PVCAM 3.1

57-062-001
Version 3.1
June 16, 2016

Photometrics
3440 E. Britannia Dr.
Tucson, AZ 85706
TEL: 520-889-9933
FAX: 520-573-1944

Table of Contents

Chapter 1: SDK 7
What is the SDK? 7
Contact Information 7
Chapter 2: PVCAM, A High-Level C Library 9
Introduction 9
System Overview 9
Hardware Support 9
Library Classes 10
Documentation Style 10
Defined Types 11
Naming Conventions 12
Include Files 12
Parameter Passing and const 12
Coordinates Model 12
Regions and Images 13
Binning Factors 13
Data Array 13
Display Orientation 14
Port and Speed Choices 14
Multi-tap Configuration and Readout 15
Frame Transfer 15
Image Smear 16
Sequences 17
Sequence Parameters IDs/Constants 18
Circular Buffer 18
Sequenced Multiple Acquisition Real Time (SMART) streaming 20
SMART Streaming Data Types 20
Possible Scenarios 21
Clear Modes 21
Exposure Modes 22
TIMED_MODE 23
VARIABLE_TIMED_MODE 23
TRIGGER_FIRST_MODE 23
STROBED_MODE 24
BULB_MODE 24
Extended Exposure Modes 25
Expose Out Modes 26
Open Delay, Close Delay 27

Shutter Control 27
Exposure Loops 28
Image Buffers 33
Chapter 3: Camera Communication (Class 0) 34
Introduction 34
List of Available Class 0 Functions 34
List of Available Class 0 Parameter IDs 34
Class 0 Functions 35
Class 0 Parameter IDs 49
Chapter 4: Error Reporting (Class 1) 52
Introduction 52
Error Codes 53
List of Available Class 1 Functions 53
Class 1 Functions 54
Chapter 5: Configuration / Setup (Class 2) 56
Introduction 56
List of Available Class 2 Functions 57
List of Available Class 2 Parameter IDs 57
Class 2 Functions 59
Class 2 Parameter IDs 70
Chapter 6: Data Acquisition (Class 3) 84
Introduction 84
List of Available Class 3 Functions 84
List of Available Class 3 Parameter IDs 85
Defining Exposures 86
New Structures 86
Exposure Mode Constants 87
Embedded Frame Metadata 88
Multiple-ROI Acquisition 89
Centroids 90
Binning Factor Discovery 91
Triggering Table 92
Class 3 Functions 93
Class 3 Parameter IDs 118
Index 123

List of Tables

Table 1 New Number Types 11
Table 2 Standard Abbreviations 12
Table 3 Two Port Camera Example 15

Table 4 Legend of sample command sequences 28
Table 5 TIMED_MODE command sequences 29
Table 6 TRIGGER_FIRST_MODE command sequences 30
Table 7 STROBED_MODE command sequences 31
Table 8 BULB_MODE command sequences 32
Table 9 Single exposure – full image 33
Table 10 Single exposure – custom region 33
Table 11 Multiple exposures – full image 33
Table 12 Multiple exposures – custom region 33

This page intentionally left blank.

Chapter 1: SDK

What is the SDK?

SDK — Photometrics' Software Development Kit — allows programmers to access and use the capabilities of PVCAM© — Programmable Virtual Camera Access Method Library. (PVCAM is described in detail in the chapters that follow.)

For developer convenience, we have included a Windows environment variable into the PVCAM installer, which will guide developers to the location of all binaries and header files needed to develop with PVCAM as well as Visual Studio and similar environments. This environment variable is called PVCAM_SDK_PATH and can be discovered in the Windows explorer using %PVCAM_SDK_PATH% syntax and Visual Studio using $(PVCAM_SDK_PATH) syntax; additionally, if the developer desires to build with a special version of PVCAM different from the version installed the "system" environment variable may be overwritten using a "user" environment variable with the same name. All the developer must do to use the custom location is build a similar directory structure and create a "user" environment variable with the path of the folder (i.e. – "C:\alt_pvcam_sdk"). Please consult this Read Me file for information on:

- Linking PVCAM to your software
- Initializing PVCAM
- Basic Acquisition using PVCAM

Contact Information

Photometrics' headquarters are located at the following addresses:

Photometrics

3440 East Britannia Drive

Tucson, Arizona 85706 (USA)

TEL: 800-874-9789 / 520-889-9933

FAX: 520-573-1944

Tech Support E-mail: support@photometrics.com

For technical support and service outside the United States, see our web page at www.photometrics.com.

An up-to-date list of addresses, telephone numbers, and e-mail addresses of Photometrics' overseas offices and representatives is maintained on the web page.

This page intentionally left blank.

Chapter 2:
PVCAM, A High-Level C Library

Introduction

PVCAM is an ANSI C library of camera control and data acquisition functions. This library, which is identical across platforms and operating systems, provides an interface that allows developers to specify the camera's setup, exposure, and data storage attributes.

System Overview

To use PVCAM, a system must include camera hardware and software, a host computer, and the PVCAM library.

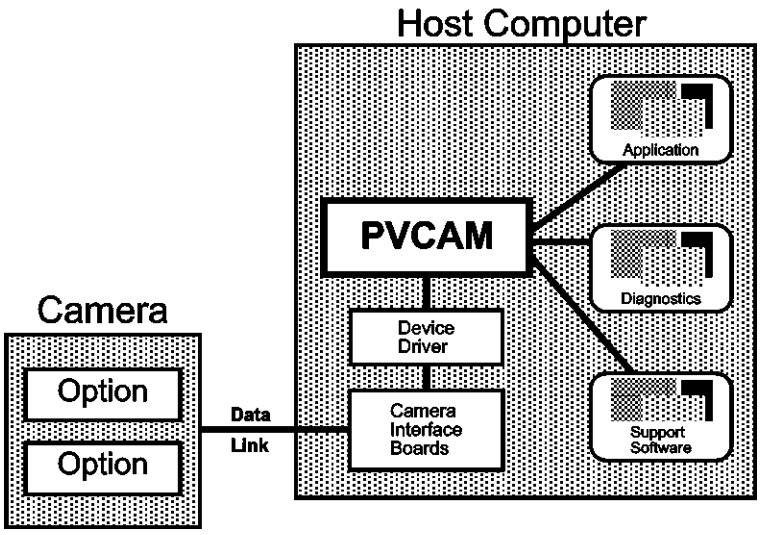

Figure 1 System Overview

Hardware Support

PVCAM library supports all Photometrics brand hardware.

Library Classes

The basic PVCAM library supports the following five classes of camera and buffer control:

0 – Camera Communication	These functions establish communication paths between the high-level application software and the device driver. They also establish some low-level functions for controlling the camera hardware.
1 – Error Reporting	These functions monitor and report on other library functions. When an error occurs, a function can be called to return a unique error code.
2 – Configuration/Setup	These functions initialize the library and set up the hardware and software environments. They also control and monitor the camera hardware, and allow the user to set parameters such as camera gain and temperature.
3. Data Acquisition	These functions define how the image data are collected.

Documentation Style

This manual describes the functional aspects of using PVCAM and various controls for Photometrics® cameras (*Chapter 2*), gives reference pages for all of the function calls (*Chapter 3* through *Chapter 6*) and provides code examples.

Defined Types

In order to work effectively across platforms, the number of bytes in a variable must be consistent. Therefore, new types have been defined for PVCAM. These typedefs are given in one of PVCAM public header files.

Type	Explanation
rs_bool+	TRUE (non-0) or FALSE (0) value
int8	signed 8-bit integral value
uns8	unsigned 8-bit integral value
int16	signed 16-bit integral value
uns16	unsigned 16-bit integral value
int32	signed 32-bit integral value
uns32	unsigned 32-bit integral value
ulong64	unsigned 64-bit integral value
long64	signed 64-bit integral value
enum	treat as signed 32-bit integral value
flt32	32-bit floating point value
flt64	64-bit floating point value
smart_stream_type	structure for S.M.A.R.T. streaming
rgn_type	structure holding additional region information, see pvcam.h
FRAME_INFO	structure holding additional frame information, see pvcam.h

Table 1 New Number Types

+**Note:** The type rs_bool has replaced the deprecated boolean type. This is due to a size difference of the boolean type on the Windows platform. Namely, windows.h defines a boolean type of a different size. Including windows.h in the same translation unit as master.h compiles the wrong boolean and causes subtle memory access violations. It is strongly recommended to use the new rs_bool type instead to avoid this potential clash.

Since Photometrics® camera data and analyses depend on bit depth, the new types give values that are consistent with the size of the bit depth.

Each new type is composed of the appropriate combinations of int, short, long, or other types that give the appropriate length for each value. The 8-bit types are the smallest type that holds 8 bits, 16-bit types are the smallest type holding 16 bits, and so forth.

For historical reasons and also for backward compatibility PVCAM public headers contain definitions of pointer type and constant pointer type for each basic data type as listed in Table 1. Those have usually suffix _ptr or _const_ptr. It is not recommended to use them in any new code.

Naming Conventions

To shorten names and improve readability, standard abbreviations are used for common words and phrases. These abbreviations are used in function and variable names.

adc=analog-to-digital converter	exp=exposure	par=parallel
addr=address	hcam=camera handle	pix=pixel
bin=binning	hi=high	pp=post processing
buf=buffer	init=initialize	ptr=pointer
cam=camera	len=length	rgn=region
clr=clear	lo=low	ser=serial
dly=delay	mem=memory	shtr=shutter
err = error	num=number	spd=speed

Table 2 Standard Abbreviations

In PVCAM, `num` always means **current selection number**, while `total` or `entries` is used for **different possibilities**.

A leading `h` usually signifies a type of handle, such as the camera handle (`hcam`). A handle is a 16-bit number that is used to uniquely identify an object.

Include Files

Any program using PVCAM must include the following files:

- `master.h` - system-specific definitions and types (must be included before `pvcam.h`)
- `pvcam.h` - constants and prototypes for all functions

Parameter Passing and const

When parameters are passed in or out of functions, it may be difficult to determine which parameters the user should set and which parameters are set by the function. This is particularly difficult in PVCAM, because virtually all information is exchanged through parameters (the function return value is reserved for indicating errors).

A few simple rules help resolve the confusion:

- Non-pointers always send information **to** a function.
- Constant pointers always send information **to** a function. The data is not altered.
- Non-constant pointers generally return information **from** a function.

In a few cases the non-constant pointer is used also for input function argument. For instance the camera name string in `pl_cam_open` function is passed as non-const pointer and cannot be change without breaking binary compatibility. Another example is void pointer to raw frame data.

Coordinates Model

In many cameras, the sensor orientation is fixed. This fixed position places the origin in a predetermined location and gives each pixel an x,y location.

In Photometrics cameras, the sensor orientation is not only different from camera to camera, but the orientation may also change when the application changes. Therefore, we use a **serial, parallel** (s,p)

coordinates system. In this system, the origin is located in the corner closest to the serial register readout, and the coordinates increase as the locations move away from the origin. The diagram below illustrates how the coordinates are unaffected by the sensor orientation.

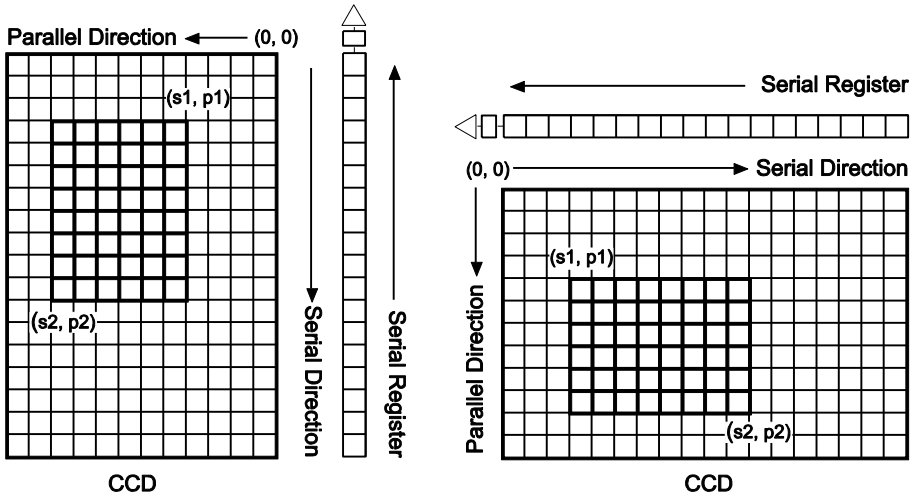

Figure 2 Serial and parallel registers

Regions and Images

A region is a user-defined, rectangular exposure area on the sensor. As seen in the diagram above, the user defines the region by selecting `s1,p1` and `s2,p2`, the diagonal corners of the region.

An image is the data collected from a region. PVCAM reads out the image, then stores it in a buffer.

Binning Factors

For data collection, two other parameters are needed: the serial and parallel binning factors. A binning of 1 in both directions reads out each pixel at full lateral resolution. A binning of 2 in both directions combines four pixels, cutting the lateral resolution in half, but quadrupling the light-collecting area. The number of pixels read out are determined as `(s2-s1+1)/sbin` in the serial direction, and `(p2-p1+1)/pbin` in the parallel direction. If these equations do not produce an integer result, the remaining pixels are ignored.

Including binning, a data collection region can be fully specified with six parameters: `s1,p1,s2,p2,sbin,pbin`. Since these values are 0 indexed, the following is true:

```
smax = serial_size - 1
pmax = parallel_size - 1
```

Data Array

When pixels are read out, they are placed in the data array indicated by the pointer passed into pl_exp_start_cont or pl_exp_start_seq. The pixels are placed into an array in the following order:

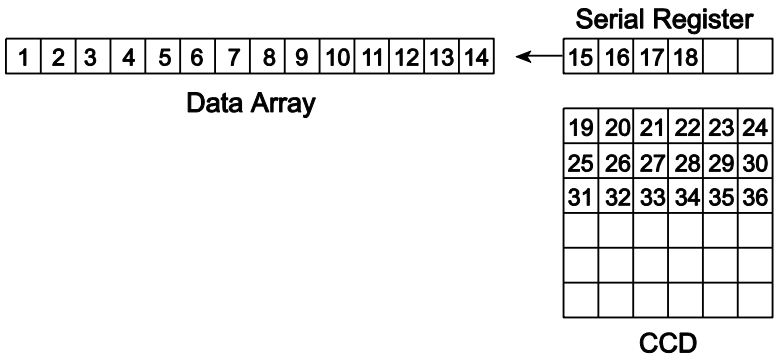

Figure 3 Shifting data out of serial register

Display Orientation

0,0 is displayed in the upper left corner, and subsequent pixels are painted from left to right. Although video coordinate configuration can be done in the display routine, factors such as the optical path, the camera rotation, and which readout port is selected may cause the image to appear in a different position.

Port and Speed Choices

The sensor in a camera will have one or more output nodes from which the pixel stream will be read. These nodes are referred to as “Readout Ports”. The signal from a readout port will be passed through an analog to digital converter (ADC) in the case of a CCD and in the case of the CMOS style sensor, will be digitized internally. The ADC (either internal or external to the sensor) operates at one or more digitization rates and has a set of parameters associated with it. In PVCAM, the choice of speed (digitization rate) and associated ADC parameters are organized into a "speed table". In some cameras, different readout ports will be connected to different analog processing chains and different ADCs. The most general method for setting up the port and speed choices is to make the speed choices dependent upon the port selection.

To display more descriptive information about the current port settings, recall `pl_get_param` with `PARAM_READOUT_PORT` with the `ATTR_CURRENT`. Next, retrieve the relevant descriptive string related to that readout port by calling `pl_get_enum_param` with `PARAM_READOUT_PORT` on the associated value.

To build the speed table, for each valid port call `pl_get_param` with `PARAM_SPDTAB_INDEX` with the `ATTR_COUNT` attribute to determine how many speed entries are allowed on your camera. Then iterate through each choice to get the associated information for that entry. The steps you should take in setting up the readout ports and associated speed tables are as follows:

1) `pl_get_param` with `PARAM_READOUT_PORT` with `ATTR_COUNT` to get the total number of valid ports.
2) `pl_get_enum_param` with `PARAM_READOUT_PORT` to get the enumerated port constants.
3) For each port constant, `pl_set_param` with `PARAM_READOUT_PORT`, and build a speed table for each.

Table 3 Two Port Camera Example is an example of a camera with two readout ports. Port 1 has one speed associated with it and Port 2 has three speeds. Note that the terms "Port 1" and "Port 2" are generic and are only being used to illustrate the example.

The user chooses the port and then the speed table entry number, and the camera is configured accordingly. The user can then choose one of the gain settings available for that speed table entry number. For example, the user chooses Port 2 and speed index one. This selection provides a 16-bit camera with a

pixel time of 500 nanoseconds (a 2 MHz readout rate). The sensor is reading out of Port 2. The gain is set to 2.

Readout Port	Entry	Bit Depth	Pixel Time	Current Gain	Max Gain
	PARAM_SPDTAB_INDEX	PARAM_BIT_DEPTH	PARAM_PIX_TIME	PARAM_GAIN_INDEX	PARAM_GAIN_INDEX (ATTR_MAX)
PORT 1	0	12	500	2	16
PORT 2	0	12	100	1	3
	1	16	500	2	3
	2	12	500	2	3

Table 3 Two Port Camera Example

It is the responsibility of the application program to remember variables associated with port and speed selections. The application must resend gain values when the user changes the current port or speed. Additionally, the application must resend the desired speed whenever a readout port is changed. Read-only values, such as bit depth, must be assumed to be unique for each speed-port combination. Once a selection is made, all settings remain in effect until the user resets them or until the camera hardware is powered down or reset.

Multi-tap Configuration and Readout

The term tap will be used to indicate a port that can participate in simultaneous readout. The configuration of multi-tap will be made using the speed table, and the frame data will be spliced together using firmware in the camera, and does not require any modifications to application functionality to support.

Frame Transfer

A frame transfer CCD is divided into two areas: one for image collection and one for image storage. After the CCD is exposed, the image is shifted to the storage array that is not light sensitive. A split clock allows the CCD to expose the next frame of the image array while simultaneously reading out from the storage array.

Since shifting an image to the storage array is many times faster than reading out the same image, frame transfer speeds up many sequences, in comparison to full frame sensors.

With most frame transfer CCDs, the image in the storage array must be completely read out before the next image is shifted into the storage array. Therefore, assuming that the `exposure_time` for each image within a sequence is equal, the shortest possible `exposure_time` would be exactly equal to the image readout time.

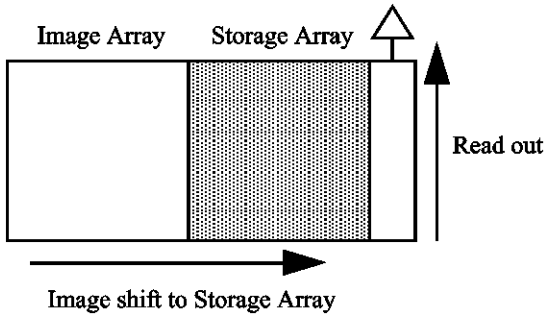

Figure 4 Shifting image into the storage array

Image Smear

If an image is shifted while the shutter is open, the charge that collects while the image is moving makes the image look smeared. Smearing can occur in several situations: if the camera is set to read out without closing the shutter, if the shutter is set to close too slowly, or in frame transfer sequences where the shutter stays open while the image is shifted to the storage array.

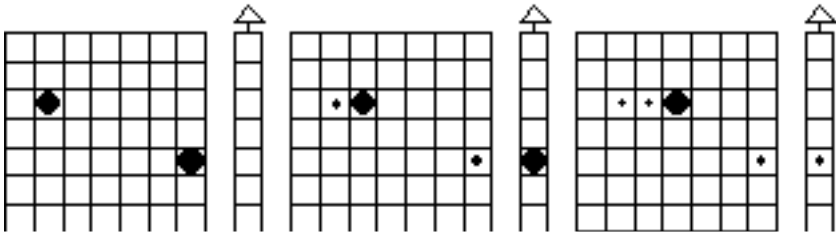

Figure 5 Smearing

In most frame transfer applications, the shutter opens before the sequence begins and closes after the sequence ends. The charge gathered during the shift creates a smear across the image array.

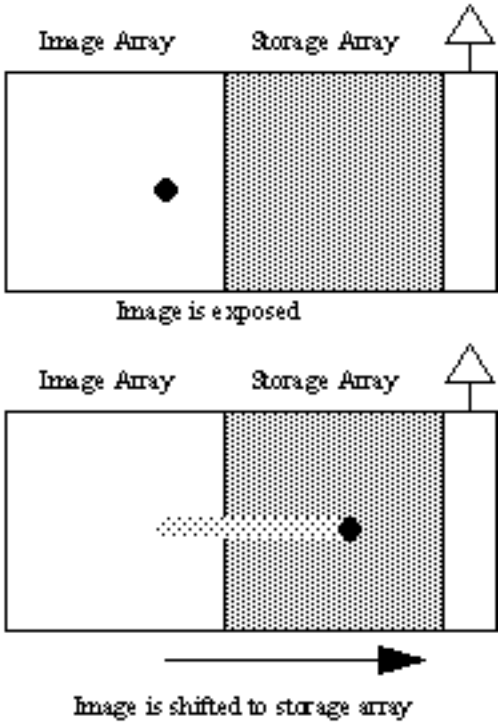

Figure 6 Smearing in frame transfer sequence

Although the frame transfer time is usually small compared to the smallest useful exposure time, smearing cannot be eliminated when the shutter is left open for the entire sequence. The higher the ratio of the `exposure_time` to the frame transfer time, the brighter the image is in comparison to the pattern caused by smearing. An `exposure_time` that is too long will saturate the pixels and cause the image to lose all contrast.

Sequences

A sequence is a programmed series of exposures that is started by a single command. In the least complex sequences, a setup is called then the camera takes a series of exposures with a complete readout between each exposure. In these simple sequences, all the variables in the setup apply to all the exposures in the sequence. The diagram below illustrates a sequence of exposures taken as the day passes.

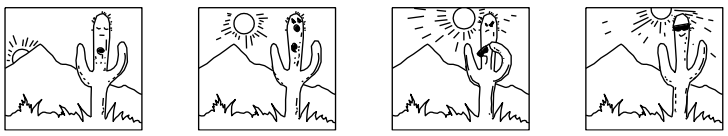

Figure 7 Sequence of exposures

In most camera modes, you must load a new setup into the camera if you want to change a variable between sequences. PVCAM offers a few exceptions to this rule. In variable timed mode, calling a command between sequences sets the `exposure_time` for the next sequence.

Sequence Parameters IDs/Constants

When constructing a sequence, the following three items determine how the camera behaves before reading out:

`PARAM_CLEAR_MODE`
: Parameter that determines if and when the sensor is cleared of charge.

