
Camera (
    CameraInterface cam_interface = CameraInterface::Any,
    DWORD serial = UNDEF_DW
);
```

**Parameter**

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

**Note**

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

### 2.1.2 Destructor

**Description** Close the activated camera and release the blocked resources.

**Prototype**

```
~Camera ();
```

### 2.1.3 getName

**Description** Return the name of the camera.

**Prototype**

```
std::string getName() const;
```

**Return value**

Name	Type	Description
name	std::string	Camera name

### 2.1.4 getSerial

**Description** Return the serial number of the camera.

**Prototype**

```
DWORD getSerial() const;
```

**Return value**

Name	Type	Description
serial_number	DWORD	Camera serial number

### 2.1.5 getRawFormat

**Description** Return the pixel format type.

**Prototype**

```
RawFormat getRawFormat() const;
```

**Return value**

Name	Type	Description
getRawFormat	RawFormat	Current raw pixel format of the camera

### 2.1.6 getInterface

**Description** Return the interface.

**Prototype**

```
CameraInterface getInterface() const;
```

**Return value**

Name	Type	Description
interface	CameraInterface	Current Interface of the camera

### 2.1.7 isRecording

**Description** Return the flag if a recording is currently active.

**Prototype**

```
bool isRecording() const noexcept;
```

**Return value**

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

### 2.1.8 isColored

**Description** Return the flag if camera is a color camera.

**Prototype**

```
bool isColored() const;
```

**Return value**

Name	Type	Description
colored	bool	Flag if camera is colored

### 2.1.9 getDescription

**Description** Return the description parameters of the camera.

**Prototype**

```
Description getDescription() const;
```

**Return value**

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

### 2.1.10 defaultConfiguration

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

**Prototype**

```
void defaultConfiguration();
```

### 2.1.11 getConfiguration

**Description** Get the current camera configuration.

**Prototype**

```
Configuration getConfiguration() const;
```

**Return value**

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

### 2.1.12 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).

### 2.1.13 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 2.1.16.1)

### 2.1.14 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 2.1.16.1)

### 2.1.15 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 2.1.16.1)
signal_type	HWIO_3_SignalType	Type of the signal the connector should have (see 2.1.16.1)

**Return value**

Name	Type	Description
signal_type_valid	bool	Flag if the <code>signal_type</code> 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.

### 2.1.16 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 2.1.16.1)
signal_type	HWIO_4_SignalType	Type of the signal the connector should have (see 2.1.16.1)
signal_timing	HWIO_StatusExpos_Timing	Timing of exposure output signal (see 2.1.16.1). Only valid for Rolling Shutter cameras and <code>signal_type status_expos</code> (default is undefined, i.e. will not be set)

**Return value**

Name	Type	Description
signal_type_valid	bool	Flag if the <code>signal_type</code> 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.

### 2.1.16.1 HWIO Types

For the **configureHWIO\_\*\*\*\*** functions we have the following enum definitions:

#### HWIO\_Polarity

```
enum class HWIO_Polarity : WORD {
    high_level      = 0x0001,
    low_level       = 0x0002
};
```

#### HWIO\_- EdgePolarity

```
enum class HWIO_Polarity : WORD {
    rising_edge     = 0x0004,
    falling_edge    = 0x0008
};
```

#### HWIO\_3\_- SignalType

```
enum class HWIO_3_SignalType : DWORD {
    status_busy = 0,
    status_line = 2,
    status_armed = 3
};
```

#### HWIO\_4\_- SignalType

```
enum class HWIO_4_SignalType : DWORD {
    status_expos = 1,
    status_line = 2,
    status_armed = 3
};
```

#### HWIO\_- StatusExpos\_- Timing

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

### 2.1.17 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

### 2.1.18 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();
```

