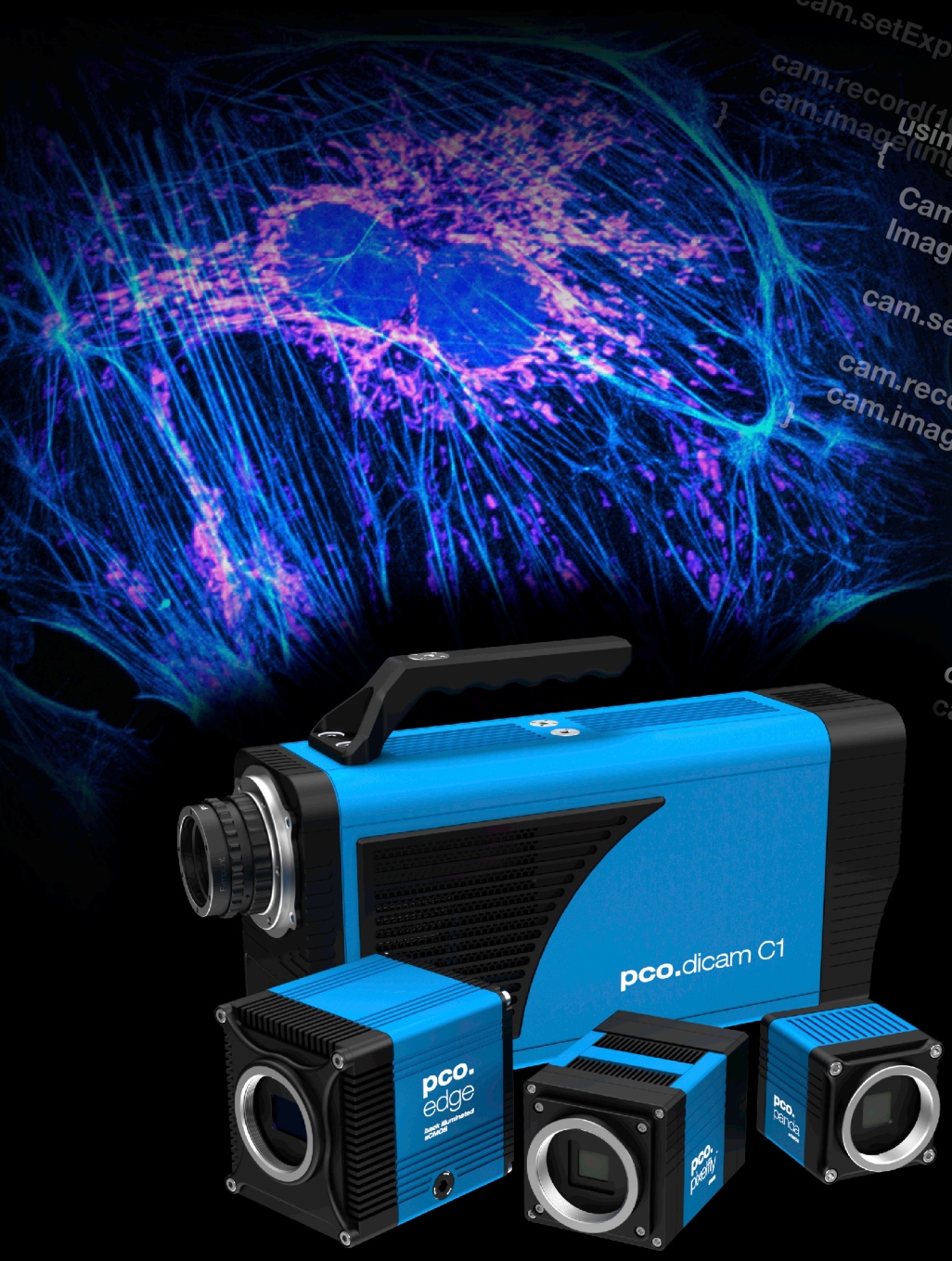
user manual

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);
}
```

EXCELITAS TECHNOLOGIES®

pco.
An Excelitas Technologies Brand

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

pco.
An Excelitas Technologies Brand

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

pco.csharp user manual 1.0.1

Released February 2024

Contents

1 General

The **pco.csharp** 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 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 Windows installer, unzip it and execute it. Simply 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*** insided 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 like it is 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
};
```

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 2.1.22, 2.1.23, 2.1.24) 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 2.1.19) or `loadlut` (see 2.1.20) 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.

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
- **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)

2.1.1 Constructor

Description Initialize the camera.

Prototype

```
Camera(CameraInterface cam_interface = CameraInterface.Any);
```

Parameter

Datatype	Name	Description
CameraInterface	cam_interface	Specific interface to search for cameras. If undefined, search on all interfaces.

