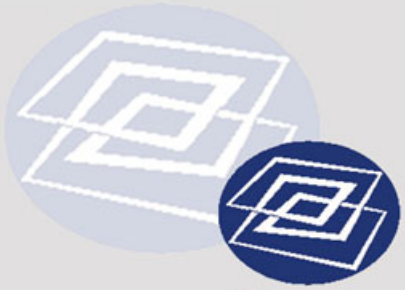
PVCAM 2.6



ROPER SCIENTIFIC™

BEYOND IMAGING

4411-0094
Version 2.6.C
April 15, 2003

Roper Scientific, Inc.
3660 Quakerbridge Rd
Trenton, NJ 08619
TEL: 800-874-9789 / 609-587-9797
FAX: 609-587-1970

Roper Scientific, Inc.
3440 E. Britannia Dr.
Tucson, AZ 85706
TEL: 800-874-9789 / 520-889-9933
FAX: 520-573-1944

Table of Contents

List of Tables

This page intentionally left blank.

Chapter 1: SDK

What is the SDK?

SDK — Roper Scientific's 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.)

Both the SDK and PVCAM are designed to be platform independent, so the functions described in this manual work with all supported operating systems. Specific information for installing and using the library with your particular platform (Windows©, Macintosh©, or UNIX©) is contained in the Read Me file included on the disk that came with your SDK. Please consult this Read Me file for information on:

- System requirements
- Linking PVCAM to your software
- Initializing PVCAM
- Device drivers
- Platform specific files

Contact Information

Roper Scientific's manufacturing facilities are located at the following addresses:

Roper Scientific, Inc.	Roper Scientific, Inc.
3440 East Britannia Drive	3660 Quakerbridge Road
Tucson, Arizona 85706 (USA)	Trenton, NJ 08619 (USA)
TEL: 800-874-9789 / 520-889-9933	TEL: 800-874-9789 / 609-587-9797
FAX: 520-573-1944	TEL: 609-587-1970

Tech Support E-mail: techsupport@roperscientific.com

Customer Service E-mail: cservice@roperscientific.com

For technical support and service outside the United States, see our web page at www.roperscientific.com. An up-to-date list of addresses, telephone numbers, and e-mail addresses of Roper Scientific's overseas offices and representatives is maintained on the web page.

This page intentionally left blank.

Chapter 2:
PVCAM, A High-Level C Library

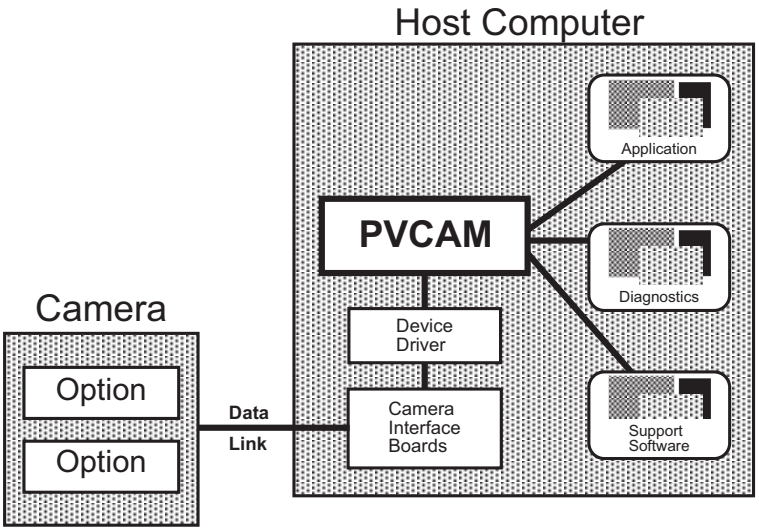
Introduction

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.

Note: Many Photometrics cameras support ICL scripting language that provides detailed low-level control of exposure and CCD readout. None of the Princeton Instruments cameras support ICL scripting.

System Overview

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



Hardware Support

Roper Scientific produces two lines of hardware: Photometrics brand and Princeton Instruments brand. Version 2.6 of the PVCAM library supports all Photometrics brand hardware. It also supports the following Princeton Instruments hardware:

- PCI Card
- PentaMAX Version 5.0
- ST-133 Controlled Cameras

Note:
Macintosh® computers are not currently supported for Princeton Instruments hardware.

