user manual

pco.software development kit

pco.

In case of any questions or comments, please contact us at PCO.

telephone	+49 (0) 9441 2005 50
fax	+49 (0) 9441 2005 20
email	info@pco.de
postal address	PCO AG Donaupark 11 93309 Kelheim, Germany

Released May 2017 © PCO AG

pco.sdk User Manual V1.21 © PCO AG, Germany

TABLE OF CONTENTS

1. GENERAL

This document describes the pco.software development kit. The application interface can be used for all cameras listed in the API function matrix (see chapter 6).

The pco.sdk is a collection of libraries and sample projects for Windows operating systems. All libraries are designed as ***dynamic link C libraries (DLL)*** which allow easy development of your own applications to manage one or more PCO cameras connected to a computer. Using a library with C calling convention the functionality of the DLL is also available, when writing managed C# and Visual Basic applications and can extend the capability of scripting languages e.g. Phyton and Matlab. Also PCO's own application ***Camware*** is based on the SDK.

The first chapter provides a short introduction on how to work with the SDK. An overview of all available functions, described in detail, of the pco camera application programming interface ***(pco camera API)*** can be found in the ***reference section*** (see chapter **2**). Example source code can be found in the ***examples section*** (see chapter **3**) or in the installation directory of the SDK.

Definition:

SDK	Software Development Kit	SDK is a collection of librariers, sample projects and applications to develop software
API	Application Programming Interface	API is an interface for application programming. It is a set of clearly defined methods of communication between various software components.

1.1 OVERVIEW

The ***API*** base functionality is to configure and control the camera settings and to transfer the acquired images from the camera to the PC. These functions are available through function calls inside the SC2_Cam.dll. The SC2_Cam.dll has the capability to control any PCO camera regardless of the camera type and hardware interface.

In principle the ***API*** can be divided into ***two parts***:

Control the camera settings	The camera settings define how images are acquired in the camera. (exposure time, ROI, trigger,...). All settings will be finalized with an arm command. If the arm was successful, the camera can be started and will then acquire images depending on the accomplished settings.
Start an image transfer from the camera to the PC	Image transfers can be invoked at any camera state. To successfully fulfill a transfer images must have been acquired from the camera or will be acquired within a predefined timeout. If a transfer cannot be completed an error status will be returned. When camera internal memory is available, the data transfer could also be at a later time.

Most part of the ***API*** is derived from the pco low level commands (pco telegram structures).

1.2 CONVENTIONS

The following typographic conventions are used in this manual:

Bold e.g. PCO_GetCameraType	Functions, procedures or modes used in this manual, with a cross-reference to the appropriate page
[words in brackets]: [run]	Possible values or states of the described functions
ALL CAPITAL WORDS: TRUE	Logical or boolean values such as TRUE, FALSE, ON, OFF, RISING, FALLING, HIGH, LOW
<words in arrows>: <acq enbl>	Names of hardware input / output signals
`Font Courier New: strGeneral.wSize = sizeof(strGeneral)`	C Example Code
bold italics	Important terms

1.3 BUILDING APPLICATIONS

First step to successfully operate a PCO camera is to establish a valid connection. This is the task of the **PCO_OpenCamera** call. This function scans through all available interface DLLs and is checking, if a camera can be found. On success, all internals are initialized and a ***unique handle*** will be assigned to this camera. This handle has to be used in all subsequent function calls. For multi camera operation **PCO_OpenCamera** can be called several times.

As next step camera description and status should be queried by calling **PCO_GetCameraDescription** and **PCO_GetCameraHealthStatus**.
Due to the wide variety of the PCO cameras, which can be controlled by the library, the camera description should be used to check the availability of enhanced features and the limitations of the connected camera.
If the camera is already recording, images can be transferred.
If the camera is not recording or after a stop command, the camera settings can be changed.

After any change of a camera parameter, a **PCO_ArmCamera** command ***must*** be sent. When this command is received from the camera all previous parameter settings are validated and on success the camera is prepared to start recording with the new parameter set. The ***only exception*** to this procedure is that ***exposure and delay time*** settings can be changed without following **PCO_ArmCamera** command in case the camera is recording

The current recording state can be queried with **PCO_GetRecordingState** and must be changed with **PCO_SetRecordingState**.
Setting the camera in ***Recording State*** in general does start image exposing and readout of the image sensor. Timing and expose start can be controlled through different operating mode settings of the camera.

Two different types of cameras can be found in the PCO camera family. The ones, which have ***internal memory*** like pco.dimax and the others without internal memory ***running in streaming mode*** like the pco.edge.

While recording, single images can be grabbed from both types with the image transfer function **PCO_GetImageEx**. The image transfer function **PCO_AddBufferEx**, which does setup an internal image request queue, should be used for fast readout of multiple images. When grabbing of multiple images with **PCO_AddBufferEx** is finished, the command **PCO_CancelImages** must be called to reset the internal buffer queue.
After recording is stopped, both image transfer functions can also be used to readout image data from the internal memory (if available).

The ***API*** is not thread save. This means that it is not possible to set up two or more threads getting images with different settings and sizes. However threading is possible in case the developer takes care for correct thread synchronization, e.g. one thread changes the settings and a second one grabs the images. In this case the second thread has to stop grabbing till the first one has changed the settings and has executed a **PCO_ArmCamera** command.

In principle the order of commands shown in the **TYPICAL IMPLEMENTATION** should be met.

1.4 RUNNING APPLICATIONS

To allow access to the ***API***, the ***SC2_Cam.dll*** must reside in the application directory or in the library search path when implicit linkage is used. The user can also link explicitely. In this case the SC2_Cam.dll can be placed in the application folder or search path. The dll can also be placed in a known folder, but you'll have to call LoadLibrary with the complete path then. To support all available hardware interfaces of PCO cameras the ***SC2_Cam.dll*** depends on additional interface DLLs. These are either installed during PCO driver installation or must be installed / copied to the application directory. Because all these DLLs are available in a 32 Bit and a 64 Bit version pay attention to copy the correct bitness for the used application.

1.5 COMPILING AND LINKING

To use the ***API*** Library in an application, the ***SC2_CamExport.h*** and the ***SC2_SDKStructures.h*** file must be added in addition to the standard header files. It is recommended to add also the header files with the PCO error codes ***pco_err.h*** and the error description ***pco_errtxt.h***.

For better control of the interface layer the ***SC2_SDKAddendum.h*** file is necessary.
Useful definitions for parameter settings can be found in the ***SC2_defs.h***.

The application program must be linked with the appropriate library (32Bit or a 64Bit) which can be found in the ***lib or lib64 folders***. The ***API*** can be invoked either by linking to the ***SC2_Cam.lib*** through project settings or by loading the required functions from the ***SC2_Cam.dll*** explicitly at runtime with the ***LoadLibrary function*** from the Windows-API.

A lot of functions use structures as input and output. To enhance security of the ***API*** interface, each structure includes a `wSize` parameter, which must be filled carefully (typical value is `sizeof`(***API*** structure)). For nested structures the `wSize` parameter of all structures has to be set.

Typical implementation for setting `wSize` parameter of embedded structures:

```
strGeneral.wSize = sizeof(strGeneral);
strGeneral.strCamType.wSize = sizeof(strGeneral.strCamType);
strCamType.wSize = sizeof(strCamType);
```

1.6 SDK FOLDER OVERVIEW

During installation the following files are copied to the target-directory.

\include

SC2_CamExport.h
API function declarations

SC2_defs.h
Useful camera definitions

SC2_SDKStructures.h
Structures which are used from different ***API*** functions.
Provide information about camera settings and ***API*** status.
The structures can be used to control camera settings.
To enhance security of the ***API*** interface, each structure includes a wSize parameter, which must be filled carefully (typical value is sizeof(***API*** structure)). For nested structures the wSize parameter of all structures has to be set.

SC2_SDKAddendum.h
Interface specific structures and defines

SC2_err.h, SC2_errt.h
Definition of return values and detailed error description

\lib
\lib64

SC2_Cam.lib
API functions library, which can be linked to the application.

\bin
\bin64

SC2_Cam.dll
API executable dynamic link library

SC2_cl_me4.dll
Interface DLL to Silicon Software ME4 Camera Link framegrabber family

SC2_cl_mtx.dll
Interface DLL to Matrox Camera Link Solios framegrabber family

SC2_cl_nat.dll
Interface DLL to National Instruments Camera Link frame grabber family

SC2_clhs.dll
Interface DLL to CLHS framegrabber

SC2_cl_ser.dll
Interface DLL to any Camera Link framegrabber compatible to the Camera Link specification. clserxxx interface DLL must be available from the grabber manufactorer. With the sc2_cl_ser DLL all Camera Control commands can be used, but not the functions of the chapter BUFFER MANAGEMENT, IMAGE ACQUISITION and DRIVER MANAGEMENT.

For the above Camera Link interface DLL's the Runtime/driver environment of the framegrabber manufacturer must be installed and working properly.

Additional interfaces are available through the following DLL's: ***sc2_1394.dll, sc2_usb.dll, sc2_usb3.dll, sc2_gige.dll, sc2_gige1.dll, sc2_gige2.dll***. These interface DLL's are installed to the system directory during pco.driver installation.

1.7 SDK LOGGING

All low level commands sent from the ***SC2_Cam.dll*** to the camera can be reported to a file. To enable logging, a file called '***SC2_Cam.log***' must be created in the following directory:

>systemdisc<:\ProgramData\pco\
(On Windows 7 / 8 / 10)

SC2_Cam.log will be overwritten with each session start. In case the user likes to keep older sessions, rename the logfile to ***SC2_Cam_a.log***. This will append further sessions. After ending your logging session please do not forget to delete the ***SC2_Cam(_a).log*** file, because it may cut down performance.

To get enhanced reports the logging can also be enabled on any interface DLL. To enable logging, a file called with the interface DLL name with extension '.log' must be created in the above mentioned directory. Several Loglevels can be selected. This is done through 'LOGGING=' parameter in the appropriate ****_param.ini*** file. For all Camera Link interfaces this file is named ***sc2_cl_param.ini*** the other interface DLLs follow the above naming convention.
E.g. logging of:
sc2_cl_me4.dll is written to ***sc2_cl_me4.log*** and controlled through ***sc2_cl_param.ini***
sc2_usb.dll is written to ***sc2_usb.log*** and controlled through ***sc2_usb_param.ini.***

1.8 PROTOTYPE EXAMPLE

The section shows the general representation format of the ***API*** functions in this manual.

Description:

This paragraph gives a brief summary of the function and its properties. The main behavior is described as well as the required usage and additional restrictions. If the function needs special requirements or has certain connections to other functions, this is also mentioned here.

Supported camera type:

If the described function is not available for all cameras, the supported camera types are listed here. Otherwise just ***All cameras*** is written.

Descriptor dependency:

Some functions are only available if special flags in the **PCO_Description structure** are set or cleared. If the current function has such dependencies, the names of the associated flags are listed here (***None*** otherwise). The **PCO_Description structure** can be read out from the camera by calling **PCO_GetCameraDescriptionEx**.

Prototype:

This paragraph shows the function prototype (see example below). A short comment behind each argument shows if it is input, output or in- and output.

```
SC2_SDK_FUNC int WINAPI PCO_Example (
    HANDLE* ph,                     //in
    WORD* inOut                     //in,out
    DWORD* dataOut                  //out
    );
```

Parameter:

All arguments of the function are listed in a table with their type and a short description. For the example function above, this table looks like the following:

Name	Type	Description
ph	HANDLE*	Handle to a previously opened camera device
inOut	WORD*	Pointer to a WORD used as in and output parameter
dataOut	DWORD*	Pointer to a DWORD holding some output data

Parameter dependency:

If the arguments of the functions have dependencies, which e.g. limit the allowed range or determine other constraints (such as symmertrical ROI, constant value stepping…) these flags or conditions are shown here (***None*** if there are no dependencies).

Return value:

The meaning of the return value is described. Since all functions have error codes as return value, the paragraph always looks like this:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2. API FUNCTION SECTIONS

2.1 CAMERA ACCESS

This chapter describes all functions that make it possible to access connected cameras.

2.1.1 PCO_OpenCamera

Description:

This function is used to get a connection to a camera. A unique handle is returned, which must be used for all other function calls. This function scans through all available interfaces and tries to connect to the next available camera. If more than one camera is connected to the computer this function must be called a second time to get the handle for the next camera. If a distinct camera should be accessed **PCO_OpenCameraEx** has to be used.
Because this function is using a scan process `wCamNum` parameter is not used.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_OpenCamera (
    HANDLE* ph,                         //in,out
    WORD wCamNum                        //not used
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE*`	Pointer to a HANDLE • On input the HANDLE must be set to NULL to open next available camera. • On output a unique HANDLE is returned, if a valid connection was established.
`wCamNum`	`WORD`	Not used

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.1.2 PCO_OpenCameraEx

Description:

This function is used to get a connection to a distinct camera, e.g. a camera which is connected to a specific interface port. A unique handle is returned, which must be used for all other function calls. To select the desired camera the structure **PCO_Openstruct structure** must be filled with appropriate parameters before the function is called. If no camera could be found at the selected interface an error is returned and the handle is set to ***NULL***.

As a special case this function can be used to establish a valid connection to a camera through the serial interface of any Camera Link grabber to control the camera with the PCO SDK functions. The image grab and transfer functions of the pco.sdk can not be used in this mode. The name of the grabber manufacturer ***clserxxx.dll*** must be provided in the **PCO_Openstruct structure**.

If more then one camera with GigE interface are used, only those cameras can be opened, which have established a connection with a valid IP address, when the **PCO_OpenCameraEx** function is called the first time from an application.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_OpenCameraEx (
    HANDLE* ph,                     //in,out
    PCO_OpenStruct* strOpenStruct   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE*	Pointer to a HANDLE: • On input the HANDLE must be set to NULL to open a camera at the selected interface. • On output a unique HANDLE is returned, if a valid connection was established.
strOpenStruct	PCO_OpenStruct*	Pointer to a previously filled **PCO_Openstruct structure**

If `wInterfacType` and `wCameraNum` are used for application site enumeration the application should scan until error `PCO_ERROR_DRIVER_NODRIVER (0x80002006)` occurs. For Camera Link devices some cameras need two ports (pco.edge). In this case `wCameraNum` has to be incremented a second time in order to get the next camera.

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.1.2.1 PCO_Openstruct structure

Name	Type	Description
wSize	WORD	Size of this structure
wInterfaceType	WORD	Interface type number defined in sc2_SDKStructures.h. With the interface type the according interface DLL is selected: • 1=FireWire • 2=Camera Link Matrox • 3=Camera Link Silicon Software mE III • 4=Camera Link National Instruments • 5=GigE • 6=USB 2.0 • 7=Camera Link Silicon Software mE IV • 8=USB 3.0 • 9=Reserved for WLAN • 10=Camera Link serial port only • 11=CLHS (Camera Link HS) • 0xFFFF: The SDK-DLL tries to find a camera at all known interfaces, starting with FireWire (1)
wCameraNumber	WORD	Desired camera number at the selected interface
wCameraNumAtInterface	WORD	Resulting current number of camera at the interface. Must be set to zero for successive open calls for the selected interface.
wOpenFlags[10]	WORD	List of WORDS with additional flags to control the interface DLL. • wOpenFlags[0]: CameraLink Bitfield: 0x1000: to open only serial connection 0x2000: a clserxxx.dll filename is provided • wOpenFlags[1]: • wOpenFlags[2]: 0x0001: must be set when the generic interface is of type Camera Link. (PCO_OPENFLAG_GENERIC_IS_CAMLINK)
dwOpenFlags[5]	DWORD	List of DWORDS moved on to interface dll • dwOpenFlags[0]: GigE:IP address
wOpenPtr[6]	void*	List of pointers which hold additional information • wOpenPtr[0]: moved on to interface dll. Camera Link: pointer to a character array, which holds the filename of the clserxxx.dll as ASCII string. • wOpenPtr[1]: Camera Link:reserved for configuration filename • wOpenPtr[2]: reserved • wOpenPtr[3]: reserved • wOpenPtr[4]: reserved • wOpenPtr[5]: filename of generic interface DLL as ASCII string
zzwDummy[8]	WORD	Reserved

2.1.3 PCO_CloseCamera

Description:

This function is used to close the connection to a previously opened camera.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_CloseCamera (
    HANDLE ph                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.1.4 PCO_ResetLib

Description:

This function is used to set the ***SC2_cam Library*** to an initial state. All camera handles have to be closed with **PCO_CloseCamera** before this function is called.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_ResetLib();
```

Parameter:

No parameter

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.1.5 PCO_CheckDeviceAvailability

Description:

This function is used to check, if the connection to a previously opened camera is still valid. Functionality is only supported for interfaces with ***HotPlug*** capability like USB, GigE or FireWire.
If a device is connected or disconnected from a ***HotPlug*** capable bus system, the device manager does invoke a bus reset call on the bus and afterwards is starting a new enumeration. If enumeration is finished, a ***DEVICE_CHANGE*** message is broadcasted to all applications.

Supported camera type:

Interface dependent

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_CheckDeviceAvailability (
    HANDLE ph,                          //in
    WORD wNumIf                         //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wNumIf	WORD	Number of camera which should be checked for availability at a distinct interface. The interface type to check is derived from the passed in handle

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.2 CAMERA DESCRIPTION

Because different sensors (CCD, CMOS, sCMOS) are used in the different camera models, each camera has its own description. This description should be readout shortly after access to the camera is established. In the description general margins for all sensor related settings and bitfields for available options of the camera are given. This set of information can be used to validate the input parameter for commands, which change camera settings, before they are sent to the camera. The `dwGeneralCapsDESC1` and `dwGeneralCapsDESC2` bitfields in the **PCO_Description structure** can be used to see what options are supported from the connected camera. Supported options may vary with different camera types and also between different firmware versions of one camera type.

2.2.1 PCO_GetCameraDescription

Description:

Sensor and camera specific description is queried. In the returned **PCO_Description structure** margins for all sensor related settings and bitfields for available options of the camera are given.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraDescription(
    HANDLE ph,                          //in
    PCO_Description* strDescription   //in,out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`strDescription`	`PCO_Description*`	Pointer to a **PCO_Description structure**: • On input the `wSize` parameter of this structure must be filled with the correct structure size in bytes • On output the structure is filled with the requested information from the camera

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.2.2 PCO_GetCameraDescriptionEx

Description:

Any of the available sensor and camera specific description can be queried. With input parameter `wType` the returned description structure can be selected. ***PCO_DescriptionEx*** is a generic structure which has to be casted to/from the queried structure. The `wSize` parameter has to be filled with the correct value for the requested structure. This function was introduced due to the size limitation of the standard camera descriptor and the need for describing additional features.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: ENHANCED_DESCRIPTOR_2`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraDescriptionEx(
    HANDLE ph,                                  //in
    PCO_DescriptionEx*  strDescEx,              //in,out
    WORD wType                                  //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`strDescEx`	`PCO_DescriptionEx*`	Pointer to any **PCO_Description structure** typecasted to `PCO_DescriptionEx*`: • On input the `wSize` parameter of this structure must be filled with the correct structure size in Bytes. • On output the structure is filled with the requested information from the camera.
`wType`	`WORD`	`WORD` variable to select the returned descriptor: • `0x0000` = `PCO_Description` • `0x0001` = `PCO_Description2`

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Example:

Get standard description **PCO_Description structure** and, if available, additional description **PCO_Description2 structure** by using the command **PCO_GetCameraDescriptionEx.**

```
PCO_Description StandardDesc;
PCO_Description2 ModulateDesc;

WORD wType = 0; // Set to zero initially

PCO_OpenCamera(&ph,0);

//initialize structures
memset(&StandardDesc,0,sizeof(PCO_Description));
memset(&ModulateDesc,0,sizeof(PCO_Description2));

StandardDesc.wSize=sizeof(PCO_Description);

PCO_GetCameraDescriptionEx(ph,(PCO_DescriptionEx*)&StandardDesc,wType);
if(StandardDesc.dwGeneralCaps1& GENERALCAPS1_ENHANCED_DESCRIPTOR_2)
{
 wType = 1;
 ModulateDesc.wSize= sizeof(PCO_Description2);
 PCO_GetCameraDescriptionEx(ph,(PCO_DescriptionEx*)&ModulateDesc,wType);
}

…
```

2.2.2.1 PCO_Description structure

Name	Type	Description
wSize	WORD	Size of this structure
wSensorTypeDESC	WORD	Image sensor type, see table **Sensor Type Codes**
wSensorSubTypeDESC	WORD	Image sensor subtype
wMaxHorzResStdDESC	WORD	Maximal horizontal resolution in pixels for standard format
wMaxVertResStdDESC	WORD	Maximal vertical resolution in pixels for standard format
wMaxHorzResExtDESC	WORD	Maximal horizontal resolution in pixels for extended format
wMaxVertResExtDESC	WORD	Maximal vertical resolution in pixels for extended format
wDynResDESC	WORD	Dynamic resolution in bits/pixel
wMaxBinHorzDESC	WORD	Maximal horizontal binning value
wBinHorzSteppingDESC	WORD	Stepping of horizontal binning • 0 = binary step (1, 2, 4, 8, 16…max.) • 1 = linear step (1, 2, 3, 4, 5 … max.)
wMaxBinVertDESC	WORD	Maximal vertical binning value
wBinVertSteppingDESC	WORD	Stepping of vertical binning • 0 = binary step (1, 2, 4, 8, 16…max.) • 1 = linear step (1, 2, 3, 4, 5 … max.)
wRoiHorStepsDESC	WORD	Stepping of horizontal ROI in pixel (camera ROI constraint) • 0 = no ROI setting possible • Others = camera ROI setting must always be a multiple of this value (e.g. value=10 -> wRoiX0=1, 11, 21, 31…)
wRoiVertStepsDESC	WORD	Stepping of vertical ROI in pixel (camera ROI constraint) • 0 = no ROI setting possible • Others = camera ROI setting must always be a multiple of this value (e.g. value=2 -> wRoiY0=1, 3, 5…)
wNumADCsDESC	WORD	Number of ADCs inside camera
wMinSizeHorzDESC	WORD	Minimum size in pixels in horizontal direction
wMinSizeVertDESC	WORD	Minimum size in pixels in vertical direction
dwPixelRateDESC[4]	DWORD	List of available pixel rate frequencies • 0 = not valid • others = pixel rate frequency in Hz Only values of this list can be set as pixel rate.
ZzdwDummy	DWORD	Reserved
wConvFactDESC[4]	WORD	List of available conversion factors • 0 = not valid • Others = Conversion factor *100 in electrons / count e.g. 100 = 1.0 electrons / count e.g. 610 = 6.1 electrons / count Only values of this list can be set as conversion factor.
sCoolingSetpoints[10]	SHORT	List of available cooling setpoints. List is only valid when the COOLING_SETPOINTS flag in dwGeneralCapsDESC1 is set. The value of wNumCoolingSetpoints give the number of valid entries in the list. If this list is valid only values out of this list can be used as cooling setpoint.
ZZdwDummycv	WORD	Reserved

To be continued on the next page →

wSoftRoiHorStepsDESC	WORD	Stepping of horizontal ROI in pixel (Software ROI constraint) Value is only valid when Software ROI is enabled. See **PCO_EnableSoftROI**: • 0 = no ROI setting possible • Others = ROI setting must always be a multiple of this value (e.g. value=2 -> wRoiX0=1, 3, 5…)
wSoftRoiVertStepsDESC	WORD	Stepping of vertical ROI in pixel (Software-ROI constraint) Value is only valid when Software ROI is enabled. See **PCO_EnableSoftROI**: • 0 = no ROI setting possible • Others = ROI setting must always be a multiple of this value (e.g. value=2 -> wRoiX0=1, 3, 5…)
wIRDESC	WORD	Sensor option IR sensitivity. If option is available the sensor can be switched to improved IR sensitivity: • 0 = IR sensitivity not available • 1 = IR sensitivity available
dwMinDelayDESC	DWORD	Minimum delay time in ns (non IR sensitivity mode)
dwMaxDelayDESC	DWORD	Maximum delay time in ms (non IR sensitivity mode)
dwMinDelayStepDESC	DWORD	Stepping of delay time in ns (both IR sensitivity modes)
dwMinExposDESC	DWORD	Minimum exposure time in ns (non IR sensitivity mode)
dwMaxExposDESC	DWORD	Maximum exposure time in ms (non IR sensitivity mode)
dwMinExposStepDESC	DWORD	Stepping of exposure time in ns (both IR sensitivity modes)
dwMinDelayIRDESC	DWORD	Minimum delay time in ns (IR sensitivity mode)
dwMaxDelayIRDESC	DWORD	Maximum delay time in ms (IR sensitivity mode)
dwMinExposIRDESC	DWORD	Minimum exposure time in ns (IR sensitivity mode)
dwMaxExposIRDESC	DWORD	Maximum exposure time in ms (IR sensitivity mode)
wTimeTableDESC	WORD	Camera option time table. If option is available the camera can perform a timetable with several delay/ exposures time pairs: • 0 = time table not available • 1 = time table available
wDoubleImageDESC	WORD	Camera option double image mode. If option is available, the camera can perform a double image with a short interleave time between exposures: • 0 = double mode not available • 1 = double mode available
sMinCoolSetDESC	SHORT	Minimum cooling setpoint in °C. (if all setpoints are 0, then cooling is not available)
sMaxCoolSetDESC	SHORT	Maximum cooling setpoint in °C. (if all setpoints are 0, then cooling is not available)
sDefaultCoolSetDESC	SHORT	Default cooling setpoint in °C. (if all setpoints are 0, then cooling is not available)
wPowerDownModeDESC	WORD	Camera option power down mode. If option is available, the camera can switch the sensor into power down mode for reduced dark current during long exposure times: • 0 = power down not available • 1 = power down available

To be continued on the next page →

wOffsetRegulationDESC	WORD	Camera option offset regulation. If option is available, the camera can perform an automatic offset regulation using the reference pixels of the sensor: • 0 = offset regulation not available • 1 = offset regulation available
wColorPatternDESC	WORD	Description of the color pattern of the sensor; Each of the four nibbles is describing the location and color of the color sensor. (see table **Color Pattern description (2x2 matrix)**
wPatternTypeDESC	WORD	Type of color pattern: • 0x0000 = [RGB Bayer Pattern]
wDummy1	WORD	Reserved
wDummy2	WORD	Reserved
wNumCoolingSetpoints	WORD	The number of valid entries in the sCoolingSetpoints list
dwGeneralCapsDESC1	DWORD	General capability bit field describing special features and constraints of the camera (see table **GeneralCaps1-Bits**)
dwGeneralCapsDESC2	DWORD	Advanced capability bit field describing special features and constraints of the camera
dwExtSyncFrequency[4]	DWORD	Predefined values for external sync mode. Only values of this list can be used as external frequency
dwGeneralCapsDESC3	DWORD	Advanced capability bit field describing special features and constraints of the camera (see table **GeneralCaps1-Bits**)
dwGeneralCapsDESC4	DWORD	Advanced capability bit field describing special features and constraints of the camera
ZzdwDummy	DWORD	Reserved for future use

2.2.2.2 Color Pattern description (2x2 matrix)

The ***Color Pattern*** of the sensor is declared by the four nibbles (4 bit each) of the WORD wColorPatternDESC. Each nibble holds the value of the corresponding color. The ***Color Pattern*** description is necessary for determining the color of the upper left corner of the image readout from a color sensor in full resolution. With this value the correct demosaicing algorithm can be selected. If vertical and/or horizontal ROI is used and ROI settings are not a multiple of 2, the correct demosaicing algorithm must be calculated for the current ROI offsets.

Color defines for RGB Bayer Pattern:

- RED = 0x1
- GREEN (RED LINE) = 0x2
- GREEN (BLUE LINE) = 0x3
- BLUE = 0x4

1	2
3	4

e.g.

R	GR
GB	B

The four nibbles are arranged in the following way:

MSB	bottom right	bottom left	top right	top left	LSB

For the sample this would result in:
0x4321 (Nibble4: BLUE, Nibble3: GREENB, Nibble2: GREENR, Nibble1: RED)

2.2.2.3 Sensor Type Codes

Sony:

Sensor	Code	Sensor	Code	Sensor	Code
ICX285AL	0x0010	ICX274AL	0x0030	ICX414AL	0x0050
ICX285AK	0x0011	ICX274AK	0x0031	ICX414AK	0x0051
ICX263AL	0x0020	ICX407AL	0x0040	ICX407BLA	0x0060
ICX263AK	0x0021	ICX407AK	0x0041		

Kodak:

Sensor	Code	Sensor	Code	Sensor	Code
KAI2000M	0x0110	KAI4010M	0x0130	KAI4020M	0x0140
KAI2000CM	0x0111	KAI4010CM	0x0131	KAI4020CM	0x0141
KAI2001M	0x0120	KAI4011M	0x0132	KAI4021M	0x0142
KAI2001CM	0x0121	KAI4011CM	0x0133	KAI4021CM	0x0143
KAI2002M	0x0122			KAI4022M	0x0144
KAI2002CM	0x0123			KAI4022CM	0x0145
KAI11000M	0x0150	KAI11002M	0x0152		
KAI11000CM	0x0151	KAI11002CM	0x0153		

sCMOS:

Sensor	Code	Sensor	Code
CIS2051_V1_FI_BW	0x2000	CIS1042_V1_FI_BW	0x2002
CIS2051_V1_FI_COL	0x2001	CIS2051_V1_BI_BW	0x2010
GPIXEL_X2_BW	0x5000		

Others:

Sensor	Code	Sensor	Code
MV13BW	0x1010	CYPRESS_RR_V1_BW	0x3000
MV13COL	0x1011	CYPRESS_RR_V1_COL	0x3001
QMFLIM_V2B_BW	0x4000		

2.2.2.4 GeneralCaps1-Bits

Flag name	Bitmask Value	Description
NOISE_FILTER	0x00000001	Noise filter is available
HOTPIX_FILTER	0x00000002	Hot pixel filter is available
HOTPIX_ONLY_WITH_NOISE_FILTER	0x00000004	Hot pixel correction does not work without noise filter
TIMESTAMP_ASCII_ONLY	0x00000008	Time stamp without binary is available
DATAFORMAT2X12	0x00000010	Camera Link data format 2x12bit available
RECORD_STOP	0x00000020	Record stop event mode is available
HOT_PIXEL_CORRECTION	0x00000040	Hot pixel correction is available
NO_EXTEXPCTRL	0x00000080	External exposure control is not available
NO_TIMESTAMP	0x00000100	Time stamp is not available
NO_ACQUIREMODE	0x00000200	Acquire mode is not available
DATAFORMAT4X16	0x00000400	Camera Link data format 4x16Bit available
DATAFORMAT5X16	0x00000800	Camera Link data format 5x16Bit available
NO_RECORDER	0x00001000	No internal recorder is available
FAST_TIMING	0x00002000	Fast timing mode is available
METADATA	0x00004000	Meta Data is available
SETFRAMERATE_ENABLED	0x00008000	Set/GetFrameRate available
CDI_MODE	0x00010000	Correlated double image mode is available
CCM	0x00020000	Internal color correction matrix is available
EXTERNAL_SYNC	0x00040000	Trigger mode external sync is available
NO_GLOBAL_SHUTTER	0x00080000	Global shutter operation mode not available
GLOBAL_RESET_MODE	0x00100000	Global reset operation mode not available
EXT_ACQUIRE	0x00200000	Extended acquire is available
FAN_CONTROL	0x00400000	Fan control is available
ROI_VERT_SYMM_TO_HORZ_AXIS	0x00800000	Vertical ROI must be symmetrical to horizontal axis (camera ROI constraint)
ROI_HORZ_SYMM_TO_VERT_AXIS	0x01000000	Horizontal ROI must be symmetrical to vertical axis (camera ROI constraint)
COOLING_SETPOINTS	0x02000000	Table with predefined cooling setpoints is available.
HW_IO_SIGNAL_DESCRIPTOR	0x40000000	Hardware IO description is available
ENHANCED_DESCRIPTOR_2	0x80000000	Enhanced description 2 is available

All flags are also defined in header file sc2_defs.h. To get the defined names of the flags preceed the above Flag name with "GENERALCAPS1_".
E.g. flag NOISE_FILTER is defined as GENERALCAPS1_NOISE_FILTER

2.2.2.5 GeneralCaps3-Bits

Flag name	Bitmask Value	Description
HDSDI_1G5	0x00000001	HDSDI interface with1.5 Gbit datarate available
HDSDI_3G	0x00000002	HDSDI interface with 3 Gbit datarate available
IRIG_B_UNMODULATED	0x00000004	Unmodulated IRIG B can be evaluated
IRIG_B_MODULATED	0x00000008	Modulated IRIG B can be evaluated
CAMERA_SYNC	0x00000010	Camera Sync mode is available
HS_READOUT_MODE	0x00000020	Fast Sensor readout is available
EXT_SYNC_1HZ_MODE	0x00000040	In trigger mode [external synchronized] multiples of 1Hz can be evaluated

All flags are also defined in header file sc2_defs.h. To get the defined names of the flags preceed the above Flag name with "GENERALCAPS3_".
E.g. flag HDSDI_1G5 is defined as GENERALCAPS3_HDSDI_1G5

2.2.2.6 PCO_Description2 structure

Name	Type	Description
wSize	WORD	Size of this structure
ZZwAlignDummy1	WORD	Reserved
dwMinPeriodicalTimeDESC2	DWORD	Minimum periodical time in ns
dwMaxPeriodicalTimeDESC2	DWORD	Maximum periodical time in ms
dwMinPeriodicalConditionDESC2	DWORD	Minimum periodical time condition Periodical time – exposure time must not be smaller than 'min per additional'
dwMaxNumberOfExposuresDESC2	DWORD	Maximum number of exposures in one frame
lMinMonitorSignalOffsetDESC2	LONG	Minimum monitor signal offset time in ns
dwMaxMonitorSignalOffsetDESC2	DWORD	Maximum monitor signal offset Maximum negative monitor signal offset in ns
dwMinPeriodicalStepDESC2	DWORD	Minimum periodical time step in ns
dwStartTimeDelayDESC2	DWORD	Constant maximum value for monitor signal offset in nsec in case of delay = 0
dwMinMonitorStepDESC2	DWORD	Minimum monitor step time in ns
dwMinDelayModDESC2	DWORD	Minimum delay time in ns in modulate mode
dwMaxDelayModDESC2	DWORD	Maximum delay time in ms in modulate mode
dwMinDelayStepModDESC2	DWORD	Minimum delay time step in ns in modulate mode
dwMinExposureModDESC2	DWORD	Minimum exposure time in ns in modulate mode
dwMaxExposureModDESC2	DWORD	Maximum exposure time in ms in modulate mode
dwMinExposureStepModDESC2	DWORD	Minimum exposure time step in ns in modulate mode
dwModulateCapsDESC2	DWORD	Modulate capability bit field describing the availability of optional functionality. (see table **ModulateCaps-Bits**)
dwReserved	DWORD	Reserved
ZZdwDummy	DWORD	Reserved

2.2.2.7 ModulateCaps-Bits

Flag name	Value	Description
MODULATE	0x00000001	Modulate is available

2.3 GENERAL CAMERA STATUS

The general status group contains functions to get access to the information, which camera is connected and if the camera is operating in good condition. Additionally there exist functions to set the camera to a default operating state.

2.3.1 PCO_GetGeneral

Description:

General information is queried from the camera and the variables of the **PCO_General structure** are filled with this information. This function is a combined version of the following functions, which request information about camera type, hardware/firmware version, serial number, current temperatures and camera status.

The **PCO_General structure** contains an embedded structure. Please fill in the regarding `wSize` parameters.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetGeneral(
    HANDLE ph,                          //in
    PCO_General* strGeneral             //in,out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`strGeneral`	`PCO_General*`	Pointer to a **PCO_General structure**: • On input the `wSize` parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes • On output the structure is filled with the requested information from the camera

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.3.1.1 PCO_General structure

Name	Type	Description
wSize	WORD	Size of this structure
ZZwAlignDummy1	WORD	Reserved
strCamType	PCO_CameraType	See **PCO_GetCameraType**
dwCamHealthWarnings	DWORD	Bitmask of warnings in camera system
dwCamHealthErrors	DWORD	Bitmask of errors in camera system
dwCamHealthStatus	DWORD	Bitmask of camera system status
sCCDTemperature	SHORT	Temperature of image sensor in tenth of a degree. e.g. 100 = 10.0 °C
sCamTemperature	SHORT	Temperature inside camera housing
sPowerSupplyTemperature	SHORT	Temperature of additional device (e.g. power supply)
ZZwDummy[]	WORD	Reserved

2.3.2 PCO_GetCameraType

Description:

This function retrieves the following parameters of the camera: camera type code, hardware/firmware version, serial number and interface type.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraType(
    HANDLE ph,                      //in
    PCO_CameraType* strCamType      //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strCamType	PCO_CameraType*	Pointer to a **PCO_CameraType structure**: • On input the wSize parameter of this structure must be filled with the correct structure size in bytes • On output the structure is filled with the requested information from the camera

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.3.2.1 PCO_CameraType structure

Name	Type	Description
wSize	WORD	Size of this structure
wCamType	WORD	Camera type code, see table **Camera Type Codes**
wCamSubType	WORD	Camera subtype code
ZZwAlignDummy1	WORD	Reserved
dwSerialNumber	DWORD	Serial number of the camera
dwHWVersion	DWORD	Global hardware version. The most significant WORD is the version number and the lower significant WORD is the revision number. (e.g. 0x00020001 = version revision 2.01)
dwFWVersion	DWORD	Global firmware version (deprecated). The most significant WORD is the version number and the lower significant WORD is the revision number. The variable is not valid in newer firmware. The firmware version structure must be used instead to get reliable information
wInterfaceType	WORD	Interface type code, see table **Interface Type Codes**
strHardwareVersion	PCO_HW_Vers	Hardware versions of installed devices. An array of up to 10 hardware info structures. One structure for each existing device.
strFirmwareVersion	PCO_FW_Vers	Firmware versions of all devices. An array of up to 10 firmware info structures. One structure for each existing device. In case more than 10 devices exist in the camera **PCO_GetFirmwareInfo** must be used to retrieve the structures of the additional devices.
ZZwDummy[]	WORD	Reserved

2.3.2.2 Camera Type Codes

Camera	Value	Camera	Value	Camera	Value
pco.edge 5.5 CL	0x1300	pco.dimax	0x1000	pco.1200HS	0x0100
pco.edge 4.2 CL	0x1302	pco.dimax_TV	0x1010	pco.1300	0x0200
pco.edge GL	0x1310	pco.dimax CS	0x1020	pco.1600	0x0220
pco.edge USB3	0x1320	pco.flim	0x1400	pco.2000	0x0240
pco.edge CLHS	0x1340	pco.panda	0x1500	pco.4000	0x0260
pco.edge MT	0x1304	pco.pixelfly usb	0x0800	pco.1400	0x0830

2.3.2.3 Interface Type Codes

Interface type	Value	Interface type	Value	Interface type	Value
FireWire	0x0001	GigE	0x0004	CLHS	0x0007
Camera Link	0x0002	Serial Interface	0x0005		
USB 2.0	0x0003	USB 3.0	0x0006		

2.3.3 PCO_GetCameraHealthStatus

Description:

The **PCO_GetCameraHealthStatus** function retrieves information about the current camera status. The returned parameters are presented as a bit field, where each bit is describing a distinct camera condition. Cleared bits in the bitfield indicate that the particular condition is not valid, set bits show valid (error, warning, status) conditions.
In case an error condition is recognized the hardware might get damaged, when operation of the camera is continued. Therefore the application should report the error condition to the user and prompt him to switch off the camera as soon as possible.
In case a warning condition is recognized, the operation of the camera can continue, but the image quality might suffer.
The status information does give hints about current camera state. It can be determined, if the camera is in the default state (power up), if a **PCO_ArmCamera** was successfully executed and if camera is currently recording.
It is recommended to call this function frequently (e.g. every 5s or after calling **PCO_ArmCamera**) in order to recognize camera internal problems. This helps to prevent camera hardware from damage.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraHealthStatus(
    HANDLE ph,                    //in
    DWORD* dwWarn,                //out
    DWORD* dwErr,                 //out
    DWORD* dwStatus               //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwWarn	DWORD*	Pointer to a DWORD variable to get warning bit field (see **Warning Bits**)
dwErr	DWORD*	Pointer to a DWORD variable to get error bit field (see **Error Bits)**
dwStatus	DWORD*	Pointer to a DWORD variable to get the status bit field (see **Status Bits**)

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Example:

```
// -- C/C++ example -------------------------------------------
if (errorcode & 0x00000001) //    power supply voltage range error
{
// report error to user etc.
}
// ------------------------------------------------------------
```

2.3.3.1 Warning Bits

Value	Description
0x00000001	Power supply voltage near limits
0x00000002	Power supply temperature near limit
0x00000004	Camera temperature near limit (board temperature / FPGA temperature)
0x00000008	Image sensor temperature near limit (for cooled camera versions only)
0x00000010	External battery nearly discharged
0x00000020	Offset regulation range near limit

2.3.3.2 Error Bits

Value	Description
0x00000001	Power supply voltage out of limits
0x00000002	Power supply temperature out of limit
0x00000004	Camera temperature out of limit (board temperature / FPGA temperature)
0x00000008	Image sensor temperature out of limit (for cooled camera versions only)
0x00000010	External battery completely discharged
0x00010000	Camera interface failure
0x00020000	Camera RAM module failure
0x00040000	Camera main board failure
0x00080000	Camera head board failure

2.3.3.3 Status Bits

Name	Description	
0x00000001	Default state:	
	Bit set:	No settings changed, camera is in default state
	Bit cleared:	Settings were changed since power up or reset
0x00000002	Settings valid:	
	Bit set:	Settings are valid. Last **PCO_ArmCamera** was successful and no settings were changed since then (except exposure time)
	Bit cleared:	Settings were changed but not yet checked and accepted by **PCO_ArmCamera** command
0x00000004	Recording state:	
	Bit set:	Recording state is on
	Bit cleared:	Recording state is off
0x00000008	Sensor readout state:	
	Bit set:	Sensor data readout is running
	Bit cleared:	No sensor data readout at the moment
0x00000010	Frame rate state:	
	Bit set:	Valid image timing was set from **PCO_SetFrameRate** call
	Bit cleared:	Valid image timing was set from **PCO_SetDelayExposureTime** call
0x00000020	State of trigger signal for sequence stop:	
	Bit set:	A trigger signal for stopping the sequence has already arrived and the camera does capture the additional frames of the sequence
	Bit cleared:	Sequence trigger cleared
0x00000040	Camera locked to external sync	
	Bit set:	The internal PLL is locked to the external sync signal
	Bit cleared:	No external sync signal or signal not locked.
0x00000080	Battery status	
	Bit set:	A rechargable battery pack is connected
	Bit cleared:	No battery available
0x00000100	Power save (only valid if battery is connected)	
	Bit set:	Camera is in power save mode. Normal operation is not possible, but recorded image data is maintained as long as possible. To readout the data the camera must be connected to the normal power supply
	Bit cleared:	Camera is in normal operation mode
0x00000200	Power save left	
	Bit set:	Camera has been in power save mode and power was reconnected. Image data from last recording can be readout, but no other settings are valid
	Bit cleared:	Camera is in normal operation mode
0x00000400	Camera locked to IRIG time code generator	
	Bit set:	An IRIG time code signal is connected to the appropriate input and the camera is locked to this signal. Camera timestamp information (date and time) is adopted to the external time code
	Bit cleared:	No IRIG information available
0x80000000	Reserved	

2.3.4 PCO_GetTemperature

Description:

This function retrieves the current temperatures in °C of the imaging sensor, camera and additional devices e.g. power supply. The image sensor and the additional device temperature are not available for all cameras. In this case the following values will be returned:
Image sensor temperature missing: `sCCDTemp = 0x8000`
Additional device temperature missing: `sPowTemp = 0x0000`

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetTemperature(
    HANDLE ph,                      //in
    SHORT* sCCDTemp,                //out
    SHORT* sCamTemp,                //out
    SHORT* sPowTemp                 //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
sCCDTemp	SHORT*	Pointer to a SHORT variable to get the image sensor temperature in tenth of a degree. e.g. 100 = 10.0 °C
sCamTemp	SHORT*	Pointer to a SHORT variable to get the internal temperature of the camera in °C
sPowTemp	SHORT*	Pointer to a SHORT variable to get the temperature of additional devices (e.g. power supply) in °C

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.3.5 PCO_GetInfoString

Description:

This function retrieves some information about the camera, e.g. sensor name. A zero terminated ASCII string will be returned in the provided array. This array must be large enough to hold the complete string and the termination value, if not, an error will be returned. At most 500 bytes will be returned from the camera. If a specific info type is not available for the camera an error will be returned.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetInfoString(
    HANDLE ph,                      //in
    DWORD dwinfotype,               //in
    char *buf,                      //out
    WORD  size_in                   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwinfotype	DWORD	Specifies the camera information to inquire, see table **InfoType**
buf	char*	Pointer to a character array. The requested information, as ASCII string.
ilen	WORD	Size of the character array, which is passed in

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.3.5.1 InfoType

Value	Name	Description
0x00000000	INFO_STRING_PCO_INTERFACE	Camera name & interface information
0x00000001	INFO_STRING_CAMERA	Camera name
0x00000002	INFO_STRING_SENSOR	Sensor name
0x00000003	INFO_STRING_PCO_MATERIALNUMBER	Production number
0x00000004	INFO_STRING_BUILD	Firmware build number and date
0x00000005	INFO_STRING_PCO_INCLUDE	Firmware build include revision

2.3.6 PCO_GetCameraName

Description:

This function retrieves the name of the camera. A zero terminated ASCII string will be returned in the provided array. This array must be large enough to hold the complete string and the termination value, if not, an error will be returned. At most 40 bytes will be returned from the camera.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraName(
    HANDLE ph,                          //in
    char* szCameraName,                 //out
    WORD wSZCameraNameLen               //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
szCameraName	char*	Pointer to a character array (40 byte) The camera name, as ASCII string
wSZCameraNameLen	WORD	Size of the array szCameraName, which has passed in

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.3.7 PCO_GetFirmwareInfo

Description:

Query firmware versions of all devices in the camera such as main microprocessor, main FPGA and coprocessors of the interface boards. Because the number of devices can exceed the number of information fields of the `PCO_FW_Vers` structure additional blocks of information can be requested using the `wDeviceBlock` parameter. The first call should be made with `wDeviceBlock` parameter set to 0. On return the `wDeviceNum` parameter of the `PCO_FW_Vers` will be filled with the number of ***all*** installed devices in the camera. Up to this number, each `Device` structure will contain the firmware information for a particular device. Further calls with increasing `wDeviceBlock` parameter might be necessary to get all available firmware versions.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFirmwareInfo(
    HANDLE ph,                          //in
    WORD wDeviceBlock,                  //in
    PCO_FW_Vers* pstrFirmWareVersion //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wDeviceBlock`	`WORD`	Address a block of information; 0 gets the first 10 devices
`pstrFirmWare Version`	`PCO_FW_Vers*`	Pointer to a `PCO_FW_Vers` structure On output the structure is filled with following information: • `pstrFirmWareVersion.wDeviceNum`: Number of available devices in the camera • `pstrFirmWareVersion.Device[0…9]`: An array of 10 **PCO_SC2_Firmware_DESC Structure** filled with the version information of available devices

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.3.7.1 PCO_SC2_Firmware_DESC Structure

Name	Type	Description
szName[16]	char	The device name, as ASCII string 16 bytes long
bMinorRev	BYTE	The minor revison of the device
bMajorRev	BYTE	The major revison of the device
wVariant	WORD	The variant of the device
ZZwDummy[22]	WORD	Reserved

2.3.8 PCO_GetColorCorrectionMatrix

Description:

This function returns the color multiplier matrix from the camera. The color multiplier matrix can be used to normalize the color values of a color camera to a color temperature of 6500k.
The color multiplier matrix is specific for each camera and is determined through a special calibration procedure.

Supported camera type:

pco.dimax, pco.edge, pco.pixelfly usb

Descriptor dependency:

dwGeneralCapsDESC1: CCM

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetColorCorrectionMatrix (
    HANDLE ph,                          //in
    double* pdMatrix                    //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
pdMatrix	double*	Pointer to an array of nine double values. The array is arranged as a 3x3 matrix containing the color coefficients

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4 GENERAL CAMERA CONTROL

2.4.1 PCO_ArmCamera

Description:

This function does arm, this means prepare the camera for a following recording. All configurations and settings made up to this moment are accepted, validated and the internal settings of the camera are prepared. If the arm was successful the camera state is changed to [armed] and the camera is able to start image recording immediately, when ***Recording State*** is set to [run].

The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

On power up the camera is in state [not armed] and ***Recording State*** [stop].
Camera arm state is changed to [not armed], when settings are changed, with the following exception. Camera arm state is not changed, when settings related to exposure time will be done during ***Recording State*** [run].

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_ArmCamera (
    HANDLE ph,                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4.2 PCO_CamLinkSetImageParameters (obsolete)

Description:

This function is marked as obsolete and will be removed in future releases of the SDK. Function **PCO_SetImageParameters** should be used instead.
This function does set the image parameters for internal allocated resources. While using Camera Link, Camera Link HS (CLHS) or GigE interface this function must be called, before an image transfer is started from the camera and the image size has been changed since the last **PCO_ArmCamera** call. Because for all other interfaces this is a dummy call, which always returns ***PCO_NOERROR***, this function can remain in the program, regardless which camera interface is used. The size parameters are adapted internally, if ***Meta Data*** mode is enabled (see **PCO_SetMetaDataMode**).

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_CamLinkSetImageParameters (
    HANDLE ph,                          //in
    WORD wxres,                         //in
    WORD wyres                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wxres	WORD	Current horizontal resolution of the image to be transferred
wyres	WORD	Current vertical resolution of the image to be transferred

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4.3 PCO_SetImageParameters

Description:

This function does set the image parameters for internal allocated resources. This function must be called, before an image transfer is started from the camera and the image size has been changed since the last **PCO_ArmCamera** call, this does also include setting a new segment for image readout of the camera internal memory (CamRam).
The size parameters are adapted internally, if ***Meta Data*** mode or ***Soft ROI*** are enabled.
In case ***Soft ROI***, see **PCO_EnableSoftROI**, is in use, `dwFlags` parameter must be set according to the subsequent program sequence, to ensure that the correct ***Soft ROI*** parameters are used. If next images will be transferred from a recording camera, flag `IMAGEPARAMETERS_READ_WHILE_RECORDING` must be set. If next action is to readout images from the camera internal memory, flag `IMAGEPARAMETERS_READ_FROM_SEGMENTS` must be set.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetImageParameters (
    HANDLE ph,                          //in
    WORD wxres,                         //in
    WORD wyres,                         //in
    DWORD dwFlags,                      //in
    void *param,                        //in
    int ilen                            //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wxres`	`WORD`	Current horizontal resolution of the image to be transferred
`wyres`	`WORD`	Current vertical resolution of the image to be transferred
`dwFlags`	`DWORD`	Soft ROI action bit field , see table **Image Parameter Bits** Only valid, if **PCO_EnableSoftROI** is enabled
`param`	`void*`	Reserved
`ilen`	`int`	Reserved

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4.3.1 Image Parameter Bits

Flag name	Value	Description
IMAGEPARAMETERS_READ_WHILE_RECORDING	0x00000001	Next image transfers will be done from a recording camera
IMAGEPARAMETERS_READ_FROM_SEGMENTS	0x00000002	Next image transfers will be done from the camera internal memory
	Bit2-31	Reserved

2.4.4 PCO_ResetSettingsToDefault

Description:

This function can be used to reset all camera settings to its default values. This function is also executed during a power-up sequence. The camera must be stopped before calling this command. Default settings are slightly different for all cameras.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_ResetSettingsToDefault(
    HANDLE ph                       //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4.4.1 Default settings

Setting	Default
Sensor Format	Standard
ROI	Full resolution
Binning	No binning
Pixel Rate	Depending on camera type
Gain	Normal gain (if setting available due to sensor)
Double Image Mode	Off
IR sensitivity	Off (if setting available due to sensor)
Cooling Setpoint	Depending on camera type
ADC mode	Using one ADC (if option available)
Exposure Time	Depending on camera type (10-20ms)
Delay Time	0
Trigger Mode	Auto Trigger
Recording state	Stopped
Memory Segmentation	Total memory allocated to first segment (if option available)
Storage Mode	Recorder ***Ring Buffer*** and ***Live View*** on
Acquire Mode	Auto

2.4.5 PCO_SetTimeouts

Description:

This function does set the internal timeout values for different tasks. Usually there is no need to change these values.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetTimeouts (
    HANDLE ph,                          //in
    void *buf_in,                       //in
    unsigned int size_in                //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
buf_in	void*	Pointer to an array of unsigned int values : • buf_in[0] = command timeout in ms A command sequence will be aborted and a timout error returned, if there is no response from the camera within the command timeout value • buf_in[1] = image timeout in ms An image request will be aborted and a timout error returned, if no image is transferred from the camera within the image timeout value. Only valid for the **PCO_GetImageEx** command • buf_in[2] = transfer timeout in ms Specifies the time to hold transfer resources. While image sequences are running transfer resources are allocated in some of the driver layer. To enable faster start time for the next image sequence these resources are held the set 'transfer timeout' time, after the last image of the sequence is transferred. **PCO_CancelImages** always deallocates the hold resources
size_in	unsigned int	Number of valid values in the array in bytes

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4.6 PCO_RebootCamera

Description:

This function will reboot the camera. The function will return immediately and the reboot process in the camera is started. After calling this command the handle to this camera should be closed with **PCO_CloseCamera**.
When reboot is finished (approximately after 6 to 10 seconds, up to 40 seconds for cameras with GigE interface) the camera can be reopened with a **PCO_OpenCameraEx** call. The reboot command is used during firmware update and is necessary when camera setup is changed.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_RebootCamera(
    HANDLE ph                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4.7 PCO_GetCameraSetup

Description:

Command can be used to get the shutter mode of a pco.edge.This function is used to query the current operation mode of the camera. Some cameras can work at different operation modes with different descriptor settings.

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraSetup(
    HANDLE ph,                          //in
    WORD* wType,                        //in,out
    DWORD* dwSetup,                     //out
    WORD* wLen                          //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wType	WORD*	Pointer to a WORD variable to receive the current setup type: • On input this variable must be set to zero • On output the variable indicates the current available setup type, which must be used in the **PCO_SetCameraSetup** function
dwSetup	DWORD*	Pointer to a DWORD array with 4 DWORDs: • On output the fields are filled with the available information
wLen	WORD*	Pointer to a WORD variable: • On input to indicate the length of the dwSetup array in DWORDs. Usually this parameter is set to 4

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.4.7.1 pco.edge dwSetup[0]

Value	Type	Description
0x00000001	PCO_EDGE_SETUP_ROLLING_SHUTTER	Camera is in rolling shutter operation mode
0x00000002	PCO_EDGE_SETUP_GLOBAL_SHUTTER	Camera is in global shutter operation mode
0x00000004	PCO_EDGE_SETUP_GLOBAL_RESET	Camera is in global reset operation mode

2.4.8 PCO_SetCameraSetup

Description

Command can be used to set the shutter mode of a pco.edge.This function is used to set the operation mode of the camera. If operation mode is changed, **PCO_RebootCamera** must be called afterwards. It is recommended to set the command timeout to 2000 ms by calling **PCO_SetTimeouts** before changing the setup.

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetCameraSetup(
    HANDLE ph,                          //in
    WORD wType,                         //in
    DWORD* dwSetup,                     //in
    WORD wLen                           //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wType	WORD	WORD to set the current setup type. Must be set to the value returned from a previous **PCO_GetCameraSetup** call
dwSetup	DWORD*	Pointer to a DWORD array with 4 DWORDs. For the pco.edge the values from table **pco.edge dwSetup[0]** can be used
wLen	WORD	WORD to indicate the the length of the dwSetup array in DWORDs

Parameter dependency:

dwGeneralCapsDESC1: NO_GLOBAL_SHUTTER,GLOBAL_RESET_MODE

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Example:

Get the current operation mode of the camera, change timeout values and set the camera to Global Shutter mode. Afterwards send reboot command and close the handle to the camera. After the camera is closed it is recommended to wait for 10s before reopening again.

```
DWORD m_dwSetup[4];
WORD m_wLen = sizeof(m_dwSetup)/sizeof(DWORD);
WORD m_wType = 0; // Set to zero initially
int ts[3] = { 2000, 3000, 250}; // command, image, channel timeout

PCO_OpenCamera(&ph,0);
PCO_GetCameraSetup(ph, &m_wType, &m_dwSetup[0], &m_wLen);
m_dwSetup[0] = PCO_EDGE_SETUP_GLOBAL_SHUTTER;
PCO_SetTimeouts(ph, &ts[0], sizeof(ts));
PCO_SetCameraSetup(ph, m_wType, &m_dwSetup[0], m_wLen);
PCO_RebootCamera(ph);
PCO_CloseCamera(ph);
```

2.4.9 PCO_ControlCommandCall

Description:

This function issues a low level command to the camera. This call is part of most of the other calls. Normally calling this function is not needed. It can be used to cover those camera commands, which are not implemented in regular SDK functions.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_ControlCommandCall (
    HANDLE ph,                      //in
    void *buf_in,                   //in
    unsigned int size_in,           //in
    void *buf_out,                  //in,out
    unsigned int size_out           //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
*buf_in	void	Pointer to a buffer which does hold the camera command telegram
size_in	unsigned int	Size of the input buffer in bytes
*buf_out	void	Pointer to a buffer which does hold the camera response telegram
size_out	unsigned int	Size of the output buffer in bytes

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5 IMAGE SENSOR

This function group can be used to get or set parameters regarding the image readout of the imaging sensor.
If one parameter of these settings is changed in most cases also other parameters must be changed to ensure, that the validation of all settings during **PCO_ArmCamera** is successful.
Setting parameters in this group can only be done, if ***Recording State*** is [stop], see **PCO_GetRecordingStruct.**

2.5.1 PCO_GetSensorStruct

Description:

Sensor related information is queried from the camera and the variables of the **PCO_Sensor structure** are filled with this information. This function is a combined version of the functions, which request information about the installed imaging sensor and the current settings of sensor related parameter like binning, ROI, pixel clock and others. For a detailed description of each parameter see the functions in this chapter.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetSensorStruct(
    HANDLE ph,                          //in
    PCO_Sensor* strSensor               //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strSensor	PCO_Sensor*	Pointer to a **PCO_Sensor structure**: • On input the wSize parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes • On output the structure is filled with the requested information from the camera

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.2 PCO_SetSensorStruct

Description:

This function does set the complete set of sensor settings at once. For the sake of clarity it is better to use the functions which change distinct parameter, despite changing all settings at once. An invalid value for one of the parameters will result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetSensorStruct(
    HANDLE ph,                          //in
    PCO_Sensor* strSensor               //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strSensor	PCO_Sensor*	Pointer to a **PCO_Sensor structure** filled with appropriate parameters The wSize parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Typical implementation for setting wSize parameter of embedded structures

```
PCO_Sensor strSensor;
strSensor.wSize = sizeof(strSensor);
strSensor.strDescription.wSize = sizeof(strSensor.strDescription);
strSensor.strDescription2.wSize = sizeof(strSensor.strDescription2);
strSensor.strSignalDesc.wSize = sizeof(strSensor.strSignalDesc);
for (int i = 0; i < NUM_MAX_SIGNALS; i++){
     strSensor.strSignalDesc.strSingeSignalDesc[i].wSize =
     sizeof(strSensor.strSignalDesc.strSingeSignalDesc[i]);
}
```

2.5.2.1 PCO_Sensor structure

Name	Type	Description
wSize	WORD	Size of this structure
ZZwAlignDummy1	WORD	Reserved
strDescription	PCO_Description	See **PCO_Description structure**
strDescription2	PCO_Description2	See **PCO_Description2 structure**
ZZdwDummy2[]	DWORD	Reserved
wSensorformat	WORD	Sensor format
wRoiX0	WORD	Left horizontal ROI, starting with 1
wRoiY0	WORD	Upper vertical ROI, starting with 1
wRoiX1	WORD	Right horizontal ROI, up to the maximal horizontal size of the sensor
wRoiY1	WORD	Lower vertical ROI, up to the maximal vertical size of the sensor
wBinHorz	WORD	Horizontal binning
wBinVert	WORD	Vertical binning
ZZwAlignDummy2	WORD	Reserved
dwPixelRate	DWORD	Pixel rate in Hz. Only the values in the dwPixelRateDESC array of the **PCO_Description structure** can be used.
wConvFact	WORD	Conversion factor. Only the values in the wConvFactDESC array of the **PCO_Description structure** can be used.
wDoubleImage	WORD	Double image mode
wADCOperation	WORD	Number of used ADCs
wIR	WORD	IR sensitivity mode
sCoolSet	SHORT	Cooling setpoint
wOffsetRegulation	WORD	Offset regulation mode
wNoiseFilterMode	WORD	Noise filter mode
wFastReadoutMode	WORD	Fast readout mode
wDSNUAdjustMode	WORD	Dark signal non uniformaty adjustment mode
wCDIMode	WORD	Correlated double image mode
ZZwDummy[]	WORD	Reserved
strSignalDesc	PCO_Signal_Description	Signal descriptor for camera input / output connectors
ZZdwDummy[]	DWORD	Reserved

2.5.3 PCO_GetSizes

Description:

This function does return the current armed image size of the camera. If the user recently changed the size influencing values without issuing a **PCO_ArmCamera**, the **PCO_GetSizes** function will return the sizes from the last recording. If no recording occurred, it will return the last ROI settings. In case ***Soft ROI*** is enabled, **PCO_GetSizes** returns the sizes of the current ***Soft ROI***. The values `wXResAct` and `wYResAct` return the current size, which should be used to allocate the buffers for image transfer. The values `wXResMax` and `wYResMax` return the maximum possible resolution including doubleshutter mode if available.

PCO recommends the following order of commands: **PCO_SetBinning, PCO_SetROI, PCO_ArmCamera, PCO_GetSizes** and **PCO_AllocateBuffer**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetSizes(
    HANDLE ph,                          //in
    WORD* wXResAct,                     //out
    WORD* wYResAct,                     //out
    WORD*  wXResMax,                    //out
    WORD* wYResMax                      //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wXResAct	WORD*	Pointer to a WORD variable to get the current horizontal resolution
wYResAct	WORD*	Pointer to a WORD variable to get the current vertical resolution
wXResMax	WORD*	Pointer to a WORD variable to get the maximum horizontal resolution
wYResMax	WORD*	Pointer to a WORD variable to get the maximum vertical resolution

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.4 PCO_GetSensorFormat

Description:

This function retrieves the current sensor format. In the format [standard] only effective pixels are readout from the sensor. The readout in the format [extended] is camera dependent. Either a distinct region of the sensor is selected or the full sensor including effective, dark, reference and dummy pixels.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetSensorFormat(
    HANDLE ph,                          //in
    WORD* wSensor                       //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSensor	WORD*	Pointer to a WORD variable to get the sensor format: • 0x0000 = [standard] • 0x0001 = [extended]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.5 PCO_SetSensorFormat

Description:

This function does set the sensor format. In the format [standard] only effective pixels are readout from the sensor. The readout in the format [extended] is camera dependent. Either a distinct region of the sensor is selected or the full sensor including effective, dark, reference and dummy pixels. Invalid values result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetSensorFormat(
    HANDLE ph,                          //in
    WORD wSensor                        //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSensor	WORD	WORD variable to set the sensor format: • 0x0000 = [standard] • 0x0001 = [extended]

Parameter dependency:

None

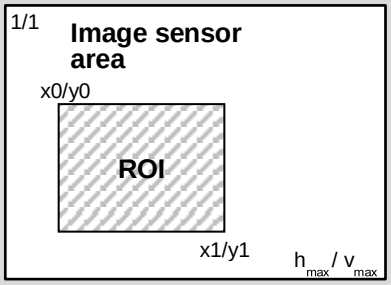
Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.6 PCO_GetROI

Description:

This function returns the current ROI (region of interest) setting in pixels, see figure. If ***Soft ROI*** is enabled (see **PCO_EnableSoftROI**) the current setting of the ***Soft ROI*** are returned otherwise the ROI registers in the camera are readout. The returned ROI is always equal to or smaller than the active image area, which is defined by the settings of ***format*** and ***binning*** (see chapter **IMAGE AREA SELECTION** (ROI)).



Supported camera type:

All cameras

Descriptor dependency:

`wRoiHorStepsDESC, wRoiVertStepsDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetROI(
    HANDLE ph,                          //in
    WORD* wRoiX0,                       //out
    WORD* wRoiY0,                       //out
    WORD* wRoiX1,                       //out
    WORD* wRoiY1                        //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wRoiX0	WORD*	Pointer to a WORD variable to get the horizontal coordinate of the ROI
wRoiY0	WORD*	Pointer to a WORD variable to get the vertical coordinate of the ROI
wRoiX1	WORD*	Pointer to a WORD variable to get the horizontal coordinate of the ROI
wRoiY1	WORD*	Pointer to a WORD variable to get the vertical coordinate of the ROI

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.7 PCO_SetROI

Description:

This function does set a ROI (region of interest) area on the sensor of the camera.
See chapter **IMAGE AREA SELECTION (ROI)** how valid input parameters can be determined.
Invalid values will result in a failure response message either immediately or from next **PCO_ArmCamera** call.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

pco.edge, pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`wRoiHorStepsDESC, wRoiVertStepsDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetROI(
    HANDLE ph,                          //in
    WORD wRoiX0,                        //in
    WORD wRoiY0,                        //in
    WORD wRoiX1,                        //in
    WORD wRoiY1                         //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wRoiX0	WORD	WORD variable to set the horizontal start coordinate of the ROI
wRoiY0	WORD	WORD variable to set the vertical start coordinate of the ROI
wRoiX1	WORD	WORD variable to set the horizontal end coordinate of the ROI
wRoiY1	WORD	WORD variable to set the vertical end coordinate of the ROI

Parameter dependency:

`wMaxHorzResStdDESC, wMaxVertResStdDESC`
`wMaxHorzResExtDES, wMaxVertResExtDESC`
`wRoiHorStepsDESC, wRoiVertStepsDESC`
`wMinSizeHorzDESC, wMinSizeVertDESC`
`wSoftRoiHorStepsDESC, wSoftRoiVertStepsDESC`
`dwGeneralCapsDESC1:`
`ROI_VERT_SYMM_TO_HORZ_AXIS, ROI_VERT_SYMM_TO_VERT_AXIS`

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.8 PCO_GetBinning

Description:

This function returns the current binning setting of the camera values for horizontal and vertical direction.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetBinning(
    HANDLE ph,                          //in
    WORD* wBinHorz,                     //out
    WORD* wBinVert                      //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wBinHorz	WORD*	Pointer to a WORD variable to get the horizontal binning
wBinVert	WORD*	Pointer to a WORD variable to get the vertical binning

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.9 PCO_SetBinning

Description:

This function does set the horizontal and vertical binning of the camera. Possible values can be calculated from the binning parameter in the **PCO_Description structure**. If the binning settings are changed, the ROI (region of interest) setting must be adapted, before **PCO_ArmCamera** is called. See chapter **IMAGE AREA SELECTION (ROI)**.
Invalid values result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetBinning(
    HANDLE ph,                          //in
    WORD wBinHorz,                      //in
    WORD wBinVert                       //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wBinHorz	WORD	WORD variable to set the horizontal binning
wBinVert	WORD	WORD variable to set the vertical binning

Parameter dependency:

wMaxBinHorzDESC, wMaxBinVertDESC
wBinHorzSteppingDESC, wBinVertSteppingDESC

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.10 PCO_GetPixelRate

Description:

This function returns the current pixel rate of the camera in Hz. The pixel rate determines the sensor readout speed.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetPixelRate(
    HANDLE ph,                          //in
    DWORD* dwPixelRate                  //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwPixelRate	DWORD*	Pointer to a DWORD variable to get the pixel rate in Hz

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.11 PCO_SetPixelRate

Description:

This function does set the pixel rate for the sensor readout. Only values, which are listed in the parameter list `dwPixelRateDESC` of the **PCO_Description structure**, are accepted.
For pco.edge 5.5 with Camera Link interface **PCO_SetTransferParameter** and **PCO_SetActiveLookuptable** with appropriate parameters must be called. See chapter **IMAGE AREA SELECTION (ROI)**.
Invalid values result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetPixelRate(
    HANDLE ph,                          //in
    DWORD dwPixelRate                   //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`dwPixelRate`	`DWORD`	DWORD variable to set the pixel rate in Hz

Parameter dependency:

`dwPixelRateDESC`

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.12 PCO_GetConversionFactor

Description:

This function returns the current conversion factor setting of the image sensor multiplied with the factor 100.
To get the current conversion factor in electrons / count the returned value must be divided by 100.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetConversionFactor(
    HANDLE ph,                          //in
    WORD* wConvFact                     //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wConvFact	WORD*	Pointer to a WORD variable to get the conversion factor

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.13 PCO_SetConversionFactor

Description:

This function does set the conversion factor of the camera. Only values, which are listed in the parameter list `wConvFactDESC` of the **PCO_Description structure**, are accepted. The input value is calculated from the conversion factor in electrons / count multiplied with 100.
Invalid values result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetConversionFactor(
    HANDLE ph,                          //in
    WORD wConvFact                      //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wConvFact`	`WORD`	`WORD` variable to set the conversion factor

Parameter dependency:

`wConvFactDESC`

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.14 PCO_GetDoubleImageMode

Description:

This function returns whether the camera is running in double image mode or not.

Supported camera type:

All cameras

Descriptor dependency:

`wDoubleImageDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetDoubleImageMode(
    HANDLE ph,                          //in
    WORD* wDoubleImage                  //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wDoubleImage`	`WORD*`	Pointer to a `WORD` variable to get the double image mode: • `0x0000` = double image mode [OFF] • `0x0001` = double image mode [ON]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.15 PCO_SetDoubleImageMode

Description:

This function does set the double image operating mode. Cameras with activated ***double image mode*** readout two exposures separated by a short interleaving time. The resulting double image is transferred as one frame that means the two images resulting from the two / double exposures are stitched together as one and are counted as one.
Thus the buffer size of all allocated buffers has to be doubled. The first half of the buffer will be filled with the first exposed frame (image A). The second exposed frame (image B) will be transferred to the second half of the buffer.
Invalid values result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

`wDoubleImageDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetDoubleImageMode(
      HANDLE ph,                    //in
      WORD wDoubleImage             //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wDoubleImage`	`WORD`	`WORD` variable to set the double image mode: • `0x0000` = double image mode [OFF] • `0x0001` = double image mode [ON]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.16 PCO_GetADCOperation

Description:

This function returns the ADC (analog / digital converter) operating mode (single / dual) currently used from the camera.

Supported camera type:

All cameras

Descriptor dependency:

`wNumADCsDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetADCOperation(
      HANDLE ph,                    //in
      WORD* wADCOperation           //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wADCOperation`	`WORD*`	Pointer to a `WORD` variable to get the ADC operation: • `0x0001` = [single ADC] • `0x0002` = [dual ADC]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.17 PCO_SetADCOperation

Description:

This function does set the ADC (analog-digital-converter) operating mode. Possible values are given through the parameter wNumADCsDESC of the **PCO_Description structure**. If sensor data is read out using single ADC operation linearity of image data is enhanced, using dual ADC operation readout is faster and allows higher frame rates. If dual ADC operating mode is set, horizontal ROI must be adapted to symmetrical values.

Supported camera type:

pco.1600, pco.2000, pco.4000

Descriptor dependency:

wNumADCsDESC

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetADCOperation(
      HANDLE ph,                    //in
      WORD wADCOperation            //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wADCOperation	WORD	WORD variable to set the ADC operation mode: • 0x0001 = [single ADC] • 0x0002 = [dual ADC]

Parameter dependency:

wNumADCsDESC

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.18 PCO_GetIRSensitivity

Description:

This function returns the ***IR sensitivity*** operating mode currently used from the camera.

Supported camera type:

All cameras

Descriptor dependency:

`wIRDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetIRSensitivity (
    HANDLE ph,                          //in
    WORD* wIR                           //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wIR`	`WORD*`	Pointer to a `WORD` variable to get the IR sensitivity: • `0x0000` = IR-Sensitivity [OFF] • `0x0001` = IR-Sensitivity [ON]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.19 PCO_SetIRSensitivity

Description:

This function does set the ***IR sensitivity*** operating mode. This option is only available for special camera models with image sensors that have improved IR sensitivity. Availability of this option can be checked with the parameter `wIRDESC` of the **PCO_Description structure**. If ***IR sensitivity*** is activated limits for the delay and exposure times are defined through parameters `dwMinDelayIRDESC`, `dwMaxDelayIRDESC`, `dwMinExposIRDESC` and `dwMaxExposIRDESC` of the **PCO_Description structure**.

Supported camera type:

pco.pixelfly_usb, pco.1300, pco.1400,

Descriptor dependency:

`wIRDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetIRSensitivity (
    HANDLE ph,                          //in
    WORD wIR                            //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wIR`	`WORD`	`WORD` variable to set the IR sensitivity: • `0x0000` = IR sensitivity [OFF] • `0x0001` = IR sensitivity [ON]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.20 PCO_GetCoolingSetpointTemperature

Description:

This function returns the temperature setpoint for the image sensor.
The current sensor temperature can be read out with the **PCO_GetTemperature** function.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: COOLING_SETPOINTS`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCoolingSetpointTemperature (
    HANDLE ph,                          //in
    SHORT* sCoolSet                     //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
sCoolSet	SHORT*	Pointer to a SHORT variable to get the current cooling temperature setpoint as signed value in °C units

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.21 PCO_SetCoolingSetpointTemperature

Description:

This function does set the ***temperature setpoint*** for the image sensor ***in °C***. A peltier cooling unit is used to regulate the temperature of the sensor to the given temperature setpoint. Thus reduces dark current noise and improves image quality. Valid values for the setpoint must be in the range between `sMinCoolSetDESC` and `sMaxCoolSetDESC`. The current temperature must be checked to see when the setpoint temperature is reached.

Default temperature regulation setpoint is defined in `sDefaultCoolSetDESC` parameter of the **PCO_Description structure**. Temperature regulation for the sensor is not available, when both temperature range parameters `sMinCoolSetDESC` and `sMaxCoolSetDESC` of the **PCO_Description structure** are zero.

Valid range depends on camera type. Invalid values result in a failure response message.
The current temperature of the sensor can be read out with the **PCO_GetTemperature** command.

Supported camera type:

pco.1300, pco.1600, pco.2000, pco.4000, pco.edge, pco.flim

Descriptor dependency:

`sMinCoolSetDESC, sMaxCoolSetDESC`
`dwGeneralCapsDESC1: COOLING_SETPOINTS`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetCoolingSetpointTemperature (
    HANDLE ph,                          //in
    SHORT sCoolSet                      //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`sCoolSet`	`SHORT`	`SHORT` variable to set the cooling setpoint

Parameter dependency:

`sMinCoolSetDESC, sMaxCoolSetDESC, sCoolingSetpoints`

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.22 PCO_GetCoolingSetpoints

Description:

This function gets the ***cooling setpoints*** of the camera. This is used when no minimum or maximum range is available.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: COOLING_SETPOINTS`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCoolingSetpoints (
    HANDLE ph,                          //in
    WORD wBlockID,                      //in
    WORD *wNumSetPoints,                //in,out
    SHORT *sCoolSetpoints               //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wBlockID`	`WORD`	Number of the block to query (currently 0)
`wNumSetPoints`	`WORD`	`WORD` pointer to set the max number of setpoints to query and to get the valid number of setpoints inside the camera. In case more than COOLING_SETPOINTS_BLOCKSIZE setpoints are valid they can be queried by incrementing the `wBlockID` till `wNumSetPoints` is reached. The valid members of the setpoints can be used to set the **PCO_SetCoolingSetpointTemperature**
`*sCoolSetpoints`	`SHORT`	Pointer to a `SHORT` array to receive the possible cooling setpoint temperatures. Size of array must be larger enough to hold, COOLING_SETPOINTS_BLOCKSIZE short values

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.23 PCO_GetOffsetMode

Description:

This function returns the current mode for the offset regulation with reference pixels (see respective camera manual for further explanations).

Supported camera type:

pco.pixelfly usb, pco.ultraviolet, pco.1300, pco.1400

Descriptor dependency:

`wOffsetRegulationDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetOffsetMode (
    HANDLE ph,                          //in
    WORD* wOffsetRegulation             //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wOffsetRegulation`	`WORD*`	Pointer to a `WORD` variable to get the offset mode: • `0x0000` = [auto] • `0x0001` = [off]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.24 PCO_SetOffsetMode

Description:

This function does set the operating mode for the offset regulation with reference pixels (see respective camera manual for further explanations).
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

pco.pixelfly usb, pco.ultraviolet, pco.1300, pco.1400

Descriptor dependency:

`wOffsetRegulationDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetOffsetMode (
    HANDLE ph,                          //in
    WORD wOffsetRegulation              //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wOffsetRegulation`	`WORD`	`WORD` variable to set the offset mode: • `0x0000` = [auto] • `0x0001` = [off]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.25 PCO_GetNoiseFilterMode

Description:

This function returns the current operating mode of the image correction in the camera.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: NOISE_FILTER, HOTPIX_ONLY_WITH_NOISE_FILTER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetNoiseFilterMode (
    HANDLE ph,                          //in
    WORD* wNoiseFilterMode              //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wNoiseFilterMode	WORD*	Pointer to a WORD variable to get the noise filter mode: • 0x0000 = [off] • 0x0001 = [on] • 0x0101 = [on + hot pixel correction]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.26 PCO_SetNoiseFilterMode

Description:

This function does set the image correction operating mode of the camera. Image correction can either be switched to totally off, noise filter only mode or noise filter plus ***hot pixel*** correction mode.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: NOISE_FILTER, HOTPIX_ONLY_WITH_NOISE_FILTER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_ SetNoiseFilterMode (
    HANDLE ph,                          //in
    WORD wNoiseFilterMode               //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wNoiseFilterMode`	`WORD`	Noise filter mode: • `0x0000` = [off] • `0x0001` = [on] • `0x0101` = [on + hot pixel correction]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.27 PCO_GetLookuptableInfo

Description:

Description of internal ***lookup tables*** (LUT) is queried. At first the number of available LUTs in the camera must be queried. This can be done by setting all pointers to NULL despite the pointer to `wNumberOffLuts`. The value returned in `wNumberOffLuts` correspondends to the number of available LUTs. Description of a certain LUT can then be queried by calling the function using input parameter `wLUTNum` and provide valid pointers for the other parameters.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetLookupTableInfo (
    HANDLE ph,                          //in
    WORD wLUTNum,                       //in
    WORD*  wNumberOfLuts,               //out
    char* Description,                  //out
    WORD wDescLen,                      //in
    WORD*  wIdentifier                  //out
    BYTE* bInputWidth,                  //out
    BYTE* bOutputWidth,                 //out
    WORD* wFormat                       //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wLUTNum	WORD	WORD variable to select number of LUT to query
wNumberOfLuts	WORD*	Pointer to a WORD variable to get number of LUTs, which can be queried
Description	char*	Pointer to a char array The LUT description as ASCII string. At most 20 bytes are returned from the camera.
wDescLen	WORD	Size of the character array, which is passed in
wIdentifier	WORD*	Pointer to a WORD variable to get the LUT identifier
bInputWidth	BYTE*	Pointer to a BYTE variable to get the number of input bits
bOutputWidth	BYTE*	Pointer to a BYTE variable to get the number of output bits
wFormat	WORD*	Pointer to a WORD variable to get the accepted data structures

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Example:

This example explains how to get the number of available LUTs in your camera using the command **PCO_GetLookupTableInfo** and then information for first LUT.