`BULB_MODE, STROBED_MODE, TIMED_MODE, TRIGGER_FIRST_MODE, VARIABLE_TIMED_MODE`
: Constants that determine if a program command or an external trigger starts and ends the exposure/nonexposure time within a sequence.

`PARAM_SHTR_OPEN_MODE`
: Parameter that determines if and when the shutter opens.

Although a single exposure may be considered a sequence of one, some options in triggering, shuttering, and sensor clearing only apply to multiple image sequences.

Circular Buffer

Circular buffer is a special case of acquisition. In a sequence, you specify the number of frames to acquire and allocate a buffer large enough to hold all of the frames. Using a circular buffer allows you to acquire a continuous acquisition; the camera will continue to acquire frames until you decide to stop it, rather than acquiring a specified number of frames. This mode is especially useful for use cases in which the user is looking for a particular event, as he has no guess as to when that event will occur. Additionally, this mode is recommended for displaying what is called a "focus loop", in which the system's optics are focused on the subject.

For a circular buffer, you allocate a buffer to hold a certain number of frames, and the data from the camera is stored in the buffer sequentially until the end of the buffer is reached. When the end is reached, the data is stored starting at the beginning of the buffer again, and so on.

The circular buffer also acts as a load leveling entity between the camera and your application. If the application momentarily cannot keep up with camera the circular buffer will hold the incoming frames and deliver them to the application in a burst once the application gets unblocked again. This is very important for high speed cameras because the operating system itself causes momentary CPU loads or delayed context switches that needs to be balanced. For reliable frame delivery we recommend to use a circular buffer that is able to hold at least one second of acquisition. This means that if the camera runs at 400FPS and the frame size is 512×512 pixels, the recommended circular buffer size would be 512×512×2 × 400 = 200MB.

To help users with choosing the correct circular buffer size we recommend to use the `PARAM_FRAME_BUFFER_SIZE` to retrieve the recommended buffer size for current acquisition.

The image buffer used for a circular buffer is passed to `pl_exp_start_cont`. The buffer is allocated by your application. Data readout is performed directly into the designated circular buffer. Depending on the circular buffer mode, overwriting this buffer may be flagged as an error.

Briefly, an example of circular buffer setup:

- `pl_cam_register_callback` is called to register for EOF camera events.
- `pl_exp_setup_cont(CIRC_OVERWRITE)`: The circular buffer mode is selected.
- `pl_get_param` for `PARAM_FRAME_BUFFER_SIZE` and `ATTR_DEFAULT` is called to retrieve the recommended size in bytes for currently configured acquisition. (optional)
- Circular buffer is allocated
- `pl_exp_start_cont`: Continuous data acquisition is started.

- Frames begin arriving in the buffer.
- Callback routine is called, notifying the application about new frame availability
- `pl_exp_get_latest_frame` is called from within the callback routine. A pointer to the latest not yet retrieved frame is obtained from the circular buffer. If there are more frames queued in the buffer another callback routine will be called immediately after the current one is exited.
- Data is processed (for example, the data is displayed).
- The callback routine is being called for each retrieved frame until continuous data acquisition is stopped with `pl_exp_stop_cont`.

Sequenced Multiple Acquisition Real Time (SMART) streaming

SMART streaming allows you to assign individual exposure settings to each frame of a continuous sequence; the camera will apply the settings just before the frame is captured. Please consult our sales department or camera manuals on information on which cameras support this mode.

The maximum number of SMART streaming entries varies from camera to camera. This parameter can be requested via the `ATTR_MAX` attribute.

The diagram below illustrates SMART streaming for a sequence of 3 frames with exposures of 10ms, 20ms, 30ms, and no delay between frames.

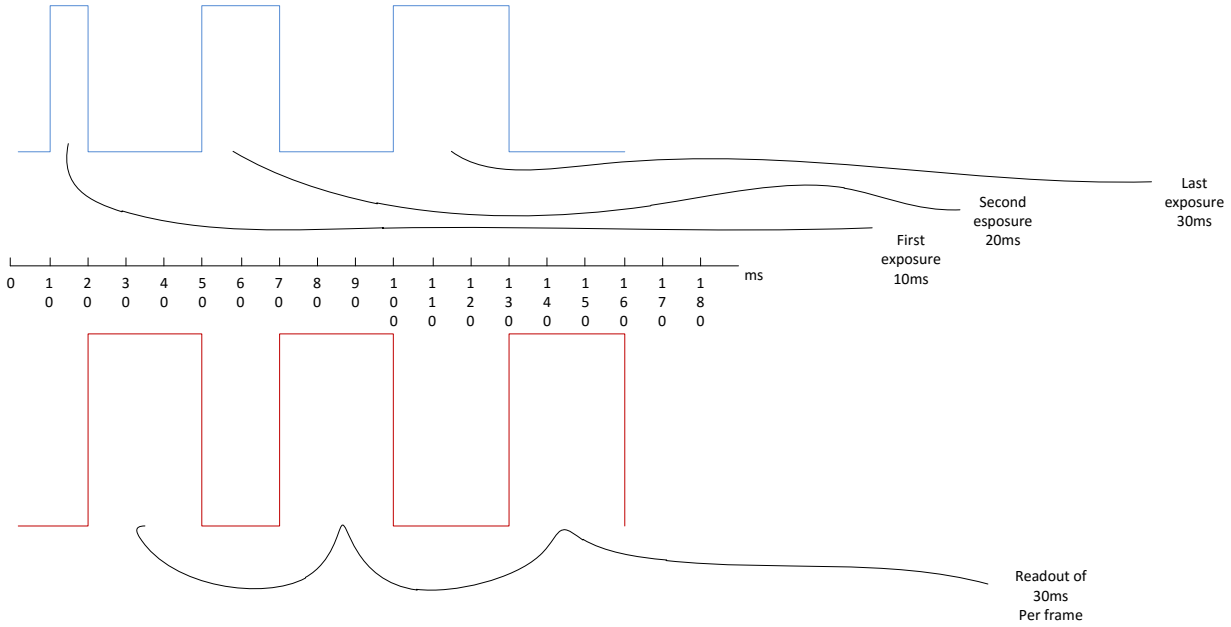

Figure 8 An example of SMART streaming acquisition

An Example SMART streaming acquisition setup:

- `pl_exp_setup_seq` or `pl_exp_setup_cont`: Either single sequence or circular buffer mode is selected. The exposure time in this call needs to be non-zero and it will be overwritten by the exposure times passed by `pl_set_param` with `PARAM_SMART_STREAM_EXP_PARAMS`.
- `pl_set_param` with `PARAM_SMART_STREAM_MODE_ENABLED`: SMART stream mode is enabled.
- `pl_set_param` with `PARAM_SMART_STREAM_EXP_PARAMS`: The exposure parameters are passed in to the camera.
- `pl_exp_start_seq` or `pl_exp_start_cont`: Data acquisition is started.
- The loop is repeated until the buffer fills up or continuous data acquisition is stopped with `pl_exp_stop_cont`.

SMART Streaming Data Types

A SMART streaming acquisition is programmed by sending the camera a list of the individual exposures or delays along with the frame count. To facilitate this process, the `smart_stream_type` encapsulates the required parameters. This data type consists of an `uns16` variable called `entries` and an array of

uns32 values called params. The params variable points to a list of exposures or delays; the entries variable contains the number of entries in the list.

A variable of type smart_stream_type can be filled in two ways:

1) Statically. For example:

```
smart_stream_type ExposureArray;
uns32 exp_values[] = {10, 20 ,30, 40};

ExposureArray.entries = sizeof(exp_values) / sizeof(uns32);
ExposureArray.params = exp_values;

/* acquire the data here */
```

2) Dynamically with the aid of the function pl_create_smart_stream_struct. For example:

```
smart_stream_type* pExposureArray;

pl_create_smart_stream_struct(&pExposureArray, 4);

for (uns16 i = 0; i < pExposureArray->entries; i++)
pExposureArray->params[i] = 10 + i * 10;

/* acquire the data here */

pl_release_smart_stream_struct(&pExposureArray);
```

Possible Scenarios

S.M.A.R.T. Streaming enabled but no arrays passed in.

In this case, only the exposure as defined in the pl_exp_setup_seq will be used.

Exposure Count (N) > Sequence Size (M)

For a finite sequence, the first N entries will be used and the rest will be ignored. For a circular buffer sequence, all of the defined entries will be used in a round-robin fashion (i.e. after a frame has been captured with the last entry defined in the SMART stream parameter, the cycle will be repeated with the first defined entry, then the second, etc.)

Exposure Count (N) < Sequence Size (M)

For a finite sequence, the defined entries will be used in a round-robin fashion and repeated until all frames are consumed. For a circular buffer sequence, all of the defined entries will be used in a round-robin fashion (i.e. after a frame has been captured with the last entry defined in the SMART stream parameter, the cycle will be repeated with the first defined entry, then the second, etc.)

Clear Modes

Clearing removes charge from the sensor by clocking the charge to the serial register then directly to ground. This process is much faster than a readout, because the charge does not go through the readout node or the amplifier. Note that not all clearing modes are available for all cameras. Be sure to check availability of a mode before attempting to set it.

The clear modes are described below:

CLEAR_NEVER
: Don't ever clear the sensor. Useful for performing a readout after an exposure has been aborted.

CLEAR_PRE_EXPOSURE

Before each exposure, clears the sensor the number of times specified by the `clear_cycles` variable. This mode can be used in a sequence. It is most useful when there is a considerable amount of time between exposures.

`CLEAR_PRE_SEQUENCE`
Before each sequence, clears the sensor the number of times specified by the `clear_cycles` variable. If no sequence is set up, this mode behaves as if the sequence has one exposure. The result is the same as using `CLEAR_PRE_EXPOSURE`.

`CLEAR_POST_SEQUENCE`
Clears continuously after the sequence ends. The camera continues clearing until a new exposure is set up or started, the abort command is sent, the speed entry number is changed, or the camera is reset.

`CLEAR_PRE_POST_SEQUENCE`
Clears `clear_cycles` times before each sequence and clears continuously after the sequence ends. The camera continues clearing until a new exposure is set up or started, the abort command is sent, the speed entry number is changed, or the camera is reset.

`CLEAR_PRE_EXPOSURE_POST_SEQ`
Clears `clear_cycles` times before each exposure and clears continuously after the sequence ends. The camera continues clearing until a new exposure is set up or started, the abort command is sent, the speed entry number is changed, or the camera is reset.

Normally during the idle period, the Camera parallel and serial clock drivers revert to a low power state that saves both power and heat. When `CLEAR_...POST` options are used, the clearing prevents these systems from entering low-power mode. This state generates a small amount of additional heat in the electronics unit and the camera head. Depending on the camera platform, this excess heat can cause "glow" in the image which can be prevented by running in a different mode.

The `pl_exp_abort` function stops the data acquisition and the camera goes into the clean cycle. Again, the sensor chip is continuously being cleaned.

Clear Modes decide when to empty the sensor wells.

Exposure Modes

During sequences, the exposure mode determines how and when each exposure begins and ends:

`TIMED_MODE`	`STROBED_MODE`
`VARIABLE_TIMED_MODE`	`BULB_MODE`
`TRIGGER_FIRST_MODE`	

In general, the settings in `pl_exp_setup_seq` apply to each exposure within a sequence. They also apply to every sequence until the setup is reset. The only exceptions are in `VARIABLE_TIMED_MODE` and `BULB_MODE`. These two modes ignore the `exposure_time` parameter in setup, and rely on a function or trigger to determine the exposure time.

Every sequence has alternating periods of exposure and nonexposure time. During the time the sensor is not exposing, the camera could be in several states, such as waiting for `pl_exp_start_seq`, reading out, or performing clearing. In the diagrams that follow, each exposure mode shows the exposure time in white and the time between exposures in gray.

TIMED_MODE

In TIMED_MODE, all settings are read from the setup parameters, making the duration of each exposure time constant and the interval times between exposures constant. In this mode, every sequence has the same settings.

The diagram below represents a sequence in TIMED_MODE.

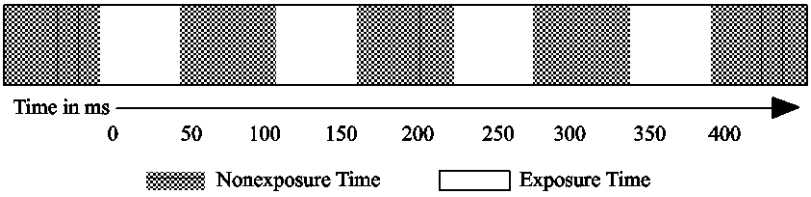

Figure 9 Sequence in TIMED_MODE

VARIABLE_TIMED_MODE

Use VARIABLE_TIMED_MODE when you want to change the exposure_time between sequences.

In VARIABLE_TIMED_MODE, all settings except exposure_time are read from the setup parameters. The exposure_time must be set with parameter ID PARAM_EXP_TIME. If you do not call PARAM_EXP_TIME before the first sequence, a random time will be assigned. The camera will not read the first exposure time from the exposure_time in setup, because this mode ignores the exposure_time parameter.

Application example: A filter wheel is used to change the filter color between sequences. The exposure time needed for the darkest filter saturates the pixels when lighter filters are used. The diagram on the next page shows two sample sequences from this example.

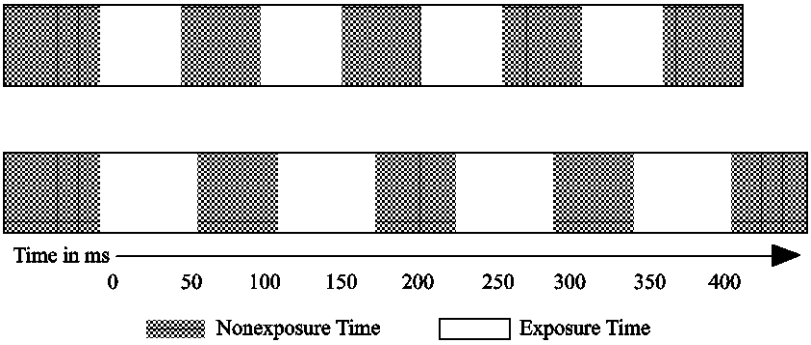

Figure 10 Sequence in VARIABLE_TIMED_MODE

The first sequence runs with a filter that uses exposure and nonexposure times that are equal. In the second sequence, the exposure time is longer, but the time between exposures remains the same as in the first sequence.

TRIGGER_FIRST_MODE

Use TRIGGER_FIRST_MODE when you want an external trigger to signal the start of the sequence.

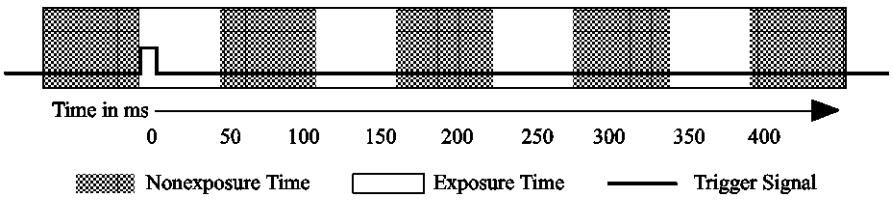

Figure 11 Sequence in TRIGGER_FIRST_MODE

In `TRIGGER_FIRST_MODE`, `pl_exp_start_seq` starts the camera, which enters the clear mode while it waits for a trigger signal. The black line in the diagram illustrates a trigger signal coming from an external trigger source.

Once the outside event triggers the camera to start exposing, the sequence follows the conditions generated in `pl_exp_setup_seq`. Note that all exposure times are equal, and the time intervals between exposures are equal.

You must have an external trigger signal connected to your camera for `TRIGGER_FIRST_MODE` to function. If your equipment fails to send a trigger signal, you can stop the sequence by calling `pl_exp_abort`.

> **Note:** If you do not use one of the `CLEAR_PRE_EXPOSURE` modes, the sensor will begin exposing immediately after `pl_exp_start_seq` is called. Once the trigger is received, the sensor will continue to expose for the `exposure_time` specified in `pl_exp_setup_seq`. In other words, the first exposure in your sequence may have a longer exposure time than the subsequent exposures.

STROBED_MODE

Use `STROBED_MODE` when you want an external trigger to start each exposure in the sequence.

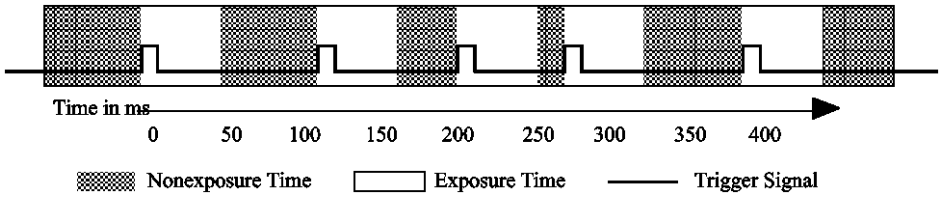

Figure 12 Sequence in STROBED_MODE

In `STROBED_MODE`, `pl_exp_start_seq` starts the camera. The camera enters clear mode while it waits for the first trigger signal to start the first exposure. As shown in the diagram above, each new exposure waits for an external trigger signal. Notice that the intervals between exposures can vary greatly, but the exposure times are constant.

You must have an external trigger signal connected to your camera for this mode to function. If your equipment fails to send a trigger signal, you can stop the sequence by calling `pl_exp_abort`.

Application example: In a nature study of birds passing through a restricted area, the motion of each bird sends a trigger signal to the camera. The camera exposes, reads out, and waits for the next trigger signal. The result is an image of each bird as it crosses the camera's field of view.

> **Note:** If you do not use one of the `CLEAR_PRE_EXPOSURE` modes, the sensor will begin exposing immediately after `pl_exp_start_seq` is called. Once the trigger is received, the sensor will continue to expose for the `exposure_time` specified in `pl_exp_setup_seq`. In other words, the first exposure in your sequence may have a longer exposure time than the subsequent exposures.

BULB_MODE

Use `BULB_MODE`, when you want an external trigger signal to control the beginning and end of each exposure.

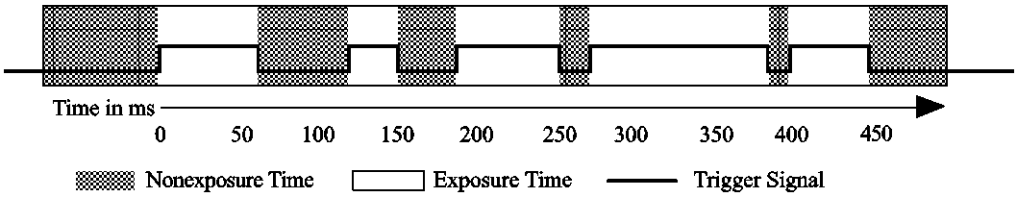

Figure 13 Sequence in BULB_MODE

In `BULB_MODE`, `pl_exp_start_seq` calls the setup. The camera enters clear mode while it waits for a **true** external trigger signal to start each exposure. The sensor continues to expose until a **false** trigger signal ends the exposure. In the diagram above, the trigger signal line moves up to represent a **true** trigger and down to represent a **false** trigger.

Notice that the exposure times and the intervals between exposures vary greatly. Since the **true** and **false** signals determine exposure time, the `exposure_time` set in `pl_exp_setup_seq` is ignored.

You must have an external trigger signal connected to your camera for `BULB_MODE` to function. If your equipment fails to send a trigger signal, you can stop the sequence by calling `pl_exp_abort`.

> **Note:** If you do not use one of the `CLEAR_PRE_EXPOSURE` modes, the sensor exposes until receiving a false trigger signal, then reads out. After reading out, the sensor exposes again without clearing and waits for the true trigger. Once the external event causes a true trigger, the sensor continues to expose until receiving a false trigger, then reads out. In other words, the sensor will expose from the end of readout until the next false trigger.

Extended Exposure Modes

Since PVCAM 3.0.1 the newly added Exposure Modes can be retrieved from the camera directly and cross-correlated with values in the PVCAM header files so that applications can narrow which modes they would like to operate with. Application developers are encouraged to not hard code the exposure modes in the source code but read the supported modes from the camera dynamically. This change was introduced together with the addition of the Expose Out Modes.

The following example shows how to use the Extended Exposure Modes together with Expose Out Modes and still keep the backward compatibility with older cameras.

Define a helper function to retrieve available parameter values:

```
// A helper function that enumerates a given parameter from the camera
void EnumerateParameter(int16 hCam, uns32 paramID, std::vector<int32>& values,
                        std::vector<std::string>& names)
{
    rs_bool bAvail = FALSE;
    uns32 count = 0;
    values.clear();
    names.clear();
    // Check the availability of the parameter
    if (pl_get_param(hCam, paramID, ATTR_AVAIL, &bAvail) != PV_OK || bAvail == FALSE)
        return;
    // Get the number of expose out modes
    if (pl_get_param(hCam, paramID, ATTR_COUNT, &count) != PV_OK)
        return;
    // Get the mode values and names
    for (uns32 i = 0; i < count; ++i)
    {
        uns32 enumStrLen;
        if (pl_enum_str_length(hCam, paramID, i, &enumStrLen) == PV_OK)
        {
            char* enumStr = new char[enumStrLen]; // Allocate a string buffer
            int32 enumVal;
            if (pl_get_enum_param(hCam, paramID, i, &enumVal, enumStr, enumStrLen)
                    == PV_OK)
```

```
            {
                values.push_back(enumVal);
                names.push_back(std::string(enumStr));
            }
            delete[] enumStr;
        }
    }
}
```

Setup the acquisition:

```
void SetupAcquisition(int16 hCam)
{
    rs_bool bAvail;

    std::vector<int32> trigModeVals;
    std::vector<std::string> trigModeStrs;
    int16 selectedTrigMode = 0;

    std::vector<int32> expOutModeVals;
    std::vector<std::string> expOutModeStrs;
    int16 selectedExpOutMode = 0;

    if (pl_get_param(hCam, PARAM_EXPOSURE_MODE, ATTR_AVAIL, &bAvail) != PV_OK
            || bAvail == FALSE)
    {
        selectedTrigMode = TIMED_MODE;
    }
    else
    {
        // The enumeration should be done upon opening the camera when an UI element
        // can be populated with available exposure modes
        EnumerateParameter(hCam, PARAM_EXPOSURE_MODE, trigModeVals, trigModeStrs);
        selectedTrigMode = (int16)trigModeVals[0]; // Or any other selected by user
    }

    if (pl_get_param(hCam, PARAM_EXPOSE_OUT_MODE, ATTR_AVAIL, &bAvail) != PV_OK
        || bAvail == FALSE)
    {
        selectedExpOutMode = 0; // This will have no effect when doing bitwise OR
    }
    else
    {
        EnumerateParameter(hCam, PARAM_EXPOSE_OUT_MODE, expOutModeVals, expOutModeStrs);
        selectedExpOutMode = (int16)expOutModeVals[0]; // Or any other selected by user
    }

    const int16 finalExpMode = selectedTrigMode | selectedExpOutMode;

    const rgn_type roi = { 0, m_SerSz-1, 1, 0, m_ParSz-1, 1 }; // Acquire full frame
    uns32 bufferSize;
    // Setup the acquisition, 1 frame, 1 roi, 10ms exposure
    if (pl_exp_setup_seq(hCam, 1, 1, &roi, finalExpMode, 10, &bufferSize) != PV_OK)
        return false;

    // Next: Allocate the buffer and call pl_exp_start_cont()
}
```

Expose Out Modes

Expose Out Modes determine the behavior of the camera expose out IO signal. This parameter is camera dependent, please refer to your camera manual for more information.