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

Datatype	Name	Description
bool	recording	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

Datatype	Name	Description
bool	colored	Flag if camera is colored

2.1.5 getDescription

Description Return the description parameters of the camera.

Prototype

```
Description getDescription();
```

Return value

Datatype	Name	Description
Description	description	Structure containing the camera description (see 2.4.4)

2.1.6 defaultConfiguration

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

Prototype

```
void defaultConfiguration();
```

2.1.7 getConfiguration

Description Get the current camera configuration.

Prototype

```
Configuration getConfiguration();
```

Return value

Datatype	Name	Description
`Configuration`	`configuration`	Structure containing the current configuration of the camera (see 2.4.3)

2.1.8 setConfiguration

Description Set a configuration to the camera.

Prototype

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

Parameter

Datatype	Name	Description
`Configuration`	`config`	Configuration that should be set (see 2.4.3).

2.1.9 getExposureTime

Description Get the current exposure time of the camera.

Prototype

```
double getExposureTime();
```

Return value

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

2.1.10 setExposureTime

Description Set a new exposure time to the camera.

Prototype

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

Parameter

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

2.1.11 getDelayTime

Description Get the current delay time of the camera.

Prototype

```
double getDelayTime();
```

Return value

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

2.1.12 setDelayTime

Description Set a new delay time to the camera.

Prototype

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

Parameter

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

2.1.13 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
);
```

Parameter

Datatype	Name	Description
`int`	`num_images`	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.
`RecordMode`	`record_mode`	Defines the recording mode for this record (see 1.3).
`string`	`file_path`	Path where the image file(s) should be stored (only for modes who directly save to file, see 1.3).

2.1.14 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();
```

2.1.15 waitForFirstImage

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

Prototype

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

Parameter

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

2.1.16 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

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

2.1.17 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

Datatype	Name	Description
UInt32	recorded_image_count	Number of currently recorded images

2.1.18 getConvertControl

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

Prototype

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

Parameter

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

Return value

Datatype	Name	Description
ConvertControl	convert_control	Structure containing the current convert settings for the specified data format(see 2.4.5)

2.1.19 setConvertControl

Description Set convert control settings for the specified data format.

Prototype

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

Parameter

Datatype	Name	Description
DataFormat	data_format	Data format for which the convert settings should be set.
ConvertControl	convert_control	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);
}
```

2.1.20 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: 2.1.19).

Prototype

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

Parameter

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

2.1.21 adaptWhiteBalance

Description Do a white-balance using a transferred image.

Prototype

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

Parameter

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

2.1.22 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 roi = default,
  DataFormat data_format = DataFormat.Mono16,
  PCO_Recorder_CompressionParams comp_params = default
);
```

Parameter

Datatype	Name	Description
`Image`	`image`	`Image` object for storing the image
`uint`	`image_index`	Index of the image that should be queried, use `PCO_RECORDER_LATEST_IMAGE` for latest image (for recorder modes `fifo`/`fifo_dpcore` always use 0 (see 1.3))
`Roi`	`roi`	Soft ROI to be applied, i.e. get only the ROI portion of the image (see 2.4.2 for the `Roi` structure)
`DataFormat`	`data_format`	Data format the image should have (see 1.4)
`PCO_Recorder_CompressionParams`	`comp_params`	Compression parameters, not implemented yet

2.1.23 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 roi = default,
  uint start_index = 0,
  DataFormat data_format = DataFormat.Mono16,
  PCO_Recorder_CompressionParams comp_params = default
);
```

Parameter

Datatype	Name	Description
`List<Image>`	`images`	A List of `Image` objects for storing the images
`Roi`	`roi`	Soft ROI to be applied, i.e. get only the ROI portion of the images (see 2.4.2 for the `Roi` structure)
`uint`	`start_index`	Index of the first image that should be queried (the number of images is defined by the length of the image vector)
`DataFormat`	`data_format`	Data format the images should have (see 1.4)
`PCO_Recorder_CompressionParams`	`comp_params`	Compression parameters, not implemented yet

2.1.24 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 roi = default,
   DataFormat data_format = DataFormat.Mono16,
);
```

Parameter

Datatype	Name	Description
`Image`	`image`	`Image` object for storing the averaged image
`Roi`	`roi`	Soft ROI to be applied, i.e. get only the ROI portion of the image (see 2.4.2 for the `Roi` structure).
`DataFormat`	`data_format`	Data format the averaged image should have (see 1.4)

2.1.25 hasRam

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

Prototype

```
bool hasRam();
```

Return value

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

2.1.26 switchToCamRam

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

Prototype

```
void switchToCamRam();

void switchToCamRam(ushort segment);
```

Parameter

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

2.1.27 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

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

2.1.28 getCamRamSegment

Description Get segment number of active camram segment.