```
WORD wNumberOfLuts;

PCO_OpenCamera(&ph,0);

PCO_GetLookupTableInfo(ph, 0, &wNumberOfLuts, NULL, 0, NULL, NULL, NULL,
NULL);

char lutname[20];
wORD wDescLen=20;
WORD wIdentifier;
BYTE bInputWidth;
BYTE bOutputWidth;
WORD wFormat;

PCO_GetLookupTableInfo(ph, 0, &wNumberOfLuts, lutname, wDescLen,
&wIdentifier, &bInputWidth, &bOutputWidth, &wFormat);

…
```

2.5.28 PCO_GetActiveLookuptable

Description:

This function returns the active ***lookup table*** (LUT) in the camera.

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetActiveLookupTable(
    HANDLE ph,                          //in
    WORD* wIdentifier,                  //out
    WORD* wOffset                       //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wIdentifier	WORD*	Pointer to a WORD variable to get the identifier of the currently selected LUT: • 0x0000 = [lookup table is disabled] • 0x#### = [identifier of the active lookup table]
wOffset	WORD*	Pointer to a WORD variable to get the currently used offset value for the calculation of the LUT input data

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.5.29 PCO_SetActiveLookuptable

Description:

This function does set an active ***lookup table*** (LUT) in the camera. If `wIdentifier` is set to 0 while calling this function, the lookup table functionality will be disabled and data values from the sensor are directly send to the interface. If `wIdentifier` is one of the available LUT identifiers of the camera data handling is as follows: at first the offset value given by parameter `wOffset` is subtracted from the data values from the sensor. The resulting value is the input to the current selected LUT.
Valid values for the LUT identifier `wIdentifier` can be retrieved with function **PCO_GetLookuptableInfo.**

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetActiveLookupTable (
    HANDLE ph,                          //in
    WORD* wIdentifier,                  //in
    WORD* wOffset                       //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wIdentifier`	`WORD*`	Pointer to a `WORD` variable to select the current LUTs: • `0x0000` = [disable lookup table] • `0x####` = [ID of the LUT to activate]
`wOffset`	`WORD*`	Pointer to a `WORD` variable for the offset

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6 TIMING CONTROL

This function group can be used to get or set parameters regarding the image timing of the camera like trigger mode, exposure time, frame rate and others.
With the function **PCO_GetCOCRunTime** maximum possible frame rate can be evaluated, which does also determine the maximum possible trigger rate for an external triggered camera.
Changing the delay and / or exposure time of the camera either directly or through one of the frame rate functions can be done also when **PCO_SetRecordingState** is [run]. The changed setting is done best possible in the camera, but it might need several image transfers until the effects can be seen in the image data.
It is recommended to use always the ***PCO_Get...*** functions according to the ***PCO_Set..*** function when the image timing parameters should be checked. Mixing different functions might result in wrong return values.
Although delay and exposure time values can be given as a table of values, most cameras support only a single pair of values. Only cameras which have option `wTimeTableDESC` set, can change time values for subsequent images.

2.6.1 PCO_GetTimingStruct

Description:

Timing related information is queried from the camera and the variables of the **PCO_Timing structure** are filled with this information. This function is a combined version of the functions which request information about the current settings of timing related parameter. For a detailed description of each parameter see the functions in this chapter.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetTimingStruct (
    HANDLE ph,                          //in
    PCO_Timing* strTiming               //in,out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`strTiming`	`PCO_Timing*`	Pointer to a **PCO_Timing structure** structure: • On input the `wSize` parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes. • On output the structure is filled with the requested information from the camera to get the timing settings

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.2 PCO_SetTimingStruct

Description:

This function does set the complete set of the timing settings. For the sake of clarity it is better to use the functions which change distinct parameter despite changing all settings at once.
An invalid value for one of the parameters will result in a failure response message.

If a single exposure/delay pair is to be set, the user must set all of the table members to zero except the first member 0. The table member 0 will hold the value for the single delay / exposure pair.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetTimingStruct (
    HANDLE ph,                          //in
    PCO_Timing* strTiming               //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strTiming	PCO_Timing*	Pointer to a PCO_Timing structure filled with appropriate parameters. The wSize parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes

Descriptor dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.2.1 PCO_Timing structure

Name	Type	Description
wSize	WORD	Size of this structure
wTimeBaseDelay	WORD	Time base delay: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
wTimeBaseExposure	WORD	Time base exposure: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
ZZwAlignDummy1	WORD	Reserved
ZZdwDummy0[2]	DWORD	Reserved
dwDelayTable[16]	DWORD	Table with delay time values (for subsequent images)
ZZdwDummy1[114]	DWORD	Reserved
dwExposureTable[16]	DWORD	Table with exposure time values (for subsequent images)
ZZdwDummy2[112]	DWORD	Reserved
wTriggerMode	WORD	Trigger mode: • 0x0000 = [auto] • 0x0001 = [software trigger] • 0x0002 = [extern] • 0x0003 = [external exposure control] • 0x0004 = [external synchronized]
wForceTrigger	WORD	Force trigger (Auto reset flag!)
wCameraBusyStatus	WORD	Camera busy status: • 0x0000 = [idle] • 0x0001 = [busy]
wPowerDownMode	WORD	Power down mode : • 0x0000 = [auto] • 0x0001 = [user]
dwPowerDownTime	DWORD	Power down time 0 ms...49.7 d
wExpTrgSignal	WORD	Exposure trigger signal status
wFPSExposureMode	WORD	CMOS sensor FPS exposure mode
dwFPSExposureTime	DWORD	Resulting exposure time in FPS mode
wModulationMode	WORD	Mode for modulation: • 0x0000 = [modulation off] • 0x0001 = [modulation on]
wCameraSynchMode	WORD	Camera synchronization mode: • 0x0000 = [off] • 0x0001 = [master] • 0x0002 = [slave]
dwPeriodicalTime	DWORD	Periodical time for modulation
wTimeBasePeriodical	WORD	Time base for periodical time for modulation: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
ZZwAlignDummy3	WORD	Reserved
dwNumberOfExposures	DWORD	Number of exposures during modulation

lMonitorOffset	LONG	Monitor offset value in ns
strSignal[20]	PCO_Signal	Signal settings
wStatusFrameRate	WORD	Frame rate status
wFrameRateMode	WORD	Mode for frame rate
dwFrameRate	DWORD	Frame rate in mHz
dwFrameRateExposure	DWORD	Exposure time in ns
wTimingControlMode	WORD	Timing control mode: • 0x0000 = [exposure / delay] • 0x0001 = [fps]
wFastTimingMode	WORD	Fast timing mode: • 0x0000 = [off] • 0x0001 = [on]
ZZwDummy[24]	WORD	Reserved

2.6.3 PCO_GetCOCRunTime

Description:

This function can be used to calculate the current frame rate of the camera.
The returned values describe exactly how much time is required to take a single image. The resulting time is calculated inside the camera and depends on the settings of the timing and sensor parameters. To cover the full range of possible times it is splitted in two parts. Parameter `dwTime_s` gives the number of seconds and `dwTime_ns` gives the number of nano seconds in the range from 0 to 999999999 ns.
If external exposure is active, the returned value does describe the readout time only.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCOCRuntime (
    HANDLE ph,                          //in
    DWORD* dwTime_s,                    //out
    DWORD*  dwTime_ns                   //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`dwTime_s`	`DWORD*`	Pointer to a `DWORD` variable to get the COC runtime part in seconds
`dwTime_ns`	`DWORD*`	Pointer to a `DWORD` variable to get the COC runtime part in nanoseconds

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.4 PCO_GetDelayExposureTime

Description:

This function returns the current setting of the delay and exposure time values and the associated time base values.
Returned values are only valid if last timing command was **PCO_SetDelayExposureTime**.

Due to hardware limitations the returned values for the pco.edge, pco.1300 and pco.1400 are rounded values. To get exact timing values for the pco.edge please use function **PCO_GetImageTiming**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetDelayExposureTime (
    HANDLE ph,                          //in
    DWORD* dwDelay,                     //out
    DWORD*  dwExposure,                 //out
    WORD* wTimeBaseDelay,               //out
    WORD* wTimeBaseExposure             //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwDelay	DWORD*	Pointer to a DWORD variable to get the delay time
dwExposure	DWORD*	Pointer to a DWORD variable to get the exposure time
wTimeBaseDelay	WORD*	Pointer to a WORD variable to get the time base of the delay time: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
wTimeBaseExposure	WORD*	Pointer to a WORD variable to get the time base of the exposure time: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.5 PCO_SetDelayExposureTime

Description:

This function does set the delay and exposure time and the associated time base values.
When the ***Recording State*** of the camera is [run], camera timing is changed immediately (best possible), else new settings will be valid after a call of **PCO_ArmCamera**.
Restrictions for the parameter values are defined through the following values in the camera description **PCO_Description structure**: `dwMinDelayDESC`, `dwMaxDelayDESC`, `dwMinDelayStepDESC`, `dwMinExposDESC`, `dwMaxExposDESC`, `dwMinExposStepDESC`
Due to hardware limitations the input values cannot be set exactly for the pco.edge, pco.dimax and pco.1300 and therefore are changed to the next possible values in the camera. To retrieve the exact timing values, which are used in the pco.edge, please use function **PCO_GetImageTiming**.
Because frame rate and exposure time are also affected by the **PCO_SetFrameRate** command, it is strongly recommended to use either the **PCO_SetFrameRate** or the **PCO_SetDelayExposureTime** command.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetDelayExposureTime (
    HANDLE ph,                      //in
    DWORD dwDelay,                  //in
    DWORD  dwExposure,              //in
    WORD wTimeBaseDelay,            //in
    WORD wTimeBaseExposure          //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`dwDelay`	`DWORD`	`DWORD` variable to set the delay time
`dwExposure`	`DWORD`	`DWORD` variable to set the exposure time
`wTimeBaseDelay`	`WORD`	`WORD` variable to set the time base of the delay time: • `0x0000` = [ns] • `0x0001` = [µs] • `0x0002` = [ms]
`wTimeBaseExposure`	`WORD`	`WORD` variable to set the time base of the exposure time: • `0x0000` = [ns] • `0x0001` = [µs] • `0x0002` = [ms]

Parameter dependency:

`dwMinDelayDESC`, `dwMaxDelayDESC`, `dwMinDelayStepDESC`
`dwMinExposDESC`, `dwMaxExposDESC`, `dwMinExposStepDESC`
`dwMinDelayIRDESC`, `dwMaxDelayIRDESC`
`dwMinExposIRDESC`, `dwMaxExposIRDESC`

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.6 PCO_GetDelayExposureTimeTable

Description:

This function returns the current setting of the delay and exposure time table values and the associated time base values. Maximum size of each array is 16 DWORD entries. Returned values are only valid if the last timing command was **PCO_SetDelayExposureTimeTable**.
See **PCO_SetDelayExposureTimeTable** for a more detailed description of the delay / exposure time table usage.

Supported camera type:

All cameras

Descriptor dependency:

wTimeTableDESC

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetDelayExposureTimeTable (
    HANDLE ph,                          //in
    DWORD* dwDelay,                     //out
    DWORD* dwExposure,                  //out
    WORD* wTimeBaseDelay,               //out
    WORD* wTimeBaseExposure,            //out
    WORD wCount                         //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwDelay	DWORD*	Pointer to a DWORD array to get the delay times
dwExposure	DWORD*	Pointer to a DWORD array to get the exposure times
wTimeBaseDelay	WORD*	Pointer to a WORD variable to get the time base of the delay times
wTimeBaseExposure	WORD*	Pointer to a WORD variable to get the time base of the exposure times
wCount	WORD	WORD variable to define the number of DWORDs, which can be hold from the time table arrays

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.7 PCO_SetDelayExposureTimeTable

Description:

This function does set the delay and exposure time pairs in the time tables and the associated time base values. Maximum size of each table array is 16 DWORD entries. Delay / exposure time table operation is supported, if wTimeTableDESC in the camera description is set. After the camera is started it will take a series of consecutive images with delay and exposure times, as defined in the table. The first found exposure time entry with value zero does break the sequence and operation starts again from the beginning of the table. This results in a sequence of 1 to 16 images with different delay and exposure time settings. External or automatic image triggering is fully functional for every image in the sequence. If the user wants maximum speed (at CCDs overlapping exposure and read out is taken), [auto trigger] should be selected and the sequence should be controlled with the <acq enbl> input.
The commands **PCO_SetDelayExposureTime** and **PCO_SetDelayExposureTimeTable** can only be used alternatively. Using **PCO_SetDelayExposureTime** has the same effect as using the **PCO_SetDelayExposureTimeTable** command and setting all but the first delay / exposure entry to zero.

Restrictions for the parameter values are defined through the following values in the camera description **PCO_Description structure**: dwMinDelayDESC, dwMaxDelayDESC, dwMinDelayStepDESC, dwMinExposDESC, dwMaxExposDESC, dwMinExposStepDESC

Supported camera type:

All cameras

Descriptor dependency:

wTimeTableDESC

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetDelayExposureTimeTable (
    HANDLE ph,                          //in
    DWORD* dwDelay,                     //in
    DWORD* dwExposure,                  //in
    WORD wTimeBaseDelay,                //in
    WORD wTimeBaseExposure,             //in
    WORD wCount                         //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwDelay	DWORD*	Pointer to a DWORD array to set the delay times
dwExposure	DWORD*	Pointer to a DWORD array to set the exposure times
wTimeBaseDelay	WORD	WORD variable to set the time base of the delay times: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
wTimeBaseExposure	WORD	WORD variable to set the time base of the exposure times: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
wCount	WORD	WORD variable to set the number of DWORD entries in the tables

Parameter dependency:

dwMinDelayDESC, dwMaxDelayDESC, dwMinDelayStepDESC
dwMinExposDESC, dwMaxExposDESC, dwMinExposStepDESC

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.8 PCO_GetFrameRate

Description:

This function returns the current frame rate and exposure time settings of the camera.
Returned values are only valid if last timing command was **PCO_SetFrameRate.**

Supported camera type:

pco.edge, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: SETFRAMERATE_ENABLED`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFrameRate (
    HANDLE ph,                          //in
    WORD* wFrameRateStatus,             //out
    DWORD*  dwFrameRate,                //out
    DWORD* dwFrameRateExposure          //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wFrameRateStatus`	`WORD*`	Pointer to a `WORD` variable to get the frame rate status: • `0x0000` = Settings consistent, all conditions met • `0x0001` = Frame rate trimmed, frame rate was limited by readout time • `0x0002` = Frame rate trimmed, frame rate was limited by exposure time • `0x0004` = Exposure time trimmed, exposure time cut to frame time • `0x8000` = Return values `dwFrameRate` and `dwFrameRateExposure` are not yet validated. The values returned are the values which were passed with the most recent call of the **PCO_SetFrameRate** function
`dwFrameRate`	`DWORD*`	Pointer to a `DWORD` variable to get the frame rate in mHz
`dwFrameRateExposure`	`DWORD*`	Pointer to a `DWORD` variable to get the exposure time in ns

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.9 PCO_SetFrameRate

Description:

This function does directly set the frame rate and the exposure time of the camera. The frame rate is limited by the readout time and the exposure time.

$$frame\ rate \leq \frac{1}{t_{readout}} \qquad frame\ rate \leq \frac{1}{t_{expos}}$$

Please note that there are some overhead times, therefore the real values can differ slightly, e.g. the maximum frame rate will be a little bit less than 1 / exposure time. The in `wFramerateMode` parameter defines how the function works if any of the conditions are not met.

If ***Recording State*** is [run] (see **PCO_GetRecordingState**) the frame rate and exposure time of the camera is changed immediately. The input parameters are adapted according to the given rule in `wFramerateMode`. The function returns the currently configured frame rate and exposure time.
If ***Recording State*** of the camera is [stop] the given frame rate and exposure time is stored in the camera. The function does not adapt the input values for frame rate and exposure time. The next call of **PCO_ArmCamera** validates the input parameters together with other settings. Status of validation can be seen in the returned status `wFrameRateStatus`.

The following procedure is recommended:

- Set **PCO_SetRecordingState** to [stop]
- Set frame rate and exposure time using the **PCO_SetFrameRate** function
- Do other settings, before or after the **PCO_SetFrameRate** function
- Call the **PCO_ArmCamera** function in order to validate the settings
- Retrieve the currently set frame rate and exposure time using **PCO_GetFrameRate**

Because frame rate and exposure time are also affected by the **PCO_SetDelayExposureTime** command, it is strongly recommended to use either the **PCO_SetFrameRate** or the **PCO_SetDelayExposureTime** command.

Supported camera type:

pco.edge, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: SETFRAMERATE_ENABLED`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFrameRate (
    HANDLE ph,                          //in
    WORD*  wFrameRateStatus,            //out
    WORD  wFramerateMode,               //in
    DWORD* dwFrameRate,                 //in,out
    DWORD* dwFrameRateExposure          //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wFrameRateStatus	WORD*	Pointer to a WORD variable to get the frame rate status: • 0x0000= Settings consistent, all conditions met • 0x0001= Frame rate trimmed, frame rate was limited by readout time • 0x0002= Frame rate trimmed, frame rate was limited by exposure time • 0x0004= Exposure time trimmed, exposure time cut to frame time • 0x8000= Return values dwFrameRate and dwFrameRateExposure are not yet validated. In that case, the values returned are the values passed to the function
wFrameRateMode	WORD	WORD variable to set the frame rate mode: • 0x0000= Auto mode (camera decides which parameter will be trimmed) • 0x0001= Frame rate has priority, (exposure time will be trimmed) • 0x0002= Exposure time has priority, (frame rate will be trimmed) • 0x0003= Strict, function shall return with error if values are not possible
dwFrameRate	DWORD*	Pointer to a DWORD variable to set and get the frame rate • Frame rate in mHz (milliHertz), thus e.g. 1 kHz = 1000000
dwFrameRateExposure	DWORD*	Pointer to a DWORD variable to set and get the exposure time in ns

Parameter dependency:

dwMinDelayDESC, dwMaxDelayDESC, dwMinDelayStepDESC
dwMinExposDESC, dwMaxExposDESC, dwMinExposStepDESC

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.10 PCO_GetFPSExposureMode

Description:

This function returns the status of ***FPS exposure mode*** setting and according exposure time information.

Supported camera type:

pco.1200

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFPSExposureMode (
    HANDLE ph,                          //in
    WORD* wFPSExposureMode,             //out
    DWORD* dwFPSExposureTime            //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wFPSExposureMode	WORD*	Pointer to a WORD to get the FPS exposure mode: • 0x0000 = FPS exposure Mode [off] • 0x0001 = FPS exposure Mode [on]
dwFPSExposureTime	DWORD*	Pointer to a DWORD to get the valid exposure time in nanoseconds

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.11 PCO_SetFPSExposureMode

Description:

This function does set the image timing of the camera so that the maximum frame rate and the maximum exposure time for this frame rate is achieved. The maximum image frame rate (FPS = frames per second) depends on the pixel rate and the image area selection.

If ***FPS exposure mode*** is on other timing commands are ignored.

Supported camera type:

pco.1200

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFPSExposureMode (
    HANDLE ph,                          //in
    WORD wFPSExposureMode,              //in
    DWORD* dwFPSExposureTime            //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wFPSExposureMode	WORD	WORD to set the FPS-exposure mode: • 0x0000 = FPS exposure mode [off] • 0x0001 = FPS exposure mode [on], exposure time is set automatically to 1/FPSmax
dwFPSExposureTime	DWORD*	Pointer to a DWORD to get the exposure time in nanoseconds. The returned value is the exposure time that will be valid if FPS exposure mode is on.

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.12 PCO_GetTriggerMode

Description:

This function returns the current ***trigger mode setting*** of the camera.
Detailed description of trigger and acquire modes can be found in the respective camera manual.
In all trigger modes effective image exposure depends also on the acquire mode settings and the acquire signal input.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetTriggerMode (
    HANDLE ph,                          //in
    WORD* wTriggerMode                  //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wTriggerMode	WORD*	Pointer to a WORD variable to get the trigger mode: • 0x0000 = [auto sequence] • 0x0001 = [software trigger] • 0x0002 = [external exposure start & software trigger] • 0x0003 = [external exposure control] • 0x0004 = [external synchronized] • 0x0005 = [fast external exposure control] • 0x0006 = [external CDS control] • 0x0007 = [slow external exposure control] • 0x0102 = [external synchronized HDSDI]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.12.1 Explanation of available trigger modes

Function	Description
Auto Sequence	An exposure of a new image is started automatically best possible compared to the readout of an image and the current timing parameters. If a CCD is used and images are taken in a sequence, exposure and sensor readout are started simultaneously. Signals at the trigger input line are irrelevant
Soft(ware) ***Trigger***	An exposure is only started by a force trigger command
External Exposure Start & ***Soft***(ware) ***Trigger***	A delay / exposure sequence is started depending on the HW signal at the trigger input line or by a force trigger command
External Exposure Control	An exposure sequence is started depending on the HW signal at the trigger input line. The exposure time is defined by the pulse length of the HW signal. The delay and exposure time values defined by the set / request delay and exposure command are ineffective. In double image mode exposure time length of the first image is controlled through the HW signal, exposure time of the second image is given by the readout time of the first image
External Synchronized	The external synchronization signal feeds a phase locked loop (PLL) in the camera. The PLL adjusts itself to the phase of the external synchronization signal. The PLL can only lock to frequencies found in the `dwExtSyncFrequency` table in the **PCO_Description structure**. The exposure times are generated based on the frequency of the phase locked loop. Advantages of the PLL solution: • Reliability: in case of dropouts of the external synchronization signal, the synchronization is kept internally by the PLL signal with quite small deviation • Noise immunity: interference on the signal can be detected and discarded • Flexibility: cameras can even be set to different frame rates, as long as all frame rates are an integral multiple of the synchronization frequency
Fast External Exposure Control	Only available for ***pco.edge cameras*** in ***Rolling Shutter*** mode. An exposure is started depending on the HW signal at the trigger input line. The exposure time is defined by the pulse length of the HW signal. A second image can be triggered, while the first one is read out. This increases the frame rate, but leads to destructive images, if the trigger timing is not accurate: internal camera error correction is inactive in this mode
External CDS Control	Only available for ***pco.edge cameras*** in ***Global Shutter PIV mode***. The readout of the reset image can be triggered separate to reduce the trigger delay
Slow External Exposure Control	Only available for ***pco.edge cameras*** in ***Rolling Shutter mode***. An exposure is started depending on the HW signal at the trigger input line. The exposure time is defined by the pulse length of the HW signal. A second image can be triggered, while the first one is read out. This mode is optimized for longer exposure times. Set exposure time with **PCO_SetDelayExposureTime** to the expected duration of exposure. A smaller trigger pulse width leads to destructive images, while the image quality for a longer trigger pulse width is improved
External Synchronized HDSDI	Only available for ***pco.dimax*** in ***HDSDI output mode***. Ensure that HD/SDI output and image recording framerate are absolutely synchronously

2.6.13 PCO_SetTriggerMode

Description:

This function does set the trigger mode of the camera. For a short description of the available trigger modes, see table **Explanation of available trigger modes**.
Detailed description of trigger and acquire modes can be found in the respective camera manual.
In all trigger modes effective image exposure depends also on the acquire mode settings and acquire signal input.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetTriggerMode (
    HANDLE ph,                          //in
    WORD wTriggerMode                   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wTriggerMode	WORD	WORD variable to set the trigger mode: • 0x0000 = [auto sequence] • 0x0001 = [software trigger] • 0x0002 = [external exposure start & software trigger] • 0x0003 = [external exposure control] • 0x0004 = [external synchronized] • 0x0005 = [fast external exposure control] • 0x0006 = [external CDS control] • 0x0007 = [slow external exposure control] • 0x0102 = [external synchronized HDSDI]

Parameter dependency:

dwGeneralCapsDESC1: NO_EXTEXPCTRL,EXTERNAL_SYNC

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.14 PCO_ForceTrigger

Description:

This function does start an exposure, if the trigger mode is either [software trigger] or [extern exposure & software trigger]. In all other trigger modes the command has no effect.

To accept a force trigger command the camera must be in **PCO_SetRecordingState** [run] and camera ***busy state*** must be [not busy] (see **PCO_GetCameraBusyStatus**).

If a trigger command is not accepted by the camera ***it is lost*** and will not trigger future exposures.

- Due to response and processing times, e.g. caused by the interface and / or the operating system on the computer, the delay between command and current trigger may be several milliseconds
- A force trigger command will be processed independent of the selected acquire mode and independent of the state of the <acq enbl> input
- Triggers are not accumulated or buffered. A trigger will be accepted only if the camera is idle

Supported camera type:

All cameras

Parameter dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_ForceTrigger (
    HANDLE ph,                          //in
    WORD* wTriggered                    //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wTriggered	WORD*	Pointer to a WORD variable to get the trigger state: • 0x0000 = trigger command was unsuccessful because the camera is busy • 0x0001 = a new image exposure has been triggered

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.15 PCO_GetCameraBusyStatus

Description:

This function returns the current ***busy status*** of the camera. The ***busy status*** is according to the <busy> hardware signal at the camera output and can be checked before a **PCO_ForceTrigger** command to ensure that this command does start a new exposure. Due to response and processing times caused by the interface and / or the operating system, the delay between the delivered status and the current status may be several milliseconds. For exact synchronization to external events the hardware signal <busy> must be used.

Supported camera type:

pco.edge, pco.1600, pco.2000, pco.4000

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraBusyStatus (
    HANDLE ph,                          //in
    WORD* wCameraBusyState              //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wCameraBusyState	WORD*	Pointer to a WORD variable to get the camera busy status: • 0x0000 = Camera is [not busy], ready for a new trigger command • 0x0001 = Camera is [busy], not ready for a new trigger command

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.16 PCO_GetPowerDownMode

Description:

This function returns the current state of the ***power down mode***.
Detailed description of the power down mode can be found in the respective camera manual.

Supported camera type:

pco.1600, pco.2000, pco.4000

Descriptor dependency:

`wPowerDownModeDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetPowerDownMode (
    HANDLE ph,                          //in
    WORD* wPowerDownMode                //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wPowerDownMode`	`WORD*`	Pointer to a `WORD` variable to get the power down mode: • `0x0000` = [auto] • `0x0001` = [user]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.17 PCO_SetPowerDownMode

Description:

This function does set the ***power down mode*** of the camera. Descriptor flag `wPowerDownModeDESC` determines, if ***power down mode*** is available and if the camera can switch the sensor into power down mode for reduced ***dark current*** during long exposure times. By default ***power down mode*** [auto] is selected and the camera does select the threshhold time best suitable for the installed sensor. When power down mode is set to [user] the threshhold time can be set through function **PCO_SetUserPowerDownTime**.

Supported camera type:

pco.1600, pco.2000, pco.4000

Descriptor dependency:

`wPowerDownModeDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetPowerDownMode (
    HANDLE ph,                          //in
    WORD wPowerDownMode                 //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wPowerDownMode`	`WORD`	`WORD` variable to set the power down mode: • `0x0000` = [auto] • `0x0001` = [user]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.18 PCO_GetUserPowerDownTime

Description:

This function returns the current power down threshold time for power down mode [user].

Supported camera type:

pco.1600, pco.2000, pco.4000

Descriptor dependency:

`wPowerDownModeDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetPowerDownMode (
    HANDLE ph,                          //in
    DWORD* wPowerDownTime               //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wPowerDownTime`	`DWORD*`	Pointer to a DWORD variable to get the power down threshold time in ms

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.19 PCO_SetUserPowerDownTime

Description:

This function does set the ***power down threshold time*** for power down mode [user]. If the exposure time is greater than the selected threshold time the sensor is switched into a special low energy mode to reduce dark current effects. Because the ***wake-up of the camera*** from this special mode does need some time the value of the `wPowerDownTime` should not be less then 1000 ms, which is also the default value when power down mode is [auto].

Supported camera type:

pco.1600, pco.2000, pco.4000

Descriptor dependency:

`wPowerDownModeDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetUserPowerDownTime(
    HANDLE ph,                          //in
    DWORD wPowerDownTime                //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wPowerDownTime`	`DWORD`	`DWORD` variable to set the power down threshold time in ms

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.20 PCO_GetModulationMode

Description:

This function returns the current settings of the modulation mode and its corresponding parameters.

The modulation mode is an optional feature which is not available for all camera models. To determine if modulation mode is available first check if second descriptor is loadable through flag `ENHANCED_DESCRIPTOR_2` in `dwGeneralCapsDESC1` of **PCO_Description structure**. Then the presence of flag `MODULATE` in `dwModulateCapsDESC2` of **PCO_Description2 structure** must be checked.

Supported camera type:

Available for ***special*** versions of pco.1600, pco.2000 and pco.4000.

Descriptor dependency:

```
dwGeneralCapsDESC1: ENHANCED_DESCRIPTOR_2
dwModulateCapsDESC2: MODULATE
```

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetModulationMode (
    HANDLE ph,                          //in
    WORD *wModulationMode,              //out
    DWORD *dwPeriodicalTime,            //out
    WORD *wTimebasePeriodical,          //out
    DWORD *dwNumberOfExposures,         //out
    LONG *lMonitorOffset                //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
*wModulationMode	WORD	Pointer to a WORD variable to get the modulation mode: • 0x0000 = [modulation mode off] • 0x0001 = [modulation mode on]
*dwPeriodicalTime	DWORD	Pointer to a DWORD variable to get the periodical time. Periodical time as a multiple of the time base unit: The periodical time, delay and exposure time must meet the following condition : t_p - (t_e + t_d) > min per condition
*wTimebasePeriodical	WORD	Pointer to a WORD to get the time base of the periodical time: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
*dwNumberOfExposures	DWORD	Pointer to a DWORD variable to get the number of exposures for one frame
*lMonitorOffset	LONG	Pointer to a LONG variable to get the monitor offset value. The MonitorOffset [ns] controls the offset for the signal output line <status out> relative to the start of the exposure time. The possible range is limited in a very special way. See t_m in the timing diagram (**Modulation Mode Timing Diagram**). • The negative limit can be set from t_{std} to 0 • The negative limit can be enlarged by adding a delay • The maximum negative monitor offset is limited to 20 µs. No matter how long the delay will be set • The positive limit can be enlarged by longer exposure times than the minimum exposure time • The maximum positive monitor offset is limited to 20 µs; no matter how long the exposure will be set

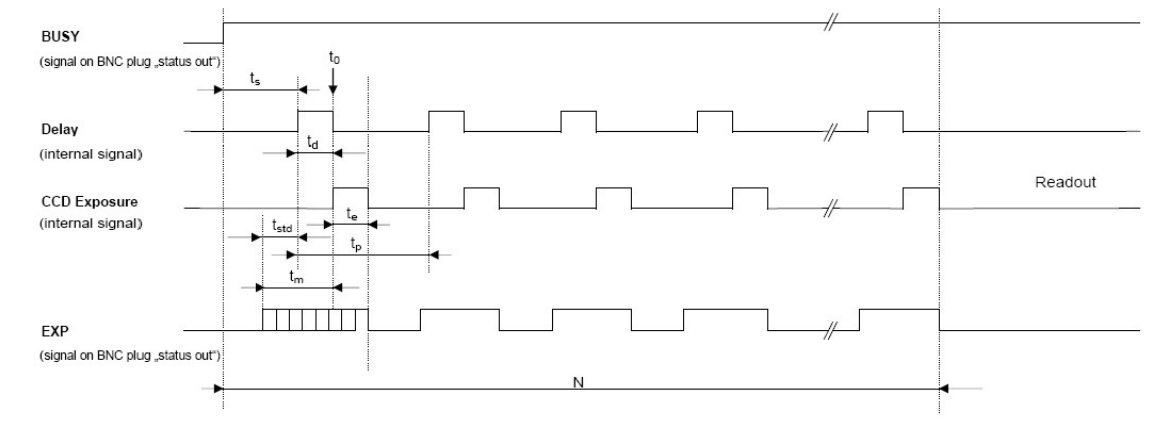
Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.20.1 Modulation Mode Timing Diagram



t_s: start time: after the initial trigger a starting sequence is worked off, then the exposures are acquired and accumulated; 7.5 µs (pco.1600, pco.2000), 8 µs (pco.4000)

t_{std}: start time delay for monitor signal; 2 µs (pco.1600), 3 µs (pco.2000, pco.4000)

t_0: exposure start

	values set by user	pco.1600	pco.2000	pco.4000	
t_d	delay time (CCD)	0…100 ms	0 … 100 ms	0 … 100 ms	500 ns steps
t_e	exposure time (CCD)	500 ns…1 ms	500 ns … 1 ms	500 ns…1 ms	500 ns steps
t_p	periodical time (only for trigger “Auto Seq.”)	20 µs…100 ms	25 µs … 100ms	50 µs…100 ms	500 ns steps
t_m	monitor offset	-15 µs…+20 µs	-20 µs…+20 µs	-20 µs…+20 µs	500 ns steps
N	number of exposures	1…500.000	1…100.000	1…100.000	steps of 1

Restrictive conditions:

- for periodical time: $t_p - (t_d + t_e) \geq 10$ µs (pco.1600, pco.2000)
  $t_p - (t_d + t_e) \geq 25$ µs (pco.4000)
- for monitor offset: $-(t_d + t_{std}) \leq t_m \leq (t_e - 0.5$ µs)
  A monitor offset of ‘0 µs’ causes a rising of the monitor output right at exposure start:
  $t_m = 0$ µs ° t0

Considerations for good image quality:

- Only runtimes of less then 10 seconds are desirable
- Totalized exposure time ($N * t_e$) should be limited to 100ms
- Keep exposure time as short as possible
- Use extensive CCD cooling, if possible

2.6.21 PCO_SetModulationMode

Description:

This function does set the modulation mode and its corresponding parameters.
The modulation mode is an optional feature which is not available for all camera models. To determine if modulation mode is available first check if second descriptor is loadable through flag `ENHANCED_DESCRIPTOR_2` in `dwGeneralCapsDESC1` of **PCO_Description structure**. Then the presence of flag `MODULATE` in `dwModulateCapsDESC2` of **PCO_Description2 structure** must be checked.
Restrictions for the parameter values are defined through the timing values in the camera description **PCO_Description2 structure .**

Supported camera type:

Available for ***special*** versions of pco.1600, pco.2000 and pco.4000.

Descriptor dependency:

```
dwGeneralCapsDESC1: ENHANCED_DESCRIPTOR_2
dwModulateCapsDESC2: MODULATE
```

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetModulationMode (
    HANDLE ph,                          //in
    WORD wModulationMode,               //in
    DWORD dwPeriodicalTime,             //in
    WORD wTimebasePeriodical,           //in
    DWORD dwNumberOfExposures,          //in
    LONG lMonitorOffset                 //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wModulationMode`	`WORD`	`WORD` variable to set the modulation mode: • `0x0000` = [modulation mode off] • `0x0001` = [modulation mode on]
`dwPeriodicalTime`	`DWORD`	`DWORD` variable to set the periodical time in time base unit The periodical time, delay and exposure time must meet the following condition : t_p - (t_e + t_d) > 'Min Per Condition'
`wTimebasePeriodical`	`WORD`	`WORD` variable to hold the time base of the periodical time: • `0x0000` = [ns] • `0x0001` = [µs] • `0x0002` = [ms]
`dwNumberOfExposures`	`DWORD`	`DWORD` variable to set the number of exposures for one frame
`lMonitorOffset`	`LONG`	`LONG` variable to set the monitor offset value in ns. The MonitorOffset controls the offset for the signal output line <status out> relative to the start of the exposure time. • the range is limited through the timing values • the maximum negative monitor offset is limited to -20 µs • the maximum positive monitor offset is limited to 20 µs See **Modulation Mode Timing Diagram**

Parameter dependency:

`dwMinPeriodicalTimeDESC2,dwMaxPeriodicalTimeDESC2`
`dwMinPeriodicalConditionDESC2,dwMaxNumberOfExposuresDESC2`
`lMinMonitorSignalOffsetDESC2,dwMaxMonitorSignalOffsetDESC2`
`dwMinPeriodicalStepDESC2,dwStartTimeDelayDESC2,dwMinMonitorStepDESC2`
`dwMinDelayModDESC2,dwMaxDelayModDESC2,dwMinDelayStepModDESC2`
`dwMinExposureModDESC2,dwMaxExposureModDESC2,dwMinExposureStepModDESC2`

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.22 PCO_GetHWIOSignalCount

Description:

This function returns the number of ***hardware I/O signal lines***, which are available at the camera. Function **PCO_GetHWIOSignalDescriptor** must be called to get a description of the available options for a distinct I/O signal line. With this information the current settings can be changed with **PCO_SetHWIOSignal** and queried with **PCO_GetHWIOSignal.**

Supported camera type:

pco.edge, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: HW_IO_SIGNAL_DESCRIPTOR`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetHWIOSignalCount (
    HANDLE ph,                          //in
    WORD* wNumSignals                   //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wNumSignals`	`WORD*`	Pointer to a WORD variable to get the number of available hardware I/O signal lines

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.23 PCO_GetHWIOSignalDescriptor

Description:

This function does retrieve the description of a distinct hardware I/O signal line. The number of available hardware I/O signal lines can be queried with **PCO_GetHWIOSignalCount**.
Only predefined signals can be routed to distinct signal lines. With the values returned in the **PCO_Single_Signal_Desc structure** available options for each I/O signal line can be determined. With these options in mind the **PCO Signal structure** for the call to **PCO_SetHWIOSignal** can be prepared.
Optional parameters can be set for some of the predefined signals, allowing better control of the signal. Optional Parameters are available, when one of the SIGNAL_DEF_PARAM bits is set in the **Signal Definitions Bits**. Descriptions for the additional parameters can be found in the appropriate listing, which is defined through the **Signal Functionality** returned from the **PCO_GetHWIOSignal** function.
Additional information about input / output lines can be found in the respective camera manual.

Supported camera type:

pco.edge, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: HW_IO_SIGNAL_DESCRIPTOR`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetHWIOSignalDescriptor (
    HANDLE ph,                              //in
    WORD wSignalNum,                        //in
    PCO_Single_Signal_Desc* pstrSignal      //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSignalNum	WORD	WORD variable to select the signal line to query. This parameter must be in the range of available hardware I/O signal lines
pstrSignal	PCO_Single_Signal_Desc*	Pointer to a **PCO_Single_Signal_Desc structure** to get the capabilities of the hardware I/O signal. On input the wSize parameter of this structure must be filled with the correct structure size in bytes

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.23.1 PCO_Single_Signal_Desc structure

Name	Type	Description
wSize	WORD	Size of this structure
ZZwAlignDummy1	WORD	Reserved
strSignalName	char[4][25]	List of available signals, which can be routed to the selected I/O signal line. Each valid list entry is an ASCII string with up to 25 characters. If the list entry is an empty string this entry is not valid. All valid entries can be selected through the wSelected parameter of the **PCO Signal structure**. At least the first list entry is always valid.
wSignalDefinitions	WORD	Flags for signal definitions see table **Signal Definitions Bits**
wSignalTypes	WORD	Flags for electrical I/O Standard availability, see **Signal I/O Standard Bits**
wSignalPolarity	WORD	Flags for signal polarity availability, see **Signal Polarity Bits**
wSignalFilter	WORD	Flags for filter options availability, see **Signal Filter Option Bits.** Time t describes the minimum pulse width of input signal

2.6.23.2 Signal Definitions Bits

Name	Value	Description
SIGNAL_DEF_ENABLE	0x00000001	I/O signal line can be enabled or disabled
SIGNAL_DEF_OUTPUT	0x00000002	I/O signal line is a status output line
	0x00000004	Reserved
	0x00000008	Reserved
SIGNAL_DEF_PARAM1	0x00000010	The signal for list entry [0] does need an additional parameter when selected for I/O signal line
SIGNAL_DEF_PARAM2	0x00000020	The signal for list entry [1] does need an additional parameter when selected for I/O signal line
SIGNAL_DEF_PARAM3	0x00000040	The signal for list entry [2] does need an additional parameter when selected for I/O signal
SIGNAL_DEF_PARAM4	0x00000080	The signal for list entry [3] does need an additional parameter when selected for I/O signal line

2.6.23.3 Signal I/O Standard Bits

Name	Value	Description
SIGNAL_TYPE_TTL	0x00000001	I/O signal line can be used as a standard TTL signal
SIGNAL_TYPE_HL_SIG	0x00000002	I/O signal line can be used as a HighLevel signal • low level: 0 V – 5 V • high level: 10 V - VCC (max. = 56 V)
SIGNAL_TYPE_CONTACT	0x00000004	I/O signal line can be used as input for a pushbutton
SIGNAL_TYPE_RS485	0x00000008	I/O signal line can be used as a standard RS485 signal

2.6.23.4 Signal Polarity Bits

Name	Value	Description
SIGNAL_POL_HIGH	0x00000001	I/O signal line can be sensed for high level
SIGNAL_POL_LOW	0x00000002	I/O signal line can be sensed for low level
SIGNAL_POL_RISE	0x00000004	I/O signal line can be sensed for rising edges
SIGNAL_POL_FALL	0x00000008	I/O signal line can be sensed for falling edges

2.6.23.5 Signal Filter Option Bits

Name	Value	Description
SIGNAL_FILTER_OFF	0x00000001	Filter can be switched off (t > ~65 ns)
SIGNAL_FILTER_MED	0x00000002	Filter can be switched to medium (t > ~1 us)
SIGNAL_FILTER_HIGH	0x00000004	Filter can be switched to high (t > ~100 ms)

2.6.23.6 Signal Functionality

Name	Value	Description
NONE	0x00000000	Signal is undefined
TRIGGER_INPUT	0x00000001	Signal is input for trigger
ACQUIRE_INPUT	0x00000002	Signal is input for acquire
BUSY_OUTPUT	0x00000003	Signal is output for camera busy state
EXPOSURE_OUTPUT	0x00000004	Signal is output for camera exposing state
READOUT_OUTPUT	0x00000005	Signal is output for camera readout state
SYNCH_INPUT	0x00000006	Signal is input for synchronization
EXPOSURE_OUTPUT_EXT	0x00000007	Signal is output for extended camera exposing state. Suitable for a pco.edge in setup mode Rolling Shutter. The additional parameter defines enhanced signal timing see table **Extended Signal Timing Rolling Shutter**

2.6.23.7 Extended Signal Timing Rolling Shutter

Name	Value	Description
HW_IO_SIGNAL_TIMING_ EXPOSURE_RS_FIRSTLINE	0x00000001	Exposure time of the first rolling shutter line ($t_{firstline}$)
HW_IO_SIGNAL_TIMING_ EXPOSURE_RS_GLOBAL	0x00000002	Core time while all lines are exposing (t_{global})
HW_IO_SIGNAL_TIMING_ EXPOSURE_RS_LASTLINE	0x00000003	Exposure time of the last rolling shutter line ($t_{lastline}$)
HW_IO_SIGNAL_TIMING_ EXPOSURE_RS_ALLLINES	0x00000004	Complete exposure time from the start of first until the end of the last rolling shutter line ($t_{alllines}$)

See respective camera manual for detailed description.

2.6.24 PCO_GetHWIOSignal

Description:

This function returns the current settings of a distinct hardware input/output (IO) signal line. To select the setting for a ***signal line*** use **PCO_SetHWIOSignal**.

Supported camera type:

pco.edge, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: HW_IO_SIGNAL_DESCRIPTOR`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetHWIOSignal (
    HANDLE ph,                          //in
    WORD wSignalNum,                    //out
    PCO_Signal*  pstrSignal             //in,out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wSignalNum`	`WORD`	Select the signal to query. This parameter must be in the range of available hardware I/O signals
`pstrSignal`	`PCO_Signal*`	Pointer to a **PCO Signal structure** to get the settings of the hardware I/O signal. On input the `wSize` parameter of this structure must be filled with the correct structure size in bytes

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.25 PCO_SetHWIOSignal

Description:

This function does select the settings of a distinct hardware IO signal line. To query the settings of a distinct signal line please use **PCO_GetHWIOSignal**.
To determine the available options for each signal line use **PCO_GetHWIOSignalDescriptor**.

Supported camera type:

pco.edge, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: HW_IO_SIGNAL_DESCRIPTOR`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetHWIOSignal (
    HANDLE ph,                          //in
    WORD wSignalNum,                    //in
    PCO_Signal* pstrSignal              //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSignalNum	WORD	Selects the signal to set. This parameter must be in the range of available hardware I/O signals
pstrSignal	PCO_Signal*	Pointer to a **PCO Signal structure** filled with appropriate parameters

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.25.1 PCO Signal structure

Name	Type	Description
wSize	WORD	Size of this structure
wSignalNum	WORD	Index of the signal
wEnabled	WORD	Enable state of the signal: • 0x0000 = Signal is off • 0x0001 = Signal is active
wType	WORD	Electrical I/O Standard • 0x0001 = TTL • 0x0002 = High Level TTL • 0x0004 = Contact Mode • 0x0008 = RS485 differential • 0x0080 = Two pin differential TTL PinA=TTL, PinB=GND
wPolarity	WORD	Signal polarity: • 0x0001 = High level • 0x0002 = Low Level • 0x0004 = Rising edge • 0x0008 = Falling edge
wFilter	WORD	Filter option: • 0x0001 = No signal filtering (t > ~65 ns) • 0x0002 = MediumFilter (t > ~1 µs) • 0x0004 = High Filter (t > ~100ms)
wSelected	WORD	Selected signal for this signal line. Choose one out of the available signals defined in the **PCO_Single_Signal_Desc structure.** e.g. Status Busy or Status Exposure
ZzwReserved	WORD	Reserved
dwParameter[4]	DWORD	Additional parameter if the selected signal requires one (when the SIGNAL_DEF_PARAM Flag is set for the selected signal in the **PCO_Single_Signal_Desc structure**). The additional parameter extends the options for a distinct signal functionality
dwSignalFunctionality[4]	DWORD	Functionality of the selected signal. Information is only valid when structure is readout. Should be set to 0 on input. • 0x0000 = None • 0x0001 = Trigger input • 0x0002 = Acquire input • 0x0003 = Busy output • 0x0004 = Exposure output • 0x0005 = Readout output • 0x0006 = Synchronization input • 0x0007 = Exposure output Rolling Shutter; requires an additional parameter to define the type of information. See table **Extended Signal Timing Rolling Shutter**
ZzdwReserved[3]	DWORD	Reserved

2.6.26 PCO_GetImageTiming

Description:

This function returns the current ***image timing*** in ***nanosecond resolution*** and ***additional trigger system information***.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

The ***maximum real trigger delay*** in ns can be calculated as:
Real Trigger Delay = `TriggerSystemDelay_ns` + `TriggerSystemJitter_ns` + `TriggerDelay_ns` + `TriggerDelay_s*1000000000`

The ***minimum real trigger delay*** in ns can be calculated as:
Real Trigger Delay = `TriggerSystemDelay_ns` + `0` + `TriggerDelay_ns` + `TriggerDelay_s*1000000000`

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetImageTiming (
    HANDLE ph,                          //in
    PCO_ImageTiming* pstrImageTiming  //in,out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`PCO_ImageTiming*`	`pstrImageTiming`	Pointer to a **PCO ImageTiming structure** to get the timing of the current camera settings. On input the `wSize` parameter of this structure must be filled with the correct structure size in bytes

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.26.1 PCO ImageTiming structure

Name	Type	Description
wSize	WORD	Size of this structure
wDummy	WORD	Reserved
FrameTime_ns	DWORD	Nanoseconds part of the time to expose and readout a single image
FrameTime_s	DWORD	Seconds part of the time to expose and readout a single image
ExposureTime_ns	DWORD	Nanoseconds part of the exposure time
ExposureTime_s	DWORD	Seconds part of the exposure time
TriggerSystemDelay_ns	DWORD	System internal trigger delay in ns. This is the time until a exposure is started after a trigger is recognized, when delay time is set to zero.
TriggerSystemJitter_ns	DWORD	Maximum possible trigger jitter time in ns
TriggerDelay_ns	DWORD	Nanoseconds part of the trigger delay, which is set through one of the timing functions
TriggerDelay_s	DWORD	Seconds part of the trigger delay, which is set through one of the timing functions
ZZdwDummy	DWORD	Reserved

2.6.27 PCO_GetCameraSynchMode

Description:

This function returns the current camera synchronization mode. This ***Master / Slave synchronization*** mode is especially for multi-camera use.
When cameras are cascaded through an external synchronization line at least one camera must be in master mode. This camera determines the timing of all other cameras in the line and therefore is the only camera, which accepts timing (exposure, delay,…) settings. All cameras which are set to slave mode synchronize its exposures with the master camera.

To get reliable results the follow start / stop rules should be observed:
In order to get the same number of images in the recorder, please set all slaves to ***Recording State*** [run] ((see **PCO_GetRecordingState**), before the master is set. When setting ***Recording State*** [stop] (see **PCO_GetRecordingState**), please stop the master as first. Please observe the start / stop sequence.

Supported camera type:

pco.dimax

Descriptor dependency:

`dwGeneralCaps3: CAMERA_SYNC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraSynchMode (
    HANDLE ph,                          //in
    WORD* wCameraSynchMode              //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wCameraSynchMode	WORD*	Pointer to a WORD variable to get the camera synchronization mode: • 0x0000 = [off] • 0x0001 = [master] • 0x0002 = [slave]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.28 PCO_SetCameraSynchMode

Description:

This function does set the current camera synchronization mode. This ***Master / Slave synchronization*** mode is especially for multi-camera use.
When cameras are cascaded through an external synchronization line at least one camera must be in master mode. This camera determines the timing of all other cameras in the line and therefore is the only camera, which accepts timing (exposure, delay,…) settings. All cameras which are set to slave mode synchronize its exposures with the master camera.
Synchronization must be set to off for all cameras, which are not connected to a cascaded line.

To get reliable results the follow start / stop rules should be observed:
In order to get the same number of images in the recorder, please set all slaves to ***Recording State*** [run] ((see **PCO_GetRecordingState**), before the master is set. When setting ***Recording State*** [stop] (see **PCO_GetRecordingState**), please stop the master as first. Please observe the start / stop sequence.

Supported camera type:

pco.dimax

Descriptor dependency:

`dwGeneralCaps3: CAMERA_SYNC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetCameraSynchMode (
    HANDLE ph,                          //in
    WORD wCameraSynchMode               //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wCameraSynchMode`	`WORD`	`WORD` variable to set the camera synchronization mode: • `0x0000` = [off] • `0x0001` = [master] • `0x0002` = [slave]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.29 PCO_GetExpTrigSignalStatus

Description:

This function returns the current status of the ***<exp trig>*** input line.
See respective camera manual for more information about hardware signals.

Due to response and processing times the delay between the software delivered status and the current status may be several milliseconds. For instance caused by interface and / or operating system.

Supported camera type:

pco.pixelfly usb, pco.ultraviolet, pco.1300, pco.1400

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetExpTrigSignalStatus (
    HANDLE ph,                          //in
    WORD* wExpTrgSignal                 //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wExpTrgSignal	WORD*	Pointer to a WORD variable to get the current state of the <exp trig> input line: • 0x0000 = [off] • 0x0001 = [on]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.30 PCO_GetFastTimingMode

Description:

This function returns the current fast timing mode setting of the camera.

If the camera is set to fast timing mode image timing is changed. The interframing time between two images is reduced to to 3.5 µs despite the normal value of about 75 µs. While running in fast timing mode image quality is reduced, which might be acceptable for special applications like PIV.

Supported camera type:

pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: FAST_TIMING`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFastTimingMode (
    HANDLE ph,                          //in
    WORD* wFastTimingMode               //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wFastTimingMode`	`WORD*`	Pointer to a `WORD` variable to get the camera fast timing mode: • `0x0000` = [off] • `0x0001` = [on]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.6.31 PCO_SetFastTimingMode

Description:

This function does set the fast timing mode of the camera.

If the camera is set to fast timing mode image timing is changed. The interframing time between two images is reduced to to 3.5 µs despite the normal value of about 75 µs. While running in fast timing mode image quality is reduced, which might be acceptable for special applications like PIV.

Supported camera type:

pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: FAST_TIMING`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFastTimingMode (
    HANDLE ph,                          //in
    WORD wFastTimingMode                //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wFastTimingMode`	`WORD`	`WORD` variable to set the camera fast timing mode: • `0x0000` = [off] • `0x0001` = [on]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7 RECORDING CONTROL

This function group can be used to control the recording state and also to get or set parameters for enhanced recording control.
During recording images can be grabbed with any of the image readout functions of the SDK. Function **PCO_AddBufferEx** allows overlapped transfers, while **PCO_GetImageEx** is a synchronous call.
Cameras ***without*** internal memory transfer the ***latest aquired image***.
Cameras ***with*** internal memory store all images to the ***camera internal memory (CamRam)***.
If storage mode is [recorder], the last aquired image is transferred.
If storage mode is set to [FIFO buffer mode], the images are transferred in the order in which they have been written into the FIFO buffer.
The image transfer does not affect CamRam recording. CamRam recording does run independently without the need of application intervention. The possible frame rate of the CamRam recording is completely different to the interface transfer frame rate. The camera frame rate can be determined by calling the **PCO_GetCOCRunTime**.

2.7.1 PCO_GetRecordingStruct

Description:

Recording control information is queried from the camera and the variables of the **PCO_Recording structure** are filled with this information. This function is a combined version of the functions, which request information about the recording control related parameter. For a detailed description of each parameter see the functions in this chapter.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetRecordingStruct (
    HANDLE ph,                          //in
    PCO_Recording* strRecording         //in,out
);
```

Parameters:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strRecording	PCO_Recording*	Pointer to a **PCO_Recording structure**. • On input the wSize parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes • On output the structure is filled with the requested information from the camera

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.2 PCO_SetRecordingStruct

Description:

This function does set the complete set of recording settings at once. For the sake of clarity it is better to use the functions which change distinct parameter despite changing all settings at once. An invalid value for one of the parameter will result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetRecordingStruct (
    HANDLE ph,                          //in
    PCO_Recording* strRecording         //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strRecording	PCO_Recording*	Pointer to a **PCO_Recording structure** filled with appropriate parameters. The wSize parameter of this structure and all nested structures must be filled with the correct structure size in bytes

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.2.1 PCO_Recording structure

Name	Type	Description
wSize	WORD	Size of this structure
wStorageMode	WORD	Storage mode: • 0x0000 = [recorder] • 0x0001 = [FIFO buffer]
wRecSubmode	WORD	Recorder sub mode: • 0x0000 = [sequence] • 0x0001 = [ring buffer]
wRecState	WORD	Recording state: • 0x0000 = [off] • 0x0001 = [on]
wAcquMode	WORD	Aquire mode: • 0x0000 = [internal auto] • 0x0001 = [external] • 0x0002 = [external frame] • 0x0003 = reserved • 0x0004 = [external sequence]
wAcquEnableStatus	WORD	Acquire status: • 0x0000 = [disabled] • 0x0001 = [enabled]
ucDay	BYTE	Timestamp data week day (1-31)
ucMonth	BYTE	Timestamp data month (1-12)
wYear	WORD	Timestamp data year
wHour	WORD	Timestamp data hour (0-23)
ucMin	BYTE	Timestamp data minutes (0-59)
ucSec	BYTE	Timestamp data seconds (0-59)
wTimeStampMode	WORD	Timestamp mode: • 0x0000 = [no stamp] • 0x0001 = [BCD coded] • 0x0002 = [BCD coded + ASCII] • 0x0003 = [ASCII]
wRecordStopEventMode	WORD	Record stop event mode: • 0x0000 = [off] • 0x0001 = [on]
dwRecordStopDelayImages	DWORD	Number of images which should pass by until stop event is executed
wMetaDataMode	WORD	Meta data mode: • 0x0000= [off] • 0x0001= [enabled]
wMetaDataSize	WORD	Size of Meta Data in number of pixels
wMetaDataVersion	WORD	Version info for Meta Data
ZZwDummy1	WORD	Reserved
dwAcquModeExNumberImages	DWORD	Number of images in one acquire sequence; Valid when in acquire mode [external sequence]
dwAcquModeExReserved[4]	DWORD	Reserved
ZZwDummy[22]	WORD	Reserved

2.7.3 PCO_GetRecordingState

Description:

This function returns the current ***Recording State*** of the camera.

The ***Recording State*** can change from [run] to [stop] through:

- Call to function **PCO_SetRecordingState** [stop]
- **PCO_SetStorageMode** is [recorder], **PCO_SetRecorderSubmode** is [sequence] and active segment is full
- **PCO_SetStorageMode** is [recorder], **PCO_SetRecorderSubmode** is [ring buffer], **PCO_SetRecordStopEvent** is [on] and the given number of images is recorded.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetRecordingState (
    HANDLE ph,                          //in
    WORD* wRecState                     //out
);
```

Parameters:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wRecState	WORD*	Pointer to a WORD variable to get the current recording state: • 0x0000 = camera is stopped, recording state [stop] • 0x0001 = camera is running, recording state [run]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.4 PCO_SetRecordingState

Description:

This function sets the ***Recording State*** and waits until the state is valid. If the requested state is already set the function will return a warning. If the state cannot be set within one second (+ current frametime for [stop]), the function will return an error.

The ***Recording State*** controls the run state of the camera. If the ***Recording State*** is [run], sensor exposure and readout sequences are started depending on current camera settings (trigger mode, acquire mode, external signals…).
The ***Recording State*** has the highest priority compared to functions like <acq enbl> or exposure trigger.
When the ***Recording State*** is set to [stop], sensor exposure and readout sequences are stopped. If the camera is currently in [sensor_readout] state, this readout is finished, before camera run state is changed to [sensor_idle]. If the camera is currently in [sensor_exposing] state, the exposure is cancelled and camera run state is changed immediately to [sensor_idle].
In run state [sensor_idle] the camera is running a special idle mode to prevent dark charge accumulation.

If any camera parameter was ***changed***: before setting the ***Recording State*** to [run], the function **PCO_ArmCamera** must be called. This is to ensure that all settings were correctly and are accepted by the camera.
If a successful ***Recording State*** [run] command is sent and recording is started, the images from a previous record to the active segment are lost.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetRecordingState (
    HANDLE ph,                          //in
    WORD wRecState                      //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wRecState	WORD	WORD variable to set the active recording state: • 0x0000 = stop camera and wait until recording state = [stop] • 0x0001 = start camera and wait until recording state = [run]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.5 PCO_GetStorageMode

Description:

This function returns the current storage mode of the camera.
Storage mode is either [recorder] or [FIFO buffer].

Supported camera type:

pco.dimax, pco.pixelfly usb, pco.ultraviolet, pco.1200, pco.1300, pco.1400, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetStorageMode (
    HANDLE ph,                          //in
    WORD* wStorageMode                  //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wStorageMode`	`WORD*`	Pointer to a `WORD` to get the current storage mode: • `0x0000` = [recorder] mode • `0x0001` = [FIFO buffer] mode

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.6 PCO_SetStorageMode

Description:

This function does set the storage mode of the camera.
Storage mode can be set to either [recorder] or [FIFO buffer] mode.

StorageMode [recorder]	StorageMode [FIFO buffer]
Images are recorded and stored in the current selected segment of the camera internal memory (CamRAM)	Camera internal memory (CamRAM) is used as huge FIFO buffer to bypass short bottlenecks in data transmission
If **PCO_SetRecorderSubmode** is [sequence] recording is stopped, when the last buffer in the segment is reached	If buffer overflows, the oldest images are overwritten
If **PCO_SetRecorderSubmode** is [ring buffer] the oldest image is overwritten, when the segment is full	While **PCO_SetRecordingState** is [run] the oldest image is transferred on an image request
While **PCO_SetRecordingState** is [run] the most recent image is transferred on an image request	When **PCO_SetRecordingState** is [stop] the recorded and not already transferred images can be read from the camera memory using an image number index. Image number 1 is always the oldest image in the segment
When **PCO_SetRecordingState** is [stop] the recorded images can be readout from the camera memory using an image number index. Image number 1 is always the oldest image in the segment	No **PCO_SetRecorderSubmode** available

Supported camera type:

pco.dimax, pco.pixelfly usb, pco.ultraviolet, pco.1200, pco.1300, pco.1400, pco.1600, pco.2000, pco.4000

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetStorageMode (
    HANDLE ph,                          //in
    WORD wStorageMode                   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wStorageMode	WORD	WORD variable to set the current storage mode: • 0x0000 = [recorder] mode • 0x0001 = [FIFO buffer] mode

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.7 PCO_GetRecorderSubmode

Description:

This function returns the current recorder submode of the camera. Recorder submode is only available if the storage mode is set to [recorder].
Recorder submode is either [sequence] or [ring buffer].

Supported camera type:

pco.dimax, pco.pixelfly usb, pco.ultraviolet, pco.1200, pco.1300, pco.1400, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetRecorderSubmode (
    HANDLE ph,                          //in
    WORD* wRecSubmode                   //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wRecSubmode	WORD*	Pointer to a WORD to get the current recorder submode: • 0x0000 = [sequence] • 0x0001 = [ring buffer]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.8 PCO_SetRecorderSubmode

Description:

This function sets the recorder submode of the camera. Recorder submode is only available if **PCO_SetStorageMode** is set to [recorder].
Recorder submode can be set to [sequence] or [ring buffer].

RecorderSubmode: [sequence]	RecorderSubmode: [ring buffer]
Recording is stopped, when the last buffer in the segment is reached	Camera records continuously into ring buffer
No images are overwritten	The oldest images are overwritten, if a buffer overflows occures due to long recording times
Recording can be stopped by software	Recording must be stopped by software or with an stop event

Supported camera type:

pco.dimax, pco.pixelfly usb, pco.ultraviolet, pco.1200, pco.1300, pco.1400, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetRecorderSubmode (
    HANDLE ph,                          //in
    WORD wRecSubmode                    //in
);
```

Parameters:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wRecSubmode	WORD	WORD variable to set the active recorder sub mode: • 0x0000 = [sequence] • 0x0001 = [ring buffer]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.9 PCO_GetAcquireMode

Description:

This function returns the current acquire mode of the camera.
Acquire mode can be either [auto], [external] or [external modulate].

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: NO_ACQUIREMODE`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetAcquireMode (
    HANDLE ph,                          //in
    WORD* wAcquMode                     //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wAcquMode	WORD*	Pointer to a WORD variable to get the current acquire mode: • 0x0000 = [auto] All images will be acquired and stored. The external <acq enbl> input is ignored. • 0x0001 = [external] The external control input <acq enbl> is a static enable signal for image acquisition. Depending on the I/O configuration a high or low level at the external <acq enbl> input does set the acquire enable state to TRUE. If the acquire enable state is TRUE exposure triggers are accepted and images are acquired. If the acquire enable state is FALSE, all exposure triggers are ignored and no images will be acquired and stored. • 0x0002 = [external modulate] The external control input <acq enbl> is a dynamic frame start signal. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will start a single frame in modulation mode.

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.10 PCO_SetAcquireMode

Description:

This function sets the acquire mode of the camera.
Acquire mode can be either [auto], [external] or [external modulate].

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: NO_ACQUIREMODE`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetAcquireMode (
    HANDLE ph,                          //in
    WORD wAcquMode                      //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wAcquMode	WORD	WORD variable to set the acquire mode: • 0x0000 = [auto] All images will be acquired and stored. The external <acq enbl> input is ignored • 0x0001 = [external] The external control input <acq enbl> is a static enable signal for image acquisition. Depending on the I/O configuration a high or low level at the external <acq enbl> input does set the acquire enable state to TRUE. If the acquire enable state is TRUE exposure triggers are accepted and images are acquired. If the acquire enable state is FALSE, all exposure triggers are ignored and no images will be acquired and stored • 0x0002 = [external modulate] The external control input <acq enbl> is a dynamic frame start signal. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will start a single frame in modulation mode

Parameter dependency:

`dwGeneralCapsDESC1: ENHANCED_DESCRIPTOR_2`
`dwModulateCapsDESC2: MODULATE`

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.11 PCO_GetAcquireModeEx

Description:

This function returns the current acquire mode of the camera. Acquire mode can be either [auto], [external], [external modulate] or [sequence trigger].
This function is an extended version of the **PCO_GetAcquireMode** function with an additional parameter `dwNumberImages`, which is needed for the [sequence trigger] mode.

Supported camera type:

pco.edge

Descriptor dependency:

`dwGeneralCapsDESC1: NO_ACQUIREMODE,EXT_ACQUIRE`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetAcquireModeEx (
    HANDLE ph,                          //in
    WORD* wAcquMode,                    //out
    DWORD*  dwNumberImages,             //out
    DWORD* dwReserved                   //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wAcquMode`	`WORD*`	Pointer to a WORD variable to get the acquire mode: • `0x0000` = [auto] All images will be acquired and stored. The external <acq enbl> input is ignored. • `0x0001` = [external] The external control input <acq enbl> is a static enable signal for image acquisition. Depending on the I/O configuration a high or low level at the external <acq enbl> input does set the acquire enable state to TRUE. If the acquire enable state is TRUE exposure triggers are accepted and images are acquired. If the acquire enable state is FALSE, all exposure triggers are ignored and no images will be acquired and stored. • `0x0002` = [external modulate] The external control input <acq enbl> is a dynamic frame start signal. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will start a single frame in modulation mode. • `0x0004` = [sequence trigger] The external control input <acq enbl> is a dynamic sequence start signal. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will start a sequence of images until the current number of images is acquired. Additional triggers during the sequence are rejected.
`dwNumber Images`	`DWORD*`	Pointer to a DWORD variable to get the number of images to acquire: Number of images, which will be acquired when a rising or falling edge at the acquire input triggers a sequence. This parameter is only valid for acquire mode [sequence trigger]
`dwReserved`	`DWORD*`	Reserved. Set to NULL at input

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.12 PCO_SetAcquireModeEx

Description:

This function sets the ***acquire mode*** of the camera. ***Acquire mode*** can be either [auto], [external], [external modulate] or [sequence trigger]. This function is an extended version of the **PCO_SetAcquireMode** function with an additional parameter `dwNumberImages`, which is needed for the [sequence trigger] mode.

Supported camera type:

pco.edge

Descriptor dependency:

`dwGeneralCapsDESC1: NO_ACQUIREMODE,EXT_ACQUIRE`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetAcquireModeEx (
    HANDLE ph,                          //in
    WORD wAcquMode,                     //in
    DWORD  dwNumberImages,              //in
    DWORD* dwReserved                   //in,out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wAcquMode`	`WORD`	`WORD` variable to set the acquire mode: • `0x0000` = [auto]: All images will be acquired and stored. The external <acq enbl> input is ignored • `0x0001` = [external]: The external control input <acq enbl> is a static enable signal for image acquisition. Depending on the I/O configuration a high or low level at the external <acq enbl> input does set the acquire enable state to TRUE. If the acquire enable state is TRUE exposure triggers are accepted and images are acquired. If the acquire enable state is FALSE, all exposure triggers are ignored and no images will be acquired and stored. • `0x0002` = [external modulate]: The external control input <acq enbl> is a dynamic frame start signal. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will start a single frame in modulation mode. • `0x0004` = [sequence trigger]: The external control input <acq enbl> is a dynamic sequence start signal. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will start a sequence of images until the current number of images is acquired. Additional triggers during the sequence are rejected.
`dwNumber Images`	`DWORD`	`DWORD` to set the number of images to acquire: Number of images, which will be acquired when a rising or falling edge at the acquire input triggers a sequence. This parameter is only valid for acquire mode [sequence trigger]
`dwReserved`	`DWORD*`	Pointer to a `DWORD` array (4 members for future use): Set array values to zero. A `NULL`-pointer is also accepted

Parameter dependency:

`dwGeneralCapsDESC1: ENHANCED_DESCRIPTOR_2`
`dwModulateCapsDESC2: MODULATE`

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.13 PCO_GetAcqEnblSignalStatus

Description:

This function returns the current status of the external <acq enbl> input. Depending on the I/O configuration a high or low level at the external <acq enbl> input does set the acquire enable state to TRUE.
Due to response and processing times the delay between the delivered status and the current status may be several 10 ms e.g. caused by the interface and/or the operating system.
If timing is critical it is strongly recommended to use other trigger modes.

Supported camera type:

pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_ACQUIREMODE`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetAcqEnblSignalStatus (
    HANDLE ph,                          //in
    WORD* wAcquEnableState              //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wAcquEnableState`	`WORD*`	Pointer to a `WORD` to get the acquire enable state: • `0x0000` = [FALSE] • `0x0001` = [TRUE]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.14 PCO_GetMetaDataMode

Description:

This function returns the current ***Meta Data*** mode of the camera and information about size and version of the ***Meta Data*** block.
When ***Meta Data*** mode is enabled, a ***Meta Data*** block with additional information is added at the end of each image. The internal buffers allocated with **PCO_AllocateBuffer** are adapted automatically. If the buffers are allocated externally, further line(s) must be added, where the number of lines depends on horizontal resolution and the size of the additional ***Meta Data*** block.

Supported camera type:

pco.dimax, pco.edge

Descriptor dependency:

`dwGeneralCapsDESC1: METADATA`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetMetaDataMode (
    HANDLE ph,                          //in
    WORD*  wMetaDataMode,               //out
    WORD*  wMetaDataSize,               //out
    WORD* wMetaDataVersion              //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wMetaDataMode`	`WORD*`	Pointer to a `WORD` to get the Meta Data mode: • `0x0000` = [off] • `0x0001` = [on]
`wMetaDataSize`	`WORD*`	Pointer to a `WORD` variable to get the size of the Meta Data block, which will be added to the image (size of Meta Data block in additional pixels)
`wMetaDataVersion`	`WORD*`	Pointer to a `WORD` variable to get the version of the Meta Data mode

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.15 PCO_SetMetaDataMode

Description:

This function does to set the mode for ***Meta Data*** and returns information about size and version of the ***Meta Data*** block.
When ***Meta Data*** mode is set to [on], a ***Meta Data*** block with additional information is added at the end of each image. The internal buffers allocated with **PCO_AllocateBuffer** are adapted automatically. If the buffers are allocated externally, the user is responsible to add further line(s), where the number of lines depends on horizontal resolution and the size of the additional ***Meta Data*** block.

Note: Meta Data mode must not be changed during one session. Failure to follow this rule might result in an application crash.

Supported camera type:

pco.dimax, pco.edge

Descriptor dependency:

`dwGeneralCapsDESC1: METADATA`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetMetaDataMode (
    HANDLE ph,                          //in
    WORD wMetaDataMode,                 //in
    WORD* wMetaDataSize,                //out
    WORD* wMetaDataVersion              //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wMetaDataMode`	`WORD`	`WORD` variable to set the Meta Data mode: • `0x0000` = [off] • `0x0001` = [on]
`wMetaDataSize`	`WORD*`	Pointer to a `WORD` variable to get the size of the Meta Data block, which will be added to the image (size of Meta Data block in additional pixels)
`wMetaDataVersion`	`WORD*`	Pointer to a `WORD` variable to get the version of the Meta Data mode

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.16 PCO_GetRecordStopEvent

Description:

This function returns the current record stop event mode and the number of images, which will be recorded after a recorder stop event is triggered.
The record stop event mode is only valid, if storage mode is [recorder] and recorder submode is [ring buffer].

Supported camera type:

pco.1200, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: RECORD_STOP`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetRecordStopEvent (
    HANDLE ph,                          //in
    WORD* wRecordStopEventMode,         //out
    DWORD* dwRecordStopDelayImages      //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wRecordStopEventMode`	`WORD*`	Pointer to a `WORD` variable to get the record stop event mode: • `0x0000` = [off] • `0x0001` = [software] Trigger stop event by command • `0x0002` = [extern] The external control input <acq enbl> is a dynamic trigger signal for the stop event. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will trigger the stop event. The stop event can also be triggered by software command.
`dwRecordStopDelayImages`	`DWORD*`	Pointer to a `DWORD` variable to get the number of images recorded after the record stop event is triggered

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.17 PCO_SetRecordStopEvent

Description:

This function does set the record stop event mode and as an additional parameter the number of images, which will be recorded after a recorder stop event is triggered.
The record stop event mode is useful to record a series of images with the ability to review the scene before and after the stop event.
A record stop event can be triggered through the software command **PCO_StopRecord** or a signal at the <acq enbl> input. After the stop event is triggered the camera records the configured number of images and stops after that. The record stop event function can only be used if storage mode is set to [recorder] and recorder submode is set to [ring buffer].
Due to internal timing constrains the current number of images taken after the event may differ by +/- 1 from the configured number.

Supported camera type:

pco.1200, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: RECORD_STOP`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetRecordStopEvent (
    HANDLE ph,                          //in
    WORD wRecordStopEventMode,          //in
    DWORD dwRecordStopDelayImages       //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wRecordStopEventMode`	`WORD`	`WORD` variable to set the record stop event mode: • `0x0000` = [off] • `0x0001` = [software] Trigger stop event by command • `0x0002` = [extern] The external control input <acq enbl> is a dynamic trigger signal for the stop event. Depending on the I/O configuration a rising or falling edge at the <acq enbl> input will trigger the stop event. The stop event can also be triggered by software command.
`dwRecordStopDelayImages`	`DWORD`	`DWORD` variable to set the number of images recorded after the record stop event occurred. If the given number of images is recorded, the current recording will be stopped automatically

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.18 PCO_StopRecord

Description:

This function does generate a stop event for the record stop event mode. See also **PCO_GetRecordStopEvent** and **PCO_SetRecordStopEvent**.
Due to internal timing constrains the current number of images taken after the event may differ by +/- 1 from the configured number.

Supported camera type:

pco.1200, pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: RECORD_STOP`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_StopRecord (
    HANDLE ph,                          //in
    WORD* wReserved0,                   //in
    DWORD *dwReserved1                  //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wReserved0	WORD	Reserved for future use, set to zero
dwReserved1	DWORD	Reserved for future use, set to zero

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.19 PCO_SetDateTime

Description:

This function does set date and time information for the internal camera clock, which is used for the timestamp function. When powering up the camera the camera clock is reset and all date and time information is set to zero. If timestamp data should be synchronized with the PC time, this function must be called at least once. It might be necessary to call the function again in distinct time intervals, because some deviation between PC time and camera time might occur after some time. When this function is called the [ms] and [µs] values of the camera clock are set to zero. All parameter values must be set in packed BCD code.
The **PCO_SetDateTime** function is called during a **PCO_OpenCamera** call to synchronize PC time with camera time.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetDateTime (
    HANDLE ph,                          //in
    BYTE ucDay,                         //in
    BYTE ucMonth,                       //in
    WORD wYear,                         //in
    WORD wHour,                         //in
    BYTE ucMin,                         //in
    BYTE ucSec                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
ucDay	BYTE	BYTE variable to set the day of month (1 - 31)
ucMonth	BYTE	BYTE variable to set the month (1 - 12)
wYear	WORD	WORD variable to set the year (4 digits e.g. 2017)
wHour	WORD	WORD variable to set the hour (0 - 24)
ucMin	BYTE	BYTE variable to set the minute (0 - 60)
ucSec	BYTE	BYTE variable to set the second (0 - 60)

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Example:
10th November 2017, 14h 15min 35s

```
PCO_SetDateTime (ph, 10, 11, 2017, 14, 15, 35);
```

2.7.20 PCO_GetTimestampMode

Description:

This function returns the current timestamp mode. To obtain information about the recording time of images a timestamp can be included in the raw image data. This timestamp consists of a continuous image number and the date and time information of the camera clock. The first 14 pixels of the image data array are used to hold this information. Image numbering always starts from ***1***. In mode [binary] the timestamp information is packed BCD coded in the lower byte of each pixel value, so every pixel holds 2 digits of information. If the bit alignment of the camera is set to [MSB aligned] the pixel value must be shifted to the right before decoding of data can be done. In mode [ASCII] the information is written as ASCII text replacing the original image data. An 8 by 8 pixel matrix is used per ASCII digit showing white on black characters. There also exists mode [binary+ASCII] which is a combination of the both methods described above.

Format of BCD coded pixels:

Pixel 1	Pixel 2	Pixel 3	Pixel 4	Pixel 5	Pixel 6	Pixel 7
Image counter(MSB) (00…99)	Image counter (00…99)	Image counter (00…99)	Image counter(LSB) (00…99)	Year (MSB) (20)	Year (LSB) (03…99)	Month (01..12)
Pixel 8	**Pixel 9**	**Pixel 10**	**Pixel 11**	**Pixel 12**	**Pixel 13**	**Pixel 14**
Day (01…31)	Hour (00..23)	Minutes (00…59)	Seconds (00…59)	µs * 10000 (00…99)	µs * 100 (00…99)	µs (00…90)

Format of ASCII text:

Number, date and time are separated by blanks

image number:	8 digits	[1 … 99999999]
date:	9 digits	[01JAN2003…31DEZ2099]
time:	15 digits	[00:00:00.000000…23:59:59.999990]

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: NO_TIMESTAMP`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetTimestampMode (
    HANDLE ph,                          //in
    WORD* wTimeStampMode                //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wTimeStampMode`	`WORD*`	Pointer to a `WORD` variable to get the timestamp mode: • `0x0000` = [off] • `0x0001` = [binary] BCD coded timestamp in the first 14 pixel • `0x0002` = [binary+ASCII] BCD coded timestamp in the first 14 pixel + ASCII text • `0x0003` = [ASCII] ASCII text only

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.7.21 PCO_SetTimestampMode

Description:

This function does set the timestamp mode of the camera. Details about the timestamp modes are explained in the previous command **PCO_GetTimestampMode**.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: NO_TIMESTAMP`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetTimestampMode (
    HANDLE ph,                          //in
    WORD wTimeStampMode                 //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wTimeStampMode	WORD	WORD variable to set the timestamp mode: • 0x0000 = [off] • 0x0001 = [binary] BCD coded timestamp in the first 14 pixel • 0x0002 = [binary+ASCII] BCD coded timestamp in the first 14 pixel + ASCII text • 0x0003 = [ASCII] ASCII text only (see camera descriptor for availability)

Parameter dependency:

`dwGeneralCapsDESC1: TIMESTAMP_ASCII_ONLY`

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8 STORAGE CONTROL

This function group can be used to get or set parameters regarding the ***camera internal memory*** (CamRAM). The ***camera internal memory*** is arranged as an array with four segments.

The overall size of the internal memory can be readout, distributed to any of the four memory segments and the active segment can be selected. Each segment can store images with individual settings. In default state all memory is distributed to segment 1 and segment 1 is also set as the active segment.

Segment size is always set as a multiple of CamRAM pages with a predefined page size. One CamRAM page is the smallest unit for RAM segmentation as well as for storing images. The size reserved for one image is also calculated as a multiple of whole pages. Therefore some unused RAM memory exists for each image, if the CamRAM page size is not exactly a multiple of the image size. The number of CamRAM pages needed for one image is calculated as image size in pixel divided by CamRAM page size. The result must be rounded up to the next integer. With this value of 'pages per image' the number of images fitting into one segment can be calculated.
Because camera internal structures must be changed when parameters in this group are set, the setting can only be done, if **PCO_SetRecordingState** is [stop] and must be followed by a **PCO_ArmCamera** command.

All storage functions can only be used with cameras which have internal recorder memory. Flag `NO_RECORDER` must not be set in the camera descriptor.