Since PVCAM 3.0.1 the new enumerable parameter can be retrieved dynamically from the currently connected camera and cross-correlated with the PVCAM header so that applications can decide which modes they prefer to operate in without user intervention.

Please refer to the *Extended Exposure Modes* chapter on page 25 for a code example showing how to retrieve and use the Expose Out Modes with new cameras.

Open Delay, Close Delay

In order to ensure that the entire sensor is exposed for the specified `exposure_time`, the mechanical limitations of the shutter must be considered. Open delay (`PARAM_SHTR_OPEN_DELAY`) and close delay (`PARAM_SHTR_CLOSE_DELAY`) account for the time necessary for the shutter to open and close. Remember that the camera is exposing while the shutter is opening and closing, so some pixels are exposed longer than others.

An Iris shutter opens in an expanding circular pattern.

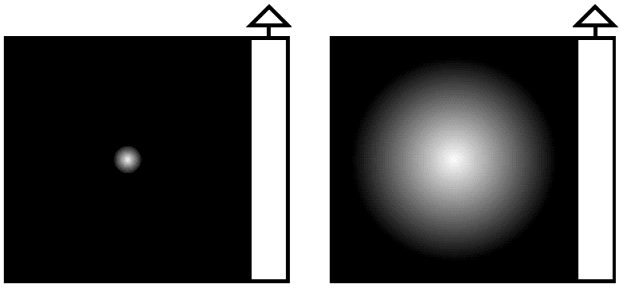

Figure 14 Iris Shutter

A Barn Door shutter slides across the exposure area.

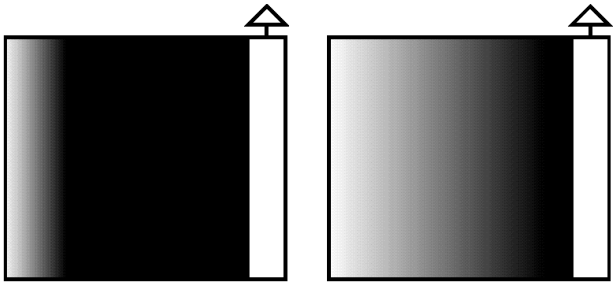

Figure 15 Barn Door Shutter

If the shutter is still closing when the image shifts for a frame transfer or readout, the image will smear. (See the section "*Image Smear*" for a more complete explanation on smearing.) `PARAM_SHTR_CLOSE_DELAY` allows time for the shutter to close before the image shifts.

The default open and close delay values will vary depending on the brand of camera and the shutter used. Open delay may be up to 15 milliseconds with a close delay of up to 30 milliseconds. Change the default values only if you are using a shutter other than the shutter shipped with your camera. **If you are using a standard Photometrics shutter, changing `PARAM_SHTR_OPEN_DELAY/CLOSE_DELAY` default values will not increase the frame transfer rate.**

Shutter Control

The shutter open modes determine how the shutter in a camera behaves when a single exposure is taken or when a sequence is run. Remember that the camera is exposing while the shutter is opening. Because not all supported cameras have programmable shutter control, remember to check for availability of a particular mode.

`OPEN_PRE_EXPOSURE`

Opens the shutter before every exposure, then closes the shutter after the exposure is finished.

`OPEN_PRE_SEQUENCE`
Opens the shutter before the sequence begins, then closes the shutter after the sequence is finished.

`OPEN_PRE_TRIGGER`
Opens the shutter, then clears or exposes (set in clear mode) until a trigger signal starts the exposure.

`OPEN_NEVER`
Keeps shutter closed during the exposure. Used for dark exposures.

`OPEN_NO_CHANGE`
Sends no signals to open or close the shutter.

Exposure Loops

Within an exposure loop, the interaction of the exposure, clear, and shutter open modes determines how the camera behaves during a sequence. In the following pages, sample command sequences show how each exposure mode acts in combination with each clear and shutter open mode. As mentioned above in "Shutter Control", not all supported cameras have programmable shutter control, remember to check for availability of a particular mode.

Key	Description
ClearN	Clear sensor N times as specified in `clear_cycles`
OS	Open shutter and perform `PARAM_SHTR_OPEN_DELAY`
CS	Close shutter and perform `PARAM_SHTR_CLOSE_DELAY`
EXP	Expose sensor for `exposure_time`
I->S	Transfer image array to storage array (frame transfer)
Readout	Readout sensor (readout storage array for frame transfer)
WaitT	Wait until trigger
EXP Until notT	Expose sensor until trigger end `(BULB_MODE)`
Items in *ITALICS* repeat M times for a sequence of M exposures.	
Items in **BOLD** are outside of the sequence loop.	

Table 4 Legend of sample command sequences

Clear Mode	Shutter Mode	Command Sequence
CLEAR_PRE_EXPOSURE	OPEN_PRE_EXPOSURE	*ClearN, OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS**, *ClearN, EXP, I->S, Readout*, **CS**
	OPEN_PRE_TRIGGER	*ClearN, OS, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	*ClearN, EXP, I->S, Readout*
	OPEN_NEVER	***CS,*** *ClearN, EXP, I->S, Readout*
CLEAR_PRE_SEQUENCE	OPEN_PRE_EXPOSURE	**ClearN**,*OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS, ClearN**, *EXP, I->S, Readout*, **CS**
	OPEN_PRE_TRIGGER	**ClearN**, *OS, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	**ClearN**, *EXP, I->S, Readout*
	OPEN_NEVER	***CS,*** **ClearN**, *EXP, I->S, Readout*
CLEAR_NEVER	OPEN_PRE_EXPOSURE	*OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS,** *EXP, I->S, Readout*, **CS**
	OPEN_PRE_TRIGGER	*OS, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	*EXP, I->S, Readout*
	OPEN_NEVER	***CS,*** *EXP, I->S, Readout*

Table 5 TIMED_MODE command sequences

Clear Mode	Shutter Mode	Command Sequence
CLEAR_PRE_EXPOSURE	OPEN_PRE_EXPOSURE	**EXP+WaitT,** *ClearN, OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS, EXP+WaitT,** *ClearN, EXP, I->S, Readout, CS*
	OPEN_PRE_TRIGGER	**EXP+WaitT,** *OS, ClearN, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	**EXP+WaitT,** *ClearN, EXP, I->S, Readout*
	OPEN_NEVER	***CS,* EXP+WaitT,** *ClearN, EXP, I->S, Readout*
CLEAR_PRE_SEQUENCE	OPEN_PRE_EXPOSURE	**Clear+WaitT,** *ClearN, OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS, Clear+WaitT,** *EXP, I->S, Readout, CS*
	OPEN_PRE_TRIGGER	**Clear+WaitT,** *OS, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	**Clear+WaitT,** *EXP, I->S, Readout*
	OPEN_NEVER	***CS,* Clear+WaitT,** *EXP, I->S, Readout*
CLEAR_NEVER	OPEN_PRE_EXPOSURE	**EXP+WaitT,** *ClearN, OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS, EXP+WaitT,** *EXP, I->S, Readout, CS*
	OPEN_PRE_TRIGGER	**EXP+WaitT,** *OS, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	**EXP+WaitT,** *EXP, I->S, Readout*
	OPEN_NEVER	***CS,* EXP+WaitT,** *EXP, I->S, Readout*

Table 6 TRIGGER_FIRST_MODE command sequences

Clear Mode	Shutter Mode	Command Sequence
CLEAR_PRE_EXPOSURE	OPEN_PRE_EXPOSURE	*Clear+WaitT, OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS**, *Clear+WaitT, EXP, I->S, Readout,* **CS**
	OPEN_PRE_TRIGGER	*OS, Clear+WaitT, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	*Clear+WaitT, EXP, I->S, Readout*
	OPEN_NEVER	***CS,*** *Clear+WaitT, EXP, I->S, Readout*
CLEAR_PRE_SEQUENCE	OPEN_PRE_EXPOSURE	**ClearN**, *EXP+WaitT, OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS, ClearN,** *EXP+WaitT, EXP, I->S, Readout,* **CS**
	OPEN_PRE_TRIGGER	**ClearN**, *OS, EXP+WaitT, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	**ClearN**, *EXP+WaitT, EXP, I->S, Readout*
	OPEN_NEVER	***CS,*** **ClearN**, *EXP+WaitT, EXP, I->S, Readout*
CLEAR_NEVER	OPEN_PRE_EXPOSURE	*EXP+WaitT, OS, EXP, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS,** *EXP+WaitT, EXP, I->S, Readout,* **CS**
	OPEN_PRE_TRIGGER	*OS, EXP+WaitT, EXP, CS, I->S, Readout*
	OPEN_NO_CHANGE	*EXP+WaitT, EXP, I->S, Readout*
	OPEN_NEVER	***CS,*** *EXP+WaitT, EXP, I->S, Readout*

Table 7 STROBED_MODE command sequences

Clear Mode	Shutter Mode	Command Sequence
CLEAR_PRE_EXPOSURE	OPEN_PRE_EXPOSURE	*Clear+WaitT, OS, EXP Until notT, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS**, *Clear+WaitT, EXP Until notT, I->S, Readout*, **CS**
	OPEN_PRE_TRIGGER	*OS, Clear+WaitT, EXP Until notT, CS, I->S, Readout*
	OPEN_NO_CHANGE	*Clear+WaitT, EXP Until notT, I->S, Readout*
	OPEN_NEVER	***CS,*** *Clear+WaitT, EXP Until notT, I->S, Readout*
CLEAR_PRE_SEQUENCE	OPEN_PRE_EXPOSURE	**ClearN,** *EXP+WaitT, OS, EXP Until notT, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS, ClearN,** *EXP+WaitT, EXP Until notT, I->S, Readout*, **CS**
	OPEN_PRE_TRIGGER	**ClearN**, *OS, EXP+WaitT, EXP Until notT, CS, I->S, Readout*
	OPEN_NO_CHANGE	**ClearN**, *EXP+WaitT, EXP Until notT, I->S, Readout*
	OPEN_NEVER	***CS,*** **ClearN**, *EXP+WaitT, EXP Until notT, I->S, Readout*
CLEAR_NEVER	OPEN_PRE_EXPOSURE	*EXP+WaitT, OS, EXP Until notT, CS, I->S, Readout*
	OPEN_PRE_SEQUENCE	**OS**, *EXP+WaitT, EXP Until notT, I->S, Readout*, **CS**
	OPEN_PRE_TRIGGER	*OS, EXP+WaitT, EXP Until notT, CS, I->S, Readout*
	OPEN_NO_CHANGE	*EXP+WaitT, EXP Until notT, I->S, Readout*
	OPEN_NEVER	***CS,*** *EXP+WaitT, EXP Until notT, I->S, Readout*

Table 8 BULB_MODE command sequences

Image Buffers

When exposures include multiple images and complex sequences, you may choose to store the images in a buffer. PVCAM has a number of buffer routines that handle memory allocation and freeing. The following list describes images you may choose to store in a buffer.

- **Full Sensor:** A single exposure where the entire sensor is treated as one region and image data are collected over the full sensor. All the data are stored in a single buffer.

Table 9 Single exposure – full image

- **Single Exposure, Custom Region:** A single exposure with a region. Less data than the full sensor produces is stored in the single buffer.

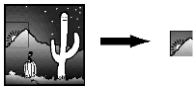

Table 10 Single exposure – custom region

- **Sequences:** A series of exposures with identical regions. The data are stored in several image arrays that are stored inside a single buffer.

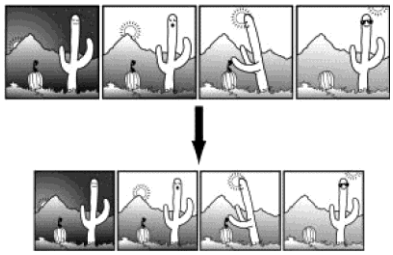

Table 11 Multiple exposures – full image

- **Multiple Exposures, Custom Region:** A series of exposures with a single region. Each exposure must have an identical region. The data is all stored in a single buffer.

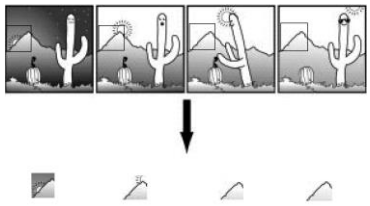

Table 12 Multiple exposures – custom region

PVCAM collects data very efficiently, but moving the data in and out of a buffer involves extra processing time. If speed is crucial, the following options may minimize processing time:

- Don’t use an extra buffer. The data are collected in a user-specified pixel stream at maximum efficiency (see `pl_exp_start_seq`). As discussed in "*Data Array*", this array can be accessed directly. However, when a region is collected, the stream becomes more complex.

Chapter 3: Camera Communication (Class 0)

Introduction

The functions in this category provide a pipeline for bidirectional communication. The table below lists the current Class 0 functions, and the "*Class 0 Functions*" section provides detailed descriptions of each. For more information about the `pl_get_param` and `pl_set_param` parameter IDs, refer to *Chapter 5*, starting on page 56.

List of Available Class 0 Functions

Class 0 functions listed below can be split into two groups. First few functions work with PVCAM library itself and do not take camera handle as an argument. The functions from second group communicate with camera thus require camera handle.

Library	Camera
`pl_pvcam_init`	`pl_cam_close`
`pl_pvcam_uninit`	`pl_cam_get_name`
`pl_pvcam_get_ver`	`pl_cam_get_total`
	`pl_cam_open`
	`pl_cam_register_callback`
	`pl_cam_register_callback_ex`
	`pl_cam_register_callback_ex2`
	`pl_cam_register_callback_ex3`
	`pl_cam_deregister_callback`

List of Available Class 0 Parameter IDs

The following are available Class 0 parameters used with `pl_get_param`, `pl_set_param`, `pl_get_enum_param`, and `pl_enum_str_length` functions specified in *Chapter 5*.

`PARAM_DD_INFO`	`PARAM_DD_TIMEOUT`
`PARAM_DD_INFO_LENGTH`	`PARAM_DD_VERSION`
`PARAM_DD_RETRIES`	

Class 0 Functions

PVCAM	**Class 0: Camera Communication**	**pl_cam_close(0)**
NAME	`pl_cam_close` — frees the current camera, prepares it for power-down.	
SYNOPSIS	**`rs_bool`** **`  pl_cam_close(`** **`  int16 hcam)`**	
DESCRIPTION	This has two effects. First, it removes the listed camera from the reserved list, allowing other users to open and use the hardware. Second, it performs all cleanup, close-down, and shutdown preparations needed by the hardware. A camera can only be closed if it was previously opened; `hcam` must be a valid camera handle.	
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.	
SEE ALSO	`pl_cam_open(0), pl_pvcam_init(0), pl_pvcam_uninit(0)`	
NOTES	pl_pvcam_uninit automatically calls a `pl_cam_close` on all cameras opened by the current user.	

PVCAM **Class 0: Camera Communication** **pl_cam_get_name(0)**

NAME

`pl_cam_get_name` — returns the name of a camera.

SYNOPSIS

```
rs_bool
  pl_cam_get_name(
            int16 cam_num,
            char* cam_name)
```

DESCRIPTION

This function allows a user to learn the string identifier associated with every camera on the current system. This is a companion to the `pl_cam_get_total` function. `cam_num` input can run from 0 to (`total_cams` - 1), inclusive. The user must pass in a string that is at least `CAM_NAME_LEN` characters long; `pl_cam_get_name` then fills that string with an appropriate null-terminated string. `cam_name` can be passed directly into the `pl_cam_open` function. It has no other use, aside from providing a brief description of the camera.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_cam_get_total(0), pl_cam_open(0), pl_cam_close(0)`

NOTES

This call reports the names of all cameras on the system, even if all the cameras are not available. If the hardware is turned off, or if another user has a camera open, the camera name is reported, but is not available.

`pl_cam_get_name` returns a name, and `pl_cam_open` gives information on availability of that camera. This function actually searches for all device drivers on the system, without checking hardware. To build a complete list of every camera on the system, it is necessary to cycle through all entries, as shown below:

```
int total_cameras;
char name[CAM_NAME_LEN];
...
pl_cam_get_total(&total_cameras);
for (i = 0; i < total_cameras; i++) {
    pl_cam_get_name(i, name);
    printf("Camera %d is called '%s'\n", i, name);
}
```

PVCAM	**Class 0: Camera Communication** **pl_cam_get_total(0)**
NAME	`pl_cam_get_total` — returns the number of cameras attached to the system.
SYNOPSIS	**`rs_bool`** **`pl_cam_get_total(`** **`int16* total_cams)`**
DESCRIPTION	This reports on the number of cameras on the system. All listed cameras may not all be available; on multi-tasking systems, some cameras may already be in use by other users. A companion function, `pl_cam_get_name`, can be used to learn the string identifier associated with each camera.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_cam_get_name(0), pl_cam_open(0), pl_cam_close(0)`
NOTES	This function actually searches for all device drivers on the system, without checking hardware. The list of cameras is obtained during `pl_pvcam_init`. Thus, if a new camera (new device driver) is added after the library was opened, the system won't know that the new camera is there. The system also will not notice if a camera is removed. (Obviously, this is only important on multi-tasking systems). A cycle of `uninit/init` regenerates the list of available cameras, updating the system for any additions or deletions.

PVCAM **Class 0: Camera Communication** **pl_cam_open(0)**

NAME

`pl_cam_open` — reserves and initializes the camera hardware.

SYNOPSIS

```
rs_bool
   pl_cam_open(
               char* cam_name,
               int16* hcam,
               int16 o_mode)
```

DESCRIPTION

The string cam_name should be identical to one of the valid camera names returned by `pl_cam_get_name`. If the name is valid, `pl_cam_open` completes a short set of checks and diagnostics as it attempts to establish communication with the camera electronics unit. If successful, the camera is opened and a valid camera handle is passed back in `hcam`. Otherwise, `pl_cam_open` returns with a failure. An explanation is shown in `pl_error_code`.

The `o_mode` setting controls the mode under which the camera is opened. Currently, the only possible choice is `OPEN_EXCLUSIVE`. On multi-user systems, opening a camera under the exclusive mode reserves it for the current user, locking out all other users on the system. If `pl_cam_open` is successful, the user has sole access to that camera until the camera is closed or `pl_pvcam_uninit` is called.

WARNING

Despite the above paragraph, a **successful** `pl_cam_open` does not mean that the camera is in working order. It **does** mean that you can communicate with the device driver associated with the camera. After a successful `pl_cam_open`, call `pl_error_message`, which reports any error conditions.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_cam_get_name(0), pl_cam_get_total(0), pl_cam_close(0), pl_pvcam_init(0), pl_pvcam_uninit(0)`

NOTES

PVCAM **Class 0: Camera Communication** **pl_pvcam_get_ver(0)**

NAME

`pl_pvcam_get_ver` — returns the PVCAM version number.

SYNOPSIS

```
rs_bool
   pl_pvcam_get_ver(
              uns16* version)
```

DESCRIPTION

This returns a version number for this edition of PVCAM. The version is a highly formatted hexadecimal number, of the style:

High byte	Low byte	
	High nibble	Low nibble
Major version	Minor version	Trivial version

For example, the number 0x11F1 indicates major release 17, minor release 15, and trivial change 1.

A major release is defined as anything that alters the interface, calling sequence, parameter list, or interpretation of any function in the library. This includes new functions and alterations to existing functions, but it does not include alterations to the options libraries, which sit on top of PVCAM (each option library includes its own, independent version number).

A new major release often requires a change in the PVCAM library, but wherever possible, major releases are backward compatible with earlier releases.

A minor release should be completely transparent to higher-level software (PVCAM) but may include internal enhancements. The trivial version is reserved for use by the software staff to keep track of extremely minor variations. The last digit is used for build numbers, and should be ignored. Minor and trivial releases should require no change in the calling software.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`PARAM_DD_VERSION`

NOTES

PVCAM	**Class 0: Camera Communication**	**pl_pvcam_init(0)**

NAME

`pl_pvcam_init` — opens and initializes the library.

SYNOPSIS

```
rs_bool
   pl_pvcam_init(void)
```

DESCRIPTION

The PVCAM library requires significant system resources: memory, hardware access, etc. `pl_pvcam_init` prepares these resources for use, as well as allocating whatever static memory the library needs. Until `pl_pvcam_init` is called, every PVCAM function (except for the error reporting functions) will fail and return an error message that corresponds to "`library has not been initialized`".

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_pvcam_uninit(0), pl_cam_open(0), pl_error_code(1)`

NOTES

If this call fails, `pl_error_code` contains the code that lists the reason for failure.

PVCAM	Class 0: Camera Communication	pl_pvcam_uninit(0)

NAME

`pl_pvcam_uninit` — closes the library, closes all devices, frees memory.

SYNOPSIS

```
rs_bool
   pl_pvcam_uninit(void)
```

DESCRIPTION

This releases all system resources that `pl_pvcam_init` acquired. It also searches for all cameras that the user has opened. If it finds any, it will close them before exiting. It will also unlock and free memory, and clean up after itself as much as possible.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_pvcam_init(0), pl_cam_close(0), pl_error_code(1)`

KNOWN BUGS

If the hardware is involved in acquiring data, the system may not be able to disconnect immediately.

PVCAM **Class 0: Camera Communication** **pl_cam_register_callback(0)**

NAME

`pl_cam_register_callback` — installs a function that will be called when an event occurs in a camera system.

SYNOPSIS

```
rs_bool
   pl_cam_register_callback(
               int16 hcam,
               PL_CALLBACK_EVENT event,
               void* Callback)
```

DESCRIPTION

Use this API call to install a function that will be called when the specified event occurs with respect to the camera system indicated.

The `hcam` parameter must reference an open camera system.

The `event` parameter must be one of the following:

```
PL_CALLBACK_BOF
PL_CALLBACK_EOF
PL_CALLBACK_CHECK_CAMS
PL_CALLBACK_CAM_REMOVED
PL_CALLBACK_CAM_RESUMED
```

The `Callback` function must be a function taking no parameters and returning no value. For example:

```
void BOFCallback(void)
{
    BOFCount++;
    return;
}
```

WARNING

`pl_exp_finish_seq` must be called if acquiring in sequential mode (using `pl_exp_setup_seq` and `pl_exp_start_seq`) with callbacks notification after all frames are read out and before new exposure is started by calling `pl_exp_start_seq`.