### 2.1.19 autoExposureOff

**Description** Deactivate the auto exposure feature.

**Prototype**

```
void autoExposureOff();
```

### 2.1.20 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]

### 2.1.21 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

### 2.1.22 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]

### 2.1.23 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

### 2.1.24 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(
    DWORD num_images = 1,
    RecordMode record_mode = RecordMode::sequence,
    std::filesystem::path file_path = "",
    WORD record_flags = 0
);
```

**Parameter**

Name	Type	Description
num_images	DWORD	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	std::filesystem::path	Path where the image file(s) should be stored (only for modes who directly save to file, see 1.3).
record_flags	WORD	Optional recorder flag (see 1.3)

### 2.1.25 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();
```

### 2.1.26 waitForFirstImage

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

**Prototype**

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

**Parameter**

Name	Type	Description
delay	bool	Flag if a small delay should be used in the waiting loop (typically recommended to reduce CPU load)

Continued on next page

Continued from previous page

Name	Type	Description
timeout_s	double	If defined, the waiting loop will be aborted if no image was recorded during timeout_s seconds.

### 2.1.27 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 = NAN
);
```

**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.

### 2.1.28 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**

```
size_t getRecordedImageCount() const;
```

**Return value**

Name	Type	Description
recorded_image_count	size_t	Number of currently recorded images

### 2.1.29 getConvertControl

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

**Note** Data formats can share memory, see 2.4.6.

**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)

**Example**

```
pco::ConvertControlPseudoColor cc = std::get<pco::↔
    ConvertControlPseudoColor>(cam.getConvertControl(pco::DataFormat↔
        ::BGR8));
cc.lut_file = lut_file;
cam.setConvertControl(pco::DataFormat::BGR8, cc);
```

### 2.1.30 setConvertControl

**Description** Set convert control settings for the specified data format.

**Note** Data formats can share memory, see 2.4.6.

**Prototype**

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

**Parameter**

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

**Example**

```
pco::ConvertControlPseudoColor cc = std::get<pco::↔
    ConvertControlPseudoColor>(cam.getConvertControl(pco::DataFormat↔
        ::BGR8));
cc.lut_file = lut_file;
cam.setConvertControl(pco::DataFormat::BGR8, cc);
```

### 2.1.31 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: 2.1.30).

**Prototype**

```
void loadLut(
    DataFormat data_format,
    std::filesystem::path lut_file
);
```

**Parameter**

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

### 2.1.32 adaptWhiteBalance

**Description** Do a white-balance using a transferred image.

**Prototype**

```
void adaptWhiteBalance(Image& image);
void adaptWhiteBalance(Image& image, Roi crop);
```

**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

### 2.1.33 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 via reference. 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_ref,
    DWORD image_index = 0,
    DataFormat data_format = DataFormat::Undefined,
    PCO_Recorder_CompressionParams* comp_params = nullptr
);

void image(
    Image& image_ref,
    Roi crop,
    DWORD image_index = 0,
    DataFormat data_format = DataFormat::Undefined,
    PCO_Recorder_CompressionParams* comp_params = nullptr
);
```

**Parameter**

Name	Type	Description
image_ref	Image&	Reference to the <code>Image</code> object for storing the image
crop	Roi	Soft ROI to be applied, i.e. get only the ROI portion of the image (see 2.4.3 for the <code>Roi</code> structure)
image_index	DWORD	Index of the image that should be queried, use <code>PCO_RECORDER_LATEST_IMAGE</code> for latest image (for recorder modes <code>fifo/fifo_dpcore</code> always use 0 (see 1.3))
data_format	DataFormat	Data format the image should have (see 1.4)
comp_params	PCO_Recorder_CompressionParams*	Pointer to compression parameters, not implemented yet

### 2.1.34 images

**Description** Get a series of images in the given format as `std::vector`. 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 vector that should hold the images (i.e. there is no additional length parameter).

The `Image` vector has to be created by the caller and transferred to the function via reference. Internally, the function automatically checks the allocated buffer sizes and adapts them 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 of the vector's `Image` objects.