2.8.1 PCO_GetStorageStruct

Description:

Information about camera internal memory (CamRAM) is queried from the camera and the variables of the **PCO_Storage structure** are filled with this information. This function is a combined version of the functions, which request information about the current settings of storage related parameter. For a detailed description of each parameter see the functions in this chapter.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetStorageStruct (
    HANDLE ph,                          //in
    PCO_Storage* strStorage             //in,out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`strStorage`	`PCO_Storage*`	Pointer to a **PCO_Storage structure**: • On input the `wSize` parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes • On output the structure is filled with the requested information from the camera

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8.2 PCO_SetStorageStruct

Description:

This function does set the complete set of storage settings at once. For the sake of clarity it is better to use the functions which change distinct parameter despite changing all settings at once.
An invalid value for one of the parameter will result in a failure response message.
The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetStorageStruct (
    HANDLE ph,                          //in
    PCO_Storage* strStorage             //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`strStorage`	`PCO_Storage*`	Pointer to a **PCO_Storage structure** filled with appropriate parameters. The `wSize` parameter of this structure and also of all nested structures must be filled with the correct structure size in bytes

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8.2.1 PCO_Storage structure

Name	Type	Description
`wSize`	`WORD`	Size of this structure
`ZZwAlignDummy1`	`WORD`	Reserved
`dwRamSize`	`DWORD`	Size of camera internal memory in CamRAM pages
`wPageSize`	`WORD`	CamRAM page size in pixel
`ZZwAlignDummy4`	`WORD`	Reserved
`dwRamSegSize[4]`	`DWORD`	List of memory segment sizes in CamRAM pages
`ZZdwDummyrs[20]`	`DWORD`	Reserved
`wActSeg`	`WORD`	Number of active segment
`ZZwDummy[]`	`WORD`	Reserved

2.8.3 PCO_GetCameraRamSize

Description:

This function returns the size of the camera internal memory in CamRAM pages and the CamRAM page size in pixels.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraRamSize (
    HANDLE ph,                          //in
    DWORD* dwRamSize,                   //out
    WORD*  wPageSize                    //out
);
```

Parameters:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwRamSize	DWORD*	Pointer to a DWORD variable to get the size of camera internal memory in CamRAM pages
wPageSize	WORD*	Pointer to a WORD variable to get the CamRAM page size in pixels

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8.4 PCO_GetCameraRamSegmentSize

Description:

This function returns a list of memory segment sizes in CamRAM pages.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCameraRamSegmentSize (
    HANDLE ph,                          //in
    DWORD* dwRamSegSize                 //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwRamSegSize	DWORD*	Pointer to a DWORD array to get the segment sizes. The array must have at least 4 DWORD entries

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8.5 PCO_SetCameraRamSegmentSize

Description:

This function does set segment size in CamRAM pages of the four memory segments. The segment size must be large enough to hold at least two images. All image data in all segments is cleared.

All previously recorded images will be lost.

- The sum of all segment sizes must not be larger than the total size of the RAM (as multiples of pages).
- A single segment size can have the value `0x0000`, but the sum of all four segments must be bigger than the size of two images.
- Pay attention that the array `dwRamSegSize` is zero based indexed while the segment number is 1 based, e.g. RAM size of segment 1 is stored in `dwRamSegSize[0]`.
- The command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetCameraRamSegmentSize (
    HANDLE ph,                          //in
    DWORD* dwRamSegSize                 //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`dwRamSegSize`	`DWORD*`	Pointer to a `DWORD` array to set the segment sizes. The array must have at least 4 `DWORD` entries

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8.6 PCO_ClearRamSegment

Description:

This function does clear the ***active memory segment***. All image data is cleared and the segment is prepared for new images.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_ClearRamSegment (
    HANDLE ph                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8.7 PCO_GetActiveRamSegment

Description:

This function returns the ***active memory segment*** of the camera.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetActiveRamSegment (
    HANDLE ph,                          //in
    WORD* wActSeg                       //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wActSeg	WORD*	Pointer to a WORD variable to get the currently active segment

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.8.8 PCO_SetActiveRamSegment

Description:

This function does set the active memory segment. Images from a subsequent recording are stored in the memory of the active segment. Relevant settings of the recorded images are also stored for each segment see also **PCO_GetSegmentImageSettings**.
This command will be rejected, if ***Recording State*** is [run], see **PCO_GetRecordingState**.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetActiveRamSegment (
    HANDLE ph,                          //in
    WORD wActSeg                        //in
);
```

Parameters:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wActSeg	WORD	WORD variable to set the active segment. Valid numbers are 1 / 2 / 3 / 4

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9 IMAGE INFORMATION

This function group can be used to get information about the layout of the images stored in the segments of the camera, bit alignment during image transfer and used image correction mode.

2.9.1 PCO_GetImageStruct

Description:

Information about previously recorded images is queried from the camera and the variables of the **PCO_Image structure** are filled with this information. This function is a combined version of the functions, which request information about the current recorded images. For a detailed description of each parameter see the functions in this chapter. For the sake of clarity and because the **PCO_Image structure** has a lot of reserved parameters it is better to use the functions which query distinct parameter.

Supported camera type:

pco.dimax, pco.1200, pco1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetImageStruct (
    HANDLE ph,                          //in
    PCO_Image* strImage                 //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strImage	PCO_Image*	Pointer to a **PCO_Image structure** to get the image settings

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9.1.1 PCO_Image structure

Name	Type	Description
wSize	WORD	Size of this structure
ZZwAlignDummy1	WORD	Reserved
strSegment[4]	PCO_Segment	Segment information structures
ZZstrDummySeg[14]	PCO_Segment	Reserved
strColorSet	PCO_Image_ColorSet	Reserved
wBitAlignment	WORD	Bit alignment
wHotPixelCorrectionMode	WORD	Hot pixel correction mode

2.9.2 PCO_GetSegmentStruct

Description:

Information about previously recorded images is queried from the camera and the variables of the **PCO_Segment structure** are filled with this information. These returned parameters depend on the camera settings, which have been active during the last recording to the dedicated segment. The **PCO_Segment structure** includes also information about count of images in the segment and the last SoftROI settings, which are pure virtual and depend only on settings in the ***API***.

Supported camera type:

pco.dimax, pco.1200, pco1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetSegmentStruct (
    HANDLE ph,                      //in
    WORD wSegment,                  //out
    PCO_Segment*  strSegment        //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSegment	WORD	WORD variable to address the desired segment (1/2/3/4)
strSegment	PCO_Segment*	Pointer to a **PCO_Segment structure** to get the segment image settings of the addressed segment

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9.2.1 PCO_Segment structure

Name	Type	Description
wSize	WORD	Size of this struct
wXRes	WORD	Resulting horizontal resolution. Depend on image area selected, while recording in this segment
wYRes	WORD	Resulting vertical resolution. Depend on image area selected, while recording in this segment
wBinHorz	WORD	Horizontal binning
wBinVert	WORD	Vertical binning
wRoiX0	WORD	ROI upper left horizontal
wRoiY0	WORD	ROI upper left vertical
wRoiX1	WORD	ROI lower right horizontal
wRoiY1	WORD	ROI lower right vertical
ZZwAlignDummy1	WORD	Reserved
dwValidImageCnt	DWORD	Number of valid images in segment
dwMaxImageCnt	DWORD	Maximum number of images in segment
wRoiSoftX0	WORD	Soft ROI upper left horizontal
wRoiSoftY0	WORD	Soft ROI upper left vertical
wRoiSoftX1	WORD	Soft ROI lower right horizontal
wRoiSoftY1	WORD	Soft ROI lower right vertical
wRoiSoftXRes	WORD	Soft ROI resulting horizontal resolution
wRoiSoftYRes	WORD	Soft ROI resulting vertical resolution
wRoiSoftDouble	WORD	Soft ROI with double image option enabled
ZZwDummy[33]	WORD	Reserved

2.9.3 PCO_GetSegmentImageSettings

Description:

Information about previously recorded images is queried from the camera. The returned parameters depend on the camera settings which have been active during the last recording to the dedicated segment.

Supported camera type:

pco.dimax, pco.1200, pco1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetSegmentImageSettings (
    HANDLE ph,                          //in
    WORD wSegment,                      //in
    WORD*  wXRes,                       //out
    WORD* wYRes,                        //out
    WORD* wBinHorz,                     //out
    WORD* wBinVert,                     //out
    WORD* wRoiX0,                       //out
    WORD* wRoiY0,                       //out
    WORD* wRoiX1,                       //out
    WORD* wRoiY1                        //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSegment	WORD	WORD variable to address the desired segment (1/2/3/4)
wXRes	WORD*	Pointer to a WORD variable to get the horizontal resolution of the recorded images
wYRes	WORD*	Pointer to a WORD variable to get the vertical resolution of the recorded images
wBinHorz	WORD*	Pointer to a WORD variable to get the horizontal binning of the recorded images
wBinVert	WORD*	Pointer to a WORD variable to get the vertical binning of the recorded images
wRoiX0	WORD*	Pointer to a WORD variable to get the upper left horizontal ROI of the recorded images
wRoiY0	WORD*	Pointer to a WORD variable to get the upper left vertical ROI of the recorded images
wRoiX1	WORD*	Pointer to a WORD variable to get the lower right horizontal ROI of the recorded images
wRoiY1	WORD*	Pointer to a WORD variable to get the lower right vertical ROI of the recorded images

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9.4 PCO_GetNumberOfImagesInSegment

Description:

This function returns the number of valid images and the maximum number of images within a distinct segment.
While recording the number of valid images is dynamic - due to read and write accesses to the CamRAM. If the recording is stopped, the variable `ValidImageCnt` does not change anymore.
If the camera is in storage mode [recorder] the variable `ValidImageCnt` is counting up until the maximum image count is reached. After that the variable remains at the same value.
If the camera is in storage mode [FIFO buffer] the variable `ValidImageCnt` can decrease also, if the amount of transferred images is greater than the recorded images. If `ValidImageCnt` does stay at ***1***, transfer rate is equal or greater than recording rate. If `ValidImageCnt` is equal to maximum image count, the transfer rate is too slow and therefore recorded images are lost.
In storage mode [FIFO buffer] the ratio of valid number of images to the maximum number of images is a kind of filling level indicator.

Supported camera type:

pco.dimax, pco.1200, pco.1600, pco.2000, pco.4000

Descriptor dependency:

`dwGeneralCapsDESC1: NO_RECORDER`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetNumberOfImagesInSegment (
    HANDLE ph,                          //in
    WORD wSegment,                      //in
    DWORD* dwValidImageCnt,             //out
    DWORD* dwMaxImageCnt                //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSegment	WORD	WORD variable to address the desired segment (1 / 2 / 3 / 4)
dwValidImageCnt	DWORD*	Pointer to a DWORD variable to get the valid number of images in the addressed segment
dwMaxImageCnt	DWORD*	Pointer to a DWORD variable to get the maximum possible number of images in the addressed segment

Parameter dependency:

None

Return value:

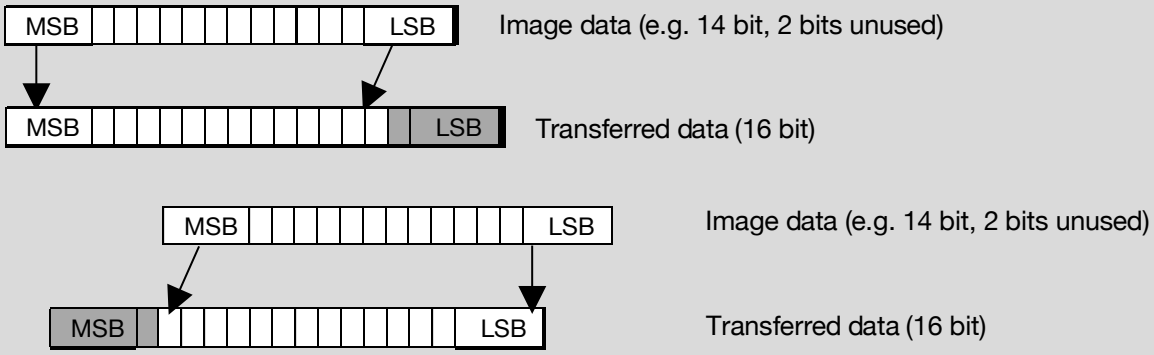
int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9.5 PCO_GetBitAlignment

Description:

This function returns the current bit alignment of the transferred image data.
If the dynamic resolution of the camera is less than 16 bit/pixel and because the transferred image data is always send as one WORD (16 bit) per pixel, the data can be either MSB or LSB aligned.

Alignment set to 0 – MSB aligned:



Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetBitAlignment (
    HANDLE ph,                          //in
    WORD* wBitAlignment                 //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wBitAlignment	WORD*	Pointer to a WORD variable to get to the bit alignment: • 0x0000 = [MSB] • 0x0001 = [LSB]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9.6 PCO_SetBitAlignment

Description:

This function does set the current bit alignment of the transferred image data.
See **PCO_GetBitAlignment** for further details.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetBitAlignment (
    HANDLE ph,                          //in
    WORD wBitAlignment                  //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wBitAlignment	WORD	WORD variable to set the bit alignment: • 0x0000 = [MSB] • 0x0001 = [LSB]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9.7 PCO_GetHotPixelCorrectionMode

Description:

This function returns the current mode of the hot pixel correction.
Hot pixel correction is used to eliminate hot, stuck or dead pixels from the raw image data, before the image data is transferred. The coordinates of all these pixels are stored in the hot pixel list of the camera.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: HOT_PIXEL_CORRECTION`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetHotPixelCorrectionMode (
    HANDLE ph,                          //in
    WORD*  wHotPixelCorrectionMode      //out
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wHotPixelCorrectionMode`	`WORD*`	Pointer to a `WORD` variable to get the hot pixel correction mode: • `0x0000` = [off] • `0x0001` = [on]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.9.8 PCO_SetHotPixelCorrectionMode

Description:

This function does set the hot pixel correction mode.

Supported camera type:

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: HOT_PIXEL_CORRECTION`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetHotPixelCorrectionMode (
    HANDLE ph,                          //in
    WORD  wHotPixelCorrectionMode       //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wHotPixelCorrectionMode`	`WORD`	`WORD` variable to set the hot pixel correction mode: • `0x0000` = [off] • `0x0001` = [on]

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.10 BUFFER MANAGEMENT

This function group can be used to allocate buffers for image transfers from the camera and to request the status of the transfer.
The functions of this group ***cannot be used***, if the connection to the camera is established through the ***serial*** connection of a ***Camera Link*** grabber. In this case the SDK of the grabber manufactorer must be used to do any buffer management.

2.10.1 PCO_AllocateBuffer

Description:

This function does set up a buffer context to receive the transferred images. A buffer index is returned, which must be used for the image transfer functions. There is a maximum of 16 buffers per camera. The buffers are attached to the camera handle. Therefore allocated buffers for one camera cannot be used for a different camera. Memory can be allocated either internal or already allocated memory can be attached to the buffer context. Because some of the image transfer functions use events to inform the application about finished transfer, an event handle is included in the buffer context. The event can be either a user allocated event or it is generated internal. Using two buffers in an alternating manner is sufficient for most applications. If more than one camera is used, the same buffer index can be returned for each camera.

To create a new buffer `*sBufNr` must be set to ***–1*** on input. If the function returns without error, `*sBufNr` contains the buffer index for this buffer context.
If the memory allocation should be done internally, `*wBuf` must be set to NULL and `dwSize` should be the current image size in bytes (Xres * Yres * sizeof(WORD)). If the function returns without error, `*wBuf` contains the pointer to the allocated memory. Larger buffers may be allocated, but the image transfer functions will always write to the returned start address of the memory and some memory will be unused.
If external allocated memory should be attached, `*wBuf` must be set to a valid address and `dwSize` must be the size of the allocated memory block in bytes. If ***Meta Data*** (**PCO_SetMetaDataMode)** is enabled, further line(s) must be added to the allocated memory area, where the number of lines depends on horizontal resolution and the size of the additional ***Meta Data*** block.

To create the event handle internal, `*hEvent` must be set to NULL. If the function returns without error, `*hEvent` contains the handle to the internal created manual reset event.
If an external created event should be used, `*hEvent` must be set to the handle of the already created event.

Allocated or attached memory is initialized to ***zero*** by this function.

After changing the image size a ***reallocation*** should be done, with all valid buffer indices.
In case of internal allocated memory: memory with the new size will be allocated.
Pay attention that the start address might change.
An external allocated buffer will be tested with the new size.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_AllocateBuffer (
    HANDLE ph,                          //in
    SHORT*  sBufNr,                     //in/out
    DWORD dwSize,                       //in
    WORD** wBuf,                        //in/out
    HANDLE* hEvent                      //in/out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
sBufNr	SHORT*	Pointer to a SHORT variable: • On input: o -1: create a new buffer o 0 … 15: buffer index from previous call • On output: o the buffer index
dwSize	DWORD	Buffer size in bytes
wBuf	WORD**	Pointer to a pointer (WORD* variable) of a memory region: • On input: o NULL: allocate memory internal o Pointer to a valid memory block • On output: o Pointer to the (allocated) memory block
hEvent	HANDLE*	Pointer to a HANDLE variable: • On input: o NULL: create event internal o Handle of event • On output: o handle of the (created) event

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.10.2 PCO_FreeBuffer

Description:

This function does free a previously allocated buffer context with the given index. If internal memory was allocated for this buffer context it will be freed. If an internal event handle was created, it will be closed.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_FreeBuffer (
    HANDLE ph,                          //in
    SHORT sBufNr                        //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
sBufNr	SHORT	Buffer index

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.10.3 PCO_GetBufferStatus

Description:

This function does query the status of the buffer context with the given index.
Two status DWORDs are returned from this function, one (StatusDll) which describes the state of the buffer context, the other (StatusDrv) the state of the last image transfer into this buffer as ***PCO errorcode***.
The StatusDrv must always be checked to see if an image transfer was successful or not, because the event is signaled also when the buffer was cancelled or when the camera cannot fulfill the requested transfer.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetBufferStatus (
    HANDLE ph,                          //in
    SHORT sBufNr,                       //in
    DWORD* dwStatusDLL,                 //out
    DWORD* dwStatusDrv                  //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
sBufNr	SHORT	Buffer index
dwStatusDll	DWORD*	Pointer to a DWORD variable to get the status inside the SDK DLL: • 0x80000000 = Buffer is allocated • 0x40000000 = Buffer event created inside the SDK DLL • 0x20000000 = Buffer is allocated externally • 0x00008000 = Buffer event is set
dwStatusDrv	DWORD*	Pointer to a DWORD variable to get the status for the image transfer: • PCO_NOERROR = Image transfer succeeded • others = See error codes

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.10.4 PCO_GetBuffer

Description:

This function is used to query the objects of the buffer context with the given index.
The pointer to the allocated or attached memory region and the assigned event handle are returned.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetBuffer (
    HANDLE ph,                          //in
    SHORT sBufNr,                       //in
    WORD** wBuf,                        //out
    HANDLE* hEvent                      //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
sBufNr	SHORT	Buffer index
wBuf	WORD**	Pointer to a pointer (WORD* variable) of a memory region
hEvent	HANDLE*	Pointer to a HANDLE variable to get the event handle

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11 IMAGE ACQUISITION

This function group can be used to handle image transfers from the camera.
Image transfers can be done with two different methods. Both methods can be used to transfer images from a recording camera or if available from the camera internal memory (CamRAM)

Transfer a single image:
Function **PCO_GetImageEx** shall be used to transfer single images. With this function an image transfer from the camera is started and the function does not return until either the image transfer has been done successfully or the image transfer has got an error status or the transfer timed out. The timeout value for the transfer can be set with function **PCO_SetTimeouts** (second `DWORD`). The function returns the errorcode of the transfer or the timeout errorcode.

Transfer a continuous image stream
Functions **PCO_AddBufferEx** or **PCO_AddBufferExtern** shall be used to add transfer requests to an internal queue. An image transfer request to a distinct buffer context is added to an internal transfer request queue and the function returns immediately. One of the following waiting methods must then be used to check the transfer state and test if the image has been transferred completely:

- Call Windows ***API*** function ***WaitForSingleObject*** or ***WaitForMultipleObjects***
- Call Function **PCO_WaitforBuffer**
- Polling with **PCO_GetBufferStatus** (should be avoided)

After end of transfer is signaled **PCO_GetBufferStatus** has to be called and the `StatusDrv` must be checked to see if the transfer was successful or not.

The functions of this group ***cannot be used***, if the connection to the camera is established through the ***serial*** connection of a ***Camera Link*** grabber. In this case the SDK of the grabber manufacturer must be used to grab images from the camera.

2.11.1 PCO_GetImageEx

Description:

This function can be used to get a single image from the camera. The function does not return until the image is transferred to the buffer or an error occured. The timeout value for the transfer can be set with function **PCO_SetTimeouts** (second DWORD), the default value is 6 seconds.
On return the image is stored in the memory area of the buffer, which is addressed through parameter sBufNr.

To get images from the ***camera internal memory*** (CamRAM) the camera must be stopped. Any segment can be selected with parameter wSegment and the parameter dw1stImage selects the image number, which should be transferred. This parameter must be in the range from ***1*** to ValidImageCnt, which is returned from **PCO_GetNumberOfImagesInSegment**. Because the feature to transfer more than one image from internal memory per call is not implemented in PCO cameras with internal memory, this function is also limited to transfer single images. Therefore the parameter dwLastImage is useless at the moment, but nevertheless must be set to the same value as dw1stImage.

To get images from a recording camera both image number values dw1stImage and dwLastImage must be set to zero.
The size parameters are used to calculate the amount of data, which is transferred from the camera. The size must match the current size of the image, which should be transferred.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetImageEx (
    HANDLE ph,                          //in
    WORD wSegment,                      //in
    DWORD dw1stImage,                   //in
    DWORD dwLastImage,                  //in
    SHORT sBufNr,                       //in
    WORD wXRes,                         //in
    WORD wYRes,                         //in
    WORD wBitPerPixel                   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSegment	WORD	WORD variable to select a segment
dw1stImage	DWORD	DWORD variable to select the image number: • 1 to ValidImageCnt if **PCO_SetRecordingState** is [stop] • 0 if **PCO_SetRecordingState** is [run]
dwLastImage	DWORD	Must be set to same value as dw1stImage
sBufNr	SHORT	Buffer index
wXRes	WORD	Current horizontal resolution of the image which should be transferred
wYRes	WORD	Current vertical resolution of the image which should be transferred
wBitPerPixel	WORD	Bit resolution of the image which should be transferred

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.2 PCO_GetImage (obsolete)

Description:

This function can be used to get a single image from the camera.

Obsolete, please use **PCO_GetImageEx**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetImage (
    HANDLE ph,                          //in
    WORD wSegment,                      //in
    DWORD dw1stImage,                   //in
    DWORD dwLastImage,                  //in
    SHORT sBufNr                        //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSegment	WORD	WORD variable to select the segment
dw1stImage	DWORD	DWORD variable to select the image number: • 1 to ValidImageCnt, if **PCO_SetRecordingState** is [stop] • 0 if **PCO_SetRecordingState** is [run]
dwLastImage	DWORD	Must be set to same value as dw1stImage
sBufNr	SHORT	Buffer index

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

StatusDLL = status of buffer context (signalled or not)

StatusDrv = status of last image transfer into respective buffer (succesful or not)

2.11.3 PCO_AddBufferEx

buffers are added to request queue. When transfer of buffer is completed in can be set again into request queue. The transfer status of each buffer is stored in the StatusDrv variable

Description:

This function can be used to setup a request for a single image transfer from the camera. The transfer request is added to the internal request queue and this function returns immediately.
If the desired image has been transferred, the buffer event will be signaled and the appropriate bit is set in the `StatusDll` variable of the buffer context. The `StatusDrv` variable of the buffer context will hold the transfer status, which is either ***PCO_NOERROR*** if the transfer was successful or any error value.
More than one buffer can be added into the request queue. A buffer must not be set twice into the request queue at the same time. But when the transfer to a buffer is completed, it can be set again into the request queue. Any operation, which should be done on the image, must be finished before the buffer is added again.
After all image transfers are done or in case of errors **PCO_CancelImages** must be called to clear the internal queue and also to reset the transfer state machine in the camera.

To readout images from ***camera internal memory*** (CamRAM) the camera must be stopped. The current selected segment is used and the parameter `dw1stImage` selects the image number, which should be transferred. This value must be in the range from ***1*** to `ValidImageCnt`, which is returned from **PCO_GetNumberOfImagesInSegment**.
The parameter `dwLastImage` must always be set to the same value as `dw1stImage`.

To get images from a recording camera both image number values `dw1stImage` and `dwLastImage` must be set to zero. In this case **PCO_AddBufferEx** should be called after setting the ***Recording State*** to [on] (see **PCO_GetRecordingState**) to avoid error returns from the camera.

Only exception to this rule is when operating a ***pco.edge with Camera Link*** interface. Because with the first **PCO_AddBufferEx** call the internal request queue is setup and this might be a time consuming operation, first images of the camera might get lost. Therefore **PCO_AddBufferEx** should be called before setting the **PCO_SetRecordingState** to [on].
When a separate thread is used for image grabbing, synchronization between camera control thread and image transfer thread must be designed carefully.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_AddBufferEx (
    HANDLE ph,                          //in
    DWORD dw1stImage,                   //in
    DWORD dwLastImage,                  //in
    SHORT sBufNr,                       //in
    WORD wXRes,                         //in
    WORD wYRes,                         //in
    WORD wBitPerPixel                   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dw1stImage	DWORD	DWORD variable to select the image number: • 1 to ValidImageCnt if recording state is [stop] • 0 if recording state is [run]
dwLastImage	DWORD	Must be set to same value as dw1stImage
sBufNr	SHORT	Buffer index
wXRes	WORD	Current horizontal resolution of the image which should be transferred
wYRes	WORD	Current vertical resolution of the image which should be transferred
wBitPerPixel	WORD	Bit resolution of the image which should be transferred

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.4 PCO_AddBuffer (obsolete)

Description:

Obsolete, please use **PCO_AddBufferEx**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_AddBuffer (
    HANDLE ph,                          //in
    DWORD dw1stImage,                   //in
    DWORD dwLastImage,                  //in
    SHORT sBufNr                        //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dw1stImage	DWORD	DWORD variable to select the image number: • 1 to ValidImageCnt, if recording state is [stop] • 0 if recording state is [run]
dwLastImage	DWORD	Must be set to same value as dw1stImage
sBufNr	SHORT	Buffer index

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.5 PCO_AddBufferExtern

WRONG USE OF THIS FUNCTION MAY CRASH YOUR SYSTEM. PCO IS NOT RESPONSIBLE FOR DAMAGES DUE TO IMPROPER USE OF THIS FUNCTION.

Description:

This function can be used to setup a request for a single image transfer from the camera. The transfer request is added to the internal request queue and this function returns immediately.
If the desired image has been transferred, the event will be signaled and the value of the `dwStatus` variable will hold the transfer status, which is either ***PCO_NOERROR***, if the transfer was successful, or any error value. If the transfer was successful, the memory area, which was passed in, is filled with the image data from the camera.
A context which does hold the pointer to the memory area, the event handle and the pointer to the status `DWORD` should be used to differentiate between the added requests. This context must be valid as long as its members are set in the internal request queue, because the members are directly accessed from the underlying functions.
More than one memory area with an associated event can be added into the request queue. A memory area with an associated event must not be set twice into the request queue at the same time. When the transfer is completed, it can be set again into the request queue. Any operation, which should be done on the image, must be finished before the buffer is added again.
After all image transfers are done or in case of errors **PCO_CancelImages** must be called, to clear the internal queue and also to reset the transfer state machine in the camera.

To readout images from from ***camera internal memory*** (CamRAM) the camera must be stopped. Any segment can be selected with parameter `wSegment` and the parameter `dw1stImage` selects the image number, which should be transferred. This value must be in the range from 1 to `ValidImageCnt`, which is returned from **PCO_GetNumberOfImagesInSegment**. The parameter `dwLastImage` must always be set to the same value as `dw1stImage`.

To get images from a recording camera both image number values `dw1stImage` and `dwLastImage` must be set to zero. In this case **PCO_AddBufferEx** should be called after setting the recording state **PCO_SetRecordingState** to [on] to avoid error returns from the camera.

Only exception to this rule is when operating a ***pco.edge with Camera Link*** interface. Because with the first ***AddBuffer call*** the internal request queue is setup and this might be a time consuming operation, first images of the camera might get lost. Therefore **PCO_AddBufferExtern** should be called before setting the recording state to [on].
When a separate thread is used for image grab, synchronization between camera control thread and image transfer thread must be designed carefully.

If ***Meta Data mode*** (see **PCO_SetMetaDataMode**) is enabled, further line(s) must be added to the allocated memory area, where the number of lines to add depends on horizontal resolution and the size of the additional ***Meta Data*** block.
The benefit of using this function is that image transfer is speed up. Due to missing parameter checking the call itself is faster and due to setting own memory addresses, there is no need for a further copy from ***API*** buffers to another memory area.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_AddBufferExtern (
    HANDLE ph,                          //in
    HANDLE hEvent,                      //in
    WORD wActSeg,                       //in
    DWORD dw1stImage,                   //in
    DWORD dwLastImage,                  //in
    DWORD dwSynch,                      //in
    void* pBuf,                         //in
    DWORD dwLen,                        //in
    DWORD*  dwStatus                    //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
hEvent	HANDLE	Handle to an event. The event is signaled, if the transfer is finished successfully or an error occurred
wActSeg	WORD	WORD variable to select the segment
dw1stImage	DWORD	DWORD variable to select the image number: • 1 to ValidImageCnt, if recording state is [stop] • 0 if **PCO_SetRecordingState** is [run]
dwLastImage	DWORD	Must be set to same value as dw1stImage
dwSynch	DWORD	Reserved, set to 0
*pBuf	void	Pointer to the start address of memory area for the transferred image
dwLen	DWORD	Size of the memory area in bytes
*dwStatus	DWORD	Pointer to a DWORD to receive the buffer status

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.6 PCO_CancelImages

Description:

This function does remove all remaining buffers from the internal queue, reset the internal queue and also reset the transfer state machine in the camera. Buffers which are removed from the internal queue will set their event handle to signaled and the `StatusDrv` is set to `PCO_ERROR_DRIVER_BUFFER_CANCELLED`.
It is ***mandatory*** to call **PCO_CancelImages** after all image transfers are done. This function can be called before or after setting **PCO_SetRecordingState** to [stop]. In case calling this function is very time consuming, please change the order of cancel and setting the ***Recording State***.
In general it is necessary to synchronize this function with any of the ***Add_Buffer functions*** (**PCO_AddBufferEx**; **PCO_AddBufferExtern**), to eliminate misbehaviour, which might occur, when buffers are added during execution of **PCO_CancelImages**.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_CancelImages(
    HANDLE ph                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.7 PCO_RemoveBuffer (obsolete)

Description:

Obsolete, please use **PCO_CancelImages** instead.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_RemoveBuffer (
    HANDLE ph                          //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.8 PCO_GetPendingBuffer

Description:

This function can be used to query the number of pending buffers in the internal queue. Even if the number of pending buffers is zero it is recommended to call **PCO_CancelImages** after all image transfers are done, to ensure that the transfer state machine in the camera is set to an idle state.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetPendingBuffer (
    HANDLE ph,                          //in
    int* number                         //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
number	int*	Pointer to an int variable to get the number of pending buffers in the internal queue

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.9 PCO_WaitforBuffer

Description:

This function can be used to wait for one or more buffers, which have been set into the internal request queue of the driver. To handle the buffers, a list of `PCO_Buflist` structures must be set up, each filled with the buffer number of the allocated buffer. On return the two status `DWORDs` reflect the current status of the buffer, `dwStatusDll` describes the state of the buffer context, `dwStatusDrv` the state of the last image transfer into this buffer as error code.
This function uses an effective ***wait function*** (e.g. `WaitforMultipleObjects`) to wait for the events of the buffers, which are set up in the internal request queue and for which a list entry exists.

Supported camera type:

All cameras

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_WaitforBuffer (
    HANDLE ph,                          //in
    INT nr_of_buffer,                   //in
    PCO_Buflist *bl,                    //in,out
    int timeout                         //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`nr_of_buffer`	`int`	Number of entries of type structure **PCO_Buflist structure**, which are setup
`bl`	`PCO_Buflist*`	Pointer to a list of structures **PCO_Buflist structure**, which hold buffer parameter, which should be processed
`timeout`	`int`	Timeout in milliseconds

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

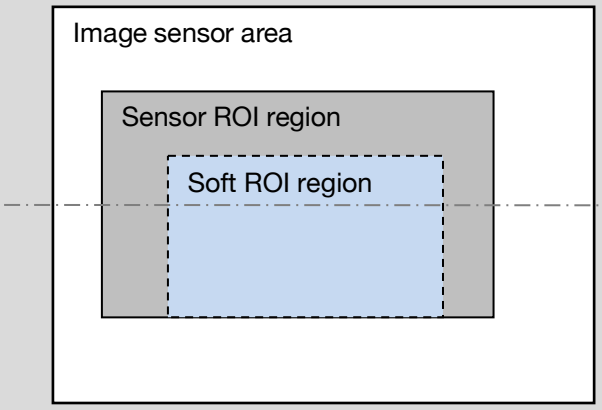
2.11.9.1 PCO_Buflist structure

Name	Type	Description
SBufNr	SHORT	Sizeof this struct
reserved	WORD	reserved
dwStatusDll	DWORD	Status inside the SDK DLL: • 0x80000000 = buffer is allocated • 0x40000000 = buffer event created inside the SDK DLL • 0x20000000 = buffer is allocated externally • 0x00008000 = buffer event is set
dwStatusDrv	DWORD	Status fot the image transfer: • PCO_NOERROR = image transfer succeeded • others = see Error codes

2.11.10 PCO_EnableSoftROI

Description:

This function enables ***Soft ROI*** functionality, which does provide a software based region of interest. Due to this functionality, the resolution of pco.edge cameras with Camera Link interface can be adjusted in steps of ***4 pixels horizontal***. Since the readout architecture of these cameras is not able to address single pixels, this downsizing is done by software in the ***API***.
When ***Soft ROI*** functionality is enabled, the camera descriptor is overwritten (granularity and symmetry restrictions). The **PCO_SetROI** and **PCO_GetROI** functions can be used as usual. The ***API*** does adapt the supplied ROI parameters to sensor appropriate settings automatically and cuts off the selected ROI region from the transferred image. Because images from the camera are exposed and transferred with camera ROI setting, current frame rate depends on camera ROI settings and ***not*** on the ***Soft ROI*** settings. Therefore the frame rate must be checked after setting a ROI to see, if it fullfills the requirement of the application.



Example pco.edge 5.5:
Image sensor area: 2560x2160

Soft-ROI setting:
X0 = 921, Y0 = 901
X1 = 1604, Y1 = 1500
Size: 684 x 600

Results in sensor ROI:
X0 = 801, Y0 = 661
X1 = 1760, Y1 = 1500
Size: 960x840

Frame rate:
285 Hz @ 256 Mhz
86 Hz @ 96 Mhz

Supported camera type

Only for cameras connected to ***Camera Link mico Enable IV (mEIV)*** grabber.

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_EnableSoftROI (
    HANDLE ph,                          //in
    WORD wSoftROIFlags                  //in
    void* param                         //in
    int ilen                            //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSoftROIFlags	WORD	WORD parameter to set Soft ROI functionality: • 0x0000 = disable Soft ROI • 0x0001 = enable Soft ROI
param	void*	Reserved, set to NULL
ilen	int	Reserved, set to 0

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.11 PCO_GetMetaData

Description:

This function can be used to query the additional image information, which the camera has attached to the transferred image, if ***Meta Data*** mode is enabled. The function does only work with buffers allocated with **PCO_AllocateBuffer**. The additional image information is returned as **PCO_METADATA_STRUCT structure**, which is defined in the file `sc2_common.h`. Header file `sc2_common.h` has to be included before `sc2_camexport.h`.

Note:
`wSize` parameter has to be set on input. **PCO_METADATA_STRUCT structure** is byte aligned.

Supported camera type

All cameras

Descriptor dependency:

`dwGeneralCapsDESC1: METADATA`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetMetaData (
    HANDLE ph,                          //in
    SHORT sBufBNr,                      //in
    PCO_METADATA_STRUCT *pMetaData,     //out
    DWORD dwReserved1,                  //in
    DWORD dwReserved2                   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
sBufNr	SHORT	Buffer index
pMetaData	PCO_METADATA_STRUCT*	Pointer to a **PCO_METADATA_STRUCT structure** to get the Meta Data
dwReserved1	DWORD	Reserved
dwReserved2	DWORD	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.11.11.1 PCO_METADATA_STRUCT structure

Name	Type	Description
wSize	WORD	Size of this structure
wVersion	WORD	Version of this structure
bIMAGE_COUNTER_BCD	BYTE[4]	0x00000001 to 0x99999999 where first byte is least significant byte
bIMAGE_TIME_US_BCD	BYTE[3]	0x000000 to 0x999999 where first byte is least significant byte
bIMAGE_TIME_SEC_BCD	BYTE	0x00 to 0x59
bIMAGE_TIME_MIN_BCD	BYTE	0x00 to 0x59
bIMAGE_TIME_HOUR_BCD	BYTE	0x00 to 0x23
bIMAGE_TIME_DAY_BCD	BYTE	0x01 to 0x31
bIMAGE_TIME_MON_BCD	BYTE	0x01 to 0x12
bIMAGE_TIME_YEAR_BCD	BYTE	0x00 to 0x99 only last two digits, 2000 has to be added
bIMAGE_TIME_STATUS	BYTE	0x00 = internal osc 0x01 = synced by IRIG 0x02 = synced by master
wEXPOSURE_TIME_BASE	WORD	Time base ns / µs / ms for following exposure time
dwEXPOSURE_TIME	DWORD	Exposure time in ns/us/ms according to timebase
dwFRAMERATE_MILLIHZ	DWORD	Frame rate in mHz, 0 if unknown
sSENSOR_TEMPERATURE	SHORT	current sensor temperature in °C 0x8000 if unknown
wIMAGE_SIZE_X	WORD	Current size of image in x direction (horizontal)
wIMAGE_SIZE_Y	WORD	Current size of image in y direction (vertical)
bBINNING_X	BYTE	Binning in x direction, 0x00 if unknown
bBINNING_Y	BYTE	Binning in y direction, 0x00 if unknown
dwSENSOR_READOUT_FREQUENCY	DWORD	Sensor readout frequency in Hz, 0 if unknown
wSENSOR_CONV_FACTOR	WORD	Sensor conversions factor in e-/ct, 0 if unknown
dwCAMERA_SERIAL_NO	DWORD	Camera serial number, 0 if unknown
wCAMERA_TYPE	WORD	Camera type, 0 if unknown
bBIT_RESOLUTION	BYTE	Dynamic resolution in bits/pixel
bSYNC_STATUS	BYTE	Status of PLL for external synchronisation 100Hz or 1kHz: • 0 = off • 1 = locked
wDARK_OFFSET	WORD	Nominal dark offset in counts, 0xFFFF if unknown current dark offset may differ
bTRIGGER_MODE	BYTE	Trigger mode
bDOUBLE_IMAGE_MODE	BYTE	0x00 = standard 0x01 = double image (PIV) mode
bCAMERA_SYNC_MODE	BYTE	0x00 = standalone; 0x01 = master 0x02 = slave
bIMAGE_TYPE	BYTE	0x01 = b/w; 0x02 = color bayer pattern 0x10 = RGB
wCOLOR_PATTERN	WORD	Bayer pattern color mask, see **Color Pattern description (2x2 matrix)**

2.12 DRIVER MANAGMENT

This function group can be used to get and set parameters for the different interface standards represented through the according interface DLL. Different options are available for each interface and therefore each interface DLL does use a different structure type for the parameter settings. Function **PCO_GetCameraType** can be used to query the interface type of the connected camera. The interface specific structure must be used to query or set the transfer parameters with the functions **PCO_SetTransferParameter** and **PCO_SetTransferParametersAuto**.

2.12.1 PCO_GetTransferParameter

Description:

Current transfer parameter settings are queried from the driver layer of the connected interface and the transfer parameter structure is filled with this information.

Supported camera type:

Depending on camera interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetTransferParameter(
    HANDLE ph,                          //in
    void* buffer,                       //out
    int ilen                            //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
buffer	void*	Pointer to a buffer to get the transfer parameter structure. The returned structure depends on the used interface DLL
ilen	int	Size of the buffer in bytes, which is passed in

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.12.2 PCO_SetTransferParameter

Description:

This function sets the transfer parameter structure of the connected interface.

Supported camera type:

Depending on camera interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetTransferParameter(
    HANDLE ph,                          //in
    void* buffer,                       //in
    int ilen                            //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
buffer	void*	Pointer to a buffer with the transfer parameter settings. Recommended type is one of the **Transfer Parameter Structures**
ilen	int	int variable to set the length of the buffer in bytes

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.12.3 Transfer Parameter Structures

Description of the structures used from each interface DLL

2.12.3.1 FireWire Interface

Structure type is PCO_1394_TRANSFER_PARAM
Changing the transfer parameters for the FireWire interface is only necessary, if simultaneous transfers from more than one camera are requested or if the FireWire bus must be shared with other devices. Then bus bandwidth must be split up among the connected cameras and a unique channel number must be selected for each camera. Decreasing the value for the usable bandwidth on the bus will increase the time, which is needed for the image transfers.
Default behaviour of the FireWire driver is to allocate bandwidth on the bus only, when an image is requested and free the bus again after the transfer is completed and the additional transfertime timeout has run out. The channel number used for this transfer is selected from the FireWire OHCI driver. With flag PCO_1394_HOLD_CHANNEL added to the number_of_isochannel parameter this behaviour is changed. When calling **PCO_SetTransferParameter**, the requested bus bandwidth is allocated at the requested channel number and hold as long as the flag is set.

Name	Type	Description
bandwidth_bytes	DWORD	Bandwidth size in bytes, which should be allocated for the image transfer on the isochronous channel of the FireWire bus. Maximum value is 4096. Values below 1024 should not be used. Default value is 4096
speed_of_isotransfer	DWORD	1, 2, 4, whereas • 1 = 100 MBit/s • 2 = 200 MBit/s • 4 = 400 MBit/s Default value is 4
number_of_isochannel	DWORD	Isochronous channel number to use 0…7, can be ored with flags PCO_1394_HOLD_CHANNEL and PCO_1394_AUTO_CHANNEL Default value is PCO_1394_AUTO_CHANNEL
number_of_isobuffers	DWORD	Maximum number of buffers to use when allocating transfer resources. Value depends on image size and is auto adjusted from the driver. Default is 128
byte_per_isoframe	DWORD	Information only: used bytes on the 1394 bus
bytes_available	DWORD	Information only: remaining bytes on the 1394 bus
reserved	DWORD[15]	Reserved

2.12.3.2 Camera Link Interface

Structure type is PCO_SC2_CL_TRANSFER_PARAM.
Transfer parameters of the Camera Link interface can be used to change the baud rate for the serial connection, which is used to send and receive the control commands and to change the settings for the image transfers in the camera and also in the interface DLL. Additionaly the camera can be setup to use the ***CClines*** of the Camera Link interface as external trigger signals.