Not all callbacks will be available for all camera systems/interfaces. The callback descriptions below indicate which callbacks are available on which interfaces.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_cam_deregister_callback(0)`

NOTES

Only `PL_CALLBACK_BOF` and `PL_CALLBACK_EOF` are fully supported by all camera types. Do not use other callback types in generic-purpose software.

Callback descriptions:

`PL_CALLBACK_BOF`
: Called when data arrives corresponding to the beginning of frame readout. This can be used as a trigger to move filter wheels, stages, etc., as depending on the clearing mode, the camera should not be exposing. This is a potentially high-frequency event; long duration processing should not be done directly in this callback, but queued for processing in another thread instead. Taking too long to process a BOF or EOF event could result in missing subsequent events.

`PL_CALLBACK_EOF`
: Called when data arrives corresponding to the end of the frame, usually indicating the beginning of exposure. This is also a potentially high-frequency event; see `PL_CALLBACK_BOF` above.

`PL_CALLBACK_CHECK_CAMS`
: On cameras with hot-pluggable buses (IEEE1394), this indicates that there is a potential for cameras to have been added to the bus. The application can use this as an indication that it should close PVCAM, re-open it, and look for new cameras.

`PL_CALLBACK_CAM_REMOVED`
: This callback is called when a hot-pluggable camera has been removed from the system, and is an indication that the camera should be closed.

`PL_CALLBACK_CAM_RESUMED`
: On camera systems supporting suspend/resume, and for camera systems with hot-pluggable buses, this indicates that the system has come back from a low-power state. If your camera is not self-powered, it probably lost power and therefore any settings that your application may have sent it. For those camera systems, this is an indication that the application should re-initialize the system.

PVCAM **Class 0: Camera Communication** **pl_cam_register_callback_ex(0)**

NAME

`pl_cam_register_callback_ex` — installs a function that will be called when an event occurs in a camera system with context.

SYNOPSIS

```
rs_bool
   pl_cam_register_callback_ex(
              int16 hcam,
              PL_CALLBACK_EVENT event,
              void* Callback,
              void* Context)
```

DESCRIPTION

Use this API call to install a function that will be called when the specified event occurs with respect to the camera system indicated supplying a context that will be echoed back when the callback is invoked.

The `hcam` parameter must reference an open camera system.

The `event` parameter must be one of the following:

```
PL_CALLBACK_BOF
PL_CALLBACK_EOF
PL_CALLBACK_CAM_REMOVED
```

The `Callback` function must be a function taking void pointer and returning no value. The contents of the context are whatever the application requires, but should be reference to the camera handle. For example:

```
void BOFCallback(void* Context)
{
    if (*(int16*)(Context) == hCamera1)
        BOFCountCamera1++;
    else if (*(int16*)(Context) == hCamera2)
        BOFCountCamera2++;
    return;
}
```

WARNING

`pl_exp_finish_seq` must be called if acquiring in sequential mode (using `pl_exp_setup_seq` and `pl_exp_start_seq`) with callbacks notification after all frames are read out and before new exposure is started by calling `pl_exp_start_seq`.

Not all callbacks will be available for all camera systems/interfaces. The callback descriptions below indicate which callbacks are available on which interfaces.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_cam_deregister_callback(0)`

NOTES

Only `PL_CALLBACK_BOF` and `PL_CALLBACK_EOF` are fully supported by all camera types. Do not use other callback types in generic-purpose software.

See *Callback descriptions* section under `pl_cam_register_callback` for details.

PVCAM **Class 0: Camera Communication** **pl_cam_register_callback_ex2(0)**

NAME

`pl_cam_register_callback_ex2` — installs a function that will be called when an event occurs in a camera providing information about frame via `FRAME_INFO` type.

SYNOPSIS

```
rs_bool
   pl_cam_register_callback_ex2(
              int16 hcam,
              PL_CALLBACK_EVENT event,
              void* Callback)
```

DESCRIPTION

Use this API call to install a function that will be called when the specified event occurs providing additional frame information. Input parameter of the callback function must be of `FRAME_INFO*` type in order to receive information about the frame (timestamp with precision of 0.1ms, frame counter number, ID (handle) of the camera that produced the frame).

The `hcam` parameter must reference an open camera system.

The `event` parameter must be one of the following:

```
PL_CALLBACK_BOF
PL_CALLBACK_EOF
PL_CALLBACK_CAM_REMOVED
```

The `Callback` function must be a function taking a pointer to `FRAME_INFO` and returning no value. For example:

```
void EOFCallbackHandler(FRAME_INFO* pNewFrameInfo)
{
    int32 frameNr = pNewFrameInfo->FrameNr;
    long64 frameTime = pNewFrameInfo->TimeStamp;
    int16 camID = pNewFrameInfo->hCam;
    // display or process frame info etc...
}
```

WARNING

`pl_exp_finish_seq` must be called if acquiring in sequential mode (using `pl_exp_setup_seq` and `pl_exp_start_seq`) with callbacks notification after all frames are read out and before new exposure is started by calling `pl_exp_start_seq`.

Not all callbacks will be available for all camera systems/interfaces. The callback descriptions below indicate which callbacks are available on which interfaces.

Variable pointed to by `pFrameInfo` must be created with `pl_create_frame_info_struct(2)`.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_cam_deregister_callback(0)`

NOTES

Only `PL_CALLBACK_BOF` and `PL_CALLBACK_EOF` are fully supported by all camera types. Do not use other callback types in generic-purpose software.

See *Callback descriptions* section under `pl_cam_register_callback` for details.

PVCAM **Class 0: Camera Communication** **pl_cam_register_callback_ex3(0)**

NAME

`pl_cam_register_callback_ex3` — installs a function that will be called when an event occurs in a camera providing information about frame via `FRAME_INFO` type and with user context information. This function combines functionality provided by `pl_cam_register_callback_ex` and `pl_cam_register_callback_ex2`.

SYNOPSIS

```
rs_bool
   pl_cam_register_callback_ex3(
              int16 hcam,
              PL_CALLBACK_EVENT event,
              void* Callback,
              void* Context)
```

DESCRIPTION

Use this API call to install a function that will be called when the specified event occurs providing additional frame information. Input parameter of the callback function must be of `FRAME_INFO*` type in order to receive information about the frame (timestamp with precision of 0.1ms, frame counter number, ID (handle) of the camera that produced the frame). Also pointer to a context that will be echoed back when the callback is invoked can be passed to PVCAM in this function.

The `hcam` parameter must reference an open camera system.

The `event` parameter must be one of the following:

```
PL_CALLBACK_BOF
PL_CALLBACK_EOF
PL_CALLBACK_CAM_REMOVED
```

The `Callback` function must be a function taking a pointer to `FRAME_INFO` and void pointer and returning no value. For example:

```
void EOFCallbackHandler(FRAME_INFO* pNewFrameInfo,
                        void* Context)
{
    int32 frameNr = pNewFrameInfo->FrameNr;
    long64 frameTime = pNewFrameInfo->TimeStamp;
    int16 camID = pNewFrameInfo->hCam;
    // display or process frame info etc...
    if (*(int16*)(Context) == hCamera1)
        EOFCountCamera1++;
    else if (*(int16*)(Context) == hCamera2)
        EOFCountCamera2++;
}
```

WARNING	`pl_exp_finish_seq` must be called if acquiring in sequential mode (using `pl_exp_setup_seq` and `pl_exp_start_seq`) with callbacks notification after all frames are read out and before new exposure is started by calling `pl_exp_start_seq`. Not all callbacks will be available for all camera systems/interfaces. The callback descriptions below indicate which callbacks are available on which interfaces. Variable pointed to by `pFrameInfo` must be created with `pl_create_frame_info_struct(2)`.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_cam_deregister_callback(0)`
NOTES	Only `PL_CALLBACK_BOF` and `PL_CALLBACK_EOF` are fully supported by all camera types. Do not use other callback types in generic-purpose software. See *Callback descriptions* section under `pl_cam_register_callback` for details.

PVCAM	**Class 0: Camera Communication** **pl_cam_deregister_callback(0)**
NAME	`pl_cam_deregister_callback` — uninstalls a function for camera system event.
SYNOPSIS	**`rs_bool`** **`pl_cam_deregister_callback(`** **`int16 hcam,`** **`PL_CALLBACK_EVENT event)`**

DESCRIPTION	Use this API call to uninstall a function for the specified camera system event.
	The `hcam` parameter must reference an open camera system.
	The `event` parameter must be one of the following:
	`PL_CALLBACK_BOF`
	`PL_CALLBACK_EOF`
	`PL_CALLBACK_CAM_REMOVED`

RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO	`pl_cam_register_callback(0),`
	`pl_cam_register_callback_ex(0),`
	`pl_cam_register_callback_ex2(0),`
	`pl_cam_register_callback_ex3(0)`

NOTES	Only `PL_CALLBACK_BOF` and `PL_CALLBACK_EOF` are fully supported by all camera types. Do not use other callback types in generic-purpose software. See *Callback descriptions* section under `pl_cam_register_callback` for details.

Class 0 Parameter IDs

The following parameter IDs are used with `pl_get_param`, `pl_set_param`, `pl_get_enum_param`, and `pl_enum_str_length` functions described in *Chapter 5*.

> **Note:** Before trying to use or retrieve more information about a parameter, it is always recommended to call an `ATTR_AVAIL` to see if the system supports it.

Class 0 Parameter ID	Description
`PARAM_DD_INFO` **Camera Dependent**	Returns an information message for each device. Some devices have no message. The user is responsible for allocating enough memory to hold the message string. Required number of bytes can be obtained via parameter `PARAM_DD_INFO_LENGTH`. **Datatype: char***
`PARAM_DD_INFO_LENGTH` **Camera Dependent**	Returns the length of an information message for each device. Some devices have no message. In other words, they return a value of 0 for bytes. **Datatype: int16**
`PARAM_DD_RETRIES` **Camera Dependent**	Reads/sets the maximum number of command retransmission attempts that are allowed. When a command or status transmission is garbled, the system signals for a retransmission. After a certain number of failed transmissions (an initial attempt + max_retries), the system abandons the attempt and concludes that the communication link has failed. The camera will not close, but the command or status read returns with an error. The maximum number of retries is initially set by the device driver, and is matched to the communication link, hardware platform, and operating system. It may also be reset by the user. **Datatype: uns16**
`PARAM_DD_TIMEOUT` **Camera Dependent**	Reads/sets the maximum time the driver waits for acknowledgment (i.e., the slowest allowable response speed from the camera). This is a crucial factor used in the device driver for communication control. If the driver sends a command to the camera and does not receive acknowledgment within the timeout period, the driver times out and returns an error. Unless reset by the user, this timeout is a default setting that is contained in the device driver and is matched to the communication link, hardware platform, and operating system. **Datatype: uns16**

Class 0 Parameter ID	Description
PARAM_DD_VERSION	Returns a version number for the device driver used to access the camera. The version is a formatted hexadecimal number, of the style: High byte | Low byte (High nibble | Low nibble) Major version | Minor version | Trivial version For example, the number 0xB1C0 indicates major release 177, minor release 12, and trivial change 0. A major release is defined as anything that alters the user interface, calling sequence, or parameter interpretation of any device driver interface function (anything that would alter the driver's API). A new major release often requires the calling software to change, but wherever possible, major releases are backward compatible with earlier releases. A minor release should be completely transparent to higher level software, but may include internal enhancements. A trivial change is reserved for use by the software staff to keep track of extremely minor variations. The last digit may also be used to flag versions of the driver constructed for unique customers or situations. Minor and trivial releases should require no change in the calling software. Open the camera before calling this parameter. Note that different cameras on the same system may use different drivers. Thus, each camera can have its own driver, and its own driver version. **Datatype: uns16**

This page intentionally left blank.

Chapter 4:
Error Reporting (Class 1)

Introduction

Every PVCAM function resets the error code to 0 (no error). This means that `pl_error_code` only reports the error status of the most recent function used. Since all PVCAM functions universally return a TRUE for no error/success, and a FALSE for a failure (except `pl_error_code` of course), you can use the following construction to report errors:

```
if (!pl_pvcam_do_something(...)) {
    char msg[ERROR_MSG_LEN];
    pl_error_message(pl_errror_code(), msg);
    printf("pl_pvcam_do_something failed with message '%s'\n", msg);
}
```

If you need to check whether the function works before executing further code, you could use the sample construction below:

```
if (pl_pvcam_do_something(...)) {
    /* function succeeded */
    ...
}
else {
    /* function failed, print message */
    char msg[ERROR_MSG_LEN];
    pl_error_message(pl_error_code(), msg);
    printf("pl_pvcam_do_something failed with message '%s'\n", msg);
}
```

Although the `(function == TRUE)` style works well in many cases, you may prefer a more explanatory comparison. In that case, the following two constants are defined for your use:

```
#define PV_OK   TRUE
#define PV_FAIL FALSE
```

Using these two constants, the code above can be rewritten as follows:

```
if (pl_pvcam_do_something(...) == PV_OK) {
    /* function succeeded */
    ...
}
```

or

```
if (pl_pvcam_do_something(...) == PV_FAIL) {
    /* function failed, print message */
    ...
}
```

Use any of the styles illustrated above in any mix. The differences are only a matter of stylistic preference.

Error Codes

All successful functions reset `pl_error_code` to 0, which produces the message "`No error`". All unsuccessful functions return a numeric value, where that value corresponds to a number linked to a published list of error code messages.

List of Available Class 1 Functions

Class 1 functions are listed below:

```
pl_error_code                    pl_error_message
```

Class 1 Functions

PVCAM	Class 1: Error Reporting	pl_error_code(1)

NAME

`pl_error_code` — returns the most recent error condition.

SYNOPSIS

```
int16
   pl_error_code(void)
```

DESCRIPTION

As every PVCAM function begins, it resets the error code to 0. If an error occurs later in the function, the error code is set to a corresponding value.

RETURN VALUE

The current error code. Note that a call to `pl_error_code` does not reset the error code.

SEE ALSO

`pl_error_message(1)`

NOTES

`pl_error_code` works even before `pl_pvcam_init` is called. This allows a message to be returned if `pl_pvcam_init` fails.

In the error codes structure, the thousands digit indicates the class of the failed function.

KNOWN BUGS

The PVCAM library does not intercept signals. Errors that interrupt the normal process (divide by zero, etc.) may cause the software to crash, and `pl_error_code` may or may not contain useful information.

PVCAM **Class 1: Error Reporting** **pl_error_message(1)**

NAME

`pl_error_message` — returns a string explaining input error code.

SYNOPSIS

```
rs_bool
   pl_error_message(
              int16 err_code,
              char* msg)
```

DESCRIPTION

This function fills in the character string `msg` with a message that corresponds to the value in `err_code`. The msg string is allocated by the user, and should be at least `ERROR_MSG_LEN` elements long.

RETURN VALUE

TRUE if a message is found corresponding to the input code, FALSE if the code is out of range or does not have a corresponding message (`msg` will be filled with the string "`unknown error`"). Even if a FALSE is returned, the value of `pl_error_code` is not altered.

SEE ALSO

`pl_error_code(1)`

NOTES

`pl_error_message` works even before `pl_pvcam_init` is called. This allows a message to be printed if `pl_pvcam_init` fails.

Most error messages are lower case sentence fragments with no ending period.

Chapter 5: Configuration / Setup (Class 2)

Note: pl_pvcam_init must be called before any other function in the library! Until it is called, all functions will fail and return a FALSE. pl_pvcam_init is necessary, even if no hardware interaction is going to occur.

Introduction

The basic idea of Get/Set functions is to determine if a feature exists in a camera set, what its attributes are, and how can it be changed (if at all). The main function is pl_get_param. This function is called with a parameter ID (param_id) and an attribute (param_attrib) and returns the attribute for that parameter. Usually, the user would start off with ATTR_AVAIL, which checks to see if the param_id is supported in the software and hardware. If FALSE is returned in the param_value, the param_id is not supported in either the software or the hardware. If TRUE is returned, the param_id is supported and the user can get the access rights (ATTR_ACCESS).

ATTR_ACCESS tells if the param_id can be written to or read or, if it cannot be written to or read, tells whether a feature is possible. If the parameter can be either written to or read the next step is to determine its data type.

Data type determination can be done by calling the parameter ID with the attribute of data type (ATTR_TYPE), this will report the data type: string (TYPE_CHAR_PTR), integer (TYPE_INT8, TYPE_UNS8, TYPE_INT16, TYPE_UNS16, TYPE_INT32, TYPE_UNS32, TYPE_INT64, TYPE_UNS64), floating point (TYPE_FLT64), boolean (TYPE_BOOLEAN), or an enumerated type (TYPE_ENUM). The user can then get the current value (ATTR_CURRENT) and the default value (ATTR_DEFAULT) for the parameter ID. If the data type is not the enumerated type, the user can also get the minimum value (ATTR_MIN), the maximum value (ATTR_MAX), and the increment (ATTR_INCREMENT). Finally, if the data type is enumerated, the user can get the number of enumerated types that are legal (ATTR_COUNT), and passing the parameter ID and index (which has to be between 0 and less than ATTR_COUNT), the user can call pl_get_enum_param and get the exact enumerated value along with a string that describes the enumerated type.

Notes:

hcam specifies which camera and which device driver are being used. hcam must be a valid camera handle.

If the data type coming back from ATTR_TYPE is TYPE_CHAR_PTR (and not an enumerated type), then the ATTR_COUNT is the number of characters in the string plus a NULL terminator.

If the data type coming back from ATTR_TYPE is TYPE_ENUM the function pl_get_param returns (and pl_set_param takes) the value of enumerated type, not its index.

List of Available Class 2 Functions

Class 2 functions represent camera settings. The current Class 2 functions are listed below according to their respective types and are further described in the "*Class 2 Func*" section, starting on page 59.

```
pl_get_param
pl_set_param
pl_get_enum_param
pl_enum_str_length
pl_pp_reset
pl_create_frame_info_struct
pl_release_frame_info_struct
pl_create_smart_stream_struct
pl_release_smart_stream_struct
```

List of Available Class 2 Parameter IDs

The following are available Class 2 parameters used with `pl_get_param`, `pl_set_param`, `pl_get_enum_param` and `pl_enum_str_length` functions specified in *Chapter 5*.

Sensor Clearing

```
PARAM_CLEAR_CYCLES
PARAM_CLEAR_MODE
```

Temperature Control

```
PARAM_COOLING_MODE
PARAM_TEMP
PARAM_TEMP_SETPOINT
PARAM_FAN_SPEED_SETPOINT
```

Gain

```
PARAM_GAIN_INDEX
PARAM_GAIN_NAME
PARAM_GAIN_MULT_ENABLE
PARAM_GAIN_MULT_FACTOR
PARAM_PREAMP_DELAY
PARAM_PREAMP_OFF_CONTROL
PARAM_ACTUAL_GAIN
```

Shutter

```
PARAM_SHTR_STATUS
PARAM_SHTR_CLOSE_DELAY
PARAM_SHTR_OPEN_DELAY
```

Sensor Physical Attributes

```
PARAM_COLOR_MODE
PARAM_FWELL_CAPACITY
PARAM_PAR_SIZE
PARAM_PIX_PAR_DIST
PARAM_PIX_PAR_SIZE
PARAM_PIX_SER_DIST
PARAM_PIX_SER_SIZE
PARAM_POSTMASK
PARAM_POSTSCAN
PARAM_PIX_TIME
PARAM_PREMASK
PARAM_PRESCAN
PARAM_SER_SIZE
PARAM_SUMMING_WELL
```

Sensor Readout

```
PARAM_PMODE
PARAM_READOUT_PORT
PARAM_READOUT_TIME
PARAM_EXPOSURE_MODE
PARAM_EXPOSE_OUT_MODE
```

PARAM_SHTR_OPEN_MODE

Capabilities

PARAM_ACCUM_CAPABLE

PARAM_FRAME_CAPABLE

PARAM_MPP_CAPABLE

I/O

PARAM_IO_ADDR

PARAM_IO_BITDEPTH

PARAM_IO_DIRECTION

PARAM_IO_STATE

PARAM_IO_TYPE

Post-Processing

PARAM_PP_INDEX

PARAM_PP_FEAT_NAME

PARAM_PP_PARAM_INDEX

PARAM_PP_PARAM_NAME

PARAM_PP_PARAM

PARAM_PP_FEAT_ID

PARAM_PP_PARAM_ID

ADC Attributes

PARAM_ADC_OFFSET

PARAM_BIT_DEPTH

PARAM_SPDTAB_INDEX

S.M.A.R.T. Streaming

PARAM_SMART_STREAM_MODE_ENABLED

PARAM_SMART_STREAM_MODE

PARAM_SMART_STREAM_EXP_PARAMS

Other

PARAM_CAM_FW_VERSION

PARAM_CHIP_NAME

PARAM_SYSTEM_NAME

PARAM_VENDOR_NAME

PARAM_PRODUCT_NAME

PARAM_CAMERA_PART_NUMBER

PARAM_HEAD_SER_NUM_ALPHA

PARAM_PCI_FW_VERSION

PARAM_READ_NOISE

Class 2 Functions

PVCAM **Class 2: Configuration/Setup** **pl_get_param(2)**

NAME

`pl_get_param` — returns the requested attribute for a PVCAM parameter.

SYNOPSIS

```
rs_bool
   pl_get_param(
              int16 hcam,
              uns32 param_id,
              int16 param_attrib,
              void* param_value)
```

DESCRIPTION

This function returns the requested attribute for a PVCAM parameter.

`param_id` is an enumerated type that indicates the parameter in question. See

Class 0 Parameter IDs,

Class 2 Parameter IDs and

Class 3 Parameter IDs for information about valid parameter IDs.

`param_value` points to the value of the requested attribute for the parameter. It is a `void*` because it can be different data types. The user is responsible for passing in the correct data type (see attribute descriptions that follow).

`param_attrib` is used to retrieve characteristics of the parameter.

Possible values for `param_attrib` are:

```
ATTR_ACCESS              ATTR_INCREMENT
ATTR_AVAIL               ATTR_MAX
ATTR_COUNT               ATTR_MIN
ATTR_CURRENT             ATTR_TYPE
ATTR_DEFAULT
```

Reading of values for attributes `ATTR_AVAIL`, `ATTR_ACCESS` and `ATTR_TYPE` should always succeed and return correct value. Values for other attributes can be read only if `ATTR_ACCESS` reports either `ACC_READ_ONLY` or `ACC_READ_WRITE`.

ATTR_ACCESS

Reports if the parameter with ID `param_id` can be written to and/or read or (if it cannot be written to and/or read) tells whether a feature exists. If the `param_id` can be either written to or read the next step is to determine its data type.