**Prototype**

```
void images(
    std::vector<Image>& images,
    DataFormat data_format = DataFormat::Undefined,
    size_t start_index = 0,
    PCO_Recorder_CompressionParams* comp_params = nullptr
);

void images(
    std::vector<Image>& images,
    Roi crop,
    DataFormat data_format = DataFormat::Undefined,
    size_t start_index = 0,
    PCO_Recorder_CompressionParams* comp_params = nullptr
);
```

**Parameter**

Name	Type	Description
images	<code>std::vector&lt;Image&gt;&amp;</code>	Reference to a vector of <code>Image</code> objects for storing the images
crop	<code>Roi</code>	Soft ROI to be applied, i.e. get only the ROI portion of the images (see 2.4.3 for the <code>Roi</code> structure)
data_format	<code>DataFormat</code>	Data format the images should have (see 1.4)
start_index	<code>size_t</code>	Index of the first image that should be queried (the number of images is defined by the length of the image vector)
comp_params	<code>PCO_Recorder_CompressionParams*</code>	Pointer to compression parameters, not implemented yet

### 2.1.35 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 via reference. 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_ref,
    DataFormat data_format = DataFormat::Undefined
);

void imageAverage(
    Image& image_ref,
    Roi crop,
    DataFormat data_format = DataFormat::Undefined
);
```

**Parameter**

Name	Type	Description
<code>image_ref</code>	<code>Image&amp;</code>	Reference to the <code>Image</code> object for storing the averaged image
<code>crop</code>	<code>Roi</code>	Soft ROI to be applied, i.e. get only the ROI portion of the image (see 2.4.3 for the <code>Roi</code> structure)
<code>data_format</code>	<code>DataFormat</code>	Data format the averaged image should have (see 1.4)

### 2.1.36 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

### 2.1.37 switchToCamRam

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

**Prototype**

```
void switchToCamRam();  
  
void switchToCamRam(WORD segment);
```

**Parameter**

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

### 2.1.38 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(std::vector<DWORD> percents);
```

**Parameter**

Name	Type	Description
percents	std::vector<DWORD>	Vector that holds percentages of segment distribution. Length: 1 <= size() <= 4

### 2.1.39 getCamRamSegment

**Description** Get segment number of active camram segment.

**Prototype**

```
WORD getCamRamSegment();
```

**Return value**

Name	Type	Description
segment_num	WORD	Number of active camram segment

### 2.1.40 getCamRamMaxImages

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

**Prototype**

```
DWORD getCamRamMaxImages();
```

**Return value**

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

### 2.1.41 getCamRamNumImages

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

**Prototype**

```
DWORD getCamRamNumImages();
```

**Return value**

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

### 2.1.42 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**

```
HANDLE sdk() const;
```

**Return value**

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

### 2.1.43 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**

```
HANDLE rec() const;
```

**Return value**

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

### 2.1.44 conv

**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**

```
HANDLE conv(DataFormat data_format) const;
```

**Parameter**

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

**Return value**

Name	Type	Description
conv	HANDLE	Handle to the pco.convert 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
- **getMetaDataPtr()** get pointer 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
- **getTimestampPtr()** get timestamp of the stored image
- **raw\_data()** get `void` pointer to the raw image data and size in bytes
- **data()** get `void` pointer to the image data and size in bytes
- **size()** get image size in pixel
- **vector\_8bit()** get image data as `std::vector` of 8-bit values (for 8-bit image formats)
- **vector\_16bit()** get image data as `std::vector` of 16-bit values (for 16-bit image formats)
- **raw\_vector\_8bit()** get raw image data as `std::vector` of 8-bit values
- **raw\_vector\_16bit()** get raw image data as `std::vector` of 16-bit values.

## 2.3 pco::CameraException

The `CameraException` class is derived from `std::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 `std::exception`.

The following list provides a short overview of the class functions:

- **Constructor()** get exception message according to the transferred error code
- **what()** get the error text as char pointer
- **error\_code()** get the error code
- **error\_text()** get the error text.