Descriptor dependency:

`dwGeneralCapsDESC1: DATAFORMAT2X12, DATAFORMAT4X16, DATAFORMAT5X16`

Name	Type	Description
`baudrate`	`DWORD`	Baud rate of the Camera Link serial port interface. Valid values: • 9600, 19200, 38400, 57600, 115200
`ClockFrequency`	`DWORD`	Clock rate of the Camera Link interface. Valid values: • pco.1600, pco.2000: 40000000, 66000000, 80000000 • pco.4000: 32000000, 64000000 • pco.dimax: 80000000 • pco.edge: 85000000 (Note: different to sensor pixel clock!)
`CCline`	`DWORD`	Bit field to enable the usage of the CC1-CC4 lines: • `0x00000001:` CC1 line is used as external trigger input • `0x00000002:` CC2 line is used as external acquire input • `0x00000004:` Reserved • `0x00000008:` CC4 line is used as transmit enable • all others reserved
`DataFormat`	`DWORD`	Data format of transferred images: • `0x00000001:` 1 x 16 bit per Camera Link clock • `0x00000002:` 2 x 12 bit per Camera Link clock • For the pco.edge see special note below
`Transmit`		Bitfield for transmit parameters: • `0x00000001:` Enable continuous image transfer • `0x00000002:` Use longer gaps between frame and line signals • All others reserved for special use, must be set to 0

Note: pco.edge 5.5

PCO_SetTransferParametersAuto can be used which does set the necessary parameters automatically.
The `DataFormat` parameter is a combination of one of the ***PCO_CL_DATAFORMAT*** settings and the ***SCCMOS Readout format*** setting. With the SCCMOS readout format the data readout direction of the camera can be controlled.

Available ***PCO_CL_DATAFORMAT*** formats are:

- `0x0005: PCO_CL_DATAFORMAT_5x16`
- `0x0007: PCO_CL_DATAFORMAT_5x12`
- `0x0008: PCO_CL_DATAFORMAT_10x8`
- `0x0009: PCO_CL_DATAFORMAT_5x12L`
- `0x000A: PCO_CL_DATAFORMAT_5x12R`

Available values for ***SCCMOS Readout format*** are, (see **SCCMOS Readout Format**):

- `0x0000: SCCMOS_FORMAT_TOP_BOTTOM`
- `0x0100: SCCMOS_FORMAT_TOP_CENTER_BOTTOM_CENTER`
- `0x0200: SCCMOS_FORMAT_CENTER_TOP_CENTER_BOTTOM`
- `0x0300: SCCMOS_FORMAT_CENTER_TOP_BOTTOM_CENTER`
- `0x0400: SCCMOS_FORMAT_TOP_CENTER_CENTER_BOTTOM`

For each of the SCCMOS format settings the correct line sorting algorithm is choosen from the interface DLL.

For ***Global Shutter*** setup the data format cannot be changed.The available data format is:
`PCO_CL_DATAFORMAT_5x12 | SCCMOS_FORMAT_TOP_CENTER_BOTTOM_CENTER`

For ***Rolling Shutter*** or ***Global Reset*** setup the data format `PCO_CL_DATAFORMAT_10x8` can be used for simplified data transfer. Different LUT's are available in the camera to select the appropriate range for the 16 bit to 8 bit conversion.

For ***Rolling Shutter*** or ***Global Reset*** setup the data format setting for 16 bit data transfers depend on the camera type, the selected pixel clock and the horizontal resolution of the current camera ROI. If ***Soft ROI*** is enabled attention should be paid to use the current camera ROI for the calculation and not the settings of the ***Soft ROI***.

pco.edge 5.5 ***Rolling Shutter*** and ***GlobalReset*** mode:

Sensor Pixelrate,horizontal Resolution	PCO_CL_Dataformat	Lookup Table
95 MHz, all	`PCO_CL_DATAFORMAT_5x16`	`0`
286 MHz, below or equal 1920	`PCO_CL_DATAFORMAT_5x16`	`0`
286 MHz, above 1920	`PCO_CL_DATAFORMAT_5x12L` `PCO_CL_DATAFORMAT_5x12R`	`0x1612`

pco.edge 4.2 ***Rolling Shutter*** and ***GlobalReset*** mode:

Sensor Pixelrate,horizontal Resolution	PCO_CL_Dataformat	Lookup Table
95 MHz, all	`PCO_CL_DATAFORMAT_5x16`	`0`
272 MHz, all	`PCO_CL_DATAFORMAT_5x16`	`0`

If data format is set to `PCO_CL_DATAFORMAT_5x12L` or `PCO_CL_DATAFORMAT_5x12R` the camera lookup table must be set with **PCO_SetActiveLookuptable** to `0x1612`, to enable square root compression of the pixel data.

If data format `PCO_CL_DATAFORMAT_5x12L` is set the 12 bit compressed and packed pixel data are recalculated to 16 bit pixel values with a recalculation function in the interface DLL.
If data format `PCO_CL_DATAFORMAT_5x12R` is set no recalculation is done and the compressed and packed pixel data is transferred to the image buffer.

2.12.3.3 USB Interface

Structure type is PCO_USB_TRANSFER_PARAM

Transfer parameters of the USB interface should not be changed, but can be used to query the current settings in the camera and the interface DLL.

Name	Type	Description
Reserved1	unsigned int	Reserved
Clock Frequency	unsigned int	Clock rate of the camera internal interface bus information only should not be changed..
Reserved2	unsigned int	Reserved
Reserved3	unsigned int	Reserved
ImgTransMode	unsigned int	Image transfer modes. Information only cannot be set • Bit 0: 0: 14 bit, 1-12 bit • Bit 1: reserved • Bit 2: 0 bit stuffing disabled; 1 bit stuffing enabled • Bit 3: 0 padding to 1024 disabled; 1: padding to 1024 enabled

2.12.3.4 GigE Interface

Structure type is PCO_GIGE_TRANSFER_PARAM
Transfer parameters of the GigE interface can be used to adapt the camera to the available NIC and subnet structure.

Packet Delay: Indicates the delay (in µs) inserted between each ethernet packet for this stream channel. This can be used as a crude flow-control mechanism, if the application or the network infrastructure cannot keep up with the ethernet packets coming from the device.
Recommended values: 0...18000.

Calculation table*:

pco.dimax cs					
Packet Delay	0	2000	4000	6000	8000
MB/s**	97	58	28	23	18
Packet Delay	10000	12000	14000	16000	18000
MB/s**	15	12	11	9	8

pco.dimax S/HD/HS & pco.1600-4000					
Packet Delay	0	2000	4000	6000	8000
MB/s**	66	43	23	19	5
Packet Delay	10000	12000	14000	16000	18000
MB/s**	13	11	9	8	7

*if more cameras are connected to one switch, make sure that the overall calculated data rate is less than 100 MByte/sec.
**estimated values (Depending on the PC's HW)

Name	Type	Description
dwPacketDelay	DWORD	Sets the delay between two stream packets in µs: • Default: 4000 • Valid range: 0 <= x <= 18000
dwResendPercent	DWORD	Information only cannot be set. Percentile part of lost packages per image, which will be re-transferred, default 30. In case more packages got lost, the complete image will be re-transferred till it times out and produces an error
dwFlags	DWORD	Sets single flags: (obsolete; can only be used with GigE driver V2.0.0.3 and older versions): • Bit 0: enable packet resend • Bit 1: enable burst mode • Bit 2: enable max speed mode • Bit 3: reserved, set to zero • Bit 4: transfer bandwidth distribution: o 0: same bandwidth for all cameras o 1 active camera gets whole bandwidth • Bit 5-7: reserved, set to zero
dwDataFormat	DWORD	Data format of the transferred data. Information only cannot be set
dwCameraIPAddress	DWORD	Current IP address of the camera Information only cannot be set
DwUDPImgPcktSize	DWORD	Size of an UDP image packet Can only be set, if a pco.dimax CS camera is used
Ui64MACAddress	UINT64	MAC address of camera Information only, cannot be set

2.13 SPECIAL COMMANDS PCO.EDGE

Special commands for pco.edge family.

2.13.1 PCO_GetSensorSignalStatus

Description:

This function returns the current action state of the camera sensor.
The delay between the software delivered status and the current status may be several milliseconds due to response and processing times caused by the interface and / or the operating system. For exact synchronization to external events, hardware signals must be used.

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetSensorSignalStatus (
    HANDLE ph,                          //in
    DWORD* dwStatus,                    //out
    DWORD* dwImageCount,                //out
    DWORD* dwReserved1,                 //out
    DWORD* dwReserved2                  //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwStatus	DWORD*	Pointer to a DWORD to get the sensor action state see table **Sensor Action State Bits**
dwImageCount	DWORD*	Pointer to a DWORD to get the image count of the last finished image.
dwReserved1	DWORD*	Reserved
dwReserved2	DWORD*	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.13.1.1 Sensor Action State Bits

Flag name	Value	Description
Sensor_Busy	0x00000001	Sensor is busy and does not accept trigger
Sensor_Idle	0x00000002	Sensor is stopped
Sensor_exposing	0x00000004	Sensor is exposing
Sensor_readout	0x00000008	Sensor is in readout state
	Bit4-31	Reserved

2.13.2 PCO_GetCmosLineTiming

Description:

This function returns the current settings of the ***Lightsheet scanning mode*** of the pco.edge.
If this mode is active, timing related settings from the functions in ***TIMING CONTROL*** are inactive and current delay and exposure times must be readout with function **PCO_GetCmosLineExposureDelay**.
The resulting exposure time in µs can be calculated as:
`int64 expos=dwExposureLines*dwLineTime*(1000^dwTimebase).`

Supported camera type:

pco.edge with Camera Link interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCmosLineTiming (
    HANDLE ph,                          //in
    WORD* wParameter,                   //out
    WORD* wTimebase,                    //out
    DWORD* dwLineTime,                  //out
    DWORD* dwReserved,                  //out
    WORD wReservedLen                   //in
    );
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wParameter	WORD*	Pointer to a WORD to receive the on/off state: • 0x0000 = [off] • 0x0001 = [on]
wTimebase	WORD*	Pointer to a WORD to receive the time base: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
dwLineTime	DWORD*	Pointer to a DWORD to receive the line time
dwReserved	DWORD*	Reserved
wReservedLen	WORD	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.13.3 PCO_SetCmosLineTiming

Description:

This function enables or disables the ***Lightsheet scanning mode*** of the pco.edge. In ***Lightsheet scanning mode*** exposure and delay time setting is correlated to the sensor line time. If this mode is enabled, timing related settings from the functions in ***TIMING CONTROL*** can not be used.
The setting of delay time and exposure time must be done with the function **PCO_SetCmosLineExposureDelay**.
The resulting exposure-time can be calculated as:
`dwExposureLines*dwLineTime*(1000^dwTimebase)`
The command is only available in operation mode ***Rolling Shutter***.
The preferred ***SCCMOS Readout format*** is `SCCMOS_FORMAT_TOP_BOTTOM`

Supported camera type:

pco.edge with Camera Link interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetCmosLineTiming (
    HANDLE ph,                          //in
    WORD wParameter,                    //in
    WORD wTimebase,                     //in
    DWORD dwLineTime,                   //in
    DWORD dwReserved,                   //in
    WORD wReservedLen                   //in
    );
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wParameter	WORD	WORD variable to set the on / off state: • 0x0000 = [off] • 0x0001 = [on]
wTimebase	WORD	WORD variable to set the time base: • 0x0000 = [ns] • 0x0001 = [µs] • 0x0002 = [ms]
dwLineTime	DWORD	DWORD variable to set the line time Minimum value is 17 µs @ 286 MHz (fast scan) 40 µs @ 95.3 MHz (slow scan) Maximum value is 100 ms
dwReserved	DWORD*	Reserved
wReservedLen	WORD	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.13.4 PCO_GetCmosLineExposureDelay

Description:

This function returns the current setting of the exposure and delay line settings for the ***Lightsheet scanning mode***. It is only available when ***Lightsheet scanning mode*** has been successfully enabled by **PCO_SetCmosLineTiming**.

Supported camera type:

pco.edge with Camera Link interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCmosLineExposureDelay (
    HANDLE ph,                          //in
    DWORD* dwExposureLines,             //out
    DWORD* dwDelayLines,                //out
    DWORD* dwReserved,                  //out
    WORD wReservedLen                   //in
    );
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwExposureLines	DWORD*	Pointer to a DWORD to receive the number of lines for exposure
dwDelayLines	DWORD*	Pointer to a DWORD to receive the number of lines for delay
dwReserved	DWORD*	Reserved
wReservedLen	WORD	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.13.5 PCO_SetCmosLineExposureDelay

Description:

This function sets the exposure and delay line settings for the ***Lightsheet scanning mode***. It is only available when Lightsheet scanning mode has been successfully enabled by **PCO_SetCmosLineTiming**.

Supported camera type:

pco.edge with Camera Link interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetCmosLineExposureDelay (
    HANDLE ph,                          //in
    DWORD dwExposureLines,              //in
    DWORD dwDelayLines,                 //in
    DWORD* dwReserved,                  //out
    WORD wReservedLen                   //in
    );
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwExposureLines	DWORD	DWORD variable to set the number of lines for exposure
dwDelayLines	DWORD	DWORD variable to set the number of lines for delay
dwReserved	DWORD*	Reserved
wReservedLen	WORD	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.13.6 PCO_SetTransferParametersAuto

Description:

This function can be used to setup the correct transfer parameters for a pco.edge 5.5 with Camera Link interface, when used in ***Rolling Shutter*** or ***Global Reset*** mode. The SDK automatically selects appropriate parameters for the transfer. Depending on the current camera settings either `PCO_CL_DATAFORMAT_5x16` or `PCO_CL_DATAFORMAT_5x12L` with the appropriate LUT is set. The current **SCCMOS Readout Format** remains as it is.
If a valid memory buffer is given as input, the current settings are returned as transfer parameter structure `PCO_SC2_CL_TRANSFER_PARAM`.

Supported camera type

pco.edge 5.5 with Camera Link interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetTransferParameterAuto (
    HANDLE ph,                      //in
    void* buffer,                   //in,out
    int ilen                        //in
);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`buffer`	`void*`	Pointer to a buffer to get the transfer parameter settings or NULL
`ilen`	`int`	Length of the buffer in bytes or 0

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.13.7 PCO_GetInterfaceOutputFormat

Description:

This function returns the current interface output format.
For the pco.edge the interface output format reflects the current setting of the **SCCMOS Readout Format** of the camera. An application note for further information is available on request.

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetInterfaceOutputFormat (
    HANDLE ph,                          //in
    WORD* wDestInterface,               //in
    WORD* wFormat,                      //out
    WORD* wReserved1,                   //out
    WORD* wReserved2                    //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wDestInterface	WORD*	Pointer to a WORD variable to set the interface to query: • 0x0002 = [edge]
wFormat	WORD*	Pointer to a WORD variable to get requested information
wReserved1	WORD*	Reserved (NULL pointer not allowed)
wReserved2	WORD*	Reserved (NULL pointer not allowed)

Parameter dependency:

None

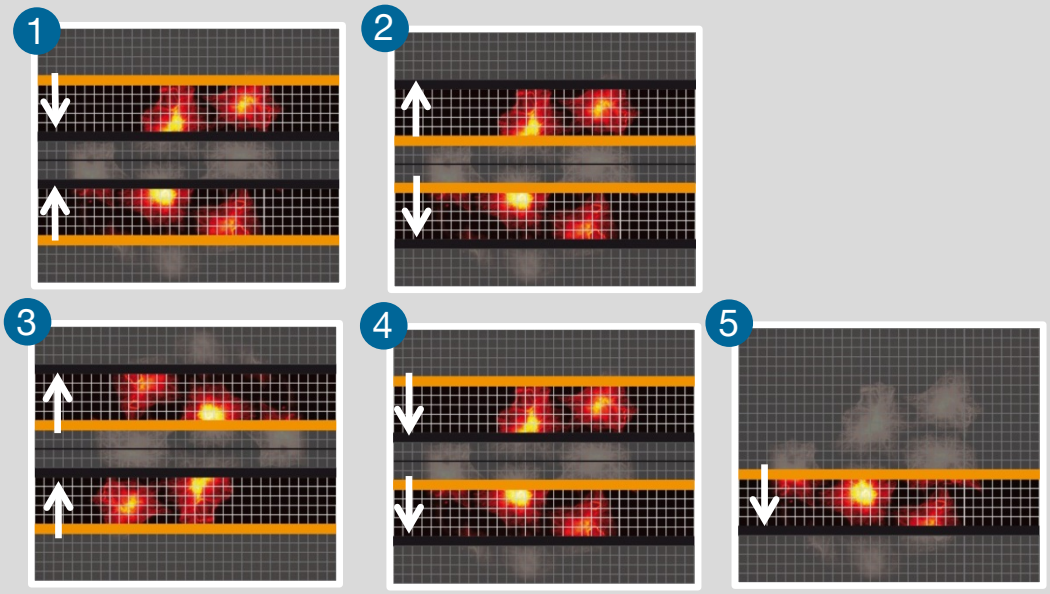
Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.13.7.1 SCCMOS Readout Format

	Value	Name	Description
1	0x0100	SCCMOS_FORMAT_TOP_CENTER_BOTTOM_CENTER	
2	0x0200	SCCMOS_FORMAT_CENTER_TOP_CENTER_BOTTOM	
3	0x0300	SCCMOS_FORMAT_CENTER_TOP_BOTTOM_CENTER	
4	0x0400	SCCMOS_FORMAT_TOP_CENTER_CENTER_BOTTOM	
5	0x0000	SCCMOS_FORMAT_TOP_BOTTOM	Linear readout

Five different readout modes are available for pco.edge in ***Rolling Shutter*** readout mode.
Standard mode is ***Dual Outside in***.
In ***Single Top down***, the pco.edge provides only half of the normal frame rate.



2.13.8 PCO_SetInterfaceOutputFormat

Description:

This function does set the current interface output format.
The interface output format reflects the currently **SCCMOS Readout Format** of the camera.
With the **SCCMOS Readout Format** the data readout direction of the camera can be controlled.
For all cameras with Camera Link interface it is recommended to use **PCO_SetTransferParameter** function instead of this function, because the driver layer must be informed about any changes in readout format to successfully rearrange the image data.

Supported camera type:

pco.edge

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetInterfaceOutputFormat (
    HANDLE ph,                          //in
    WORD wDestInterface,                //in
    WORD wFormat,                       //in
    WORD wReserved1,                    //in
    WORD wReserved2                     //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wDestInterface	WORD	WORD to set the interface to change: • 0x0002 = [edge]
wFormat	WORD	WORD to set format see table **SCCMOS Readout Format**
wReserved1	WORD	Reserved
wReserved2	WORD	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.14 SPECIAL COMMANDS PCO.DIMAX

Special commands for pco.dimax S/HD/HS and pco.dimax cs.

2.14.1 PCO_GetImageTransferMode

Description:

Current image transfer mode settings are queried from the camera and the **IMAGE_TRANSFER_MODE_PARAM structure** is filled with this information.

Supported camera type:

pco.dimax with GigE or USB interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetImageTransferMode (
    HANDLE ph,                          //in
    void *param,                        //out
    int ilen                            //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
param	void*	Pointer to a **IMAGE_TRANSFER_MODE_PARAM structure**
ilen	int	Length in bytes of the **IMAGE_TRANSFER_MODE_PARAM structure**

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.14.1.1 IMAGE_TRANSFER_MODE_PARAM structure

Name	Type	Description
wSize	WORD	Size of this struct
wMode	WORD	Transfer mode, e.g. full, scaled, cutout etc.
wImageWidth	WORD	Original image width
wImageHeight	WORD	Original image height
wTxWidth	WORD	Width of transferred image (scaled or cutout)
wTxHeight	WORD	Width of transferred image (scaled or cutout)
wParam	WORD[8]	Parameter meaning depends on selected mode set to zero if not used
ZZwDummy	WORD[10]	Reserved

2.14.1.2 Transfer Mode Definition

Name	Value	Description
IMAGE_TRANSFER_MODE_STANDARD	0x0000	Images as recorded from the camera
IMAGE_TRANSFER_MODE_SCALED_XY_8BIT	0x0001	Scaled image with TxWidth=ImageWidth/2 TxHeight=ImageHeight/2
IMAGE_TRANSFER_MODE_CUTOUT_XY_8BIT	0x0002	Region of image with TxWidth=ImageWidth/2 TxHeight=ImageHeight/2 Offset parameter are set with wParam
IMAGE_TRANSFER_MODE_FULL_RGB_24BIT	0x0003	Reserved, not implemented
IMAGE_TRANSFER_MODE_BIN_SCALED_8BIT_BW	0x0004	Binary scaled image
IMAGE_TRANSFER_MODE_BIN_SCALED_8BIT_COLOR	0x0005	Binary scaled and color transformed image
IMAGE_TRANSFER_MODE_TEST_ONLY	0x8000	Test image

2.14.1.3 Parameter Transfer Mode Cutout XY

Name	Type	Description
wParam[0]	WORD	Horizontal offset in pixel Valid range: 0 <= x <= ImageWidth/2
wParam[1]	WORD	Vertical offset in pixel Valid range: 0 <= x <= ImageHeight/2

2.14.1.4 Parameter Transfer Mode Scaled 8 bit

Name	Type	Description
wParam[0]	WORD	Scale factor of image Allowed values 1 / 2 / 4 / 8 / 16

2.14.2 PCO_SetImageTransferMode

Description:

This function does set the scaled image transfer mode of the camera. The image transfer mode can be used to transfer scaled images from the internal memory of the camera. With scaled image transfer the amount of data, which must be transferred for one image, is reduced and therefore the image display frequency can be enhanced. An application can use this mode for display of thumbnails or faster image preview, when camera setup is performed e.g. adjust and focus the camera lens.
The **IMAGE_TRANSFER_MODE_PARAM structure** must be filled with appropriate values.
Because scaled image transfer mode is a special mode inside the camera the size parameters must be set to the original camera recording size for the image allocation and acquisition functions like **PCO_AllocateBuffer**, **PCO_GetImageEx** or **PCO_AddBufferEx**.
When any size related camera settings are changed the ***PCO_SetImageTransferMode*** has to be called again in order to correctly calculate the transferred amount of data. Also when reading images from different Camera RAM segments, which have different image sizes, the ***PCO_SetImageTransferMode*** has to be called after selecting another segment and before reading the images.
Before an application is closed the scaled image transfer mode must be reset to standard mode.

Supported camera type:

pco.dimax with GigE or USB interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetImageTransferMode (
    HANDLE ph,                          //in
    void *param,                        //in
    int ilen                            //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
param	void*	Pointer to a **IMAGE_TRANSFER_MODE_PARAM structure**
ilen	int	Length in bytes of **IMAGE_TRANSFER_MODE_PARAM structure**

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.14.3 PCO_GetCDIMode

Description:

This function returns the current CDI (correlated double image) mode from the camera.

Supported camera type:

pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: CDI_MODE`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetCDIMode (
    HANDLE ph,                          //in
    WORD *wCDIMode                      //out
    );
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wCDIMode	WORD*	Pointer to a WORD to receive the current CDI mode setting: • 0x0000 = [CDI mode off] • 0x0001 = [CDI mode on]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.14.4 PCO_SetCDIMode

Description:

This function does set the CDI (correlated double image) mode in the camera.

Supported camera type:

pco.dimax

Descriptor dependency:

`dwGeneralCapsDESC1: CDI_MODE`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetCDIMode (
    HANDLE ph,                          //in
    WORD wCDIMode                       //in
    );
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wCDIMode	WORD	WORD variable to set the CDI mode • 0x0000 = [CDI mode off] • 0x0001 = [CDI mode on]

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.14.5 PCO_GetPowerSaveMode

Description:

This function returns the current power save mode from the camera.
The power save mode may be used for pco.dimax cameras with an external backup battery connected. Using the **PCO_SetPowerSaveMode** command the camera can be configured to change automatically into a special power save mode when the main power supply is disconnected or fails. The time how long the main power supply must be off until the camera changes into power save mode, can be set with the command. Note that the camera cannot be accessed by software when it is in power save, however the images recorded are kept over several hours. In order to get the camera back to normal operation, the main power supply has to be restored. An application note for further information is available on request.

Supported camera type:

pco.dimax

Descriptor dependency:

`wPowerDownModeDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetPowerSaveMode (
    HANDLE ph,                          //in
    WORD* wMode,                        //out
    WORD* wDelayMinutes                 //out
    );
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wMode`	`WORD*`	Pointer to a `WORD` variable to get the current power save mode: • `0x0000` = [off] default • `0x0001` = [on]
`wDelayMinutes`	`WORD*`	Pointer to a `WORD` variable to get the delay in minutes, after which the camera enters power save mode when main power is lost

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.14.6 PCO_SetPowerSaveMode

Description:

This function does set the power save mode of the camera.

Supported camera type:

pco.dimax

Descriptor dependency:

`wPowerDownModeDESC`

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetPowerSaveMode (
    HANDLE ph,                          //in
    WORD wMode,                         //in
    WORD wDelayMinutes                  //in
    );
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wMode`	`WORD`	`WORD` variable to set the power save mode: • `0x0000` = [off] default • `0x0001` = [on]
`wDelayMinutes`	`WORD*`	`WORD` variable to set the delay in minutes, after which the camera enters power save mode when main power is lost. The current switching delay is between `wDelayMinutes` and `wDelayMinutes` + 1. Valid range is from 1 to 60.

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.14.7 PCO_GetBatteryStatus

Description:

This function returns the current state of the battery package connected to the camera.

Supported camera type:

pco.dimax

Descriptor dependency:

wPowerDownModeDESC

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetBatteryStatus (
    HANDLE ph,                          //in
    WORD* wBatteryType,                 //out
    WORD* wBatteryLevel,                //out
    WORD* wPowerStatus,                 //out
    WORD* wReserved,                    //out
    WORD wNumReserved                   //in
    );
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wBatteryType	WORD*	Pointer to a WORD variable to get the battery type: • 0x0000 = no battery mounted • 0x0001 = nickel metal hydride type • 0x0002 = lithium ion type • 0x0003 = lithium iron phosphate type • 0x0004 = battery dimax cs • 0xFFFF = unknown battery type
wBatteryLevel	WORD*	Pointer to a WORD variable to get the charge condition of the battery calculated in percent
wPowerStatus	WORD*	Pointer to a WORD variable to get the overall power state: • 0x0001 = power supply is available • 0x0002 = battery mounted and detected • 0x0004 = battery is charged Bits can be combined e.g. 0x0003 means that camera has a battery and is running on external power, 0x0002: camera runs on battery
wReserved	WORD*	Reserved
wNumReserved	WORD	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.15 SPECIAL COMMANDS PCO.DIMAX WITH HD-SDI

Special commands for pco.dimax with HD-SDI interface.

2.15.1 PCO_GetInterfaceOutputFormat

Description:

This function returns the current interface output format. For the pco.dimax the interface output shows the selection of the active image streaming interface. If the interface format is set to [off], then image data will be transferred through the standard interface, e.g. GigE or USB. If the interface is set to any of the predefined HD-SDI modes a continuous image data stream is output on the HD-SDI connector and the current image size setting of the camera depend on the selected HD-SDI format. Setting of ROI is not possible when HD-SDI output is enabled.

Supported camera type:

pco.dimax with HD-SDI interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetInterfaceOutputFormat (
    HANDLE ph,                          //in
    WORD* wDestInterface,               //in
    WORD* wFormat,                      //out
    WORD* wReserved1,                   //out
    WORD* wReserved2                    //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wDestInterface	WORD*	Pointer to a WORD variable to set the interface to query: • 0x0001 = [HD-SDI]
wFormat	WORD*	Pointer to a WORD variable to get the interface format: • 0x0000 = [off] • see table **HD-SDI formats**
wReserved1	WORD*	Reserved (NULL pointer not allowed)
wReserved2	WORD*	Reserved (NULL pointer not allowed)

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.15.2 PCO_SetInterfaceOutputFormat

Description:

This function does set the interface output format of the pco.dimax; this will enable streaming through the active image streaming interface (→ HD-SDI interface).

If the interface format is set to [off], then image data will be transferred through the standard interface, e.g. GigE or USB. If the interface is set to any of the predefined HD-SDI modes a continuous image data stream is output on the HD-SDI connector and the current image size setting of the camera depend on the selected HD-SDI format. It is not possible to set a ROI, when HD-SDI output is enabled.

Supported camera type:

pco.dimax with HD-SDI interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetInterfaceOutputFormat (
    HANDLE ph,                          //in
    WORD wDestInterface,                //in
    WORD wFormat,                       //in
    WORD wReserved1,                    //in
    WORD wReserved2                     //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wDestInterface	WORD	WORD variable to set the interface to change: • 0x0001 = [HD-SDI]
wFormat	WORD	WORD variable to set the interface format: • 0x0000 = [off] • see table **HD-SDI formats**
wReserved1	WORD	Reserved must be set to 0
wReserved2	WORD	Reserved must be set to 0

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.15.2.1 HD-SDI formats

Name	Description
#define HDSDI_FORMAT_OUTPUT_OFF	0x0000
#define HDSDI_FORMAT_1080P25_SINGLE_LINK_RGB	0x0001
#define HDSDI_FORMAT_1080P50_DUAL_LINK_RGB	0x0003
#define HDSDI_FORMAT_1080P30_SINGLE_LINK_RGB	0x000B
#define HDSDI_FORMAT_1080P2997_SINGLE_LINK_RGB	0x000C
#define HDSDI_FORMAT_1080P24_SINGLE_LINK_RGB	0x000D
#define HDSDI_FORMAT_1080P2398_SINGLE_LINK_RGB	0x000E
#define HDSDI_FORMAT_720P24_SINGLE_LINK_RGB	0x0017
#define HDSDI_FORMAT_720P2398_SINGLE_LINK_RGB	0x0018

2.15.3 PCO_PlayImagesFromSegmentHDSDI

Description:

This function does setup the image output on the HD-SDI interface.
It is used to stream the recorded images from the camera internal memory (CamRAM) to the HD-SDI interface. The HD-SDI interface is an output only interface, therefore it does not request images, but it has to be supplied with a continuous data stream.
This function can only be used, if **PCO_SetStorageMode** is set to [recorder] and recording to the camera RAM segment is stopped.
If **PCO_PlayImagesFromSegmentHDSDI** is called, the sequence is started and the function returns immediately. Streaming time for the entire recorded sequence may take seconds or up to minutes depending on the choosen parameters.
The play speed is defined by the `wSpeed` parameter together with the `wMode` parameter:

- Fast forward: The play position is ***increased*** by `wSpeed`, so `(wSpeed-1)` images are leaped
- Fast rewind: The play position is ***decreased*** by `wSpeed`, so `(wSpeed-1)` images are leaped
- Slow forward: The current image is sent `wSpeed` times before the position is increased
- Slow rewind: The current image is sent `wSpeed` times before the position is decreased

With the play command parameters (e.g. `wSpeed`) can also be changed while a play is active. The parameters will be changed immediately. It is possible to change parameters like play speed or play direction without changing the current position by setting Start No. to -1 (as `DWORD 0xFFFFFFFF`).

Supported camera type:

pco.dimax with HD-SDI interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_PlayImagesFromSegmentHDSDI (
    HANDLE ph,                          //in
    WORD wSegment,                      //in
    WORD wInterface,                    //in
    WORD wMode,                         //in
    WORD wSpeed,                        //in
    DWORD dwRangeLow,                   //in
    DWORD dwRangeHigh,                  //in
    DWORD dwStartPos                    //in);
```

Parameter:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`wSegment`	`WORD`	Number of segment of the RAM segment to read from
`wInterface`	`WORD`	Select destination interface. Must be set to 0
`wMode`	`WORD`	`WORD` to set the play mode: • `0x0000` = Stop, switch data stream off • `0x0001` = Play (fast) forward • `0x0002` = Play (fast) backward (rewind) • `0x0003` = Play slow forward • `0x0004` = Play slow backward (rewind)

		Mode & 0x0100 = 0: At the end just repeat the last image (freeze image) Mode & 0x0100 = 1: At the end replay sequence from beginning Other values reserved for future modes
wSpeed	WORD	Either stepping (fast play mode) or repeat count (slow play mode)
dwRangeLow	DWORD	Lowest image number of range to be played
dwRangeHigh	DWORD	Highest image number of range to be played
dwStartPos	DWORD	Start with this image number or leave unchanged (-1)

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Some examples:

Assuming that a record to a segment has been finished and there are ***N*** images in the segment. The function **PCO_GetNumberOfImagesInSegment** can be used to query the current number of images ***N*** in a segment.

Desired function	Range Low	Range High	Start No.1	Speed	Mode
Play / Start complete sequence	1	N	1	1	0x0001
Fast Forward (speed x 10)	1	N	1	10	0x0001
Fast Rewind (speed x 10)	1	N	N	10	0x0002
Slow Forward (1/5th in speed)	1	N	N	5	0x0003
Slow Rewind (1/5th in speed)	1	N	N	5	0x0004
Cut out (starting with 1)	j ≥ 1	k ≤ N	1	1	0x0001
Cut out (starting with m)	j ≥ 1	k ≤ N	1...m...N	1	0x0001
Change Play Speed (to x 20)	1	N	-1	20	0x0001
Change Play Direction (to rewind)	1	N	-1	20	0x0003
Change current Play Position	1	N	1 ≤ p ≤ N	20	0x0001
Display image k as freezed image	1	N	k	0	0x0001
Switch HD/SDI off	0	0	0	0	0x0000

When changing the range and the current image position or the ***Start Number parameter*** is out of range, the position will be set to the following positions:

- Play forward: Range Low (with replay) or Range High (without replay)
- Play reverse: Range High (with replay) or Range Low (without replay)

Record frame rate and play frame rate:

Please note that the speed parameter does not depend on the recorded frame rate at all. Speed parameter 1 always means that the recorded images are sent one after another without leaps, as fast as possible for the selected interface and the selected format.

Thus if the record frame rate is 1000 fps and the output frame rate defined by the interface and the output format is 50 fps, it will result in a play speed which is 20 times slower than the record frame rate. So with speed parameter set to 1, the sequence will appear as a slow motion when played. To see the sequence as it really happened the speed parameter has to be set to 20.

2.15.4 PCO_GetPlayPositionHDSDI

Description:

The function **PCO_GetPlayPositionHDSDI** queries the current position of the play pointer of the currently started sequence. Due to time necessary for communication and processing of the command, the current pointer may be 1 or 2 steps images ahead at the time, when the function returns.

Supported camera type:

pco.dimax with HD-SDI interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetPlayPositionHDSDI (
    HANDLE ph,                          //in
    WORD *wStatus                       //out
    DWORD  *dwPlayPosition              //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wStatus	WORD*	Pointer to a WORD variable to get current play state: • 0x0000 = no play active or play has already stopped • 0x0001 = play is active
dwPlayPosition	DWORD*	Number of the image currently streamed to the HD-SDI interface. It is between range low and range high, as set by **PCO_PlayImagesFromSegmentHDSDI** Only valid, when sequence play is still active

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.15.5 PCO_GetColorSettings

Description:

Gets the current color convert parameters of the camera.

Supported camera type:

pco.dimax with HD-SDI interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetColorSettings (
    HANDLE ph,                          //in
    PCO_Image_ColorSet* strColorSet   //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strColorSet	PCO_Image_ColorSet*	Pointer to a **PCO_Image_ColorSet structure** to receive the color convert parameter

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.15.6 PCO_SetColorSettings

Description:

Sets the color convert parameters of the camera.

Supported camera type:

pco.dimax with HD-SDI interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetColorSettings (
    HANDLE ph,                          //in
    PCO_Image_ColorSet* strColorSet   //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
strColorSet	PCO_Image_ColorSet*	Pointer to a **PCO_Image_ColorSet structure** to set the color convert parameters

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.15.6.1 PCO_Image_ColorSet structure

Name	Type	Description
wSize	WORD	Size of this struct
sSaturation	SHORT	Saturation from -100 to 100, 0 is default
sVibrance	SHORT	Vibrance from -100 to 100, 0 is default
wColorTemp	WORD	Color temperature from 2000K to 20000K; can be used for manual white balance
sTint	SHORT	Tint from -100 to 100, 0 is default; can be used for manual white balance
wMulNormR	WORD	Not used, must be set to 0x8000
wMulNormG	WORD	Not used, must be set to 0x8000
wMulNormB	WORD	Not used, must be set to 0x8000
sContrast	SHORT	Contrast from -100 to 100, 0 is default; must be set to 0, if any of the LUT's is used
wGamma	WORD	Gamma in percent Valid range: 40 <= x <= 250, where 100 corresponds to the gamma value of 1.00; Not used 0, if any of the LUT's is used
wSharpFixed	WORD	0 = off, 100 = maximum
wSharpAdaptive	WORD	0 = off, 100 = maximum
wScaleMin	WORD	0 to 4095
wScaleMax	WORD	0 to 4095
wProcOptions	WORD	Processing options as bit mask: • 0x0001 = ColorRefine Filter On When the color refine filter is set to ON color artefacts from the debayering process are reduced
Lut-Selection	WORD	Selection of LUT, which is used as last step in the the color processing. Either predefined or user loadable LUT's can be selected. If a LUT is selected parameters sContrast and wGamma are not used for color processing. Parameter sContrast must be set to value 0. • 0x0000 = No LUT selected • 0x0001 - 0x001C (1-28) Select one of the user loadable LUT's. • 0x1001 = predefined LUT according REC709 definition • 0x1002 = predefined LUT according LOG90 algorithm.
ZZwDummy[92]	WORD	Reserved

2.15.7 PCO_DoWhiteBalance

Description:

This function does start a white balance calculation process. The function must only be called, when images are transmitted to the HD-SDI interface and one of the color formats is selected. The function does return immediately. The camera uses a 50% image region in the center of the image to calculate new values for `wColorTemp` and `sTint` of the **PCO_Image_ColorSet structure**.

Supported camera type:

pco.dimax with HD-SDI interface

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_DoWhiteBalance (
    HANDLE ph,                          //in
    WORD wMode,                         //in
    WORD* wParam,                       //in
    WORD wParamLen                      //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wMode	WORD	WORD variable: • 0x0001 = start white balance process
wParam	WORD*	Pointer to a WORD array for additional parameters. Not used at the moment
wParamLen	WORD	WORD variable which holds the number of entries in the wParam array

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.16 SPECIAL COMMANDS PCO.FLIM

Special commands for pco.flim camera system.

2.16.1 PCO_GetFlimModulationParameter

Description:

This function can be used to query the current modulation signal settings of the pco.flim.

Supported camera type

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFlimModulationParameter (
    HANDLE ph,                          //in
    WORD* wSourceSelect,                //out
    WORD* wOutputWaveform,              //out
    WORD* wReserved1,                   //out
    WORD* wReserved2                    //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSource Select	WORD*	Pointer to a WORD variable to receive the modulation source: • 0x0000 = [intern]: The modulation signal is generated internally by the camera. The camera acts as frequency master. • 0x0001 = [extern]: The camera is using an external clock source present at the modulation input <mod - in>. The camera acts as frequency slave.
wOutput Waveform	WORD*	Pointer to a WORD variable to receive the modulation waveform of the homodyne modulation signal: • 0x0000 = [none]: The modulation output <out - mod> is disabled • 0x0001 = [sinusoidal]: The modulation output <out - mod> is enabled and generates a sinusoidal waveform • 0x0002 = [rectangular]: The modulation output <out-mod> is enabled and generates a rectangular waveform
wReservedx	WORD*	Reserved

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.16.2 PCO_SetFlimModulationParameter

Description:

This function does set the modulation mode of the pco.flim. With this function the source of the modulation frequency signal can be selected and whether the signal is sent to the modulation output line <out – mod>. Furthermore the shape of the output waveform can be selected. No **PCO_ArmCamera** command is required to change these settings.

Supported camera type

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFlimModulationParameter (
    HANDLE ph,                          //in
    WORD wSourceSelect,                 //in
    WORD wOutputWaveform,               //in
    WORD wReserved1,                    //in
    WORD wReserved2                     //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wSource Select	WORD	WORD variable to set the modulation source: • 0x0000 = [intern] The modulation signal is generated internally by the camera. The camera acts as frequency master. The modulation frequency is set with function **PCO_SetFlimMasterModulationFrequency.** • 0x0001 = [extern] The camera is set to use an external clock source present at the modulation input <mod - in>. The input frequency has to be stable and within the valid frequency range (see pco.flim datasheet).
wOutput Waveform	WORD	WORD variable to set the modulation waveform: • 0x0000 = [none] The modulation output <out - mod> is disabled • 0x0001 = [sinusoidal] The modulation output <out - mod> is enabled and generates a sinusoidal waveform • 0x0002 = [rectangular] The modulation output <out - mod> is enabled and generates a rectangular waveform
wReserved	WORD	Reserved for future use, set to zero

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success
	`0x80001100`=PCO_ERROR_FIRMWARE_FLICAM_EXT_MOD_OUT_OF_RANGE If the modulation frequency of the external signal is out of range
	`0x80001101`=PCO_ERROR_FIRMWARE_FLICAM_SYNC_PLL_NOT_LOCKED If the camera can not lock its internal frequency to the modulation frequency of the external signal.
	any other value, see **ERROR / WARNING CODES**

2.16.3 PCO_GetFlimMasterModulationFrequency

Description:

This function can be used to query the current modulation frequency, which is used when the camera is configured as frequency master (see function **PCO_SetFlimModulationParameter**).

Supported camera type

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFlimMasterModulationFrequency (
    HANDLE ph,                          //in
    DWORD* dwFrequency                  //out
);
```

Parameters:

Name	Type	Description
`ph`	`HANDLE`	Handle to a previously opened camera device
`dwFrequency`	`DWORD*`	Pointer to a DWORD variable to receive the modulation frequency in units of Hertz (Hz)

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.16.4 PCO_SetFlimMasterModulationFrequency

Description:

This function does set the modulation frequency of the camera. The camera has to be configured as frequency master (see function **PCO_SetFlimModulationParameter**).
No **PCO_ArmCamera** is required to change this setting.