The access types are enumerated:

```
ACC_EXIST_CHECK_ONLY     ACC_READ_ONLY
ACC_WRITE_ONLY           ACC_READ_WRITE
```

The data type for this attribute is `TYPE_UNS16`.

ATTR_AVAIL

Feature available with attached hardware and software.

The data type for this attribute is `TYPE_BOOLEAN`.

PVCAM	**Class 2: Configuration/Setup** **pl_get_param(2)**
ATTR_COUNT	Number of possible values for enumerated and array data types. If the data type returned by ATTR_TYPE is TYPE_CHAR_PTR (and not an enumerated type), then the ATTR_COUNT is the number of characters in the string including a space for NULL terminator. If 0 or 1 is returned, ATTR_COUNT is a scalar (single element) of the following data types: TYPE_INT8, TYPE_UNS8, TYPE_INT16, TYPE_UNS16, TYPE_INT32, TYPE_UNS32, TYPE_INT64, TYPE_UNS64, TYPE_FLT64 and TYPE_BOOLEAN. The data type for this attribute is TYPE_UNS32.
ATTR_CURRENT	Current value. For the enumerated type the value returned here is the value assigned to current enum item not the item index. The data type for this attribute is defined by ATTR_TYPE.
ATTR_DEFAULT	Default value. This value should be equal to the current value set in camera after power cycle. For the enumerated type the value returned here is the value assigned to current enum item not the item index. The data type for this attribute is defined by ATTR_TYPE.
ATTR_INCREMENT	Step size for values (zero if non-linear or has no increment). The value is only valid for the following data types: TYPE_INT8, TYPE_UNS8, TYPE_INT16, TYPE_UNS16, TYPE_INT32, TYPE_UNS32, TYPE_INT64, TYPE_UNS64 and TYPE_FLT64. The value for this attribute is never negative. If the value is not zero valid values can be easily calculated. First valid value is the value reported for attribute ATTR_MIN, second value is minimum value plus increment (ATTR_INCREMENT), and so on up to the maximum value (ATTR_MAX). The data type for this attribute is defined by ATTR_TYPE.
ATTR_MAX	Maximum value. The value is only valid for the following data types: TYPE_INT8, TYPE_UNS8, TYPE_INT16, TYPE_UNS16, TYPE_INT32, TYPE_UNS32, TYPE_INT64, TYPE_UNS64, TYPE_FLT64 and TYPE_BOOLEAN. The data type for this attribute is defined by ATTR_TYPE.
ATTR_MIN	Minimum value. The value is only valid for the following data types: TYPE_INT8, TYPE_UNS8, TYPE_INT16, TYPE_UNS16, TYPE_INT32, TYPE_UNS32, TYPE_INT64, TYPE_UNS64, TYPE_FLT64 and TYPE_BOOLEAN. The data type for this attribute is defined by ATTR_TYPE.

PVCAM	**Class 2: Configuration/Setup** **pl_get_param(2)**

ATTR_TYPE	Data type of parameter.
	Data types used by `pl_get_param` with attribute type (`ATTR_TYPE`) are:
	`TYPE_BOOLEAN` – `rs_bool`
	`TYPE_INT8` – `int8`
	`TYPE_UNS8` – `uns8`
	`TYPE_INT16` – `int16`
	`TYPE_UNS16` – `uns16`
	`TYPE_INT32` – `int32`
	`TYPE_UNS32` – `uns32`
	`TYPE_INT64` – `int64`
	`TYPE_UNS64` – `uns64`
	`TYPE_FLT64` – `flt64`
	`TYPE_ENUM` – each has type-specific enum but is treated as `int32`
	`TYPE_CHAR_PTR` – `char*` – NULL-terminated string
	`TYPE_SMART_STREAM_TYPE_PTR` – `smart_stream_type*`
	The data type for this attribute is `TYPE_UNS16`.

RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_set_param(2), pl_get_enum_param(2)`
NOTES	The data type of `param_value` is documented in `pvcam.h` for each `param_id`.

PVCAM	**Class 2: Configuration/Setup**	**pl_set_param(2)**

NAME

`pl_set_param` — sets the current value for a PVCAM parameter.

SYNOPSIS

```
rs_bool
  pl_set_param(
             int16 hcam,
             uns32 param_id,
             void* param_value)
```

DESCRIPTION

This function sets the current value for a PVCAM parameter.

`param_id` is an enumerated type that indicates the parameter in question. See *Class 0 Parameter* IDs, *Class 2 Parameter* IDs and *Class 3 Parameter* IDs for information about valid parameter IDs.

`param_value` points to the new value of the parameter. For the enumerated type this value is the value assigned to current enum item not the item index.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_get_param(2), pl_get_enum_param(2)`

NOTES

The data type of `param_value` is documented in `pvcam.h` for each `param_id`. It can be retrieved using the `pl_get_param` function, using the `ATTR_TYPE` attribute.

The user should call the `pl_get_param` function with the attribute `ATTR_ACCESS`, to verify that the parameter ID is writeable (settable), before calling the `pl_set_param` function.

PVCAM	**Class 2: Configuration/Setup** **pl_get_enum_param(2)**
NAME	`pl_get_enum_param` — returns the enumerated value of the parameter `param_id` at `index`.
SYNOPSIS	**`rs_bool`** **`  pl_get_enum_param(`** **`  int16 hcam,`** **`  uns32 param_id,`** **`  uns32 index,`** **`  int32* value,`** **`  char* desc,`** **`  uns32 length)`**
DESCRIPTION	This function will return the enumerated value of the parameter `param_id` at `index`. It also returns a string associated with the enumerated type (`desc`). `length` indicates the maximum length allowed for the returned description. See *Class 0 Parameter* IDs, *Class 2 Parameter* IDs and *Class 3 Parameter* IDs for information about valid parameter IDs.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_get_param(2), pl_set_param(2), pl_enum_str_length(2)`
NOTES	The user should call the `pl_get_param` function with the attribute `ATTR_TYPE`, to verify that the parameter ID is an enumerated data type before calling the `pl_get_enum_param`. The user should also call the `pl_get_param` function with the attribute `ATTR_COUNT` to determine how many valid enumerated values the parameter ID has. **Example:** Suppose there is a parameter for camera readout speed. This parameter can be set to 1MHz, 5MHz or 10MHz with the appropriate values 1, 5 and 10. If the readout speed is currently set to 5MHz, a call to `pl_get_param` with `ATTR_CURRENT` returns a value of 5. A call to `pl_get_enum_param` for the readout speed parameter at `index` 1 (the second item) returns the enumerated type 5MHz with the `value` equal to 5 and the `desc` would contain "5MHz".

PVCAM **Class 2: Configuration/Setup** **pl_enum_str_length(2)**

NAME

`pl_enum_str_length` — returns the length of the descriptive string for the parameter `param_id` at index.

SYNOPSIS

```
rs_bool
   pl_enum_str_length(
              int16 hcam,
              uns32 param_id,
              uns32 index,
              uns32* length)
```

DESCRIPTION

This function will return the length (`length`) of the descriptive string for the parameter `param_id` at index. The length includes the terminating null (‘\0’) character.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_get_enum_param(2)`

NOTES

This function can be used to determine the amount of memory to allocate for the descriptive string when calling the `pl_get_enum_param` function. Using the example in `pl_get_enum_param`, the length returned would be 5 (4 printable characters plus 1 null character).

PVCAM	Class 2: Configuration/Setup	pl_pp_reset

NAME

`pl_pp_reset` — fails if post-processing modules are not available in current camera or if `hcam` is not the handle of an open camera.

SYNOPSIS

```
rs_bool
   pl_pp_reset(
               int16 hcam)
```

DESCRIPTION

This function will reset all post-processing modules to their default values.

RETURN VALUE

TRUE for a successful reset, FALSE for an unsuccessful reset.

SEE ALSO

`PARAM_PP_FEAT_NAME, PARAM_PP_PARAM_INDEX, PARAM_PP_PARAM_NAME, PARAM_PP_PARAM, PARAM_PP_FEAT_ID, PARAM_PP_PARAM_ID`

PVCAM **Class 2: Configuration/Setup** **pl_create_frame_info_struct(2)**

NAME

`pl_create_frame_info_struct` — creates and allocates variable of `FRAME_INFO` type and returns pointer to it.

SYNOPSIS

```
rs_bool
   pl_create_frame_info_struct(
               FRAME_INFO** pNewFrameInfo)
```

DESCRIPTION

This function will create a variable of `FRAME_INFO` type and return a pointer to access it. The GUID field of the `FRAME_INFO` structure is assigned by this function. Other fields are updated by PVCAM at the time of frame reception.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_release_frame_info_struct(2)`,
`pl_exp_get_latest_frame_ex(3)`,
`pl_exp_get_oldest_frame_ex(3)`,
`pl_exp_check_cont_status_ex(3)`,
`pl_cam_register_callback_ex2(0)`,
`pl_cam_register_callback_ex3(0)`

NOTES

PVCAM	**Class 2: Configuration/Setup** **pl_release_frame_info_struct(2)**
NAME	`pl_release_frame_info_struct` — deletes variable of `FRAME_INFO` type.
SYNOPSIS	**`rs_bool`** **`  pl_release_frame_info_struct(`** **`  FRAME_INFO* pFrameInfoToDel)`**
DESCRIPTION	This function will deallocate `FRAME_INFO` variable created by `pl_create_frame_info_struct`.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO	`pl_create_frame_info_struct(2),`
	`pl_exp_get_latest_frame_ex(3),`
	`pl_exp_get_oldest_frame_ex(3),`
	`pl_exp_check_cont_status_ex(3),`
	`pl_cam_register_callback_ex2(0),`
	`pl_cam_register_callback_ex3(0)`

NOTES	

PVCAM	**Class 2: Configuration/Setup** **pl_create_smart_stream_struct(2)**
NAME	`pl_create_smart_stream_struct` — creates and allocates variable of `smart_stream_type` type with the number of entries passed in via the `entries` parameter and returns pointer to it.
SYNOPSIS	**`rs_bool`** **`  pl_create_smart_stream_struct(`** **`  smart_stream_type** pSmtStruct,`** **`  uns16 entries)`**
DESCRIPTION	This function will create a variable of `smart_stream_type` type and return a pointer to access it. The `entries` parameter passed by the user determines how many entries the structure will contain.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_release_smart_stream_struct(2)`
NOTES	

PVCAM	**Class 2: Configuration/Setup** **pl_release_smart_stream_struct(2)**
NAME	`pl_release_smart_stream_struct` — frees the space previously allocated by the `pl_create_smart_stream_struct` function.
SYNOPSIS	**`rs_bool`** **`pl_release_smart_stream_struct(`** **`smart_stream_type** pSmtStruct)`**
DESCRIPTION	This function will deallocate a `smart_stream_type` variable created by `pl_create_smart_stream_struct`.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_create_smart_stream_struct(2)`
NOTES	

Class 2 Parameter IDs

The following parameter IDs are used with `pl_get_param`, `pl_set_param`, `pl_get_enum_param`, and `pl_enum_str_length` functions described in *Chapter 5*.

> **Note:** Before trying to use or retrieve more information about a parameter, it is always recommended to call an `ATTR_AVAIL` to see if the system supports it.

Class 2 Parameter ID	Description
`PARAM_ACCUM_CAPABLE` **Camera Dependent**	Returns TRUE for `ATTR_AVAIL` if the camera has accumulation capability. Accumulation functionality is provided with the Class 93 FF plug-in. **Datatype: rs_bool**
`PARAM_ADC_OFFSET` **Camera Dependent**	Bias offset voltage. The units do not correspond to the output pixel values in any simple fashion (the conversion rate should be linear, but may differ from system to system) but a lower offset voltage will yield a lower value for all output pixels. Pixels brought below zero by this method will be clipped at zero. Pixels raised above saturation will be clipped at saturation. Before you can change the offset level, you must read the current offset level. The default offset level will also vary from system to system and may change with each speed and gain setting. **Note:** THIS VALUE IS SET AT THE FACTORY AND SHOULD NOT BE CHANGED. If you would like to change this value, please contact customer service before doing so. **Datatype: int16**
`PARAM_BIT_DEPTH`	Number of bits output by the currently selected speed choice. Although this number might range between 6 and 16, the data will always be returned in an unsigned 16-bit word. This value indicates the number of valid bits within that word. **Datatype: int16**
`PARAM_CAM_FW_VERSION` **Camera Dependent**	Returns the firmware version of the camera, as a hexadecimal number in the form MMmm, where MM is the major version and mm is the minor version. For example, 0x0814 corresponds to version 8.20. **Datatype: uns16**
`PARAM_CHIP_NAME`	The name of the sensor. The name is a null-terminated text string. The user must pass in a character array that is at least `CCD_NAME_LEN` elements long. **Datatype: char***

Class 2 Parameter ID	Description
PARAM_SYSTEM_NAME **Camera Dependent**	The name of the system. The name is a null-terminated text string. The user must pass in a character array that is at least MAX_SYSTEM_NAME_LEN elements long. It is meant to replace the purpose of PARAM_CHIP_NAME behavior on some cameras which were reporting their friendly product name with this parameter, and in turn help future cameras go back to reporting the name of the sensor with PARAM_CHIP_NAME. **Datatype: char***
PARAM_VENDOR_NAME **Camera Dependent**	The name of the vendor. The name is a null-terminated text string. The user must pass in a character array that is at least MAX_VENDOR_NAME_LEN elements long. This is meant to differentiate between “QImaging” and “Photometrics” products moving forward. **Datatype: char***
PARAM_PRODUCT_NAME **Camera Dependent**	The name of the product. The name is a null-terminated text string. The user must pass in a character array that is at least MAX_PRODUCT_NAME_LEN elements long. This is meant to report camera name like “Prime” or “Retiga R6”. OEMs should also consider using this for branding their cameras. **Datatype: char***
PARAM_CAMERA_PART_NUMBER **Camera Dependent**	The part number of the camera. The part number is a null-terminated text string. The user must pass in a character array that is at least MAX_CAM_PART_NUM_LEN elements long. **Datatype: char***
PARAM_CLEAR_CYCLES	This is the number of times the sensor must be cleared to completely remove charge from the parallel register. **Datatype: uns16**

Class 2 Parameter ID	Description
PARAM_CLEAR_MODE **Camera Dependent**	This defines when clearing takes place. See enum below for possible values. CLEAR_NEVER Do not ever clear the sensor. CLEAR_PRE_EXPOSURE Clear clear_cycles times before each exposure starts. CLEAR_PRE_SEQUENCE Clear clear_cycles times before the sequence starts. CLEAR_POST_SEQUENCE Do continuous clearing after the sequence ends. CLEAR_PRE_POST_SEQUENCE Clear clear_cycles times before the sequence starts and continuous clearing after the sequence ends. CLEAR_PRE_EXPOSURE_POST_SEQ Clear clear_cycles times before each exposure starts and continuous clearing after the sequence ends. The CLEAR_NEVER setting is particularly useful for performing a readout after an exposure has been aborted. Note that normally during the idle period, the CCS parallel clock drivers and serial drivers revert to a low power state. This saves on both power and heat. If any CLEAR_..._POST options are used, these systems will not enter low power mode. This will generate extra heat in both the electronics unit and the camera head. **Datatype: enum (int32)**
PARAM_COLOR_MODE **Camera Dependent**	The color mode of the sensor. See enum below for possible values. COLOR_NONE Monochrome camera COLOR_RGGB Color camera with RGGB color mask **Datatype: enum (int32)**

Class 2 Parameter ID	Description
PARAM_COOLING_MODE	This is the type of cooling used by the current camera. See enum below for possible values. NORMAL_COOL This is a thermo-electrically (TE)-cooled camera with air or liquid assisted cooling. CRYO_COOL The camera is cryogenically cooled. A camera cooled via Liquid Nitrogen (LN) in an attached Dewar is an example of a cryo-cooled camera. **Datatype: enum (int32)**
PARAM_EXPOSURE_MODE	This parameter cannot be set but its value can be retrieved. Possible values: TIMED_MODE STROBED_MODE BULB_MODE TRIGGER_FIRST_MODE VARIABLE_TIMED_MODE **Note:** See *Exposure Mode Constants* chapter on page 87 for information about these modes. **Datatype: enum (int32)**
PARAM_EXPOSE_OUT_MODE **Camera Dependent**	This parameter cannot be set but its value can be retrieved. Note: See *Extended Exposure Modes* chapter on page 25 for information about Expose Out Modes. **Datatype: enum (int32)**
PARAM_FRAME_CAPABLE **Camera Dependent**	Returns TRUE for ATTR_AVAIL if this camera can run in frame transfer mode (set through PARAM_PMODE). **Datatype: rs_bool**
PARAM_FWELL_CAPACITY **Camera Dependent**	Gets the full-well capacity of this sensor, measured in electrons. **Datatype: uns32**

Class 2 Parameter ID	Description
PARAM_GAIN_INDEX	Gain setting for the current speed choice. The valid range for a gain setting is reported via ATTR_MIN and ATTR_MAX, where the min. gain is usually 1 the max. gain may be as high as 16. Values outside this range will be ignored. Note that gain setting may not be linear! Values 1-16 may not correspond to 1x-16x, and there are holes between the values. However, when the camera is initialized, and every time a new speed is selected, the system will always reset to run at a gain of 1x. **Datatype: int16**
PARAM_GAIN_NAME **Camera Dependent**	Name of the currently selected Gain (via PARAM_GAIN_INDEX). Use ATTR_AVAIL to check for the availability. The gain name has a maximum length of MAX_GAIN_NAME_LEN and can be retrieved with the ATTR_CURRENT attribute. **Datatype: char***
PARAM_GAIN_MULT_ENABLE **Camera Dependent**	Gain multiplier on/off indicator for cameras with the multiplication gain functionality. This parameter may be read-only, in which case the gain is always on. **Datatype: rs_bool**
PARAM_GAIN_MULT_FACTOR **Camera Dependent**	Gain multiplication factor for cameras with multiplication gain functionality. The valid range is reported via ATTR_MIN and ATTR_MAX. **Datatype: uns16**
PARAM_HEAD_SER_NUM_ALPHA **Camera Dependent**	Returns the alphanumeric serial number for the camera head. The serial number for Photometrics-brand cameras has a maximum length of MAX_ALPHA_SER_NUM_LEN. **Datatype: char***
PARAM_IO_ADDR **Camera Dependent**	Sets and gets the currently active I/O address. The number of available I/O addresses can be obtained using the ATTR_COUNT. **Datatype: uns16**

Class 2 Parameter ID	Description
PARAM_IO_BITDEPTH **Camera Dependent**	Gets the bit depth for the signal at the current address. The bit depth has different meanings, depending on the I/O Type: IO_TYPE_TTL The number of bits read or written at this address. IO_TYPE_DAC The number of bits written to the DAC. **Datatype: uns16**
PARAM_IO_DIRECTION **Camera Dependent**	Gets the direction of the signal at the current address. Possible values are: IO_DIR_INPUT IO_DIR_OUTPUT IO_DIR_INPUT_OUTPUT **Datatype: enum (int32)**
PARAM_IO_STATE **Camera Dependent**	Sets and gets the state of the currently active I/O signal. The new (when setting) or return (when getting) value has different meanings, depending on the I/O type: IO_TYPE_TTL A bit pattern, indicating the current state (0 or 1) of each of the control lines (bit 0 indicates line 0 state, etc.). IO_TYPE_DAC The value of the desired analog output (only applies to pl_set_param). The minimum and maximum range for the signal can be obtained using the ATTR_MIN and ATTR_MAX attributes, respectively. When outputting signals, the state is the desired output. For example, when setting the output of a 12-bit DAC with a range of 0-5V to half-scale, the state should be 2.5 (volts), not 1024 (bits). **Datatype: flt64**
PARAM_IO_TYPE **Camera Dependent**	Gets the type of I/O available at the current address. Possible values are: IO_TYPE_TTL IO_TYPE_DAC **Datatype: enum (int32)**

Class 2 Parameter ID	Description
PARAM_MPP_CAPABLE **Camera Dependent**	Indicates whether this sensor runs in MPP mode. The actual value returned is equal to one of those constants: MPP_UNKNOWN MPP_ALWAYS_OFF MPP_ALWAYS_ON MPP_SELECTABLE **Datatype: enum (int32)**
PARAM_PAR_SIZE	This is the parallel size of the sensor chip, in active rows. The full size of the parallel register is actually (par_size + premask + postmask). **Datatype: uns16**
PARAM_PCI_FW_VERSION **Camera Dependent**	Returns the version number of the PCI firmware. This number is a single 16-bit unsigned value. **Datatype: uns16**
PARAM_PIX_PAR_DIST	This is the center-to-center distance between pixels (in the parallel direction) measured in nanometers. This is identical to PARAM_PIX_PAR_SIZE if there are no interpixel dead areas. **Datatype: uns16**
PARAM_PIX_PAR_SIZE	This is the size of the active area of a pixel, in the parallel direction, measured in nanometers. **Datatype: uns16**
PARAM_PIX_SER_DIST	This is the center-to-center distance between pixels (in the serial direction), in nanometers. This is identical to PARAM_PIX_SER_SIZE, if there are no dead areas. **Datatype: uns16**
PARAM_PIX_SER_SIZE	This is the size of a single pixel's active area, in the serial direction, measured in nanometers. **Datatype: uns16**
PARAM_PIX_TIME	This is the actual speed for the currently selected speed choice. It returns the time for each pixel, in nanoseconds. This readout time will change as new speed choices are selected. **Datatype: uns16**

Class 2 Parameter ID	Description
PARAM_PMODE	This allows the user to select the parallel clocking method. Possible values are: PMODE_NORMAL PMODE_FT PMODE_MPP PMODE_FT_MPP PMODE_ALT_NORMAL PMODE_ALT_FT PMODE_ALT_MPP PMODE_ALT_FT_MPP where _FT indicates frame transfer mode, _FT_MPP indicates both frame transfer and _MPP mode. _ALT indicates that custom parameters may be loaded. **Datatype: enum (int32)**
PARAM_POSTMASK	This is the number of masked lines at the far end of the parallel register (away from the serial register). This is the number of additional parallel shifts that need to be done after readout to clear the parallel register. **Datatype: uns16**
PARAM_POSTSCAN	This is the number of pixels to discard from the serial register after the last real data pixel. These must be read or discarded to clear the serial register. **Datatype: uns16**
PARAM_PREAMP_DELAY **Camera Dependent**	This is the number of milliseconds required for the sensor output preamp to stabilize, after it is turned on. **Datatype: uns16**
PARAM_PREAMP_OFF_CONTROL **Camera Dependent**	The exposure time limit in milliseconds above which the preamp is turned off during exposure. **Datatype: uns32**
PARAM_PREMASK	This is the number of masked lines at the near end of the parallel register, next to the serial register. 0=no mask (no normal mask). If the premask is equal to par_size, this probably indicates a frame transfer device with an ordinary mask. Accordingly, the sensor should probably be run in frame transfer mode. **Datatype: uns16**
PARAM_PRESCAN	This is the number of pixels discarded from the serial register before the first real data pixel. **Datatype: uns16**