## 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)
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

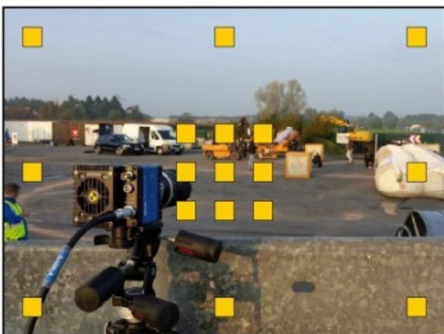
**Note**

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

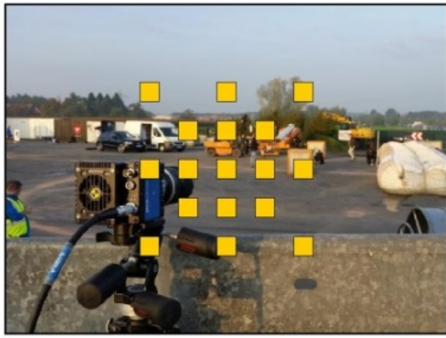
The size of the pixel clusters is fixed, but depends on the overall image size and is treated separately for width and height:

- for width/height  $\geq 1300$  the cluster size is 100
- for  $1300 > \text{width/height} \geq 650$  the cluster size is 50
- for  $650 > \text{width/height} \geq 325$  the cluster size is 25
- for width/height  $< 325$  the cluster size equal to width/height.

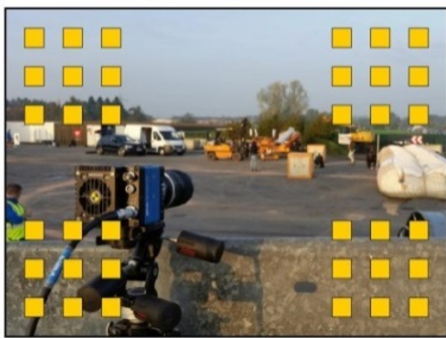
**balanced** Measurement fields positioned centrally and in all corners.



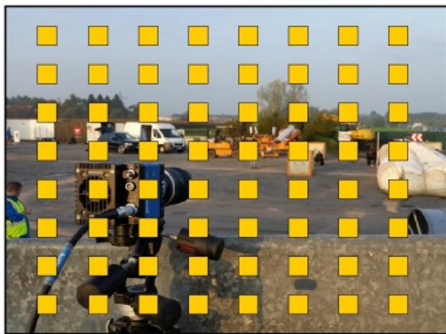
**center\_based** Measurement fields positioned centrally.



**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	WORD	Vertical binning
horz	WORD	Horizontal binning
mode	BinningMode	Binning mode (default is BinningMode::sum)

**Note**

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

## 2.4.3 Roi

**Description** Structure holding the ROI information.

Name	Type	Description
x0	size_t	Left position of ROI (starting from 1)
y0	size_t	Top position of ROI (starting from 1)
x1	size_t	Right position of ROI (up to full width)
y1	size_t	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]
roi	Roi	Hardware ROI structure (see 2.4.3)

Continued on next page

Continued from previous page

Name	Type	Description
timestamp_mode	WORD	Timestamp mode
pixelrate	DWORD	Pixelrate
trigger_mode	WORD	Trigger mode
acquire_mode	WORD	Acquire mode
metadata_mode	WORD	Metadata mode
noise_filter_mode	WORD	Noise filter mode
pixel_format	DWORD	Pixel format
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	DWORD	Serial number of the camera
type	WORD	Sensor type
sub_type	WORD	Sensor sub type
interface_type	CameraInterface	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	size_t	Minimal possible image width (hardware ROI)
min_height	size_t	Minimal possible image height (hardware ROI)
max_width	size_t	Maximal possible image width (hardware ROI)
max_height	size_t	Maximal possible image height (hardware ROI)
roi_step_horz	size_t	Horizontal ROI stepping (hardware ROI)
roi_step_vert	size_t	Vertical ROI stepping (hardware ROI)