Library Classes

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

0. Camera Communications	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.
4. Buffer Manipulation	These functions report buffer information and control buffer allocation and editing.

Note: Other classes are supported in optional plug-ins. Contact the factory for more information about plug-ins for PVCAM.

Documentation Style

This manual describes the functional aspects of using PVCAM and various controls for Roper Scientific® cameras (Chapter 2), gives reference pages for all of the function calls (Chapter 3 through Chapter 7), gives code examples (Chapter 8), provides a list of the defined error codes (Appendix A) and lists the function calls that are obsolete but still supported in the library (Appendix B).

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 the header file `master.h`.

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
enum	treat as unsigned 32-bit integral value
flt64	64-bit floating point value

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 Roper Scientific® 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.

The following list includes the new types defined for use in PVCAM. Additional derived types always begin with the base name followed by *_ptr* or *_const_ptr*.

Type	Pointer	Pointer to Constant Type
rs_bool	rs_bool_ptr	rs_bool_const_ptr
char	char_ptr	char_const_ptr
int8	int8_ptr	int8_const_ptr
uns8	uns8_ptr	uns8_const_ptr
int16	int16_ptr	int16_const_ptr
uns16	uns16_ptr	uns16_const_ptr
int32	int32_ptr	int32_const_ptr
uns32	uns32_ptr	uns32_const_ptr
flt64	flt64_ptr	flt64_const_ptr

Type	Pointer	Pointer to Constant Type
rgn_type	rgn_ptr	rgn_const_ptr
export_ctrl_type	export_ctrl_ptr	export_ctrl_const_ptr

Table 2. New Pointer Types

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	dly=delay	num=number
addr=address	dup=duplicate	ofs=offset
bin=binning	err = error	par=parallel
buf=buffer	exp=exposure	pix=pixel
cam=camera	expt=export	ptr=pointer
cfg=configuration	hbuf=buffer handle	rpt=report
chan=channel	hcam=camera handle	rgn=region
clr=clear	hi=high	ser=serial
cmd=command	hrgn=region handle	shtr=shutter
comm=communication	init=initialize	spd=speed
ctr=counter	len=length	tmp=temp
ctrl=control	lo=low	totl=total
diag=diagnostics	mem=memory	xfr=transfer

Table 3. Standard Abbreviations

In PVCAM, *num* always means **current selection number**, while *totl* or *entries* is used for **total 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 refers to an object.

Include Files

Any program using PVCAM must include the following files:

- master.h system-specific definitions and types
- pvcam.h constants and prototypes for all functions

master.h must be included before pvcam.h.

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:

- Pointers generally return information **from** a function.
- Non-pointers always send information **to** a function.

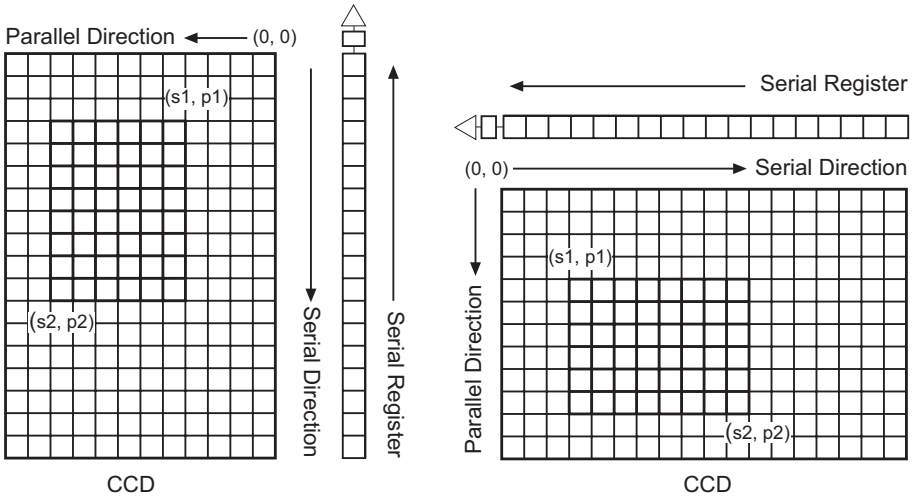
In a few cases, such as structures and arrays, a pointer is passed even though the data are being sent in to the function. This is done to reduce overhead and to speed function calls, but it conflicts with the rules above. To solve this problem, when a structure or array (pointer) is sent as input to a function, the *_const_ptr* type is used to indicate that the function will not (and can not) change the data.