Prototype

```
ushort getCamRamSegment();
```

Return value

Datatype	Name	Description
ushort	segment_num	Number of active camram segment

2.1.29 getCamRamMaxImages

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

Prototype

```
uint getCamRamMaxImages();
```

Return value

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

2.1.30 getCamRamNumImages

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

Prototype

```
uint getCamRamNumImages();
```

Return value

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

2.1.31 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

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

Return value

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

2.1.32 Accessors

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

2.1.32.1 cameraName

Description Get the name of the camera

Prototype

```
string cameraName;
```

Return value

Datatype	Name	Description
string	name	Camera name

2.1.32.2 cameraSerial

Description Get the serial number of the camera.

Prototype

```
uint cameraSerial;
```

Return value

Datatype	Name	Description
uint	serial_number	Camera serial number

2.1.32.3 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

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

2.1.32.4 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

Datatype	Name	Description
IntPtr	rec	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
- **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)
- **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)
- **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_16bit()** Get raw image data as `ushort[]` array of 16 Bit values

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 Binning

Description Structure holding the binning information.

Datatype	Name	Description
`UInt16`	`vert`	Vertical binning
`UInt16`	`horz`	Horizontal binning

2.4.2 Roi

Description Structure holding the ROI information

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

Additionally the following convenience function are available.

Datatype	Name	Description
`UInt64`	`width()`	Get width of the ROI
`UInt64`	`height()`	Get height of the ROI
`UInt64`	`size()`	Get overall size in pixel
`UInt64`	`evenPaddedWidth()`	Get padded width
`UInt64`	`paddedSize()`	Get padded overall size

2.4.3 Configuration

Description Structure holding a camera configuration.

Datatype	Name	Description
`double`	`exposure_time_s`	Exposure time [s]
`double`	`delay_time_s`	Delay time [s]
`Roi`	`roi`	Hardware ROI structure (see 2.4.2)
`UInt16`	`timestamp_mode`	Timestamp mode
`UInt32`	`pixelrate`	Pixelrate
`UInt16`	`trigger_mode`	Trigger mode
`UInt16`	`acquire_mode`	Acquire mode

Continued on next page

Continued from previous page

Datatype	Name	Description
UInt16	metadata_mode	Metadata mode
UInt16	noise_filter_mode	Noise filter mode
Binning	binning	Binning structure (see 2.4.1)

2.4.4 Description

Description Structure holding the camera description information.

Datatype	Name	Description
UInt32	serial	Serial number of the camera
UInt16	type	Sensor type
UInt16	sub_type	Sensor sub type
UInt16	interface_type	Interface type
double	min_exposure_time_s	Minimal possible exposure time
double	max_exposure_time_s	Maximal possible exposure time
double	min_exposure_step_s	Minimal possible exposure step
double	min_delay_time_s	Minimal possible delay time
double	max_delay_time_s	Maximal possible delay time
double	min_delay_step_s	Minimal possible delay step
UInt64	min_width	Minimal possible image width (hardware ROI)
UInt64	min_height	Minimal possible image height (hardware ROI)
UInt64	max_width	Maximal possible image width (hardware ROI)
UInt64	max_height	Maximal possible image height (hardware ROI)
UInt64	roi_step_horz	Horizontal ROI stepping (hardware ROI)
UInt64	roi_step_vert	Vertical ROI stepping (hardware ROI)
bool	roi_symmetric_horz	Flag if hardware ROI has to be horizontally symmetric (i.e. if x0 is increased, x1 has to be decreased by the same value)
bool	roi_symmetric_vert	Flag if hardware ROI has to be vertically symmetric (i.e. if y0 is increased, y1 has to be decreased by the same value)
UInt16	bit_resolution	Bit-resolution of the sensor
bool	has_timestamp_mode	Flag if camera supports the timestamp setting
bool	has_timestamp_mode_ascii_only	Flag if camera supports setting the timestamp to ascii-only

Continued on next page

Continued from previous page

Datatype	Name	Description
List<UInt32>	pixelrate_vec	Vector containing all possible pixelrate frequencies (index 0 is default)
bool	has_acquire_mode	Flag if camera supports the acquire mode setting
bool	has_ext_acquire_mode	Flag if camera supports the external acquire setting
bool	has_metadata_mode	Flag if metadata can be activated for the camera
bool	has_ram	Flag if camera has internal memory
List<UInt16>	binning_horz_vec	Vector containing all possible horizontal binning values
List<UInt16>	binning_vert_vec	Vector containing all possible vertical binning values

2.4.5 ConvertControl

Description Structure containing (color) convert information.