Supported camera type:

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFlimMasterModulationFrequency (
    HANDLE ph,                          //in
    DWORD dwFrequency                   //in
);
```

Parameters:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwFrequency	DWORD	DWORD variable to set the modulation frequency in units of Hertz (Hz) The specified value must be in the range from 0 Hz to 50 MHz

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.16.5 PCO_GetFlimPhaseSequenceParameter

Description:

Get configuration for the generation of phase image sequences. The combination of the described options determines the resulting length and sorting of ***phase sequences***. One ***phase sequence*** is a sequence of single (phase) images sampled at different points within the full modulation period of 360°. Because each parameter has an influence on each other, table **Image Sequences** should be used to determine the resulting image sequence.

Supported camera type:

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFlimPhaseSequenceParameter (
    HANDLE ph,                          //in
    WORD* wPhaseNumber,                 //out
    WORD* wPhaseSymmetry,               //out
    WORD* wPhaseOrder,                  //out
    WORD* wTapSelect,                   //out
    WORD* wReserved1,                   //out
    WORD* wReserved2                    //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wPhase Number	WORD*	Pointer to a WORD variable to receive the number of phases. Number of equidistant phases per modulation period of 360°, where following options are available: • 0x0000 = [manual shifting] This special mode is used in conjunction with the function **PCO_SetFlimRelativePhase**. The modulation period of 360° is divided into 2 phases, where o tap A carries the phase information at the relative phase *phi* and o tap B carries the phase information at the relative phase *phi*+180°. The relative phase *phi* can be adjusted using the function **PCO_SetFlimRelativePhase.** The relative phase *phi is* automatically set to zero when a **PCO_ArmCamera** is performed and [manual shifting] mode is selected. The options wPhaseSymmetry and wPhaseOrder have no effect in [manual shifting] mode. • 0x0001 = [2 phases] The modulation period of 360° is divided into 2 phases: 0° and 180°. • 0x0002 = [4 phases] The modulation period of 360° is divided into 4 phases: 0°, 90°, 180° and 270°.

		• 0x0003 = [8 phases] The modulation period of 360° is divided into 8 phases: 0°, 45°, 90°, 135°, 180°, 225°, 270° and 315°. • 0x0004 = [16 phases] The modulation period of 360° is divided into 16 phases: 0°, 22.5°, 45°, 67.5°, 90°, 112.5°, 135°, 157.5°, 180°, 202.5°, 225°, 247.5°, 270°, 292.5°, 315° and 337.5°.
wPhase Symmetry	WORD*	Pointer to a WORD variable to receive the phase symmetry. This parameter determines how the phase images are represented by tap A and tap B: • 0x0000 = [singular] The first half period of modulation (0° to 180°) is covered by tap A, whereas the second half period (180° to 360°) is covered by tap B. • 0x0001 = [twice] The complete modulation period of 360° is covered by both taps A and B, doubling the resulting number of single phase images per sequence.
wPhaseOrder	WORD*	Pointer to a WORD variable to receive the phase order. This parameter has only effect if wPhaseSymmetry = [twice]. While the taps A and B are always read-out alternately from the image sensor (the option wTapSelect determines which of them are output at the camera interface), the sorting of these phase image pairs (tap A and B) within a sequence is controlled by this parameter. • 0x0000 = [ascending] The phase-shifted phase image pairs (tap A and B) are within an ascending order. • 0x0001 = [opposite] The phase-shifted phase image pairs (tap A and B) are sorted in an opposite manner. Must be selected for asymmetry correction mode using the function **PCO_SetFlimImageProcessingFlow** with the parameter wAsymmetryCorrection = [average]
wTapSelect	WORD*	Pointer to a WORD variable to receive the tap selection. This parameter determines which taps (A and/or B) are output at the camera interface. (Since tap B carries the 180°-shifted information compared to tap A, it is also sometimes denoted as “tap 180”, whereas tap A is denoted as “tap 0”.) • 0x0000 = [both] Both taps A and B are output in the order A, B, A, B, … • 0x0001 = [tap A] Only tap A is output. • 0x0002 = [tap B] Only tap B is output.
wReserved1	WORD*	Reserved for future use, can be zero. Content will be set to zero
wReserved2	WORD*	Reserved for future use, can be zero. Content will be set to zero

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Examples:

Example 1:

- `wPhaseNumber` = [4 phases]
- `wPhaseSymmetry` = [twice]
- `wPhaseOrder` = [ascending]
- `wTapSelect` = [both]

```
PCO_GetFlimPhaseSequenceParameter(ph, 0x0002, 0x0001, 0x0000, 0x0000,);
```

The resulting phase image sequence out of the camera is:
0° (tap A), 180° (tap B), 90° (tap A), 270° (tap B), 180° (tap A), 0° (tap B), 270° (tap A), 90° (tap B)

Example 2:

- `wPhaseNumber` = [4 phases]
- `wPhaseSymmetry` = [twice]
- `wPhaseOrder` = [opposite]
- `wTapSelect` = [both]

```
PCO_GetFlimPhaseSequenceParameter(ph, 0x0002, 0x0001, 0x0001, 0x0000,);
```

The resulting phase image sequence out of the camera is:
0° (tap A), 180° (tap B), 180° (tap A), 0° (tap B), 90° (tap A), 270° (tap B), 270° (tap A), 90° (tap B)

Remark:

The current length of each phase image sequence depends on the options described above. The number given by `wPhaseNumber` is doubled if `wPhaseSymmetry` = [twice]. The current length is halved, if `wTapSelect` is configured to select only one tap (A or B) instead of both.
Furthermore, if the asymmetry correction mode is selected (see function **PCO_SetFlimImageProcessingFlow**), the current length is further halved.

2.16.6 PCO_SetFlimPhaseSequenceParameter

Description:

Set configuration for the generation of phase image sequences. The combination of the described options determines the resulting length and sorting of phase sequences. One phase sequence is a sequence of single (phase) images covering a modulation period of 360°. A **PCO_ArmCamera** is required to update these settings.

Supported camera type:

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFlimPhaseSequenceParameter (
    HANDLE ph,                          //in
    WORD wPhaseNumber,                  //in
    WORD wPhaseSymmetry,                //in
    WORD wPhaseOrder,                   //in
    WORD wTapSelect,                    //in
    WORD wReserved1,                    //in
    WORD wReserved2                     //in
);
```

Parameter:

See next page.

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wPhase Number	WORD	WORD variable to set the number of phases. Number of equidistant phases per modulation period of 360°, where following options are available: • 0x0000 = [manual shifting] This special mode is used in conjunction with the function **PCO_SetFlimRelativePhase**. The modulation period of 360° is divided into 2 phases, where o tap A carries the phase information at the relative phase *phi* and o tap B carries the phase information at the relative phase *phi*+180° The relative phase *phi* can be adjusted using the function **PCO_SetFlimRelativePhase.** The relative phase *phi is* automatically set to zero when a **PCO_ArmCamera** is performed and [manual shifting] mode is selected. The options wPhaseSymmetry and wPhaseOrder have no effect in [manual shifting] mode. • 0x0001 = [2 phases] The modulation period of 360° is divided into 2 phases: 0° and 180°. Depending on the option wPhaseSymmetry this phase information is carried by tap A and tap B as following: o wPhaseSymmetry = [singular] ▪ tap A carries the phase information: 0° ▪ tap B carries the phase information: 180° o wPhaseSymmetry = [twice] ▪ tap A carries the phase information: 0°, 180° ▪ tap B carries the phase information: 0°, 180° • 0x0002 = [4 phases] The modulation period of 360° is divided into 4 phases: 0°, 90°, 180° and 270°. Depending on the option wPhaseSymmetry this phase information is carried by tap A and tap B as following: o wPhaseSymmetry = [singular] ▪ tap A carries the phase information: 0°, 90° ▪ tap B carries the phase information: 180°, 270° o wPhaseSymmetry = [twice] ▪ tap A carries the phase information: 0°, 90°, 180°, 270° ▪ tap B carries the phase information: 0°, 90°, 180°, 270° • 0x0003 = [8 phases] The modulation period of 360° is divided into 8 phases: 0°, 45°, 90°, 135°, 180°, 225°, 270° and 315°. Depending on the option wPhaseSymmetry this phase information is carried by tap A and tap B as following: o wPhaseSymmetry = [singular] ▪ tap A carries the phase information: 0°, 45°, 90°, 135° ▪ tap B carries the phase information: 180°, 225°, 270°, 315° o wPhaseSymmetry = [twice] ▪ tap A carries the phase information: 0°, 45°, 90°, 135°, 180°, 225°, 270°, 315° ▪ tap B carries the phase information: 0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°

		• 0x0004 = [16 phases] The modulation period of 360° is divided into 16 phases: 0°, 22.5°, 45°, 67.5°, 90°, 112.5°, 135°, 157.5°, 180°, 202.5°, 225°, 247.5°, 270°, 292.5°, 315° and 337.5°. Depending on the option wPhaseSymmetry this phase information is carried by tap A and tap B as following: o wPhaseSymmetry = [singular] ▪ tap A carries the phase information: 0°, 22.5°, 45°, 67.5°, 90°, 112.5°, 135°, 157.5° ▪ tap B carries the phase information: 180°, 202.5°, 225°, 247.5°, 270°, 292.5°, 315°, 337.5° o wPhaseSymmetry = [twice] ▪ tap A carries the phase information: 0°, 22.5°, 45°, 67.5°, 90°, 112.5°, 135°, 157.5°, 180°, 202.5°, 225°, 247.5°, 270°, 292.5°, 315°, 337.5° ▪ tap B carries the phase information: 0°, 22.5°, 45°, 67.5°, 90°, 112.5°, 135°, 157.5°, 180°, 202.5°, 225°, 247.5°, 270°, 292.5°, 315°, 337.5°
wPhase Symmetry	WORD	WORD variable to set the phase symmetry. This parameter determines how the phase images are represented by tap A and tap B: • 0x0000 = [singular] The first half period of modulation (0° to 180°) is covered by tap A, whereas the second half period (180° to 360°) is covered by tap B. • 0x0001 = [twice] The complete modulation period of 360° is covered by both taps A and B, doubling the resulting number of single phase images per sequence
wPhase Order	WORD	WORD variable to set the phase order. This parameter has only effect if wPhaseSymmetry = [twice]. While the taps A and B are always read-out alternately from the image sensor (the option wTapSelect determines which of them are output at the camera interface), the sorting of these phase image pairs (tap A and B) within a sequence is controlled by this parameter. • 0x0000 = [ascending] The phase-shifted phase image pairs (tap A and B) are within an ascending order. • 0x0001 = [opposite] The phase-shifted phase image pairs (tap A and B) are sorted in an opposite manner. Must be selected for asymmetry correction mode using the function **PCO_SetFlimImageProcessingFlow** with the parameter wAsymmetryCorrection = [average]
wTap Select	WORD	WORD variable to set the tap selection This parameter determines which taps (A and/or B) are output at the camera interface. (Since tap B carries the 180°-shifted information compared to tap A, it is also sometimes denoted as “tap 180”, whereas tap A is denoted as “tap 0”.) • 0x0000 = [both] Both taps A and B are output in the order A, B, A, B, … • 0x0001 = [tap A] Only tap A is output. • 0x0002 = [tap B] Only tap B is output
wReserved	WORD	Reserved for future use, set to zero

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

Examples:

Example 1:

- `wPhaseNumber` = [4 phases]
- `wPhaseSymmetry` = [twice]
- `wPhaseOrder` = [ascending]
- `wTapSelect` = [both]

```
PCO_SetFlimPhaseSequenceParameter(ph, 0x0002, 0x0001, 0x0000, 0x0000,);
```

The resulting phase image sequence out of the camera is:
0° (tap A), 180° (tap B), 90° (tap A), 270° (tap B), 180° (tap A), 0° (tap B), 270° (tap A), 90° (tap B)

Example 2:

- `wPhaseNumber` = [4 phases]
- `wPhaseSymmetry` = [twice]
- `wPhaseOrder` = [opposite]
- `wTapSelect` = [both]

```
PCO_SetFlimPhaseSequenceParameter(ph, 0x0002, 0x0001, 0x0001, 0x0000,);
```

The resulting phase image sequence out of the camera is:
0° (tap A), 180° (tap B), 180° (tap A), 0° (tap B), 90° (tap A), 270° (tap B), 270° (tap A), 90° (tap B)

Remark:
The current length of each phase image sequence depends on the options described above. The number given by `wPhaseNumber` is doubled if `wPhaseSymmetry` = [twice]. The current length is halved, if `wTapSelect` is configured to select only one tap (A or B) instead of both.
Furthermore, if the asymmetry correction mode is selected (see function **PCO_SetFlimImageProcessingFlow**), the current length is further halved.

2.16.7 PCO_GetFlimRelativePhase

Description:

This function can be used to query the current relative phase setting, which is used when the camera is configured for manual phase shifting.
See function **PCO_SetFlimPhaseSequenceParameter**.

Supported camera type:

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFlimRelativePhase (
    HANDLE ph,                          //in
    DWORD* dwPhaseMilliDeg              //out
);
```

Parameters:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwPhaseMilliDeg	DWORD*	Pointer to a DWORD variable to receive the relative phase in units of millidegrees. The returned value has to be divided by 1000 to obtain the relative phase in units of degrees

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.16.8 PCO_SetFlimRelativePhase

Description:

This function does set the relative phase value, if the camera is configured for manual phase shifting (see function **PCO_SetFlimPhaseSequenceParameter**).
No **PCO_ArmCamera** is required to change the relative phase setting, but the relative phase value is reset to zero, when the current mode is set to [manual shifting] and a **PCO_ArmCamera** is performed.

Supported camera type:

pco.flim

Descriptor dependency:

None

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFlimRelativePhase (
    HANDLE ph,                          //in
    DWORD dwPhaseMilliDeg               //in,out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
dwPhaseMilliDeg	DWORD	DWORD variable to set the relative phase in units of millidegrees. Valid range: 0 <= x < 360000, where 1000 corresponds to the relative phase value of 1.000 degrees

Parameter dependency:

None

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.16.9 PCO_GetFlimImageProcessingFlow

Description:

Get settings of pco.flim specific internal image processing inside the camera.

Supported camera type:

pco.flim

Descriptor dependency:

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_GetFlimImageProcessingFlow (
    HANDLE ph,                          //in
    WORD* wAsymmetryCorrection,         //out
    WORD* wCalculationMode,             //out
    WORD* wReferencingMode,             //out
    WORD* wThresholdLow,                //out
    WORD* wThresholdHigh,               //out
    WORD* wOutputMode,                  //out
    WORD* wReserved1,                   //out
    WORD* wReserved2,                   //out
    WORD* wReserved3,                   //out
    WORD* wReserved4                    //out
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wAsymmetryCorrection	WORD*	Pointer to a WORD variable to receive the asymmetry correction mode. This parameter can be used to perform an asymmetry correction of taps A and B due to dynamic imbalances of their responsivities. This is done by the computation of the arithmetic mean of tap A and tap B, both carrying the same phase information. To use that mode, the function **PCO_SetFlimPhaseSequenceParameter** has to be called with the parameters wPhaseSymmetry = [twice], wPhaseOrder = [opposite] and wTapSelect = [both] with wPhaseNumber other than [manual shifting]. • 0x0000 = [off] Asymmetry correction mode is disabled. • 0x0001 = [average] Asymmetry correction mode using the arithmetic mean is enabled.
wCalculationMode	WORD*	Reserved. Content will be set to zero
wReferencingMode	WORD*	Reserved. Content will be set to zero
wThresholdLow	WORD*	Reserved. Content will be set to zero
wThresholdHigh	WORD*	Reserved. Content will be set to zero
wOutputMode	WORD*	Pointer to a WORD variable to receive the output mode • 0x0001 = [multiply x2] If this flag is set, the pixel values of the phase images are multiplied by two to virtually reach saturation earlier.
wReservedx	WORD*	Reserved for future use, can be NULL. Content will be set to zero

Parameter dependency:

None

Return value:

`int ErrorMessage`	0 in case of success else less than 0, see **ERROR / WARNING CODES**

2.16.10 PCO_SetFlimImageProcessingFlow

Description:

Configure parameters of pco.flim specific internal image processing inside the camera. A **PCO_ArmCamera** command is required to update these settings.

Supported camera type:

pco.flim

Descriptor dependency:

Prototype:

```
SC2_SDK_FUNC int WINAPI PCO_SetFlimImageProcessingFlow (
    HANDLE ph,                          //in
    WORD wAsymmetryCorrection,          //in
    WORD wCalculationMode,              //in
    WORD wReferencingMode,              //in
    WORD wThresholdLow,                 //in
    WORD wThresholdHigh,                //in
    WORD wOutputMode,                   //in
    WORD wReserved1,                    //in
    WORD wReserved2,                    //in
    WORD wReserved3,                    //in
    WORD wReserved4                     //in
);
```

Parameter:

Name	Type	Description
ph	HANDLE	Handle to a previously opened camera device
wAsymmetryCorrection	WORD	WORD variable to set the asymmetry correction mode. This parameter can be used to perform an asymmetry correction of taps A and B due to dynamic imbalances of their responsivities. This is done by the computation of the arithmetic mean of tap A and tap B, both carrying the same phase information. To use that mode, the function **PCO_SetFlimPhaseSequenceParameter** has to be called with the parameters wPhaseSymmetry = [twice], wPhaseOrder = [opposite] and wTapSelect = [both] with wPhaseNumber other than [manual shifting]. • 0x0000 = [off] Asymmetry correction mode is disabled • 0x0001 = [average] Asymmetry correction mode using the arithmetic mean is enabled
wCalculationMode	WORD	Reserved for future use, set to zero
wReferencingMode	WORD	Reserved for future use, set to zero
wThresholdLow	WORD	Reserved for future use, set to zero
wThresholdHigh	WORD	Reserved for future use, set to zero
wOutputMode	WORD	WORD variable to set the output mode • 0x0001 = [multiply x2] If this flag is set, the pixel values of the phase images are multiplied by two to virtually reach saturation earlier
wReservedx	WORD	Reserved for future use, set to zero

Parameter dependency:

None

Example:

- wPhaseNumber = [4 phases]
- wPhaseSymmetry = [twice]
- wPhaseOrder = [opposite]
- wTapSelect = [both]
- wAsymmetryCorrection = [average]

The resulting phase image sequence out of the camera is:
0° (mean tap A & B), 180° (mean tap A & B), 90° (mean tap A & B), 270° (mean tap A & B)

Return value:

int ErrorMessage	0 in case of success else less than 0, see **ERROR / WARNING CODES**

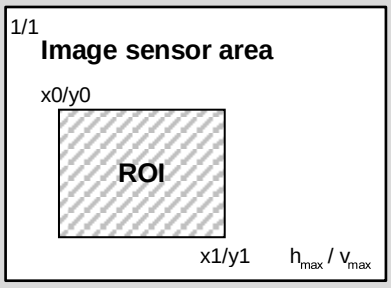
2.16.11 Image Sequences

wPhase Number	wPhase Symmetry	wPhase Order	wTap Select	wAsymmetry Correction	Resulting Sequence	Sequence Length
[manual]	don't care	don't care	[both]	[off]	phi (A), phi+180° (B)	2
[manual]	don't care	don't care	[tap A]	[off]	phi (A)	1
[manual]	don't care	don't care	[tap B]	[off]	phi+180° (B)	1
[2 phases]	[singular]	don't care	[both]	[off]	0° (A), 180° (B)	2
[2 phases]	[singular]	don't care	[tap A]	[off]	0° (A)	1
[2 phases]	[singular]	don't care	[tap B]	[off]	180° (B)	1
[2 phases]	[twice]	don't care	[both]	[off]	0° (A), 180° (B), 180° (A), 0° (B)	4
[2 phases]	[twice]	don't care	[tap A]	[off]	0° (A), 180° (A)	2
[2 phases]	[twice]	don't care	[tap B]	[off]	180° (B), 0° (B)	2
[2 phases]	[twice]	[opposite]	[both]	[average]	0° (AB), 180° (AB)	2
[4 phases]	[singular]	don't care	[both]	[off]	0° (A), 180° (B), 90° (A), 270° (B)	4
[4 phases]	[singular]	don't care	[tap A]	[off]	0° (A), 90° (A)	2
[4 phases]	[singular]	don't care	[tap B]	[off]	180° (B), 270° (B)	2
[4 phases]	[twice]	[ascending]	[both]	[off]	0° (A), 180° (B), 90° (A), 270° (B), 180° (A), 0° (B), 270° (A), 90° (B)	8
[4 phases]	[twice]	[ascending]	[tap A]	[off]	0° (A), 90° (A), 180° (A), 270° (A)	4
[4 phases]	[twice]	[ascending]	[tap B]	[off]	180° (B), 270° (B), 0° (B), 90° (B)	4
[4 phases]	[twice]	[opposite]	[both]	[off]	0° (A), 180° (B), 180° (A), 0° (B), 90° (A), 270° (B), 270° (A), 90° (B)	8
[4 phases]	[twice]	[opposite]	[both]	[average]	0° (AB), 180° (AB), 90° (AB), 270° (AB)	4
[4 phases]	[twice]	[opposite]	[tap A]	[off]	0° (A), 180° (A), 90° (A), 270° (A)	4
[4 phases]	[twice]	[opposite]	[tap B]	[off]	180° (B), 0° (B), 270° (B), 90° (B)	4
[8 phases]	[singular]	don't care	[both]	[off]	0° (A), 180° (B), 45° (A), 225° (B), 90° (A), 270° (B), 135° (A), 315° (B)	8
[8 phases]	[singular]	don't care	[tap A]	[off]	0° (A), 45° (A), 90° (A), 135° (A)	4
[8 phases]	[singular]	don't care	[tap B]	[off]	180° (B), 225° (B), 270° (B), 315° (B)	4
[8 phases]	[twice]	[ascending]	[both]	[off]	0° (A), 180° (B), 45° (A), 225° (B), 90° (A), 270° (B), 135° (A), 315° (B), 180° (A), 0° (B), 225° (A), 45° (B), 270° (A), 90° (B), 315° (A), 135° (B)	16
[8 phases]	[twice]	[ascending]	[tap A]	[off]	0° (A), 45° (A), 90° (A), 135° (A), 180° (A), 225° (A), 270° (A), 315° (A)	8
[8 phases]	[twice]	[ascending]	[tap B]	[off]	180° (B), 225° (B), 270° (B), 315° (B), 0° (B), 45° (B), 90° (B), 135° (B)	8
[8 phases]	[twice]	[opposite]	[both]	[off]	0° (A), 180° (B), 180° (A), 0° (B), 45° (A), 225° (B), 225° (A), 45° (B), 90° (A), 270° (B), 270° (A), 90° (B), 135° (A), 315° (B), 315° (A), 135° (B)	16
[8 phases]	[twice]	[opposite]	[both]	[average]	0° (AB), 180° (AB), 45° (AB), 225° (AB), 90° (AB), 270° (AB), 135° (AB), 315° (AB)	8
[8 phases]	[twice]	[opposite]	[tap A]	[off]	0° (A), 180° (A), 45° (A), 225° (A), 90° (A), 270° (A), 135° (A), 315° (A)	8
[8 phases]	[twice]	[opposite]	[tap B]	[off]	180° (B), 0° (B), 225° (B), 45° (B), 270° (B), 90° (B), 315° (B), 135° (B)	8
[16 phases]	[singular]	don't care	[both]	[off]	0° (A), 180° (B), 22.5° (A), 202.5° (B), 45° (A), 225° (B), 67.5° (A), 247.5° (B), 90° (A), 270° (B), 112.5° (A), 292.5° (B), 135° (A), 315° (B), 157.5° (A), 337.5° (B)	16
[16 phases]	[singular]	don't care	[tap A]	[off]	0° (A), 22.5° (A), 45° (A), 67.5° (A), 90° (A), 112.5° (A), 135° (A), 157.5° (A)	8

[16 phases]	[singular]	don't care	[tap B]	[off]	180° (B), 202.5° (B), 225° (B), 247.5° (B), 270° (B), 292.5° (B), 315° (B), 337.5° (B)	8
[16 phases]	[twice]	[ascending]	[both]	[off]	0° (A), 180° (B), 22.5° (A), 202.5° (B), 45° (A), 225° (B), 67.5° (A), 247.5° (B), 90° (A), 270° (B), 112.5° (A), 292.5° (B), 135° (A), 315° (B), 157.5° (A), 337.5° (B), 180° (A), 0° (B), 202.5° (A), 22.5° (B), 225° (A), 45° (B), 247.5° (A), 67.5° (B), 270° (A), 90° (B), 292.5° (A), 112.5° (B), 315° (A), 135° (B), 337.5° (A), 157.5° (B)	32
[16 phases]	[twice]	[ascending]	[tap A]	[off]	0° (A), 22.5° (A), 45° (A), 67.5° (A), 90° (A), 112.5° (A), 135° (A), 157.5° (A), 180° (A), 202.5° (A), 225° (A), 247.5° (A), 270° (A), 292.5° (A), 315° (A), 337.5° (A)	16
[16 phases]	[twice]	[ascending]	[tap B]	[off]	180° (B), 202.5° (B), 225° (B), 247.5° (B), 270° (B), 292.5° (B), 315° (B), 337.5° (B), 0° (B), 22.5° (B), 45° (B), 67.5° (B), 90° (B), 112.5° (B), 135° (B), 157.5° (B)	16
[16 phases]	[twice]	[opposite]	[both]	[off]	0° (A), 180° (B), 180° (A), 0° (B), 22.5° (A), 202.5° (B), 202.5° (A), 22.5° (B), 45° (A), 225° (B), 225° (A), 45° (B), 67.5° (A), 247.5° (B), 247.5° (A), 67.5° (B), 90° (A), 270° (B), 270° (A), 90° (B), 112.5° (A), 292.5° (B), 292.5° (A), 112.5° (B), 135° (A), 315° (B), 315° (A), 135° (B), 157.5° (A), 337.5° (B), 337.5° (A), 157.5° (B)	32
[16 phases]	[twice]	[opposite]	[both]	[average]	0° (AB), 180° (AB), 22.5° (AB), 202.5° (AB), 45° (AB), 225° (AB), 67.5° (AB), 247.5° (AB), 90° (AB), 270° (AB), 112.5° (AB), 292.5° (AB), 135° (AB), 315° (AB), 157.5° (AB), 337.5° (AB)	16
[16 phases]	[twice]	[opposite]	[tap A]	[off]	0° (A), 180° (A), 22.5° (A), 202.5° (A), 45° (A), 225° (A), 67.5° (A), 247.5° (A), 90° (A), 270° (A), 112.5° (A), 292.5° (A), 135° (A), 315° (A), 157.5° (A), 337.5° (A)	16
[16 phases]	[twice]	[opposite]	[tap B]	[off]	180° (B), 0° (B), 202.5° (B), 22.5° (B), 225° (B), 45° (B), 247.5° (B), 67.5° (B), 270° (B), 90° (B), 292.5° (B), 112.5° (B), 315° (B), 135° (B), 337.5° (B), 157.5° (B)	16

3. IMAGE AREA SELECTION (ROI)

In addition to common settings like exposure time and trigger modes the ***PCO cameras*** offer a wide range of parameter settings to adapt the camera best for the requirements of the application. One of the main features is that an image readout area can be set, which does reduce the amount of data which must be handled from the interface and the application and therefore does increase the usable frame rate.
Because the image readout area can be set in different ways and some parameters interact with others and all depend on camera constraints the following overview does show what must be considered to get the correct settings.
Because the camera constraints differ from model to model, the exact description can be loaded from the camera with **PCO_GetCameraDescription** to the **PCO_Description structure**.
All listet parameters can be found in this structure.



First of all the image sensor area is defined through the imaging sensor itself and its area of effective pixel rows and columns. Most of the sensors have additional lines and/or rows with dark reference and dummy pixels. Two different formats can be selected with the function **PCO_SetSensorFormat**. Format [standard] does use all effective pixels. Format [extended] can either define full sensor with the additional dark pixels or a smaller region of the sensor for cameras which don't support ROI settings otherwise.

The size of the active image area in format [standard] is defined through parameters `wMaxHorzResStdDESC` and `wMaxVertResStdDESC` in **PCO_Description structure**.
The size of the active image area in format [extended] is defined through parameters `wMaxHorzResExtDESC` and `wMaxVertResExtDESC` in **PCO_Description structure**.

Setting a binning value **PCO_SetBinning** does reduce the active image area by the factor of the binning. In example: setting binning 2x2 for a sensor with standard resolution 1600x1200 will result in an active image area of 800x600. Possible binning values in horizontal and vertical direction are specified through parameters `wMaxBinHorzDESC`, `wBinHorzSteppingDESC`, `wMaxBinVertDESC`, `wBinVertSteppingDESC`.

With **PCO_SetROI** a ROI (Region of interest) can be selected out of this active image area.
Minimum limits for the ROI settings are defined through parameters `wMinSizeHorzDESC` and `wMinSizeVertDESC`. The maximum limits are predetermined through the active sensor area, which depend on the settings of ***format*** and ***binning.***
Additional restrictions exist see **CAMERA CONSTRAINTS**.

If SoftROI (**PCO_EnableSoftROI,** only available for Camera Link mE IV interface) is disabled or not available several restrictions of the camera must be respected to find accurate ROI settings.
If SoftROI is enabled any value inside the active image area can be given for the ROI setting. But the ROI setting does not influence the frame rate in any case, because the restrictions on the camera remain. The function will set the camera ROI best possible, but some overhead may occure.

Valid values for the ROI setting of the first top-left pixel are from 1 up to limit – stepping+1.
Valid values for the ROI setting of the last bottom right pixel are from 1+stepping up to limit.

Wrong settings do not produce an error immediately, but the next **PCO_ArmCamera** will not succeed, because ROI settings will be validated from the camera in this command. After a successful ***Arm*** additional actions must be done.

- **PCO_SetImageParameters** must be called. This is ***mandatory*** for cameras with Camera Link, CLHS and GigE interface and recommended for all others.
- For pco.edge 5.5 cameras with Camera Link interface running in Rolling Shutter mode **PCO_SetTransferParameter** and **PCO_SetActiveLookuptable** must be called, followed by an additional **PCO_ArmCamera** call.
  For all other cameras **PCO_SetTransferParameter** is optional.
- Sizes of previous allocated buffers must be changed.

3.1 CAMERA CONSTRAINTS

No ROI allowed

- if parameter value `wRoiHorStepsDESC` and `wRoiVertStepsDESC` are zero

Symetrical requirements:

- according to the flags `ROI_VERT_SYMM_TO_HORZ_AXIS` and `ROI_VERT_SYMM_TO_VERT_AXIS` in parameter `dwGeneralCapsDESC1`
- for dual ADC mode the horizontal ROI must be symmetrical
- for a pco.dimax the horizontal and vertical ROI must be symmetrical
- for a pco.edge the vertical ROI must be symmetrical
  (despite in readout mode `SCCMOS_FORMAT_TOP_BOTTOM`)

Stepping requirement:

- according to the parameters `wRoiHorStepsDESC` and `wRoiVertStepsDESC`

Additional note for pco.edge 4.2 with Camera Link interface:

The image sensor area of the SCMOS sensor is 2048x2048, but the transmission over the Camera Link interface does only allow a horizontal stepping of 20 pixels. So without SoftROI the Region of interest can be set to either 2060 (does include 12 dark pixels) or 2040 or below getting not the full effective pixel area. When SoftRoi is selected a horizontal ROI of 2048 pixels can be set to ensure that the grabbed image does only consist of effective pixels.

4. TYPICAL IMPLEMENTATION

4.1 Basic Handling

This typical step by step implementation shows the basic handling:

1) Declarations:

- PCO_General strGeneral;
- PCO_CameraType strCamType;
- PCO_Sensor strSensor;
- PCO_Description strDescription;
- PCO_Timing strTiming;
- PCO_Storage strStorage;
- PCO_Recording strRecording;

2) Set all buffer 'size' parameters to the expected values:

- strGeneral.wSize = sizeof(strGeneral);
  strGeneral.strCamType.wSize = sizeof(strGeneral.strCamType);
  strCamType.wSize = sizeof(strCamType);
- strSensor.wSize = sizeof(strSensor);
  strSensor.strDescription.wSize = sizeof(strSensor.strDescription);
  strSensor.strDescription2.wSize = sizeof(strSensor.strDescription2);
  strDescription.wSize = sizeof(strDescription);
  strDescription2.wSize = sizeof(strDescription2);
- strTiming.wSize = sizeof(strTiming);
  strStorage.wSize = sizeof(strStorage);
  strRecording.wSize = sizeof(strRecording);

3) Open the camera and fill the structures:

- **PCO_OpenCamera**(&hCam, iBoardNumber)
- **PCO_GetGeneral**(hCam, &strGeneral)
- **PCO_GetCameraType**(hCam, &strCamType)
- **PCO_GetSensorStruct**(hCam, &strSensor)
- **PCO_GetCameraDescription**(hCam, &strDescription)
- **PCO_GetTimingStruct**(hCam,&strTiming)
- **PCO_GetRecordingStruct**(hCam, &strRecording)

4) Set camera settings (exposure, modes, etc.) and sizes (binning, ROI, etc.).

5) Arm the camera:

- **PCO_ArmCamera**(hCam)
- **PCO_GetCameraHealthStatus**(hCam, &dwWarn, &dwError, &dwStatus)

6) Get the current sizes and allocate one or more buffer:

- **PCO_GetSizes**(hCam, &actsizex, &actsizey, &ccdsizex, &ccdsizey)
- **PCO_AllocateBuffer**(hCam, &bufferNr, actsizex * actsizey * sizeof(WORD), &data, &hEvent)
- **PCO_SetImageParameters**(actsizex, actsizey ,IMAGEPARAMETERS_READ_WHILE_RECORDING, NULL,0);
  Mandatory for Camera Link,CLHS and GigE interface for others recommended.

7) Set the recording state to 'Recording' and add the allocated buffer(s):

- **PCO_SetRecordingState**(hCam, 0x0001);
- **PCO_AddBufferEx**(hCam, 0, 0, bufferNr, actualsizex, actualsizey, bitres);

8) Access the image data through the pointer returned from AllocateBuffer:

- Call WaitForSingleObject/ WaitforMultipleObjects or poll the buffer status.
- Do a convert and show the image.
- Call ResetEvent for a manual reset event before doing an AddBuffer.

9) Stop the camera:

- **PCO_CancelImages**(hCam);
- **PCO_SetRecordingState**(hCam, 0x0000);

10) If internal camera memory (CamRAM) is available images can be readout from the camera:

- **PCO_GetNumberOfImagesInSegment**(hCam, wActSeg, &dwValidImageCnt, &dwMaxImageCnt);
- **PCO_GetImageEx**(hCam, wActSeg, dw1stImage, dwLastImage, bufferNr, actualsizex, actualsizey, bitres)

11) Free allocated buffers and close the camera.

- **PCO_FreeBuffer**(hCamera, sBufNr);
- **PCO_CloseCamera**(hCamera)

4.1.1 Short code discussion

PCO_SetRecordingState: enables recording of images, depending on the trigger mode. If trigger mode is 0 (auto) and acquire mode is 0 (auto) images are transferred automatically to the camera ram.

PCO_AddBufferEx: moves a buffer to the driver queue (set firstimage=lastimage=0 while record is on), in order to transfer the most recent recorded image to the pc. At least two buffers must be used to transfer images with maximum possible performance (depending on the interface).

PCO_GetBufferStatus: gives further information about success or error states.

WaitForMultipleObjects (Windows function)**:** waits until one or multiple buffer event handles are in a signaled state. If waiting was successful and the buffer state shows successful transfer, the data in the buffer can be used in other processing functions. After processing is finished, the buffer can be added again to the driver queue.

PCO_CancelImages: must be called to remove all pending buffers from the driver queue. It also does clear resources in the driver and camera, to get a clean state for further image transfers.

PCO_SetRecordingState: to zero stops recording. The image sensor inside the camera is read out completely and set to idle

Some pitfalls:

- `wSize` is not set. ***Do not forget to set all*** wSize parameters
- Segment index is zero: The segment parameter is 1 based, whereas all structure reflections are zero based, e.g. `dwRamSegSize[0]` is the size of segment 1
- The user calls **PCO_GetImageEx** with `dw1stImage number 0`. If the user wants to access the first image inside the camera, set the image parameter to 1. Access to the camera is 1 based!
- The minimum segment size has to be at least two images

4.2 Example 'Get Single Images from running Camera'

```
#include "pco_err.h"
#include "sc2_SDKStructures.h"
#include "SC2_SDKAddendum.h"
#include "SC2_CamExport.h"
#include "SC2_Defs.h"

#ifdef _FILEFUNCTION_
char file_name[50];
#include "../file12.h"
#include "../file12.cpp"
#endif

void print_transferpar(HANDLE cam);

int main(int argc, char* argv[])
{
  int iRet;
  HANDLE cam;
  HANDLE BufEvent;
  short BufNum;
  WORD *BufAdr;

  PCO_Description strDescription;
  WORD RecordingState;

  printf("Get Handle to connected camera\n");
  iRet = PCO_OpenCamera(&cam, 0);
  if(iRet != PCO_NOERROR)
  {
     printf("No camera found\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     return -1;
  }

  strDescription.wSize=sizeof(PCO_Description);
  iRet = PCO_GetCameraDescription(cam,&strDescription);

  iRet = PCO_GetRecordingState(cam, &RecordingState);
  if(RecordingState)
  {
     iRet = PCO_SetRecordingState(cam, 0);
  }

//set camera to default state
  iRet = PCO_ResetSettingsToDefault(cam);

#ifdef _FILEFUNCTION_
  iRet = PCO_SetTimestampMode(cam,TIMESTAMP_MODE_BINARYANDASCII);
#endif

  iRet = PCO_ArmCamera(cam);

  DWORD CameraWarning, CameraError, CameraStatus;
  iRet = PCO_GetCameraHealthStatus(cam, &CameraWarning, &CameraError, &CameraStatus);
  if(CameraError!=0)
  {
     printf("Camera has ErrorStatus\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     iRet = PCO_CloseCamera(cam);
     return -1;
  }

  print_transferpar(cam);
```

```
  WORD XResAct, YResAct, XResMax, YResMax;
  DWORD bufsize;

  iRet = PCO_GetSizes(cam, &XResAct, &YResAct, &XResMax, &YResMax);
  bufsize=XResAct*YResAct*sizeof(WORD);