Note: *const_ptr* (pointers to const) always sends data ***into*** a function. The data is not altered.

CCD Coordinates Model

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

In Roper Scientific cameras, the CCD 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 CCD orientation.



Regions and Images

A region is a user-defined, rectangular exposure area on the CCD. 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

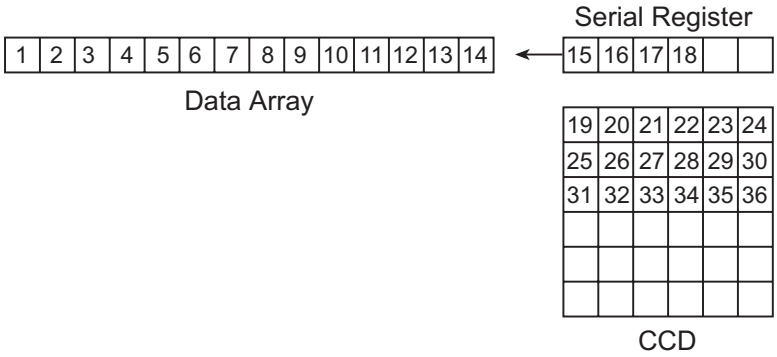
For data collection, two other parameters are needed: the serial and parallel binning factors. A binning of 1 in both directions reads out each pixel at full lateral resolution. A binning of 2 in both directions combines four pixels, cutting the lateral resolution in half, but quadrupling the light-collecting 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:



Display Orientation

Some users have expressed an interest in having the data in video coordinates. With video coordinates, 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 CCD in a camera will have one or more output nodes from which the analog pixel stream will be read. These nodes are referred to as “Readout Ports”. The signal from a readout port is passed to an analog signal processing chain and then passed to an analog to digital converter (ADC). The ADC 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 view the port settings, call `pl_get_param` with `PARAM_READOUT_PORT` with the `ATTR_COUNT` attribute to determine how many ports are available in your camera. Next, iterate through each choice, calling `pl_get_enum_param` with `PARAM_READOUT_PORT` and record the enumerated types returned for each valid port. Next, iterate through each of the enumerated valid

ports calling pl_set_param with PARAM_READOUT_PORT. For each valid port, build a speed table that will then be associated with that port.

Camera speed is determined by CCD readout speed. Since readout speed is determined by a number of constraints, getting consistent results depends on using the appropriate camera and hardware settings. To maintain consistency, each camera has the appropriate readout speeds and associated hardware controls loaded into the speed table. 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 4 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 CCD 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 with 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 4. Two Port Camera Example

It is the responsibility of the application program to remember variables associated with port and speed selections. For example, the camera maintains one gain value. Changing this value will change it for all port and speed choices. However, the application program may maintain gain values for each setting and then write them to the camera when the user changes the current port or speed. Read-only values, such as bit depth, may be read at time of open and saved in variables in the application or may be read each time a user selection changes.

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. If a camera has multiple speed entry numbers, you may choose to view the settings located in the speed table. To view the speed table settings, call pl_get_param with PARAM_SPDTAB_INDEX with the ATTR_MAX 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.

Frame Transfer

With a non-frame transfer CCD, the entire CCD is exposed, and the image read out before the CCD is exposed again. 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. 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 a standard frame transfer device, the storage array is usually masked and covers half the CCD. With this standard configuration, 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.

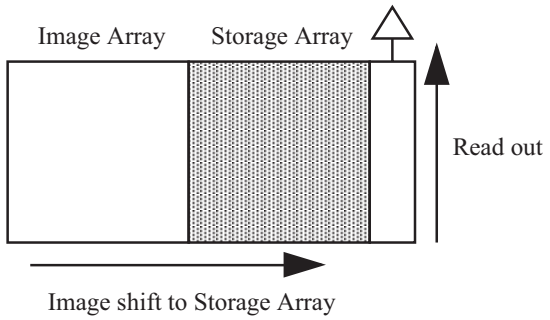
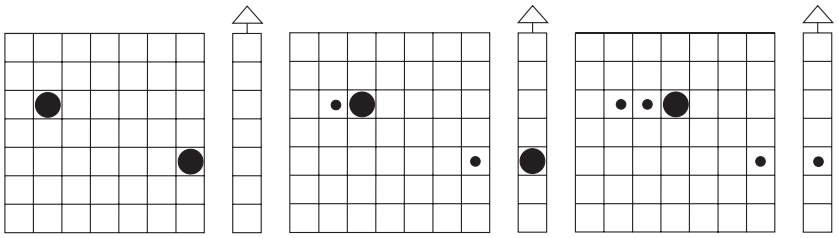


Image Smear

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.