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

Mono8 format **ConvertControlMono**

Datatype	Name	Description
bool	sharpen	Flag if the image should be sharpened
bool	adaptive_sharpen	Flag if adaptive sharpening should be enabled
bool	flip_vertical	Flag if the image should be vertically flipped
bool	auto_minmax	Flag if auto scale should be enabled
int	min_limit	Minimum scaling value (will be ignored if auto scale is enabled)
int	max_limit	Maximum scaling value (will be ignored if auto scale is enabled)
double	gamma	Gamma of the image (default is 1.0)
int	contrast	Contrast of the image (default is 0)

Color camera and color format **ConvertControlColor**

Datatype	Name	Description
bool	sharpen	Flag if the image should be sharpened
bool	adaptive_sharpen	Flag if adaptive sharpening should be enabled
bool	flip_vertical	Flag if the image should be vertically flipped
bool	auto_minmax	Flag if auto scale should be enabled
int	min_limit	Minimum scaling value (will be ignored if auto scale is enabled)

Continued on next page

Continued from previous page

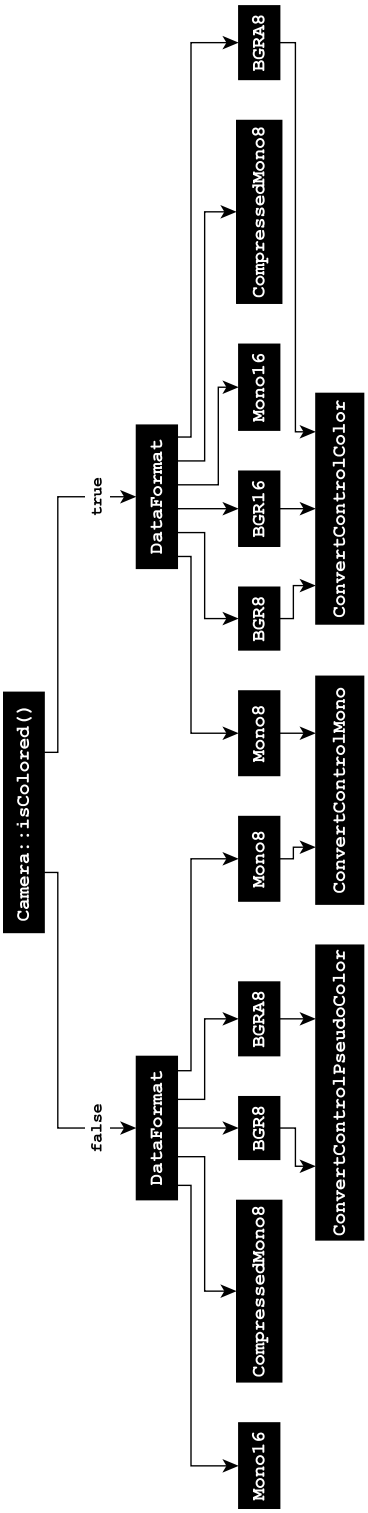
Datatype	Name	Description
int	max_limit	Maximum scaling value (will be ignored if auto scale is enabled)
double	gamma	Gamma of the image (default is 1.0)
int	contrast	Contrast of the image (default is 0)
bool	pco_debayer_algorithm	Flag if PCO debayering should be used
int	color_temperature	Color temperature of the image
int	color_saturation	Color saturation of the image
int	color_vibrance	Color vibrance of the image
int	color_tint	Color tint of the image

BW camera and color format

ConvertControlPseudoColor

Datatype	Name	Description
bool	sharpen	Flag if the image should be sharpened
bool	adaptive_sharpen	Flag if adaptive sharpening should be enabled
bool	flip_vertical	Flag if the image should be vertically flipped
bool	auto_minmax	Flag if auto scale should be enabled
int	min_limit	Minimum scaling value (will be ignored if auto scale is enabled)
int	max_limit	Maximum scaling value (will be ignored if auto scale is enabled)
double	gamma	Gamma of the image (default is 1.0)
int	contrast	Contrast of the image (default is 0)
int	color_temperature	Color temperature of the image
int	color_saturation	Color saturation of the image
int	color_vibrance	Color vibrance of the image
int	color_tint	Color tint of the image
string	lut_file	Path of the lut file that should be used

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



3 About Excelitas PCO

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

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

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

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

pco.

An Excelitas Technologies Brand

pco.

An Excelitas Technologies Brand

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

EXCELITAS TECHNOLOGIES®