  BufEvent = NULL;
  BufNum = -1;
  BufAdr=NULL;
  iRet = PCO_AllocateBuffer(cam, &BufNum, bufsize, &BufAdr, &BufEvent);

  iRet =
PCO_SetImageParameters(cam,XResAct,YResAct,IMAGEPARAMETERS_READ_WHILE_RECORDING,NULL,0);

  printf("Start camera\n");
  iRet = PCO_SetRecordingState(cam, 1);

  printf("Grab single images from running camera\n");
  for(int i=1;i<=10;i++)
  {
   printf("%02d. image ",i);
   iRet = PCO_GetImageEx(cam, 1, 0, 0, BufNum, XResAct, YResAct, 16);
   if (iRet != PCO_NOERROR)
   {
    printf("failed \n");
    break;
   }
   else
    printf("done ");

#ifdef _FILEFUNCTION_
    sprintf(file_name,"image_%02d.tif",i);
    store_tiff(file_name, XResAct, YResAct, 0, BufAdr);
    printf("and stored to %s",file_name);
#endif
     printf("\n");
  }

  printf("Stop camera and close connection\n");
  iRet = PCO_SetRecordingState(cam, 0);
  iRet = PCO_FreeBuffer(cam, BufNum);
  iRet = PCO_CloseCamera(cam);

  printf("Press <Enter> to end\n");
  iRet = getchar();
  return 0;
}

void print_transferpar(HANDLE cam)
{
  PCO_CameraType strCamType;
  DWORD iRet;
  strCamType.wSize=sizeof(PCO_CameraType);
  iRet = PCO_GetCameraType(cam,&strCamType);
  if(iRet!=PCO_NOERROR)
  {
   printf("PCO_GetCameraType failed with errorcode 0x%x\n",iRet);
   return;
  }

  if(strCamType.wInterfaceType==INTERFACE_CAMERA LINK)
  {
    PCO_SC2_CL_TRANSFER_PARAM cl_par;
    iRet = PCO_GetTransferParameter(cam,(void*)&cl_par,sizeof(PCO_SC2_CL_TRANSFER_PARAM));
    printf("Camlink Settings:\nBaudrate:    %u\nClockfreq:   %u\n",
           cl_par.baudrate,cl_par.ClockFrequency);
    printf("Dataformat:  %u 0x%x\nTransmit:
%u\n",cl_par.DataFormat,cl_par.DataFormat,cl_par.Transmit);
  }
}
```

4.3 Example 'Get Single Images from Camera Recorder'

```
#include "pco_err.h"
#include "sc2_SDKStructures.h"
#include "SC2_SDKAddendum.h"
#include "SC2_CamExport.h"
#include "SC2_Defs.h"

#ifdef _FILEFUNCTION_
char filename[50];
#include "../file12.h"
#include "../file12.cpp"
#endif

void print_transferpar(HANDLE cam);

int main(int argc, char* argv[])
{
  int iRet;
  HANDLE cam;
  HANDLE BufEvent;
  short BufNum;
  WORD *BufAdr;

  PCO_Description strDescription;
  WORD RecordingState;

  printf("Get Handle to connected camera\n");
  iRet = PCO_OpenCamera(&cam, 0);
  if (iRet != PCO_NOERROR)
  {
     printf("No camera found\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     return -1;
  }

  strDescription.wSize=sizeof(PCO_Description);
  iRet = PCO_GetCameraDescription(cam,&strDescription);
//check if camera has internal Recorder (CamRam)
  if(strDescription.dwGeneralCapsDESC1&GENERALCAPS1_NO_RECORDER)
  {
     printf("Camera found, but no recorder available\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     iRet = PCO_CloseCamera(cam);
     return -1;
  }

  iRet = PCO_GetRecordingState(cam, &RecordingState);
  if(RecordingState)
   iRet = PCO_SetRecordingState(cam, 0);

//set camera to default state
  iRet = PCO_ResetSettingsToDefault(cam);

#ifdef _FILEFUNCTION_
  iRet = PCO_SetTimestampMode(cam,TIMESTAMP_MODE_BINARYANDASCII);
#endif

  iRet = PCO_ArmCamera(cam);

  DWORD CameraWarning, CameraError, CameraStatus;
  iRet = PCO_GetCameraHealthStatus(cam, &CameraWarning, &CameraError, &CameraStatus);
  if(CameraError!=0)
  {
     printf("Camera has ErrorStatus\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     iRet = PCO_CloseCamera(cam);
     return -1;
  }
```

```
  print_transferpar(cam);

  printf("Start and after some time stop camera\n");
  iRet = PCO_SetRecordingState(cam, 1);
//wait while camera is recording
  Sleep(500);
  iRet = PCO_SetRecordingState(cam, 0);

  DWORD ValidImageCnt, MaxImageCnt;
  WORD Segment=1; //this is the default segment
  iRet = PCO_GetNumberOfImagesInSegment(cam, Segment, &ValidImageCnt, &MaxImageCnt);

  if(ValidImageCnt >= 1)
  {
    WORD XResAct, YResAct, XBin, YBin;
    WORD RoiX0, RoiY0, RoiX1, RoiY1;
    iRet = PCO_GetSegmentImageSettings(cam,Segment, &XResAct, &YResAct,
               &XBin, &YBin, &RoiX0, &RoiY0, &RoiX1, &RoiY1);

    BufEvent = NULL;
    BufNum = -1;
    BufAdr = NULL;
    DWORD bufsize = XResAct*YResAct*sizeof(WORD);

    iRet = PCO_AllocateBuffer(cam, &BufNum, bufsize, &BufAdr, &BufEvent);
    iRet = PCO_SetImageParameters(cam, XResAct,
YResAct,IMAGEPARAMETERS_READ_FROM_SEGMENTS,NULL,0);

    printf("Grab recorded images from camera current valid %d\n",ValidImageCnt);
    for(DWORD i=1;i<=10;i++)
    {
      printf("%02d. image ",i);
      if(ValidImageCnt < i)
      {
       printf("not available \n");
       break;
      }

      iRet = PCO_GetImageEx(cam, Segment, i, i, BufNum, XResAct, YResAct, 16);
      if (iRet != PCO_NOERROR)
      {
       printf("failed \n");
       break;
      }
      else
       printf("done ");

#ifdef _FILEFUNCTION_
      sprintf(filename,"rec_image_%02d.tif",i);
      store_tiff(filename, XResAct, YResAct, 0, BufAdr);
      printf("and stored to %s",filename);
#endif
     printf("\n");
    }
    iRet = PCO_FreeBuffer(cam, BufNum);
  }

  iRet = PCO_CloseCamera(cam);

  printf("Press <Enter> to end\n");
  iRet = getchar();
  return 0;
}
```

4.4 Example 'Get Multiple Images from running Camera'

```
#include "pco_err.h"
#include "SC2_SDKStructures.h"
#include "SC2_SDKAddendum.h"
#include "SC2_CamExport.h"
#include "SC2_Defs.h"

#ifdef _FILEFUNCTION_
char filename[50];
#include "../file12.h"
#include "../file12.cpp"
#endif

void print_transferpar(HANDLE cam);

#define BUFNUM 4

int main(int argc, char* argv[])
{
  int iRet;
  HANDLE cam;
  HANDLE BufEvent[BUFNUM];
  short BufNum[BUFNUM];
  WORD *BufAdr[BUFNUM];

  PCO_Description strDescription;
  WORD RecordingState;
  DWORD waitstat;

  printf("Get Handle to connected camera\n");
  iRet = PCO_OpenCamera(&cam, 0);
  if (iRet != PCO_NOERROR)
  {
     printf("No camera found\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     return -1;
  }

  strDescription.wSize=sizeof(PCO_Description);
  iRet = PCO_GetCameraDescription(cam,&strDescription);

  iRet = PCO_GetRecordingState(cam, &RecordingState);
  if(RecordingState)
  {
     iRet = PCO_SetRecordingState(cam, 0);
  }

//set camera to default state
  iRet = PCO_ResetSettingsToDefault(cam);

#ifdef _FILEFUNCTION_
  iRet = PCO_SetTimestampMode(cam,TIMESTAMP_MODE_BINARYANDASCII);
#endif

  iRet = PCO_ArmCamera(cam);

  DWORD CameraWarning, CameraError, CameraStatus;
  iRet = PCO_GetCameraHealthStatus(cam, &CameraWarning, &CameraError, &CameraStatus);
  if(CameraError!=0)
  {
     printf("Camera has ErrorStatus\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     iRet = PCO_CloseCamera(cam);
     return -1;
  }

  print_transferpar(cam);
```

```
  WORD XResAct, YResAct, XResMax, YResMax;
  DWORD bufsize,StatusDLL,StatusDrv;

  iRet = PCO_GetSizes(cam, &XResAct, &YResAct, &XResMax, &YResMax);
  bufsize=XResAct*YResAct*sizeof(WORD);

  for(int b=0;b<BUFNUM;b++)
  {
    BufEvent[b] = NULL;
    BufNum[b] = -1;
    BufAdr[b]=NULL;
  }

  for(int b=0;b<BUFNUM;b++)
  {
   iRet = PCO_AllocateBuffer(cam, &BufNum[b], bufsize, &BufAdr[b], &BufEvent[b]);
  }

  iRet = PCO_SetImageParameters(cam, XResAct,
YResAct,IMAGEPARAMETERS_READ_WHILE_RECORDING,NULL,0);

  printf("Start camera\n");
  iRet = PCO_SetRecordingState(cam, 1);

  for(int b=0;b<BUFNUM;b++)
  {
   iRet = PCO_AddBufferEx(cam,0,0, BufNum[b], XResAct, YResAct, 16);
  }

  int test,next,multi;
  test=next=multi=0;
  printf("Grab images from running camera\n");
  for(int i=1;i<=10;i++)
  {
   multi=0;
   printf("%02d. image wait ",i);
   waitstat=WaitForMultipleObjects(BUFNUM,BufEvent,FALSE,1000);
   if(waitstat==WAIT_TIMEOUT)
   {
    printf("failed\n");
    break;
   }

// WaitForMultipleObjects might return with 2 or more events set,
// so all buffers must be checked
// 'test' and 'next' help to start check at last successful buffer
// 'multi' counts the number of buffers, which have their event set
   test=next;
   for(int b=0;b<BUFNUM;b++)
   {
    waitstat=WaitForSingleObject(BufEvent[test],0);
    if(waitstat==WAIT_OBJECT_0)
    {
     multi++;
     ResetEvent(BufEvent[test]);
     iRet = PCO_GetBufferStatus(cam,BufNum[test],&StatusDLL,&StatusDrv);

//!!! IMPORTANT StatusDrv must always be checked for errors
     if(StatusDrv==PCO_NOERROR)
     {
      printf(" done buf%02d status 0x%08x ",test,StatusDrv);
      if(multi>1)
       printf("multi %02d ",multi);
     }
     else
     {
      printf("buf%02d error status 0x%08x m %02d ",test,StatusDrv,multi);
      break;
     }
```

```
// calculations on the image data can be done here, but calculation time must not exceed
// frametime of camera else images are lost
#ifdef _FILEFUNCTION_
      sprintf(filename,"add_image_%02d.tif",i);
      store_tiff(filename, XResAct, YResAct, 0, BufAdr[test]);
      printf("and stored to %s",filename);
#endif

     iRet = PCO_AddBufferEx(cam,0,0, BufNum[test], XResAct, YResAct, 16);
    }
    else
     break;
    test++;
    if(test>=BUFNUM)
     test=0;
    printf("\n");
   }
   next=test;
   fflush(stdout);
  }//end for imacount

//!!! IMPORTANT PCO_CancelImages must always be called, after PCO_AddBuffer...() loops
  iRet = PCO_CancelImages(cam);

  printf("Stop camera and close connection\n");
  iRet = PCO_SetRecordingState(cam, 0);
  for(int b=0;b<BUFNUM;b++)
   iRet = PCO_FreeBuffer(cam, BufNum[b]);
  iRet = PCO_CloseCamera(cam);

  printf("Press <Enter> to end\n");
  iRet = getchar();
  return 0;
}

void print_transferpar(HANDLE cam)
{
  PCO_CameraType strCamType;
  DWORD iRet;
  strCamType.wSize=sizeof(PCO_CameraType);
  iRet = PCO_GetCameraType(cam,&strCamType);
  if(iRet!=PCO_NOERROR)
  {
   printf("PCO_GetCameraType failed with errorcode 0x%x\n",iRet);
   return;
  }

  if(strCamType.wInterfaceType==INTERFACE_CAMERA LINK)
  {
    PCO_SC2_CL_TRANSFER_PARAM cl_par;

    iRet = PCO_GetTransferParameter(cam,(void*)&cl_par,sizeof(PCO_SC2_CL_TRANSFER_PARAM));
    printf("Camlink Settings:\nBaudrate:    %u\nClockfreq:
%u\n",cl_par.baudrate,cl_par.ClockFrequency);
    printf("Dataformat:  %u 0x%x\nTransmit:
%u\n",cl_par.DataFormat,cl_par.DataFormat,cl_par.Transmit);
  }
}
```

4.5 Example ‘Get Multiple Images from Camera Recorder’

```
#include "pco_err.h"
#include "sc2_SDKStructures.h"
#include "SC2_SDKAddendum.h"
#include "SC2_CamExport.h"
#include "SC2_Defs.h"

#ifdef _FILEFUNCTION_
char filename[50];
#include "../file12.h"
#include "../file12.cpp"
#endif

void print_transferpar(HANDLE cam);


#define BUFNUM 4

int main(int argc, char* argv[])
{
  int iRet;
  HANDLE cam;
  HANDLE BufEvent[BUFNUM];
  short BufNum[BUFNUM];
  WORD *BufAdr[BUFNUM];

  PCO_Description strDescription;
  WORD RecordingState;
  DWORD waitstat;

  printf("Get Handle to connected camera\n");
  iRet = PCO_OpenCamera(&cam, 0);
  if (iRet != PCO_NOERROR)
  {
     printf("No camera found\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     return -1;
  }

  strDescription.wSize=sizeof(PCO_Description);
  iRet = PCO_GetCameraDescription(cam,&strDescription);

  if(strDescription.dwGeneralCapsDESC1&GENERALCAPS1_NO_RECORDER)
  {
     printf("Camera found, but no recorder available\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     iRet = PCO_CloseCamera(cam);
     return -1;
  }

  iRet = PCO_GetRecordingState(cam, &RecordingState);
  if(RecordingState)
     iRet = PCO_SetRecordingState(cam, 0);

//set camera to default state
  iRet = PCO_ResetSettingsToDefault(cam);

#ifdef _FILEFUNCTION_
  iRet = PCO_SetTimestampMode(cam,TIMESTAMP_MODE_BINARYANDASCII);
#endif

  iRet = PCO_ArmCamera(cam);
```

```
  DWORD CameraWarning, CameraError, CameraStatus;
  iRet = PCO_GetCameraHealthStatus(cam, &CameraWarning, &CameraError, &CameraStatus);
  if(CameraError!=0)
  {
     printf("Camera has ErrorStatus\n");
     printf("Press <Enter> to end\n");
     iRet = getchar();
     iRet = PCO_CloseCamera(cam);
     return -1;
  }

  print_transferpar(cam);

  printf("Start and after some time stop camera\n");
  iRet = PCO_SetRecordingState(cam, 1);

//wait while camera is recording
  Sleep(500);

  iRet = PCO_SetRecordingState(cam, 0);

  DWORD ValidImageCnt, MaxImageCnt;
  WORD Segment=1; //this is the default segment

  iRet = PCO_GetNumberOfImagesInSegment(cam, Segment, &ValidImageCnt, &MaxImageCnt);

  if(ValidImageCnt >= 1)
  {
    DWORD bufsize,StatusDLL,StatusDrv,set;
    WORD XResAct, YResAct, XBin, YBin;
    WORD RoiX0, RoiY0, RoiX1, RoiY1;
    iRet = PCO_GetSegmentImageSettings(cam,Segment, &XResAct, &YResAct,
               &XBin, &YBin, &RoiX0, &RoiY0, &RoiX1, &RoiY1);


    for(int b=0;b<BUFNUM;b++)
    {
      BufEvent[b] = NULL;
      BufNum[b] = -1;
      BufAdr[b]=NULL;
    }

    bufsize = XResAct*YResAct*sizeof(WORD);
    for(int b=0;b<BUFNUM;b++)
    {
     iRet = PCO_AllocateBuffer(cam, &BufNum[b], bufsize, &BufAdr[b], &BufEvent[b]);
    }
    iRet = PCO_SetImageParameters(cam, XResAct,
YResAct,IMAGEPARAMETERS_READ_FROM_SEGMENTS,NULL,0);

    int test,next,multi;
    test=next=multi=0;
    printf("Grab recorded images from camera current valid %d\n",ValidImageCnt);

    set=1;
    for(int b=0;b<BUFNUM;b++)
    {
     if(ValidImageCnt >= set)
     {
      iRet = PCO_AddBufferEx(cam,set,set, BufNum[b], XResAct, YResAct, 16);
      set++;
     }
    }
```

```
    for(DWORD i=1;i<=10;i++)
    {
      printf("%02d. image ",i);
      if(ValidImageCnt < i)
      {
       printf("not available \n");
       break;
      }
      multi=0;
      printf("wait ");
      waitstat=WaitForMultipleObjects(BUFNUM,BufEvent,FALSE,1000);
      if(waitstat==WAIT_TIMEOUT)
      {
       printf("failed\n");
       break;
      }

// WaitForMultipleObjects might return with 2 or more events set, so all buffers must be
checked
// 'test' and 'next' help to start check at last successfull buffer
// 'multi' counts the number of buffers, which have their event set
      test=next;
      for(int b=0;b<BUFNUM;b++)
      {
        waitstat=WaitForSingleObject(BufEvent[test],0);
        if(waitstat==WAIT_OBJECT_0)
        {
          multi++;
          ResetEvent(BufEvent[test]);
          iRet = PCO_GetBufferStatus(cam,BufNum[test],&StatusDLL,&StatusDrv);

//!!! IMPORTANT StatusDrv must always be checked for errors
          if(StatusDrv==PCO_NOERROR)
          {
           printf(" done buf%02d status 0x%08x ",test,StatusDrv);
           if(multi>1)
            printf("multi %02d ",multi);
          }
          else
          {
           printf("buf%02d error status 0x%08x m %02d ",test,StatusDrv,multi);
           break;
          }

// calculations on the image data can be done here, but calculation time must not exceed
// frametime of camera else images are lost
#ifdef _FILEFUNCTION_
          sprintf(filename,"addrec_image_%02d.tif",i);
          store_tiff(filename, XResAct, YResAct, 0, BufAdr[test]);
          printf("and stored to %s",filename);
#endif

          if(ValidImageCnt >= set)
          {
           iRet = PCO_AddBufferEx(cam,set,set, BufNum[test], XResAct, YResAct, 16);
           set++;
          }
        }
        else
         break;
        test++;
        if(test>=BUFNUM)
         test=0;
        printf("\n");
      }
      next=test;
      fflush(stdout);
    }//end for imacount
```

```
//!!! IMPORTANT PCO_CancelImages must always be called, after PCO_AddBuffer...() loops
    iRet = PCO_CancelImages(cam);
    for(int b=0;b<BUFNUM;b++)
     iRet = PCO_FreeBuffer(cam, BufNum[b]);
  }

  iRet = PCO_CloseCamera(cam);

  printf("Press <Enter> to end\n");
  iRet = getchar();
  return 0;
}

void print_transferpar(HANDLE cam)
{
  PCO_CameraType strCamType;
  DWORD iRet;
  strCamType.wSize=sizeof(PCO_CameraType);
  iRet = PCO_GetCameraType(cam,&strCamType);
  if(iRet!=PCO_NOERROR)
  {
   printf("PCO_GetCameraType failed with errorcode 0x%x\n",iRet);
   return;
  }

  if(strCamType.wInterfaceType==INTERFACE_CAMERA LINK)
  {
    PCO_SC2_CL_TRANSFER_PARAM cl_par;

    iRet = PCO_GetTransferParameter(cam,(void*)&cl_par,sizeof(PCO_SC2_CL_TRANSFER_PARAM));
    printf("Camlink Settings:\nBaudrate:    %u\nClockfreq:
%u\n",cl_par.baudrate,cl_par.ClockFrequency);
    printf("Dataformat:  %u 0x%x\nTransmit:
%u\n",cl_par.DataFormat,cl_par.DataFormat,cl_par.Transmit);
  }
}
```

4.6 Debugging with GigE Interface

While debugging with the GigE interface, it might be possible to get error `0xA0322005`, which means ***time-out***. This is caused by a long break between two debugging steps (usually > 65s). Single stepping stops all threads executed till the next step. This disables the `sc2_gige.dll` thread to send heartbeat messages to the camera. The camera will generate a timeout due to lost connection. In this case please stop and restart your debug session. Keep in mind that you'll have to step quickly through your code while debugging with a GigE interface. In case your stepping performance is not quick enough you could increase the parameter `EthernetHeartbeatTimoutMs`. See file ***SC2_GigE_param.ini*** within the CSIDL_COMMON_APPDATA\pco folder.

5. ERROR / WARNING CODES

The error codes are standardized as far as possible. The error codes contain the information of the error layer, the source (microcontrollers, CPLDs, FPGAs) and an error code (error cause). All values are combined by a logical ***OR*** operation. Error codes and warnings are always negative values, if read as signed integers, or if read as unsigned integer the MSB is set. Errors have the general format 0x80######, warnings have the format 0xC0######.
The error numbers are not unique. Each layer and the common errors have its own error codes. You have to analyse the error in order to get error source. This can easily be done with a call to **PCO_GetErrorText**.

```
// e.g.: 0xC0000080 indicates a warning,
//       0x800A3001 is an error inside the SC2-SDK-DLL.
// MSB                                LSB
// XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
// |||| |||| |||| |||| |||| |||| |||| ||||
// |||| |||| |||| |||| |||| ---------------- Error or warning code
// |||| |||| |||| |||| ||||
// |||| |||| |||| |||| -------------------- Layer code
// |||| |||| |||| ||||
// |||| |||| ------------------------------ Device code
// |||| ||||
// |||------------------------------------- reserved for future use
// |||
// ||-------------------------------------- Common error code flag
// ||
// |--------------------------------------- Warning indication bit
// |
// ---------------------------------------- Error indication bit
```

Error layer:

Value	Name	Description
0x00001000	PCO_ERROR_FIRMWARE	Error inside the firmware
0x00002000	PCO_ERROR_DRIVER	Error inside the driver
0x00003000	PCO_ERROR_SDK_DLL	Error inside the SDK DLL
0x00004000	PCO_ERROR_APPLICATION	Error inside the application

Error / Warning source (some examples):

Value	Name	Description
0x00010000	SC2_ERROR_PCOCAM_POWER_CPLD	Error at CPLD in power unit
0x00020000	SC2_ERROR_PCOCAM_HEAD_UP	Error at uP of head board in camera
0x00030000	SC2_ERROR_PCOCAM_MAIN_UP	Error at uP of main board in camera
0x00040000	SC2_ERROR_PCOCAM_FWIRE_UP	Error at uP of FireWire board in camera
0x00050000	SC2_ERROR_PCOCAM_MAIN_FPGA	Error at FPGA of main board in camera
0x00060000	SC2_ERROR_PCOCAM_HEAD_FPGA	Error at FGPA of head board in camera
0x00070000	SC2_ERROR_PCOCAM_MAIN_BOARD	Error at main board in camera
0x00080000	SC2_ERROR_PCOCAM_HEAD_CPLD	Error at CPLD of head board in camera
0x00090000	SC2_ERROR_SENSOR	Error at image sensor (CCD or CMOS)
0x000A0000	SC2_ERROR_SDKDLL	Error inside the SDKDLL
0x000B0000	SC2_ERROR_DRIVER	Error inside the driver
0x000D0000	SC2_ERROR_POWER	Error within power unit
0x00100000	PCO_ERROR_CAMWARE	Error in Camware also some kind of "device"
0x00110000	PCO_ERROR_CONVERTDLL	Error inside the convert DLL

Error codes:

Please take a look at the file pco_err.h.

Warnings:

Please take a look at the file pco_err.h.

In case of successful operation `PCO_NOERROR` is returned.
To get detailed error information call the function **PCO_GetErrorText**, which is defined inside the PCO_errt.h header file.

5.1 PCO_GetErrorText

Description:

Use this command to get a detailed description for an error.
This function is part of the header file pco_errt.h. To use this function include the pco_errt.h header file and define `PCO_ERRT_H_CREATE_OBJECT` in a module.

Supported camera type:

All cameras

Prototype:

```
SC2_SDK_FUNC void WINAPI PCO_GetErrorText (
    DWORD dwerr,                    //in
    char* pbuf,                     //in,out
    DWORD dwlen                     //in
);
```

Parameter:

Name	Type	Description
`dwerr`	`DWORD`	DWORD which holds the error number
`pbuf`	`char*`	Pointer to a character array. The error description as ASCII string
`dwlen`	`DWORD`	Size of the character array `pbuf` in byte, which has passed in

6. API FUNCTION MATRIX

Chapter	Title	All cameras	pco.edge	pco.dimax	pco.flim	pco.pixelfly usb	pco.ultraviolet	pco.1200	pco.1300	pco.1400	pco.1600 pco.2000 pco.4000
2.1	**CAMERA ACCESS**										
2.1.1	**PCO_OpenCamera**	✓									
2.1.2	**PCO_OpenCameraEx**	✓									
2.1.3	**PCO_CloseCamera**	✓									
2.1.4	**PCO_ResetLib**	✓									
2.1.5	**PCO_CheckDeviceAvailability**	IF									
2.2	**CAMERA DESCRIPTION**										
2.2.1	**PCO_GetCameraDescription**	✓									
2.2.2	**PCO_GetCameraDescriptionEx**	✓									
2.3	**GENERAL CAMERA STATUS**										
2.3.1	**PCO_GetGeneral**	✓									
2.3.2	**PCO_GetCameraType**	✓									
2.3.3	**PCO_GetCameraHealthStatus**	✓									
2.3.4	**PCO_GetTemperature**	✓									
2.3.5	**PCO_GetInfoString**	✓									
2.3.6	**PCO_GetCameraName**	✓									
2.3.7	**PCO_GetFirmwareInfo**	✓									
2.3.8	**PCO_GetColorCorrectionMatrix**		✓	✓		✓					
2.4	**GENERAL CAMERA CONTROL**										
2.4.1	**PCO_ArmCamera**	✓									
2.4.2	**PCO_CamLinkSetImageParameters (obsolete)**	✓									
2.4.3	**PCO_SetImageParameters**	✓									
2.4.4	**PCO_ResetSettingsToDefault**	✓									
2.4.5	**PCO_SetTimeouts**	✓									
2.4.6	**PCO_RebootCamera**	✓									
2.4.7	**PCO_GetCameraSetup**		✓								
2.4.8	**PCO_SetCameraSetup**		✓								
2.4.9	**PCO_ControlCommandCall**	✓									
2.5	**IMAGE SENSOR**										
2.5.1	**PCO_GetSensorStruct**	✓									
2.5.2	**PCO_SetSensorStruct**	✓									
2.5.3	**PCO_GetSizes**	✓									
2.5.4	**PCO_GetSensorFormat**	✓									
2.5.5	**PCO_SetSensorFormat**	✓									
2.5.6	**PCO_GetROI**	✓									
2.5.7	**PCO_SetROI**		✓	✓				✓			✓
2.5.8	**PCO_GetBinning**	✓									
2.5.9	**PCO_SetBinning**	✓									
2.5.10	**PCO_GetPixelRate**	✓									
2.5.11	**PCO_SetPixelRate**	✓									
2.5.12	**PCO_GetConversionFactor**	✓									

Chapter	Title	All cameras	pco.edge	pco.dimax	pco.flim	pco.pixelfly usb	pco.ultraviolet	pco.1200	pco.1300	pco.1400	pco.1600 pco.2000 pco.4000
2.5.13	**PCO_SetConversionFactor**	✓									
2.5.14	**PCO_GetDoubleImageMode**	✓									
2.5.15	**PCO_SetDoubleImageMode**	✓									
2.5.16	**PCO_GetADCOperation**										✓
2.5.17	**PCO_SetADCOperation**										✓
2.5.18	**PCO_GetIRSensitivity**					✓			✓	✓	
2.5.19	**PCO_SetIRSensitivity**					✓			✓	✓	
2.5.20	**PCO_GetCoolingSetpointTemperature**		✓		✓				✓		✓
2.5.21	**PCO_SetCoolingSetpointTemperature**		✓		✓				✓		✓
2.5.22	**PCO_GetCoolingSetpoints**		✓								
2.5.23	**PCO_GetOffsetMode**					✓	✓		✓	✓	
2.5.24	**PCO_SetOffsetMode**					✓	✓		✓	✓	
2.5.25	**PCO_GetNoiseFilterMode**	✓									
2.5.26	**PCO_SetNoiseFilterMode**	✓									
2.5.27	**PCO_GetLookuptableInfo**		✓								
2.5.28	**PCO_GetActiveLookuptable**		✓								
2.5.29	**PCO_SetActiveLookuptable**		✓								
2.6	**TIMING CONTROL**										
2.6.1	**PCO_GetTimingStruct**	✓									
2.6.2	**PCO_SetTimingStruct**	✓									
2.6.3	**PCO_GetCOCRunTime**	✓									
2.6.4	**PCO_GetDelayExposureTime**	✓									
2.6.5	**PCO_SetDelayExposureTime**	✓									
2.6.6	**PCO_GetDelayExposureTimeTable**	✓									
2.6.7	**PCO_SetDelayExposureTimeTable**	✓									
2.6.8	**PCO_GetFrameRate**		IF	✓							
2.6.9	**PCO_SetFrameRate**		IF	✓							
2.6.10	**PCO_GetFPSExposureMode**							✓			
2.6.11	**PCO_SetFPSExposureMode**							✓			
2.6.12	**PCO_GetTriggerMode**	✓									
2.6.13	**PCO_SetTriggerMode**	✓									
2.6.14	**PCO_ForceTrigger**	✓									
2.6.15	**PCO_GetCameraBusyStatus**		✓								✓
2.6.16	**PCO_GetPowerDownMode**										✓
2.6.17	**PCO_SetPowerDownMode**										✓
2.6.18	**PCO_GetUserPowerDownTime**										✓
2.6.19	**PCO_SetUserPowerDownTime**										✓
2.6.20	**PCO_GetModulationMode** (only special versions)										✓
2.6.21	**PCO_SetModulationMode** (only special versions)										✓
2.6.22	**PCO_GetHWIOSignalCount**		✓	✓							
2.6.23	**PCO_GetHWIOSignalDescriptor**		✓	✓							
2.6.24	**PCO_GetHWIOSignal**		✓	✓							

Chapter	Title	All cameras	pco.edge	pco.dimax	pco.flim	pco.pixelfly usb	pco.ultraviolet	pco.1200	pco.1300	pco.1400	pco.1600 pco.2000 pco.4000
2.6.25	PCO_SetHWIOSignal		✓	✓							
2.6.26	PCO_GetImageTiming		✓								
2.6.27	PCO_GetCameraSynchMode			✓							
2.6.28	PCO_SetCameraSynchMode			✓							
2.6.29	PCO_GetExpTrigSignalStatus					✓	✓		✓	✓	
2.6.30	PCO_GetFastTimingMode			✓							
2.6.31	PCO_SetFastTimingMode			✓							
2.7	RECORDING CONTROL										
2.7.1	PCO_GetRecordingStruct	✓									
2.7.2	PCO_SetRecordingStruct	✓									
2.7.3	PCO_GetRecordingState	✓									
2.7.4	PCO_SetRecordingState	✓									
2.7.5	PCO_GetStorageMode		IF	✓		✓	✓	✓	✓	✓	✓
2.7.6	PCO_SetStorageMode		IF	✓		✓	✓	✓	✓	✓	✓
2.7.7	PCO_GetRecorderSubmode		IF	✓		✓	✓	✓	✓	✓	✓
2.7.8	PCO_SetRecorderSubmode		IF	✓		✓	✓	✓	✓	✓	✓
2.7.9	PCO_GetAcquireMode		✓	✓		✓	✓	✓	✓	✓	✓
2.7.10	PCO_SetAcquireMode		✓	✓		✓	✓	✓	✓	✓	✓
2.7.11	PCO_GetAcquireModeEx		✓								
2.7.12	PCO_SetAcquireModeEx		✓								
2.7.13	PCO_GetAcqEnblSignalStatus										✓
2.7.14	PCO_GetMetaDataMode		IF	✓							
2.7.15	PCO_SetMetaDataMode		IF	✓							
2.7.16	PCO_GetRecordStopEvent			✓				✓			
2.7.17	PCO_SetRecordStopEvent			✓				✓			
2.7.18	PCO_StopRecord			✓				✓			
2.7.19	PCO_SetDateTime	✓									
2.7.20	PCO_GetTimestampMode	✓									
2.7.21	PCO_SetTimestampMode	✓									
2.8	STORAGE CONTROL										
2.8.1	PCO_GetStorageStruct			✓				✓			✓
2.8.2	PCO_SetStorageStruct			✓				✓			✓
2.8.3	PCO_GetCameraRamSize			✓				✓			✓
2.8.4	PCO_GetCameraRamSegmentSize			✓				✓			✓
2.8.5	PCO_SetCameraRamSegmentSize			✓				✓			✓
2.8.6	PCO_ClearRamSegment			✓				✓			✓
2.8.7	PCO_GetActiveRamSegment			✓				✓			✓
2.8.8	PCO_SetActiveRamSegment			✓				✓			✓
2.9	IMAGE INFORMATION										
2.9.1	PCO_GetImageStruct			✓				✓			✓
2.9.2	PCO_GetSegmentStruct			✓				✓			✓
2.9.3	PCO_GetSegmentImageSettings			✓				✓			✓

Chapter	Title	All cameras	pco.edge	pco.dimax	pco.flim	pco.pixelfly usb	pco.ultraviolet	pco.1200	pco.1300	pco.1400	pco.1600 pco.2000 pco.4000
2.9.4	**PCO_GetNumberOfImagesInSegment**			✓				✓			✓
2.9.5	**PCO_GetBitAlignment**	✓									
2.9.6	**PCO_SetBitAlignment**	✓									
2.9.7	**PCO_GetHotPixelCorrectionMode**	✓									
2.9.8	**PCO_SetHotPixelCorrectionMode**	✓									
2.10	**BUFFER MANAGEMENT**										
2.10.1	**PCO_AllocateBuffer**	✓									
2.10.2	**PCO_FreeBuffer**	✓									
2.10.3	**PCO_GetBufferStatus**	✓									
2.10.4	**PCO_GetBuffer**	✓									
2.11	**IMAGE ACQUISITION**										
2.11.1	**PCO_GetImageEx**	✓									
2.11.2	**PCO_GetImage (obsolete)**	✓									
2.11.3	**PCO_AddBufferEx**	✓									
2.11.4	**PCO_AddBuffer (obsolete)**	✓									
2.11.5	**PCO_AddBufferExtern**	✓									
2.11.6	**PCO_CancelImages**	✓									
2.11.7	**PCO_RemoveBuffer (obsolete)**	✓									
2.11.8	**PCO_GetPendingBuffer**	✓									
2.11.9	**PCO_WaitforBuffer**	✓									
2.11.10	**PCO_EnableSoftROI** (only Camera Link MEIV)		✓								
2.11.11	**PCO_GetMetaData**		✓	✓							
2.12	**DRIVER MANAGMENT**										
2.12.1	**PCO_GetTransferParameter**	IF									
2.12.2	**PCO_SetTransferParameter**	IF									
2.12.3	**Transfer Parameter Structures**	IF									
2.13	**SPECIAL COMMANDS PCO.EDGE**										
2.13.1	**PCO_GetSensorSignalStatus**		✓								
2.13.2	**PCO_GetCmosLineTiming** (only Camera Link)		IF								
2.13.3	**PCO_SetCmosLineTiming** (only Camera Link)		IF								
2.13.4	**PCO_GetCmosLineExposureDelay** (only Camera Link)		IF								
2.13.5	**PCO_SetCmosLineExposureDelay** (only Camera Link)		IF								
2.13.6	**PCO_SetTransferParametersAuto** (only 5.5 Camera Link)		IF								
2.13.7	**PCO_GetInterfaceOutputFormat**		✓								
2.13.8	**PCO_SetInterfaceOutputFormat**		✓								
2.14	**SPECIAL COMMANDS PCO.DIMAX**										
2.14.1	**PCO_GetImageTransferMode** (only GigE / USB interface)			IF							
2.14.2	**PCO_SetImageTransferMode** (only GigE / USB interface)			IF							
2.14.3	**PCO_GetCDIMode**			✓							
2.14.4	**PCO_SetCDIMode**			✓							
2.14.5	**PCO_GetPowerSaveMode**			✓							
2.14.6	**PCO_SetPowerSaveMode**			✓							

Chapter	Title	All cameras	pco.edge	pco.dimax	pco.flim	pco.pixelfly usb	pco.ultraviolet	pco.1200	pco.1300	pco.1400	pco.1600 pco.2000 pco.4000
2.14.7	**PCO_GetBatteryStatus**			✓							
2.15	**SPECIAL COMMANDS PCO.DIMAX WITH HD-SDI**										
2.15.1	**PCO_GetInterfaceOutputFormat**			✓							
2.15.2	**PCO_SetInterfaceOutputFormat**			✓							
2.15.3	**PCO_PlayImagesFromSegmentHDSDI**			✓							
2.15.4	**PCO_GetPlayPositionHDSDI**			✓							
2.15.5	**PCO_GetColorSettings**			✓							
2.15.6	**PCO_SetColorSettings**			✓							
2.15.7	**PCO_DoWhiteBalance**			✓							
2.16	**SPECIAL COMMANDS PCO.FLIM**										
2.16.1	**PCO_GetFlimModulationParameter**				✓						
2.16.2	**PCO_SetFlimModulationParameter**				✓						
2.16.3	**PCO_GetFlimMasterModulationFrequency**				✓						
2.16.4	**PCO_SetFlimMasterModulationFrequency**				✓						
2.16.5	**PCO_GetFlimPhaseSequenceParameter**				✓						
2.16.6	**PCO_SetFlimPhaseSequenceParameter**				✓						
2.16.7	**PCO_GetFlimRelativePhase**				✓						
2.16.8	**PCO_SetFlimRelativePhase**				✓						
2.16.9	**PCO_GetFlimImageProcessingFlow**				✓						
2.16.10	**PCO_SetFlimImageProcessingFlow**				✓						

(✓: function available; FW: depends on firmware version; IF: depends on interface)

ABOUT PCO

In 1987, PCO was founded with the objective to develop and to produce specialized, fast and sensitive video camera systems, mainly for scientific applications. Meanwhile the product range of PCO cameras covers digital camera systems with high dynamic range, high resolution, high speed and low noise, which are sold in the scientific and industrial market all over the world.

Currently PCO is one of the leading manufacturers of scientific cameras. Worldwide representatives, together with our own sales department and technical support assure that we keep in touch with our customers and their needs. The current wide range of specialized camera systems is the result of technical challenge and product specific know-how. A design according to advanced techniques, a high standard of production and strict quality controls guarantee a reliable operation of the cameras. Our own developments in conjunction with an excellent contact to leading manufacturers of image sensors ensure our access to state-of-the-art CCD- and CMOS-technology for our cameras.

Since 2001, PCO is located in its own facility building in Kelheim at the shore of the beautiful and international river Danube. Here in the county Bavaria, which is well known for its excellent support and conditions for high technology companies, we share the benefits of the simple access to high performance products and services in the surrounding area.

Kelheim itself is a historical town, first documented in 866. The small city is founded at the confluence of the Danube and the Altmühl, which has been converted into the Rhine-Main-Danube bypass channel for water transport. Located in Danube-valley, it is the heart of a beautiful river and forest covered lime plateau landscape. It´s landmark, the Hall of Liberation, was built by Ludwig I. in 1863 on the Mount Michael and is visible from all over the city and valley. The beautiful Danube-Gorge, which is protected as natural monument since 1840, is located between Kelheim and the famous abbey Weltenburg.

pco.