Class 2 Parameter ID	Description
PARAM_READOUT_PORT **Camera Dependent**	Sensor readout port being used by the currently selected speed. Different readout ports (used for alternate speeds) flip the image in serial, parallel, or both. READOUT_PORT_MULT_GAIN READOUT_PORT_NORMAL Use PARAM_READOUT_PORT with ATTR_COUNT to read out the number of ports on the system. **Datatype: enum (int32)**
PARAM_READOUT_TIME **Camera Dependent**	Time it will take to read out the image from the sensor with the current camera settings, in microseconds. Settings have to be applied with pl_exp_setup_seq or pl_exp_setup_cont before the camera will calculate the readout time for the new settings. **Datatype: uns32**
PARAM_SER_SIZE	Defines the serial dimension of the active area of the sensor chip. **Datatype: uns16**
PARAM_SHTR_CLOSE_DELAY **Camera Dependent**	This is the shutter close delay. This is the number of milliseconds required for the shutter to close. The software default values compensate for the standard shutter that is shipped with all cameras. You only need to set this value if you are using a shutter with characteristics that differ from the standard shutter. **Datatype: uns16**
PARAM_SHTR_OPEN_DELAY **Camera Dependent**	This is the shutter open delay. This is the number of milliseconds required for the shutter to open. The software default values compensate for the standard shutter that is shipped with all cameras. You only need to set this value if you are using a shutter with characteristics that differ from the standard shutter. **Datatype: uns16**

Class 2 Parameter ID	Description
PARAM_SHTR_OPEN_MODE **Camera Dependent**	This is the shutter opening condition. See enum below for possible values. OPEN_NEVER The shutter closes before the exposure and stays closed during the exposure. OPEN_PRE_EXPOSURE Opens each exposure. Normal mode. OPEN_PRE_SEQUENCE Opens the shutter at the start of each sequence. Useful for frame transfer and external strobe devices. OPEN_PRE_TRIGGER If using a triggered mode, this function causes the shutter to open before the external trigger is armed. If using a non-triggered mode, this function operates identical to OPEN_PRE_EXPOSURE. OPEN_NO_CHANGE Sends no signals to open or close the shutter. Useful for frame transfer when you want to open the shutter and leave it open (see pl_exp_abort). For detailed scripts, see *Exposure Loops* chapter. **Datatype: enum (int32)**
PARAM_SHTR_STATUS **Camera Dependent**	This is the current state of the camera shutter. SHTR_FAULT SHTR_OPENING SHTR_OPEN SHTR_CLOSING SHTR_CLOSED SHTR_UNKNOWN If the shutter is run too fast, it will overheat and trigger SHTR_FAULT. The shutter electronics will disconnect until the temperature returns to a suitable range. Note that although the electronics have reset the voltages to open or close the shutter, there is a lag time for the physical mechanism to respond. See also PARAM_SHTR_OPEN_DLY and PARAM_SHTR_CLOSE_DLY. **Datatype: enum (int32)**

Class 2 Parameter ID	Description
PARAM_SPDTAB_INDEX	This selects the sensor readout speed from a table of available choices. Entries are 0-based and the range of possible values is 0 to max_entries, where max_entries can be determined using ATTR_MAX attribute. This setting relates to other speed table values, including PARAM_BIT_DEPTH, PARAM_PIX_TIME, PARAM_READOUT_PORT, PARAM_GAIN_INDEX and PARAM_GAIN_NAME. After setting PARAM_SPDTAB_INDEX, the gain setting is always reset to a value corresponding to 1x gain. To use a different gain setting, call pl_set_param with PARAM_GAIN_INDEX after setting the speed table index. **Datatype: int16**
PARAM_SUMMING_WELL **Camera Dependent**	Checks to see if the summing well exists. When a TRUE is returned for ATTR_AVAIL, the summing well exists. **Datatype: rs_bool**
PARAM_TEMP **Camera Dependent**	Returns the current measured temperature of the sensor in C°x 100. For example, a temperature of minus 35° would be read as -3500. **Datatype: int16**
PARAM_TEMP_SETPOINT **Camera Dependent**	Sets the desired sensor temperature in hundredths of degrees Celsius (minus 35°C is represented as -3500). The hardware attempts to heat or cool the sensor to this temperature. The min/max allowable temperatures are given ATTR_MIN and ATTR_MAX. Settings outside this range are ignored. Note that this function only sets the desired temperature. Even if the desired temperature is in a legal range, it still may be impossible to achieve. If the ambient temperature is too high, it is difficult to get much cooling on an air-cooled camera. **Datatype: int16**

Class 2 Parameter ID	Description
PARAM_FAN_SPEED_SETPOINT **Camera Dependent**	Sets the desired fan speed. However, camera can automatically adjust the fan speed to higher level due to sensor overheating or completely shut down power to the sensor board to protect camera from damage. The default fan speed is supposed to be changed only temporarily during experiments to reduce sound noise or vibrations. Use this parameter with caution. See enum below for possible values. FAN_SPEED_HIGH The full fan speed, set in factory, it is also the default value. FAN_SPEED_MEDIUM Medium fan speed. FAN_SPEED_LOW Low fan speed. FAN_SPEED_OFF The fan is turned off. **Datatype: enum (int32)**
PARAM_ACTUAL_GAIN **Camera Dependent**	Gets the actual e/ADU for the current gain setting (read only). **Datatype: uns16**
PARAM_READ_NOISE **Camera Dependent**	Gets the read noise for the current speed (read only). **Datatype: uns16**
PARAM_PP_INDEX **Camera Dependent**	This selects the current post-processing feature from a table of available choices. The entries are 0-based and the range of possible values is 0 to max_entries. max_entries can be determined with the ATTR_MAX attribute. This setting relates to other post-processing table values, including PARAM_PP_FEAT_NAME, PARAM_PP_FEAT_ID and PARAM_PP_PARAM_INDEX **Datatype: int16**
PARAM_PP_FEAT_NAME **Camera Dependent**	This returns the name of the currently-selected post-processing feature. User is responsible for buffer allocation with at least MAX_PP_NAME_LEN bytes. **Datatype: char***

Class 2 Parameter ID	Description
PARAM_PP_PARAM_INDEX **Camera Dependent**	This selects the current post-processing parameter from a table of available choices. The entries are 0-based and the range of possible values is 0 to max_entries. max_entries can be determined with the ATTR_MAX attribute. This setting relates to other post-processing table values, including PARAM_PP_PARAM_NAME, PARAM_PP_PARAM_ID and PARAM_PP_PARAM. **Datatype: int16**
PARAM_PP_PARAM_NAME **Camera Dependent**	Gets the name of the currently-selected post-processing parameter for the currently-selected post-processing feature. User is responsible for buffer allocation with at least MAX_PP_NAME_LEN bytes. **Datatype: char***
PARAM_PP_PARAM **Camera Dependent**	This gets or sets the post-processing parameter for the currently-selected post-processing parameter in the index. **Datatype: uns32**
PARAM_PP_FEAT_ID **Camera Dependent**	This returns the ID of the currently-selected post-processing feature. This maps a specific post-processing module across cameras to help applications filter for camera features they need to expose and those that they don't. It helps to identify similarities between camera post-processing features. **Datatype: uns32**
PARAM_PP_PARAM_ID **Camera Dependent**	This returns the ID of the currently-selected post-processing parameter. This maps a specific post-processing parameter across cameras to help applications filter for camera features they need to expose and those that they don't. It helps to identify similarities between camera post-processing features. **Datatype: uns32**
PARAM_SMART_STREAM_MODE **Camera Dependent**	This parameter allows the user to select between available S.MA.R.T. streaming modes. Currently the only available mode is SMTMODE_ARBITRARY_ALL **Datatype: uns16**
PARAM_SMART_STREAM_MODE_ENABLED **Camera Dependent**	This parameter allows the user to retrieve or set the state of the S.M.A.R.T. streaming mode. When a TRUE is returned by the camera, S.M.A.R.T. streaming is enabled. **Datatype: rs_bool**

Class 2 Parameter ID	Description
PARAM_SMART_STREAM_EXP_PARAMS **Camera Dependent**	This parameter allows the user to set or read the current exposure parameters for S.M.A.R.T. streaming. **Datatype: smart_stream_type***

Chapter 6: Data Acquisition (Class 3)

Introduction

Class 3 defines sensor readout and specifies regions and binning factors. This class gives you complete control over exposures and exposure sequences. Camera configurations set in Class 2 must be considered when defining the functions in Class 3.

The current Class 3 functions are listed below. Although these functions have been superseded by `pl_get_param` and `pl_set_param` parameter IDs, the list of these functions and their descriptions have been included for reference purposes.

List of Available Class 3 Functions

The Class 3 functions are listed below:

```
pl_exp_abort
pl_exp_check_cont_status
pl_exp_check_status
pl_exp_finish_seq
pl_exp_get_latest_frame
pl_exp_get_oldest_frame
pl_exp_setup_cont
pl_exp_get_latest_frame_ex
pl_md_frame_decode
pl_md_create_frame_struct
pl_md_release_frame_struct
pl_exp_get_oldest_frame_ex
pl_exp_setup_seq
pl_exp_start_cont
pl_exp_start_seq
pl_exp_stop_cont
pl_exp_unlock_oldest_frame
pl_io_clear_script_control
pl_io_script_control
pl_exp_check_cont_status_ex
pl_md_frame_recompose
pl_md_create_frame_struct_cont
pl_md_read_extended
```

List of Available Class 3 Parameter IDs

The following are available Class 3 parameters used with `pl_get_param`, `pl_set_param`, `pl_get_enum_param`, and `pl_enum_str_length` functions specified in *Chapter 5*.

PARAM_BOF_EOF_CLR	PARAM_EXP_RES
PARAM_BOF_EOF_COUNT	PARAM_EXP_RES_INDEX
PARAM_BOF_EOF_ENABLE	PARAM_EXP_TIME
PARAM_ROI_COUNT	PARAM_EXPOSURE_TIME
PARAM_CENTROIDS_ENABLED	PARAM_METADATA_ENABLED
PARAM_CENTROIDS_COUNT	PARAM_BINNING_SER
PARAM_CENTROIDS_RADIUS	PARAM_BINNING_PAR
PARAM_TRIGTAB_SIGNAL	
PARAM_LAST_MUXED_SIGNAL	

Defining Exposures

To define an exposure or exposure sequence, you must follow the steps below:

- Define the region(s) to be collected by filling a `rgn_type`
- Define the exposure time and mode

Configure any desired camera parameters:

- Apply the settings to the hardware by calling `pl_exp_setup_cont` or `pl_exp_setup_seq`
- Start the acquisition by calling `pl_exp_start_cont` or `pl_exp_start_seq`
- Monitor the progress of data collection by calling `pl_exp_check_cont_status` or `pl_exp_check_status` or utilize callback handlers
- If needed, interrupt the acquisition at any time by calling `pl_exp_abort` (optional)
- Continuous acquisition can be interrupted also by calling `pl_exp_stop_cont`
- Sequence acquisition stops automatically at the end of the sequence, but `pl_exp_finish_seq` should be called before new acquisition is started.

New Structures

To handle these tasks, a new structure is used. It is defined in the include file pvcam.h.

```
typedef struct rgn_type {
    uns16 s1;   /* Starting pixel in the serial register   */
    uns16 s2;   /* Ending pixel in the serial register     */
    uns16 sbin; /* Serial binning for this region          */
    uns16 p1;   /* Starting pixel in the parallel register */
    uns16 p2;   /* Ending pixel in the parallel register   */
    uns16 pbin; /* Parallel binning for this region        */
} rgn_type;
```

Exposure Mode Constants

The six constants below define the exposure mode:

TIMED_MODE	STROBED_MODE
VARIABLE_TIMED_MODE	BULB_MODE
TRIGGER_FIRST_MODE	

These modes describe how the exposure is controlled:

TIMED_MODE	Begins a single exposure or the first exposure of a sequence. The internal timer controls the exposure duration.
VARIABLE_TIMED_MODE	Begins a single exposure or the first exposure of a sequence. This mode ignores the `exposure_time` parameter in setup. Instead, you must call `pl_exp_set_time` to set the exposure duration before each sequence. In this mode, you can change the exposure duration between sequences, and readout in rapid succession, while maintaining the same readout parameters.
TRIGGER_FIRST_MODE	Waits for a trigger to begin a single exposure or a sequence of exposures. The exposure duration is controlled by the internal timer.
STROBED_MODE	Waits for a trigger to begin each exposure in a sequence. The exposure duration is controlled by the internal timer.
BULB_MODE	Waits for a trigger to begin each exposure in a sequence, then waits for the end of the trigger to end the exposure. This mode ignores `exposure_time` parameters in `setup`.

Note about Extended Exposure Modes:

The Exposure Mode enum values and descriptions are hard-coded in PVCAM libraries to supply slightly more information to the developer about enums that were present in PVCAM cameras. These values can be used for applications to run in specific operating modes without advanced user knowledge or intervention. This does continue to apply for the new *Extended Exposure Modes* reported by the firmware directly.

Please refer to *Extended Exposure Modes* chapter on page 25 for more details and a code example.

Embedded Frame Metadata

This feature allows the camera to insert various metadata directly into the frame buffer. The metadata is generated by the camera itself, appended to the user buffer and transferred together with the image data inside the same buffer. The metadata include all information necessary to reconstruct the frame along with other data-describing information such as timestamps, exposure time or other related information.

The Frame Metadata feature is enabled by the `PARAM_METADATA_ENABLED` parameter. This feature is essential for Multiple ROIs and Centroids features. However for single ROI acquisition the Metadata can be disabled to order PVCAM to return the raw frame buffer without any additional information.

Once enabled the `pl_exp_setup_seq` and `pl_exp_setup_cont` functions will report slightly larger frame sizes and the caller is required to allocate the frame buffer accordingly.

Please note that after enabling the feature the frame data will no longer contain raw data only and cannot be directly displayed without additional processing.

To properly display the metadata-enabled frame the following sequence of steps shall be followed:

1) User retrieves a pointer to the image buffer, preferably via the `pl_exp_get_latest_frame` function.
2) The pointer is passed to `pl_md_frame_decode` function that decodes the buffer and fills a frame descriptor helper structure.
3) The frame descriptor structure can be used to directly access each ROI data in the frame. In case the frame contains several ROIs it's up to the user to decide whether to process and display each ROI data separately or use the helper function `pl_md_frame_recompose` to generate a black-filled directly-displayable frame.

Please refer to SDK example applications to obtain more detailed information about how to use this feature.

Multiple-ROI Acquisition

Some cameras allow users to configure more than one ROI in the `pl_exp_setup_seq` or `pl_exp_setup_cont` functions. To verify whether and how many ROIs the camera supports the user should call `PARAM_ROI_COUNT` with `ATTR_MAX` prior configuring the acquisition.

The *Embedded Frame Metadata* must be enabled in order to use multiple ROIs.

To acquire frame with multiple ROIs the following sequence of steps shall be followed:

1) Embedded Frame Metadata feature is enabled.
2) `pl_exp_setup_seq` or `pl_exp_setup_cont` is called with `rgn_array` and `rgn_total` describing the array of desired regions.
3) `pl_exp_start_seq` or `pl_exp_start_cont` is called to start the acquisition.
4) Once the frame arrives it can be decoded or recomposed using the same process as described in the *Embedded Frame Metadata* chapter.

Centroids

Centroids can be thought of as multiple ROIs that are chosen and generated by the camera itself. The user only specifies how many ROIs they are interested in and what should be their size. The size and number of ROIs cannot change during acquisition, however locations of the ROIs may change with each frame. Because of that an embedded frame metadata must be included for every acquired frame. The frame may be re-composed using the same re-composing function(s) as described in the *Embedded Frame Metadata* chapter.

The Centroids feature is enabled by `PARAM_CENTROIDS_ENABLED` parameter. Another two parameters, the `PARAM_CENTROIDS_COUNT` and `PARAM_CENTROIDS_RADIUS` are used to define the desired number of centroids and their size. The *Embedded Frame Metadata* feature must be enabled before starting the Centroids acquisition.

By calling the `pl_exp_setup_seq/pl_exp_setup_cont` functions the user only specifies a single ROI where the Centroids shall be generated. User-defined multiple ROIs cannot be combined with this feature.

To use the centroids feature in an application the following sequence of steps shall be followed:

1) The Embedded Frame Metadata feature is enabled with the `PARAM_METADATA_ENABLED`.
2) The Centroids feature is enabled with `PARAM_CENTROIDS_ENABLED`.
3) Number and size of Centroids is configured with `PARAM_CENTROIDS_COUNT` and `PARAM_CENTROIDS_RADIUS`.
4) `pl_exp_setup_seq` or `pl_exp_setup_cont` is called and frame buffer is allocated accordingly.
5) Acquisition is started.
6) Once the frame arrives it can be decoded or recomposed using the same process as described in the *Embedded Frame Metadata* chapter.

Please refer to SDK example applications to obtain more detailed information about how to use this feature.

Binning Factor Discovery

Binning factors entered during acquisition setup as a part of region have only two restrictions – both serial and parallel factors have to be positive 16-bit numbers and in each direction the factors must not be greater than sensor resolution. Cameras support only very small subset of possible values. Symmetrical binning factors that are power of two (1, 2, 4, 8 …) are usually well supported but applications previously had no way how to check which other values are allowed.

To enable this discover, two new read-only parameters were introduced: `PARAM_BINNING_SER` and `PARAM_BINNING_PAR`. These parameters must be implemented on each camera separately, so it is essential to ensure the `ATTR_AVAIL` returns TRUE before assuming the parameters are implemented.

These parameters are intended to be used together in a pair as they provide a list with all the supported binning permutations. `ATTR_COUNT` reports the same number of permutations for both parameters. Each parameter can and most probably will contain duplicate values but serial × parallel pairs are all unique. Also both parameters return the same string for each value.

Parameters are available if camera has extended binning capability and does not support arbitrary binning. In all other cases user is free to enter any binning theoretically up to sensor resolution. This solution has been chosen because not all cameras might support all permutations of serial and parallel binning factors. For instance, the camera could support only symmetrical binning, 1×1, 2×2, 4×4, etc.

Note: Keep in mind that older cameras don't report supported binning even though they might support only limited subset of binning factors and that behavior for unsupported binning is not defined. This is why this feature has been added to newer cameras: to help applications/users explicitly understand compatibility of binning factors before trying an acquisition.

Example:

Values and strings provided for binning factors 1×1, 1×2, 1×4, 1×8, 2×1, 2×2, 2×4, 3×3 and 4×4.

Attribute	PARAM_BINNING_SER	PARAM_BINNING_PAR
ATTR_MIN	1	1
ATTR_MAX	4	8
ATTR_COUNT	9	9

Complete list of values, note the duplicates, order matters:

Index	0	1	2	3	4	5	6	7	8
PARAM_BINNING_SER	1	1	1	1	2	2	2	3	4
PARAM_BINNING_PAR	1	2	4	8	1	2	4	3	4
Common string	"1×1"	"1×2"	"1×4"	"1×8"	"2×1"	"2×2"	"2×4"	"3×3"	"4×4"

Triggering Table

The *triggering table* feature deals with output signals available on camera connector and has nothing in common with triggering/exposure modes. If camera supports this feature it has a one or more multiplexers connected output signals. The multiplexer can cycle the signal it is connected to (in round-robin fashion) over all or configured number of its outputs.

The configuration is do in two steps. The output signal to be configured can be selected via parameter `PARAM_TRIGTAB_SIGNAL`. The number of active output wires to be used for selected signal can be changed by parameter `PARAM_LAST_MUXED_SIGNAL`.

By default the configured number of multiplexed outputs is set to one for all signals. This has the same effect as there would be no multiplexer connected in the signal path.

Class 3 Functions

PVCAM	**Class 3: Data Acquisition** **pl_exp_finish_seq(3)**
NAME	`pl_exp_finish_seq` — finishes and cleans up after `pl_exp_start_seq`.
SYNOPSIS	**`rs_bool`** **`pl_exp_finish_seq(`** **`int16 hcam,`** **`void* pixel_stream,`** **`int16 hbuf)`**
DESCRIPTION	This cleans up after an exposure started through `pl_exp_start_seq` has finished readout. If the exposure has not finished readout, this function returns with an error. Argument `hbuf` is not used at the moment.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_exp_abort(3), pl_exp_check_status(3), pl_exp_setup_seq(3), pl_exp_start_seq(3)`
NOTES	This function must also be called if acquiring in sequential mode (using `pl_exp_setup_seq` and `pl_exp_start_seq)` with callbacks notification after a frame is read out and before new exposure is started by calling `pl_exp_start_seq`.

PVCAM | **Class 3: Data Acquisition** | **pl_exp_get_latest_frame(3)**

NAME

`pl_exp_get_latest_frame` — returns pointer to most recent frame in circular buffer.

SYNOPSIS

```
rs_bool
   pl_exp_get_latest_frame(
              int16 hcam,
              void** frame)
```

DESCRIPTION

This function returns a pointer to the most recently acquired frame in the circular buffer. `frame` is a pointer to the most recent frame.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3), pl_exp_start_cont(3), pl_exp_check_cont_status(3), pl_exp_stop_cont(3)`

NOTES

If the camera in use is not able to return the latest frame for the current operating mode, this function will fail. For example, some cameras cannot return the latest frame in `CIRC_NO_OVERWRITE` mode. Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

PVCAM | **Class 3: Data Acquisition** | **pl_exp_get_latest_frame_ex(3)**

NAME

`pl_exp_get_latest_frame_ex` — returns pointer to most recent frame in circular buffer and updates values of timestamps (with precision of 0.1ms), frame counter numbers and readout time in variable of `FRAME_INFO` type.

SYNOPSIS

```
rs_bool
   pl_exp_get_latest_frame_ex(
              int16 hcam,
              void** frame,
              FRAME_INFO* pFrameInfo)
```

DESCRIPTION

This function returns a pointer to the most recently acquired frame in the circular buffer. `frame` is a pointer to the most recent frame. Additionally this function updates the values in variable pointed to by `pFrameInfo` with the data collected at the time of frame reception by the device driver (e.g. timestamp, frame counter value).

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3)`, `pl_exp_start_cont(3)`, `pl_exp_check_cont_status(3)`, `pl_exp_stop_cont(3)`, `pl_exp_get_oldest_frame_ex(3)`, `pl_exp_check_cont_status_ex(3)`, `pl_cam_register_callback_ex2(0)`, `pl_create_frame_info_struct(2)`, `pl_release_frame_info_struct(2)`

NOTES

If the camera in use is not able to return the latest frame for the current operating mode, this function will fail. For example, some cameras cannot return the latest frame in `CIRC_NO_OVERWRITE` mode. Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

Variable pointed to by `pFrameInfo` must be created with `pl_create_frame_info_struct(2)`.