Continued on next page

Continued from previous page

Name	Type	Description
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)
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	WORD	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	std::vector<DWORD>	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	std::vector<WORD>	Vector containing all possible horizontal binning values
binning_vert_vec	std::vector<WORD>	Vector containing all possible vertical binning values
has_average_binning	bool	Flag if camera supports average binning
pixel_format_vec	std::vector<WORD>	Vector containing all possible pixel formats

## 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	WORD	Variable to set additional flags for image/color conversion (default is 0)
min_limit	WORD	Minimum scaling value (will be ignored if auto scale is enabled)
max_limit	WORD	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**

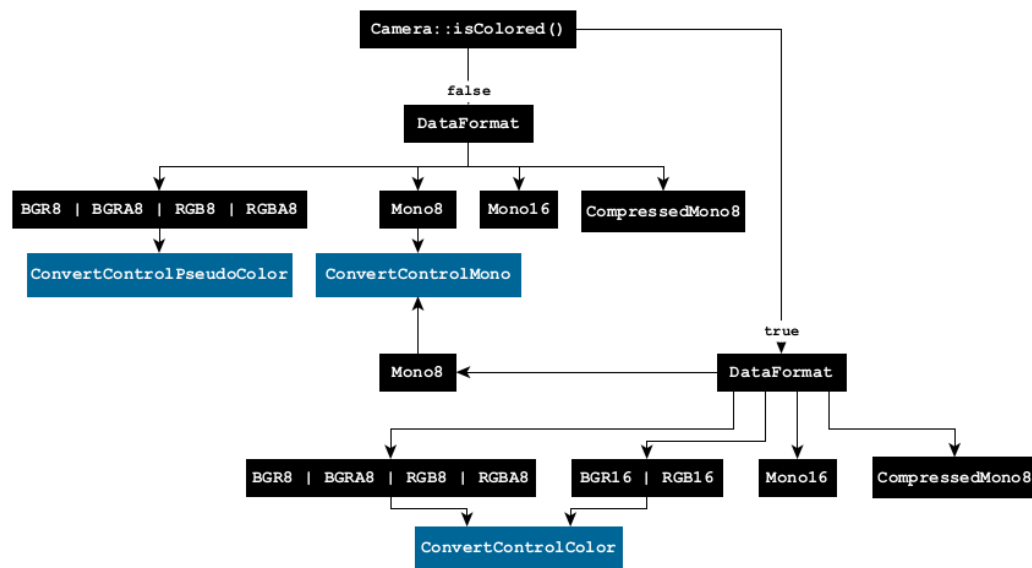
The data formats for 16-bit/8-bit share memory for this structure respectively (i.e. BGR changes will be applied to RGB/RGBA/BGRA as well.).

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	WORD	Variable to set additional flags for image/color conversion (default is 0)
min_limit	WORD	Minimum scaling value (will be ignored if auto scale is enabled)
max_limit	WORD	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**

The data formats for 16-bit/8-bit share memory for this structure respectively (i.e. BGR changes will be applied to RGB/RGBA/BGRA as well.).

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	WORD	Variable to set additional flags for image/color conversion (default is 0)
min_limit	WORD	Minimum scaling value (will be ignored if auto scale is enabled)
max_limit	WORD	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
color_vibrance	int	Color vibrance of the image
color_tint	int	Color tint of the image
lut_file	std::filesystem::path	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 a X-Cite® light source. Optionally one can specify either the device or the com port.

**Prototype**

```
XCite(
    XCiteType type = XCiteType::Any,
    const std::string &com_port = {}
);
```

**Parameter**

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

### 2.5.2 Destructor

**Description** Close the activated connection and release blocked resources.

**Prototype**

```
~XCite();
```

### 2.5.3 xcite

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

**Prototype**

```
HANDLE xcite() const;
```

**Return value**

Name	Type	Description
xcite_	HANDLE	HANDLE for the current xcite connection

### 2.5.4 getComPort

**Description** Return the Com Port of the current connection.

**Prototype**

```
std::string getComPort() const;
```

**Return value**

Name	Type	Description
sz_com_port	std::string	Com Port

### 2.5.5 getType

**Description** Return the X-Cite® device type.

**Prototype**

```
XCiteType getType() const;
```

**Return value**

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

### 2.5.6 getDescription

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

**Prototype**

```
XCITE_Description getDescription() const;
```

**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**

```
XCITE_Configuration getConfiguration() const;
```

**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**

```
void setConfiguration(
    const XCITE_Configuration& config
);
```