PVCAM **Class 3: Data Acquisition** **pl_exp_get_oldest_frame(3)**

NAME

`pl_exp_get_oldest_frame` — locks oldest frame in circular buffer and returns pointer to that frame.

SYNOPSIS

```
rs_bool
   pl_exp_get_oldest_frame(
              int16 hcam,
              void** frame)
```

DESCRIPTION

This function locks the oldest unretrieved frame in the circular buffer, and returns a pointer to that frame. `frame` is a pointer to the oldest unretrieved frame.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3), pl_exp_start_cont(3), pl_exp_check_cont_status(3), pl_exp_unlock_oldest_frame(3), pl_exp_stop_cont(3), pl_exp_get_oldest_frame_ex(3)`

NOTES

If the camera in use is not able to return the oldest frame for the current operating mode, this function will fail. For example, some cameras cannot return the oldest frame in `CIRC_OVERWRITE` mode. Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

PVCAM | **Class 3: Data Acquisition** | **pl_exp_get_oldest_frame_ex(3)**

NAME

`pl_exp_get_oldest_frame_ex` — locks oldest frame in circular buffer and returns pointer to that frame. Also updates frame counter value in the variable of `FRAME_INFO` type.

SYNOPSIS

```
rs_bool
   pl_exp_get_oldest_frame_ex(
              int16 hcam,
              void** frame,
              FRAME_INFO* pFrameInfo)
```

DESCRIPTION

This function locks the oldest unretrieved frame in the circular buffer, and returns a pointer to that frame. `frame` is a pointer to the oldest unretrieved frame. Additionally this function updates the values in the variable pointed to by `pFrameInfo` with the data collected at the time of frame reception by the device driver (e.g. frame counter value).

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3)`, `pl_exp_start_cont(3)`, `pl_exp_check_cont_status(3)`, `pl_exp_unlock_oldest_frame(3)`, `pl_exp_stop_cont(3)`, `pl_exp_check_cont_status_ex(3)`, `pl_cam_register_callback_ex2(0)`, `pl_create_frame_info_struct(2)`, `pl_release_frame_info_struct(2)`

NOTES

If the camera in use is not able to return the oldest frame for the current operating mode, this function will fail. For example, some cameras cannot return the oldest frame in `CIRC_OVERWRITE` mode. Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

Variable pointed to by `pFrameInfo` must be created with `pl_create_frame_info_struct(2)`.

PVCAM **Class 3: Data Acquisition** **pl_exp_setup_cont(3)**

NAME

pl_exp_setup_cont — sets circular buffer mode.

SYNOPSIS

```
rs_bool
   pl_exp_setup_cont(
              int16 hcam,
              uns16 rgn_total,
              const rgn_type* rgn_array,
              int16 mode,
              uns32 exposure_time,
              uns32* stream_size,
              int16 circ_mode)
```

DESCRIPTION

This function sets the mode of operation for the circular buffer. This function uses the array of regions, exposure mode, exposure time passed in, and circular buffer mode and transmits them to the camera.

The pointer rgn_array points to rgn_total region definitions.

mode specifies the bitwise OR combination of the exposure mode and expose out mode. Please refer to chapter *Extended Exposure Modes* on page 25 for more details.

exposure_time specifies the exposure time in the currently selected exposure time resolution (see PARAM_EXP_RES and PARAM_EXP_RES_INDEX).

The pointer stream_size points to a variable that will be filled with number of bytes in the pixel stream.

circ_mode can be set to either CIRC_OVERWRITE or CIRC_NO_OVERWRITE. This function must be called before calling pl_exp_start_cont.

The settings are then downloaded to the camera. If there is any problem (overlapping regions or a frame-transfer setting for a camera that lacks that capability), this function aborts and returns with a failure. pl_error_code indicates the definition problem.

The stream_size pointer is filled with the number of bytes of memory needed to buffer the full sequence. (It is the developer's responsibility to allocate a memory buffer for the pixel stream.)

When this function returns, the camera is ready to begin the exposure. pl_exp_start_cont initiates exposure and readout.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets pl_error_code.

SEE ALSO

pl_exp_start_cont(3), pl_exp_check_cont_status(3), pl_exp_get_oldest_frame(3), pl_exp_get_latest_frame(3), pl_exp_unlock_oldest_frame(3), pl_exp_stop_cont(3)

NOTES

Use the parameter ID PARAM_CIRC_BUFFER with pl_get_param to see if the system can perform circular buffer operations. The circular buffer is

passed to `pl_exp_start_cont`. The buffer is allocated by your application.

PVCAM **Class 3: Data Acquisition** **pl_exp_setup_seq(3)**

NAME

`pl_exp_setup_seq` — prepares the camera to perform a readout.

SYNOPSIS

```
rs_bool
   pl_exp_setup_seq(
              int16 hcam,
              uns16 exp_total,
              uns16 rgn_total,
              const rgn_type* rgn_array,
              int16 mode,
              uns32 exposure_time,
              uns32* stream_size)
```

DESCRIPTION

This function uses the array of regions, exposure mode, and exposure time passed in and transmits them to the camera. `exp_total` specifies the number of images to take. The pointer `rgn_array` points to `rgn_total` region definitions, `mode` specifies the bitwise OR combination of exposure mode and expose out mode (see chapter *Extended Exposure Modes* on page 25), `exposure_time` specifies the exposure time in the currently selected exposure time resolution (see `PARAM_EXP_RES` and `PARAM_EXP_RES_INDEX`). The pointer `stream_size` points to a variable that will be filled with number of bytes in the pixel stream.

The settings are then downloaded to the camera. If there is any problem (overlapping regions or a frame-transfer setting for a camera that lacks that capability), this function aborts and returns with a failure. `pl_error_code` indicates the definition problem.

The `stream_size` pointer is filled with the number of bytes of memory needed to buffer the full sequence. (It is the developer's responsibility to allocate a memory buffer for the pixel stream.)

When this function returns, the camera is ready to begin the exposure. `pl_exp_start_seq` initiates exposure and readout.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure `sets pl_error_code`.

SEE ALSO

`pl_exp_abort(3), pl_exp_check_status(3), pl_exp_start_seq(3), pl_exp_finish_seq(3)`

NOTES

This function downloads new settings. After receiving the settings, the camera merely waits in an idle state. The `pl_exp_abort` command may be used to place the camera into some other state, such as continuous clearing, but this will not alter or affect the downloaded settings. Essentially, the camera is still holding the exposure sequence and waiting to start, while it clears the sensor charge.

PVCAM **Class 3: Data Acquisition** **pl_exp_start_cont(3)**

NAME

`pl_exp_start_cont` — begins continuous readout into circular buffer

SYNOPSIS

```
rs_bool
   pl_exp_start_cont(
              int16 hcam,
              void* pixel_stream,
              uns32 size)
```

DESCRIPTION

This function will initiate a continuous readout from the camera into a circular buffer. `pixel_stream` is a pointer to the circular buffer, and `size` indicates the number of bytes the buffer can hold.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3), pl_exp_check_cont_status(3), pl_exp_get_oldest frame(3), pl_exp_get_latest_frame(3), pl_exp_unlock_oldest_frame(3), pl_exp_stop_cont(3)`

NOTES

If `pixel_stream` points to a buffer that is not an integer-multiple of the frame size for the exposure, this function will return FALSE and set an appropriate error code in `pl_error_code`. For example, a buffer size of 1000 bytes with a frame size of 250 is OK, but a buffer size of 900 bytes would cause a failure.

Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

PVCAM **Class 3: Data Acquisition** **pl_exp_start_seq(3)**

NAME

`pl_exp_start_seq` — begins exposing, returns immediately.

SYNOPSIS

```
rs_bool
   pl_exp_start_seq(
              int16 hcam,
              void* pixel_stream)
```

DESCRIPTION

This is a companion function to `pl_exp_setup_seq`. `pl_exp_setup_seq` must be called first to define the exposure and program this information into the camera. After that, `pl_exp_start_seq` may be called one or more times. Each time it is called, it starts one sequence and returns immediately (a sequence may be one or more exposures).

Progress can be monitored through `pl_exp_check_status`. The next sequence may be started as soon as the readout has finished or an abort has been performed (`pl_exp_abort`). The `hcam` parameter defines which camera is used.

The user must allocate an appropriately sized memory buffer for data collection, pointed to by `pixel_stream`. This buffer must be at least `stream_size` bytes, where `stream_size` is the value returned from `pl_exp_setup_seq`. In addition, this memory must be page-locked or similarly protected on virtual memory systems — these requirements are system specific and the responsibility of the application.

There is a special case for those users who want to use their own frame grabber (with an appropriately equipped camera). If a null pointer is passed in for `pixel_stream`, `pl_exp_start_seq` will assume that the user is routing the data to a frame grabber or other device under their control. Under these conditions, `pl_exp_start_seq` initiates the exposure, but does not attempt to collect incoming data.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets pl_error_code.

SEE ALSO

`pl_exp_check_status(3), pl_exp_setup_seq(3, pl_exp_finish_seq(3)`

NOTES

Technically, this only changes the state of the CCS program. Regardless of whether the CCS is idle or continuously clearing, this forces the CCS program into the busy state. The camera settings are not altered by this command, but it does begin executing. If the CCS is idle, there is no delay and the camera will begin running immediately. If the CCS is continuously clearing, the system finishes the current parallel shift (it finishes the current single parallel row) and then begins running. This produces a delay of up to the parallel-shift time for this CCD (1–300 microseconds, depending on the CCD). If the camera has been set up with one of the `CLEAR_PRE_` clearing modes, it will also explicitly clear the sensor as its first action.

PVCAM **Class 3: Data Acquisition** **pl_exp_abort(3)**

NAME

`pl_exp_abort` – stops collecting data, cleans up device driver, halts camera.

SYNOPSIS

```
rs_bool
   pl_exp_abort(
               int16 hcam,
               int16 cam_state)
```

DESCRIPTION

`pl_exp_abort` performs two functions: it stops the host device driver, and it may halt the camera (`hcam` specifies which camera and which device driver are being used.) Halting the camera halts `readout`,`clearing`, and all other camera activity. On the host side, data collection is controlled by a device driver. If data collection is currently enabled (the image data **active** state), this function stops collection, returns the low-level communication hardware and software to an image data **idle** state, and disables collection. In the **idle** state, any data that arrives is ignored and discarded. The **idle** state is the normal system default. On the camera side, the Camera Control Subsystem (CCS) may be in the process of collecting data, or it may be in one of several idle states.

This function always stops the data collection software. In addition, it has the option of forcing the CCS into a new state by setting the `cam_state` variable to one of the following constants, which are camera dependent:

`CCS_NO_CHANGE`
: Do not alter the current state of the CCS.

`CCS_HALT`
: Halt all CCS activity, and put the CCS into the idle state.

`CCS_HALT_CLOSE_SHTR`
: Close the shutter, then halt all CCS activity, and put the CCS into the idle state.

`CCS_CLEAR`
: Put the CCS into the continuous clearing state.

`CCS_CLEAR_CLOSE_SHTR`
: Close the shutter, then put the CCS into the continuous clearing state.

`CCS_OPEN_SHTR`
: Open the shutter, then halt all CCS activity, and put the CCS into the idle state.

`CCS_CLEAR_OPEN_SHTR`
: Open the shutter, then put the CCS into the continuous clearing state.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

NOTES

This may also be called outside of an exposure. It can explicitly open the shutter, close the shutter, or stop the CCS.

In the **idle** state, the system takes the least possible amount of action when image data arrives. On some systems, this involves placing the hardware in reset state, so it is inactive. On SCSI systems, the driver does not initiate any data transfers, although a buffer on the camera end may be filling up.

If the CCS is halted and the shutter is closed (`CCS_HALT_CLOSE_SHTR`), the current image remains on the sensor (although dark charge continues to accumulate). If `clear_cycles` is zero or the clear mode is `CLEAR_NEVER`, the image may be read off by performing a bias readout.

In frame transfer mode, you may not want to close the shutter when halting the CCS. Some frame transfer systems do not include a shutter, in which case an attempt to open or close the shutter is ignored, but does not cause an error.

PVCAM **Class 3: Data Acquisition** **pl_exp_stop_cont(3)**

NAME

`pl_exp_stop_cont` — stops continuous readout acquisition.

SYNOPSIS

```
rs_bool
   pl_exp_stop_cont(
              int16 hcam,
              int16 cam_state)
```

DESCRIPTION

This function halts a continuous readout acquisition into a circular buffer. `cam_state` defines the new state of the Camera Control Subsystem, as described in the documentation for the `pl_exp_abort` function.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3), pl_exp_start_cont(3), pl_exp_check_cont_status(3), pl_exp_get_oldest frame(3), pl_exp_get_latest_frame(3), pl_exp_unlock_oldest_frame(3)`

NOTES

Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

PVCAM **Class 3: Data Acquisition** **pl_exp_check_status(3)**

NAME

`pl_exp_check_status` — checks the status of the current exposure.

SYNOPSIS

```
rs_bool
   pl_exp_check_status(
               int16 hcam,
               int16* status,
               uns32* byte_cnt)
```

DESCRIPTION

This is only useful when data collection has been set up and started, as with a call to the Class 3 functions `pl_exp_setup_seq` and `pl_exp_start_seq`. In general, Class 3 functions start an exposure then immediately return, allowing the progress to be monitored. The status gives a quick evaluation of progress. The variable status returns one of the following values:

`READOUT_NOT_ACTIVE`
: The system is **idle**, no data is expected. If any arrives, it will be discarded.

`EXPOSURE_IN_PROGRESS`
: The data collection routines are **active**. They are waiting for data to arrive, but none has arrived yet.

`READOUT_IN_PROGRESS`
: The data collection routines are **active**. The data has started to arrive.

`READOUT_COMPLETE`
: All the expected data has arrived. Data collection is complete, and the driver has returned to **idle** state.

`READOUT_FAILED`
: Something went wrong. The function returns a FALSE and `pl_error_code` is set. (See Return Value below for more information.)

More detailed information is returned in `byte_cnt`. This reports on exactly how many bytes of data have arrived so far (divide by two to get the number of pixels). This level of feedback is unimportant to many users.

RETURN VALUE

TRUE means the status was checked successfully, FALSE indicates a bad handle, a problem communicating with the camera or driver, or some type of readout failure. Failure will set `pl_error_code`.

SEE ALSO

`pl_exp_setup_seq(3), pl_exp_start_seq(3)`

NOTES

PVCAM **Class 3: Data Acquisition** **pl_exp_check_cont_status(3)**

NAME

`pl_exp_check_cont_status` — checks the continuous readout status from the camera into a circular buffer.

SYNOPSIS

```
rs_bool
   pl_exp_check_cont_status(
              int16 hcam,
              int16* status,
              uns32* byte_cnt,
              uns32* buffer_cnt)
```

DESCRIPTION

This function will return the status of a continuous readout from the camera into a circular buffer. `status` is a pointer to one of the following values:

`READOUT_NOT_ACTIVE`
: The system is **idle**, no data is expected. If any arrives, it will be discarded.

`EXPOSURE_IN_PROGRESS`
: The data collection routines are **active**. They are waiting for data to arrive, but none has arrived yet.

`READOUT_IN_PROGRESS`
: The data collection routines are **active**. The data has started to arrive.

`FRAME_AVAILABLE`
: There is at least one frame which has not yet been retrieved from the buffer.

`READOUT_FAILED`
: Something went wrong. The function returns a FALSE and `pl_error_code` is set. (See Return Value below for more information.)

The `byte_cnt` reports on how many bytes of data have arrived since `buffer_cnt` has been incremented.

The `buffer_cnt` points to the number of times the buffer has been filled.

RETURN VALUE

TRUE is returned for success, FALSE for a failure. Failure will set `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3), pl_exp_start_cont(3), pl_exp_get_oldest frame(3), pl_exp_get_latest_frame(3), pl_exp_unlock_oldest_frame(3), pl_exp_stop_cont(3)`

NOTES

This function only returns meaningful results if a continuous readout from the camera has been initiated by a call to `pl_exp_start_cont`. Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

PVCAM **Class 3: Data Acquisition** **pl_exp_check_cont_status_ex(3)**

NAME

`pl_exp_check_cont_status_ex` — checks the continuous readout status from the camera into a circular buffer.

SYNOPSIS

```
rs_bool
   pl_exp_check_cont_status_ex(
              int16 hcam,
              int16*  status,
              uns32* byte_cnt,
              uns32* buffer_cnt,
              FRAME_INFO* pFrameInfo)
```

DESCRIPTION

This function will return the status of a continuous readout from the camera into a circular buffer. `status` is a pointer to one of the following values:

```
READOUT_NOT_ACTIVE
EXPOSURE_IN_PROGRESS
READOUT_IN_PROGRESS
ACQUISITION_IN_PROGRESS
READOUT_COMPLETE
READOUT_FAILED
```

`byte_cnt` reports on how many bytes of data have arrived since `buffer_cnt` has been incremented.

`buffer_cnt` points to the number of times the buffer has been filled.

Values in the variable pointed to by pFrameInfo will be updated with frame counters, timestamps (with precision of 0.1ms) and readout time information assigned by device driver at the moment of frame reception.

RETURN VALUE

TRUE is returned for success, FALSE for a failure. Failure will set `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3), pl_exp_start_cont(3), pl_exp_get_oldest frame(3), pl_exp_get_latest_frame(3), pl_exp_unlock_oldest_frame(3), pl_exp_stop_cont(3), pl_create_frame_info_struct(2), pl_exp_get_latest_frame_ex(3), pl_exp_get_oldest_frame_ex(3)`

NOTES

This function only returns meaningful results if a continuous readout from the camera has been initiated by a call to `pl_exp_start_cont`. Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

Variable pointed to by `pFrameInfo` must be created with `pl_create_frame_info_struct(2)`.

PVCAM **Class 3: Data Acquisition** **pl_exp_unlock_oldest_frame(3)**

NAME

`pl_exp_unlock_oldest_frame` — makes oldest frame in circular buffer overwriteable.

SYNOPSIS

```
rs_bool
   pl_exp_unlock_oldest_frame(
              int16 hcam)
```

DESCRIPTION

This function unlocks the oldest frame in the circular buffer; the frame should have been locked previously by a call to `pl_exp_get_oldest_frame`.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_exp_setup_cont(3), pl_exp_start_cont(3), pl_exp_check_cont_status(3), pl_exp_get_oldest frame(3), pl_exp_unlock_oldest_frame(3), pl_exp_stop_cont(3)`

NOTES

Failure to call this function after using the frame will cause the continuous acquisition progress to halt eventually, because the frame cannot be overwritten when it is locked.

Use the parameter ID `PARAM_CIRC_BUFFER` with `pl_get_param` to check to see if the system can perform circular buffer operations.

PVCAM	**Class 3: Data Acquisition**	**pl_io_clear_script_control(3)**

NAME

`pl_io_clear_script_control` — Clears the current setup for control of the available I/O lines within a camera script.

SYNOPSIS

```
rs_bool
   pl_io_clear_script_control(
              int16 hcam)
```

DESCRIPTION

This function allows the application program to clear the current setup for control of the available I/O lines within the script. This allows the user to enter a new setup for these lines.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_io_script_control(3)`

NOTES

PVCAM **Class 3: Data Acquisition** **pl_io_script_control(3)**

NAME

`pl_io_script_control` — Defines control of an I/O line from within a camera script.

SYNOPSIS

```
rs_bool
   pl_io_script_control(
              int16 hcam,
              uns16 addr,
              flt64  state,
              uns32 location)
```

DESCRIPTION

This function allows the application program to define control of the available I/O lines from within a script. This allows for more precise control of external devices. For example, the application could request that a linear stage be indexed immediately after integration, instead of waiting until after the data is read out, the shutter is closed, etc. `addr` specifies which I/O address to control. `state` specifies the desired setting for the address being controlled.

`state` has different meanings depending on the I/O type:

`IO_TYPE_TTL`
: The bit pattern written to this address.

`IO_TYPE_DAC`
: The value of the desired analog output written to the DAC at this address.

`location` can be set to the following values:

`SCR_PRE_OPEN SHTR`	`SCR_POST_OPEN_SHTR`
`SCR_PRE_FLASH`	`SCR_POST_FLASH`
`SCR_PRE_INTEGRATE`	`SCR_POST_INTEGRATE`
`SCR_PRE_READOUT`	`SCR_POST_READOUT`
`SCR_PRE_CLOSE_SHTR`	`SCR_POST_CLOSE_SHTR`

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_io_clear_script_control(3)`

NOTES

PVCAM	**Class 3: Data Acquisition** **pl_md_frame_decode(3)**
NAME	`pl_md_frame_decode` — decodes a metadata-enabled frame buffer into provided frame descriptor structure.
SYNOPSIS	**`rs_bool`** **`  pl_md_frame_decode(`** **`  md_frame* pDstFrame,`** **`  void* pSrcBuf,`** **`  uns32 srcBufSize)`**
DESCRIPTION	This function processes the input frame buffer and calculates pointers to frame metadata headers, ROI headers and ROI image data and stores them to previously allocated `pDstFrame` structure.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO	`pl_md_create_frame_struct(3),`
	`pl_md_create_frame_struct_cont(3),`
	`pl_md_release_frame_struct(3),`
	`pl_md_frame_recompose(3), pl_md_read_extended(3)`

NOTES	The `md_frame` structure can be used to access the image data and metadata directly, or it can be further passed to `pl_md_frame_recompose(3)` function.

PVCAM	**Class 3: Data Acquisition** **pl_md_create_frame_struct(3)**
NAME	`pl_md_create_frame_struct` — creates a helper structure – a frame descriptor that is used to decode metadata-enabled buffer. The structure must be then released with `pl_md_release_frame_struct(3)`.
SYNOPSIS	**`rs_bool`** **`pl_md_create_frame_struct(`** **`md_frame** pFrame,`** **`void* pSrcBuf,`** **`uns32 srcBufSize)`**
DESCRIPTION	The function allocates the frame descriptor structure for existing metadata-enabled frame buffer.
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.
SEE ALSO	`pl_md_frame_decode(3)`, `pl_md_create_frame_struct_cont(3)`, `pl_md_release_frame_struct(3)`, `pl_md_frame_recompose(3)`, `pl_md_read_extended(3)`
NOTES	Use this function for single frame acquisitions or where performance and memory overhead is not an issue. Also this function shall be used if a frame buffer with unknown ROI count is decoded (for example when the buffer is loaded from a disk without knowledge of the acquisition settings). For continuous mode or when many frames are acquired in a single sequence it is recommended to pre-allocate a single frame descriptor structure with `pl_md_create_frame_struct_cont(3)` and reuse this structure when decoding incoming frames.