**Parameter**

Name	Type	Description
config	const 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**

```
void defaultConfiguration();
```

### 2.5.10 SwitchOn

**Description** Switch the configured lights on.

**Prototype**

```
void switchOn();
```

### 2.5.11 SwitchOff

**Description** Switch all lights off.

**Prototype**

```
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**

```
void executeCommand(
    const std::string &cmd,
    std::string &to,
    const std::string &in_value = {}
);
```

**Parameter**

Name	Type	Description
&cmd	const std::string	Command to be sent to the X-Cite® device
&to	std::string	Response string buffer
&in_value	const std::string	Parameters if necessary for the command

## 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
<code>intensities</code>	<code>std::vector&lt;DWORD&gt;</code>	Vector of available intensities
<code>on_states</code>	<code>std::vector&lt;BYTE&gt;</code>	Vector of which lights are on

### 2.5.13.2 XCITE\_Description

**Description** Structure holding the X-Cite® description information.

Name	Type	Description
<code>serial</code>	<code>DWORD</code>	Serial number
<code>type</code>	<code>XCiteType</code>	XCite type (see 2.5.13.3)
<code>name</code>	<code>std::string</code>	Name of the X-Cite® device
<code>wavelengths_vec</code>	<code>std::vector&lt;DWORD&gt;</code>	Vector of available wavelengths
<code>exclusivity_vec</code>	<code>std::vector&lt;DWORD&gt;</code>	Value indicate which wavelengths can be set exclusively (matching wheel number). Wheel number 0: independant activation possible
<code>intensity_max_vec</code>	<code>std::vector&lt;DWORD&gt;</code>	Vector of available maximum intensities
<code>intensity_min_vec</code>	<code>std::vector&lt;DWORD&gt;</code>	Vector of available minimum intensities

### 2.5.13.3 XCiteType

**Description** Enumeration of all XCiteTypes.

```
enum XCiteType
{
    XC_120PC,
    XC_exacte,
    XC_120LED, // USB 04D8 F615
    XC_110LED,
    XC_mini,
    XC_XYLIS,
    XC_XR210,
    XC_XLED1,
    XC_XT600, // USB 04D8 F53D
    XC_XT900,
    Any = 0xFFFF,
};
```

#### 2.5.13.4 XCite\_TypeName

**Description** Structure to map the XCiteType (see 2.5.13.3) to a string name in the array xcite\_names.

Name	Type	Description
type	XCiteType	Type
szName[40]	char[]	Type as a string

```
const XCite_TypeName xcite_names[] =
{
    {XCiteType::XC_120PC, "120PC"},
    {XCiteType::XC_exacte, "exacte"},
    {XCiteType::XC_120LED, "120LED"},
    {XCiteType::XC_110LED, "110LED"},
    {XCiteType::XC_mini, "mini"},
    {XCiteType::XC_XYLIS, "XYLIS"},
    {XCiteType::XC_XR210, "XR210"},
    {XCiteType::XC_XLED1, "XLED1"},
    {XCiteType::XC_XT600, "XT600"},
    {XCiteType::XC_XT900, "XT900"},
    {XCiteType::Any, "<Undefined Type>"}
};
```

#### 2.5.13.5 XCiteException

The `etc::XciteException` class is derived from `std::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 `std::exception`.

The following list provides a short overview of the class functions:

- **Constructor()** get exception message according to the transferred error code.
- **what()** get the error text as char pointer.
- **error\_code()** get the error code.
- **error\_text()** get the error text.

## 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.



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