PVCAM	**Class 3: Data Acquisition**	**pl_md_create_frame_struct_cont(3)**

NAME

`pl_md_create_frame_struct_cont` — creates a helper structure – a frame descriptor that is used to decode metadata-enabled buffer. The structure must be then released with `pl_md_release_frame_struct(3)`.

SYNOPSIS

```
rs_bool
   pl_md_create_frame_struct_cont(
              md_frame** pFrame,
              uns16 roiCount)
```

DESCRIPTION

The function allocates the frame descriptor structure for a frame with known ROI count.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_md_frame_decode(3), pl_md_create_frame_struct(3), pl_md_release_frame_struct(3), pl_md_frame_recompose(3), pl_md_read_extended(3)`

NOTES

Use this function before starting the continuous or large sequence acquisition to create a single pre-allocated frame descriptor structure. When using multiple ROIs or Centroids feature the number of ROIs is usually known in advance, this allows the structure to be created before the acquisition and reused each time a frame is acquired.

PVCAM **Class 3: Data Acquisition** **pl_md_release_frame_struct(3)**

NAME

`pl_md_release_frame_struct` — releases a previously created frame descriptor structure from the memory.

SYNOPSIS

```
rs_bool
   pl_md_release_frame_struct(
              md_frame* pFrame)
```

DESCRIPTION

The function releases the memory of the frame descriptor structure that was previously created with `pl_md_create_frame_struct(3)` or `pl_md_create_frame_struct_cont(3)`.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_md_frame_decode(3)`, `pl_md_create_frame_struct(3)`, `pl_md_create_frame_struct_cont(3)`, `pl_md_frame_recompose(3)`, `pl_md_read_extended(3)`

NOTES

PVCAM **Class 3: Data Acquisition** **pl_md_frame_recompose(3)**

NAME

`pl_md_frame_recompose` — creates a displayable image from a metadata-enabled frame buffer containing multiple ROIs.

SYNOPSIS

```
rs_bool
   pl_md_frame_recompose(
              void* pDstBuf,
              uns16 offX,
              uns16 offY,
              uns16 dstWidth,
              uns16 dstHeight,
              md_frame* pDstFrame)
```

DESCRIPTION

The function fills in a pre-allocated user-provided display buffer with image data from metadata-enabled PVCAM buffer. The image data is copied. Use this function to create a displayable image from a buffer containing multiple ROIs.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_md_frame_decode(3), pl_md_create_frame_struct(3), pl_md_create_frame_struct_cont(3), pl_md_release_frame_struct(3), pl_md_read_extended(3)`

NOTES

The `pDstBuf` user buffer does not have to be black filled each time this function is called. This is not necessary if the buffer is being re-used for displaying live frames with user defined ROIs, in this case the ROIs are not moving across the frame and the buffer can be black-filled only once before starting the acquisition. However when Centroids are used it is likely that each acquired frame may have the Centroids positioned differently, in such scenario it is recommended to black-fill the `pDstBuf` each time this method is called otherwise a ghosts from previous use may be visible in the final image. The black-filling is responsibility of the caller.

PVCAM **Class 3: Data Acquisition** **pl_md_read_extended(3)**

NAME

`pl_md_read_extended` — reads extended frame or ROI metadata from the metadata-enabled frame buffer.

SYNOPSIS

```
rs_bool
   pl_md_read_extended(
              md_ext_item_collection* pOutput,
              void* pExtMdData,
              uns32 extMdDataSize)
```

DESCRIPTION

The function will read the extended metadata buffer and fills in a structure for easy access to this metadata.

RETURN VALUE

TRUE for success, FALSE for a failure. Failure sets `pl_error_code`.

SEE ALSO

`pl_md_frame_decode(3)`, `pl_md_create_frame_struct(3)`, `pl_md_create_frame_struct_cont(3)`, `pl_md_frame_recompose(3)`, `pl_md_release_frame_struct(3)`, `pl_md_read_extended(3)`

NOTES

The `pOutput` must be allocated before using this function. The `pExtMdData` and `extMdDataSize` shall be obtained from the `md_frame` structure that was previously filled with `pl_md_frame_decode(3)`.

Class 3 Parameter IDs

The following parameter IDs are used with pl_get_param, pl_set_param, pl_get_enum_param, and pl_enum_str_length functions described in *Chapter 5*.

Note: Before trying to use or retrieve more information about a parameter, it is always recommended to call an ATTR_AVAIL to see if the system supports it.

Class 3 Parameter ID	Description
PARAM_BOF_EOF_CLR **Camera Dependent**	Clears the BOF/EOF count when a pl_set_param is performed. This is a write-only parameter. **Datatype: rs_bool**
PARAM_BOF_EOF_COUNT **Camera Dependent**	Returns the Begin-Of-Frame and/or End-Of-Frame count. BOF/EOF counting is enabled and configured with PARAM_BOF_EOF_ENABLE. **Datatype: uns32**
PARAM_BOF_EOF_ENABLE **Camera Dependent**	Enables and configures the BOF/EOF interrupts. Possible values are: NO_FRAME_IRQS BEGIN_FRAME_IRQS END_FRAME_IRQS BEGIN_END_FRAME_IRQS **Datatype: enum (int32)**
PARAM_CIRC_BUFFER	Tests to see if the hardware/software can perform circular buffer. When a TRUE is returned for ATTR_AVAIL attribute, the circular buffer function can be used. **Datatype: rs_bool**
PARAM_FRAME_BUFFER_SIZE	Retrieves the min, max, current and recommended (default) buffer size in bytes for currently configured acquisition. This parameter becomes available only after calling the pl_exp_setup_seq or pl_ext_setup_cont. For sequence acquisition the attribute always report the full sequence size in bytes. For circular buffer acquisition use the ATTR_DEFAULT to retrieve the recommended buffer size. **Datatype: ulong64**

Class 3 Parameter ID	Description
PARAM_EXP_RES	Gets the resolution for the current resolution index, as described for PARAM_EXP_RES_INDEX. This value is an enumerated type, representing the resolution. This parameter does exactly the same as PARAM_EXP_RES_INDEX but additionally provides human-readable string for each exposure resolution. Possible values are: EXP_RES_ONE_MICROSEC EXP_RES_ONE_MILLISEC EXP_RES_ONE_SEC **Datatype: enum (int32)**
PARAM_EXP_RES_INDEX	Gets and sets the index into the exposure resolution table for the camera. The table contains the resolutions supported by the camera. The value at this index is an enumerated type, representing different resolutions (such as EXP_RES_ONE_MILLISEC or EXP_RES_ONE_MICROSEC). The number of supported resolutions can be obtained by using the ATTR_COUNT attribute. **Datatype: uns16**
PARAM_EXP_TIME	This is used to examine and change the exposure time in VARIABLE_TIMED_MODE only. The value is limited to 16-bit. For higher exposure times separate single frame acquisitions, or SMART streaming (if available), have to be used. **Datatype: uns16**
PARAM_EXPOSURE_TIME	This is used to examine current exposure time and range of valid values. The minimum and maximum value could be limited by camera HW. Use ATTR_MIN and ATTR_MAX to retrieve it. This parameter is always available but for older cameras not reporting their limits, the min. value is set to 0 and max. value set to max. 32bit value for backward compatibility. It means the range is not known (it does not mean there are no limits). In such case e.g. camera manual could specify some limits. **Datatype: ulong64**
PARAM_ROI_COUNT	Read only parameter. The ATTR_CURRENT returns the currently configured number of ROIs (via pl_exp_setup functions). The ATTR_MAX can be used to retrieve the maximum number of ROIs the camera supports. **Datatype: uns16**

Class 3 Parameter ID	Description
PARAM_METADATA_ENABLED **Camera Dependent**	This parameter is used to enable or disable the embedded frame metadata feature. Once enabled the acquired frames will contain additional information describing the frame. Please refer to *Embedded Frame Metadata* chapter for more information. **Datatype: rs_bool**
PARAM_CENTROIDS_ENABLED **Camera Dependent**	This parameter is used to enable or disable the Centroids feature. Please refer to *Centroids* chapter for more information. **Datatype: rs_bool**
PARAM_CENTROIDS_COUNT **Camera Dependent**	This read-write parameter is used to obtain the supported number of Centroids and set the desired number of Centroids to the camera. **Datatype: uns16**
PARAM_CENTROIDS_RADIUS **Camera Dependent**	This read-write parameter is used to obtain the range of Centroids radii the camera supports. Use the ATTR_MIN and ATTR_MAX to retrieve the range. The radius defines the distance from the center pixel. For example if the camera reports the radius range between 1 and 5 it means that the resulting ROIs can be configured to following sizes: 1=3×3, 2=5×5, 3=7×7, 4=9×9, 5=11×11 Use pl_set_param to set the desired Centroids radius. Once set, make sure to reconfigure the acquisition with pl_exp_setup functions. **Datatype: uns16**
PARAM_BINNING_SER **Camera Dependent**	This read-only parameter is used to obtain serial part of serial × parallel binning factors permutations. It has to be always used in pair with PARAM_BINNING_PAR parameter. Please refer to *Binning Factor Discovery* chapter for more information. **Datatype: enum (int32)**
PARAM_BINNING_PAR **Camera Dependent**	This read-only parameter is used to obtain parallel part of serial × parallel binning factors permutations. It has to be always used in pair with PARAM_BINNING_SER parameter. Please refer to *Binning Factor Discovery* chapter for more information. **Datatype: enum (int32)**
PARAM_TRIGTAB_SIGNAL **Camera Dependent**	This read-write parameter is used to select the output signal to be configured. The configuration of number of active outputs is done via PARAM_LAST_MUXED_SIGNAL parameter. Please refer to *Triggering Table* chapter for more information. **Datatype: enum (int32)**

Class 3 Parameter ID	Description
PARAM_LAST_MUXED_SIGNAL **Camera Dependent**	This read-write parameter is used to set the number of active output signals used by multiplexer for the signal selected by PARAM_TRIGTAB_SIGNAL parameter. Please refer to *Triggering Table* chapter for more information. **Datatype: uns8**

This page intentionally left blank.

Index

A

ANSI C library 9

B

Binning 13
Bit depth 11
Buffers 33

C

Callback types
  PL_CALLBACK_BOF 43
  PL_CALLBACK_CAM_REMOVED 43
  PL_CALLBACK_CAM_RESUMED 43
  PL_CALLBACK_CHECK_CAMS 43
  PL_CALLBACK_EOF 43
Camera
  communication 10, 34
  settings 58
Circular buffer ... 18, 95, 96, 97, 98, 99, 102, 106, 108, 109, 110, 119
Circular buffer modes
  CIRC_NO_OVERWRITE 95, 96
  CIRC_OVERWRITE 18, 97, 98
Class 0 ..... 10, 34
Class 0 Functions 34
  pl_cam_close 35
  pl_cam_deregister_callback 49
  pl_cam_get_name 36
  pl_cam_get_total 37
  pl_cam_open 38
  pl_cam_register_callback 42
  pl_cam_register_callback_ex 44
  pl_cam_register_callback_ex2 45
  pl_cam_register_callback_ex3 47
  pl_pvcam_get_ver 39
  pl_pvcam_init 40
  pl_pvcam_uninit 41
Class 1 10
Class 1 Functions 54
  pl_error_code 55
  pl_error_message 56
Class 2 10
Class 2 Functions 58
  pl_create_frame_info_struct 67, 68
  pl_create_smart_stream_struct 69, 70
  pl_enum_str_length 65
  pl_get_enum_param 64
  pl_get_param 60
  pl_pp_reset 66
  pl_release_frame_info_struct 68
  pl_release_smart_stream_struct 70
  pl_set_param 63
Class 3 10
Class 3 Functions 85
  pl_exp_abort 104
  pl_exp_check_cont_status 108, 109
  pl_exp_check_status 107
  pl_exp_finish_seq 94
  pl_exp_get_latest_frame 95
  pl_exp_get_latest_frame_ex 96
  pl_exp_get_oldest_frame 97
  pl_exp_get_oldest_frame_ex 98
  pl_exp_setup_cont 99
  pl_exp_setup_seq 101
  pl_exp_start_cont 102
  pl_exp_start_seq 103
  pl_exp_stop_cont 106
  pl_exp_unlock_oldest_frame 110
  pl_io_clear_script_control 111
  pl_io_script_control 112
  pl_md_create_frame_struct 114
  pl_md_create_frame_struct_cont 115
  pl_md_frame_decode 113
  pl_md_frame_recompose 117
  pl_md_read_extended 118
  pl_md_release_frame_struct 116

Clear modes 18, 73
CLEAR_NEVER 21, 73
CLEAR_POST_SEQUENCE 22, 73
CLEAR_PRE_EXPOSURE 21, 73
CLEAR_PRE_EXPOSURE_POST_SEQ ... 73
CLEAR_PRE_EXPOSURE_POST_SEQUENCE 22
CLEAR_PRE_POST_SEQUENCE 22, 73
CLEAR_PRE_SEQUENCE 22, 73
Clocking modes
PMODE_ALT_FT 78
PMODE_ALT_FT_MPP 78
PMODE_ALT_MPP 78
PMODE_ALT_NORMAL 78
PMODE_FT 78
PMODE_FT_MPP 78
PMODE_MPP 78
PMODE_NORMAL 78
Close delay 27
Color modes 74
COLOR_NONE 74
COLOR_RGGB 74
Configuration/setup 10, 57
Contact information 7
Cooling modes 74
CRYO_COOL 74
NORMAL_COOL 74
Customer service 7

D

Data acquisition 10, 85
Data arrays 13
Defined types 11
Defining exposures 87
Display orientation 13

E

Error conditions 53
Error reporting 10, 53
Expose out modes 25, 26, 27, 74
Exposure loops 28
Exposure modes 18, 22, 25, 88
BULB_MODE 24, 74, 88
STROBED_MODE 24, 74, 88
TIMED_MODE 23, 74, 88
TRIGGER_FIRST_MODE 23, 74, 88
VARIABLE_TIMED_MODE 23, 74, 88
Exposure resolutions
EXP_RES_ONE_MICROSEC 120
EXP_RES_ONE_MILLISEC 120
EXP_RES_ONE_SEC 120
Exposure scripts 28
Extended exposure modes 25, 27, 88

F

Fan speeds
FAN_SPEED_HIGH 82
FAN_SPEED_LOW 82
FAN_SPEED_MEDIUM 82
FAN_SPEED_OFF 82
Frame interrupts
BEGIN_END_FRAME_IRQS 119
BEGIN_FRAME_IRQS 119
END_FRAME_IRQS 119
NO_FRAME_IRQS 119
Frame transfer 15
Full lateral resolution 13
Full-sensor image in buffer 33

G

Gain 14
Get and Set Parameter Functions
pl_enum_str_length 57
pl_get_enum_param 57
pl_get_param 57
pl_set_param 57

H

Handle 12

I

I/O directions
IO_DIR_INPUT 76
IO_DIR_INPUT_OUTPUT 76
IO_DIR_OUTPUT 76
I/O types
IO_TYPE_DAC 76

IO_TYPE_TTL 76
Image
array 15
buffers 33
smear 16
Include files 11, 12
master.h 12
pvcam.h 12
Initialization functions 34

L

Library classes 10

M

master.h 12
MMP modes
MMP_ALWAYS_OFF 77
MMP_ALWAYS_ON 77
MMP_SELECTABLE 77
MMP_UNKNOWN 77
Multiple exposures in buffer 33
Multi-tap Configuration and Readout 15

O

Open delay, close delay 27
Orientation of Sensor 12

P

Parallel 12
Parallel binning 13
PARAM_ACCUM_CAPABLE 71
PARAM_ACTUAL_GAIN 82
PARAM_ADC_OFFSET 71
PARAM_BINNING_PAR 121
PARAM_BINNING_SER 121
PARAM_BIT_DEPTH 71
PARAM_BOF_EOF_CLR 119
PARAM_BOF_EOF_COUNT 119
PARAM_BOF_EOF_ENABLE 119
PARAM_CAM_FW_VERSION 71
PARAM_CAMERA_PART_NUMBER 72
PARAM_CENTROIDS_COUNT 91, 121
PARAM_CENTROIDS_ENABLED 91, 121
PARAM_CENTROIDS_RADIUS 91, 121
PARAM_CHIP_NAME 71
PARAM_CIRC_BUFFER 119
PARAM_CLEAR_CYCLES 72
PARAM_CLEAR_MODE 73
PARAM_COLOR_MODE 74
PARAM_COOLING_MODE 74
PARAM_DD_INFO 50
PARAM_DD_INFO_LENGTH 50
PARAM_DD_RETRIES 50
PARAM_DD_TIMEOUT 50
PARAM_DD_VERSION 51
PARAM_EXP_RES 120
PARAM_EXP_RES_INDEX 120
PARAM_EXP_TIME 23, 120
PARAM_EXPOSE_OUT_MODE 74
PARAM_EXPOSURE_MODE 74
PARAM_EXPOSURE_TIME 120
PARAM_FAN_SPEED_SETPOINT 82
PARAM_FRAME_BUFFER_SIZE 18, 119
PARAM_FRAME_CAPABLE 74
PARAM_FWELL_CAPACITY 74
PARAM_GAIN_INDEX 75
PARAM_GAIN_MULT_ENABLE 75
PARAM_GAIN_MULT_FACTOR 75
PARAM_GAIN_NAME 75
PARAM_HEAD_SER_NUM_ALPHA 75
PARAM_IO_ADDR 75
PARAM_IO_BITDEPTH 76
PARAM_IO_DIRECTION 76
PARAM_IO_STATE 76
PARAM_IO_TYPE 76
PARAM_LAST_MUXED_SIGNAL 122
PARAM_METADATA_ENABLED 121
PARAM_MPP_CAPABLE 77
PARAM_PAR_SIZE 77
PARAM_PCI_FW_VERSION 77
PARAM_PIX_PAR_DIST 77
PARAM_PIX_PAR_SIZE 77
PARAM_PIX_SER_DIST 77
PARAM_PIX_SER_SIZE 77
PARAM_PIX_TIME 77

PARAM_PMODE 78
PARAM_POSTMASK 78
PARAM_POSTSCAN 78
PARAM_PP_FEAT_ID 82, 83
PARAM_PP_FEAT_NAME 82
PARAM_PP_INDEX 82
PARAM_PP_PARAM 83
PARAM_PP_PARAM 83
PARAM_PP_PARAM_ID 83
PARAM_PP_PARAM_INDEX 82, 83
PARAM_PP_PARAM_NAME 83
PARAM_PREAMP_DELAY 78
PARAM_PREAMP_OFF_CONTROL 78
PARAM_PREMASK 78
PARAM_PRESCAN 78
PARAM_PRODUCT_NAME 72
PARAM_READ_NOISE 82
PARAM_READOUT_PORT 79
PARAM_READOUT_TIME 79
PARAM_ROI_COUNT 120
PARAM_SER_SIZE 79
PARAM_SHTR_CLOSE_DELAY 79
PARAM_SHTR_OPEN_DELAY 79
PARAM_SHTR_OPEN_MODE 80
PARAM_SHTR_STATUS 80
PARAM_SMART_STREAM_EXP_PARAMS 84
PARAM_SMART_STREAM_MODE 83
PARAM_SMART_STREAM_MODE_ENABLED 83
PARAM_SPDTAB_INDEX 81
PARAM_SUMMING_WELL 81
PARAM_SYSTEM_NAME 72
PARAM_TEMP 81
PARAM_TEMP_SETPOINT 81
PARAM_TRIGTAB_SIGNAL 121
PARAM_VENDOR_NAME 72
Parameter attributes
ATTR_ACCESS 60
ATTR_AVAIL 60
ATTR_COUNT 14, 57, 61, 64, 75, 79, 120
ATTR_CURRENT 14, 61
ATTR_DEFAULT 18, 61
ATTR_INCREMENT 61
ATTR_MAX 61
ATTR_MIN 61
ATTR_TYPE 62
Parameter passing 12
pbin 13
pl_cam_close 35
pl_cam_deregister_callback 49
pl_cam_get_name 36
pl_cam_get_total 37
pl_cam_open 38
pl_cam_register_callback 18, 42
pl_cam_register_callback_ex 44
pl_cam_register_callback_ex2 45
pl_cam_register_callback_ex3 47
pl_create_frame_info_struct 67, 68
pl_create_smart_stream_struct 69, 70
pl_enum_str_length 65
pl_error_code 54, 55
pl_error_message 54, 56
pl_exp_abort 104
pl_exp_check_cont_status 108, 109
pl_exp_check_status 107
pl_exp_finish_seq 94
pl_exp_get_latest_frame 19, 95
pl_exp_get_latest_frame_ex 96
pl_exp_get_oldest_frame 97
pl_exp_get_oldest_frame_ex 98
pl_exp_setup_cont 18, 99
pl_exp_setup_seq 22, 101
pl_exp_start_cont 18, 102
pl_exp_start_seq 13, 103
pl_exp_stop_cont 19, 106
pl_exp_unlock_oldest_frame 110
pl_get_enum_param 64
pl_get_param 18, 60
pl_io_clear_script_control 111
pl_io_script_control 112
pl_md_create_frame_struct 114

pl_md_create_frame_struct_cont....115
pl_md_frame_decode....113
pl_md_frame_recompose....117
pl_md_read_extended....118
pl_md_release_frame_struct....116
pl_pp_reset....66
pl_pvcam_get_ver....39
pl_pvcam_init....40, 57
pl_pvcam_uninit....41
pl_release_frame_info_struct....68
pl_release_smart_stream_struct....70
pl_set_param....63
PVCAM....7, 9
pvcam.h....12

R

Readout port selection....14
Readout ports
  READOUT_PORT_MULT_GAIN....79
  READOUT_PORT_NORMAL....79
Region....13

S

s,p coordinates....12
S.M.A.R.T. streaming modes
  SMTMODE_ARBITRARY_ALL....83
sbin....13
SDK....7
Sensor
  coordinates model....12
  orientation....12
  readout port....14
Sequence parameters....18
Sequences....18
Sequences in buffer....33
Serial binning....13
Serial register....13
Shutter open modes....18, 27, 80
  OPEN_NEVER....28, 80
  OPEN_NO_CHANGE....28, 80
  OPEN_PRE_EXPOSURE....27, 80
  OPEN_PRE_SEQUENCE....28, 80
  OPEN_PRE_TRIGGER....28, 80
Shutter states
  SHTR_CLOSED....80
  SHTR_CLOSING....80
  SHTR_FAULT....80
  SHTR_OPEN....80
  SHTR_OPENING....80
  SHTR_UNKNOWN....80
Single exposure, multiple images in buffer....33
Smearing....16
Specifying regions....13
Speed choices....14
Standard shutter....27
Storage array....15
System overview....9

T

Technical support....7

This page intentionally left blank.