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.

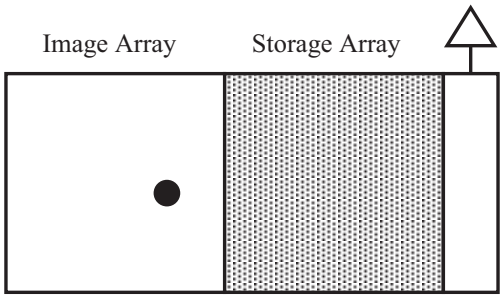


Image is exposed

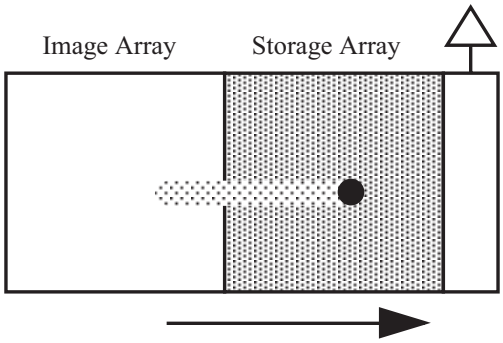
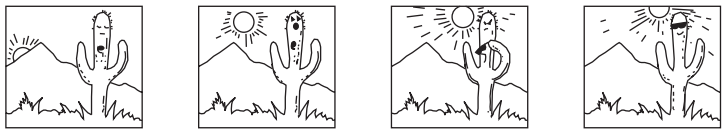


Image is shifted to storage array

Although the frame transfer time is usually only a few milliseconds, 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.



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. Since several PVCAM exposure modes ignore the setup *exposure_time*, an external trigger begins each sequence or each exposure in the sequence. In one exposure mode, calling a command between sequences sets the *exposure_time* for the next sequence.

Sequence Parameters IDs/Constants

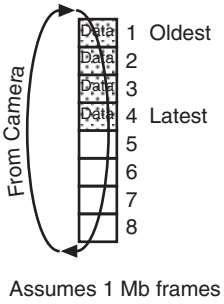
When constructing a sequence, the following three items determine how the camera behaves before reading out:

- **PARAM_CLEAR_MODE parameter id:** Determines if and when the CCD is cleared of charge.
- **BULB_MODE, FLASH_MODE, STROBED_MODE, TIMED_MODE, TRIGGER_FIRST_MODE,** or **VARIABLE_ TIMED_MODE constant :** Determines if a program command or an external trigger starts and ends the exposure/nonexposure time within a sequence.
- **PARAM_SHTR_OPEN_MODE parameter id:** Determines if and when the shutter opens.

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

Circular Buffer

Note: Because some cameras do not support circular buffer, use the parameter id *PARAM_CIRC_BUFFER* with `pl_get_param` to see if the system can perform circular buffer operations.



Assumes 1 Mb frames

Circular buffers are a special case of sequences. 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 sequence; the camera will continue to acquire frames until you decide to stop it, rather than acquiring a specified number of frames. 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 as shown in the above figure.

The image buffer used for a circular buffer is passed to `pl_exp_start_cont`. The buffer is either allocated by your application or obtained from the driver as a preallocated contiguous block of physical memory. The driver buffer pointer is retrieved using the `pl_exp_get_driver_buffer` function. Data read out of the camera is stored in the designated circular buffer until it is retrieved by the user's data processing routine, it is overwritten, or the buffer is filled. The selected circular buffer mode determines whether or not buffer data can be overwritten before being retrieved by the application.

When a circular buffer is running in `CIRC_OVERWRITE` mode, the frames in the buffer are filled as data becomes available, regardless of whether the application has retrieved the data. This allows for the fastest possible data display (on the host computer monitor) and is equivalent to the Princeton Instruments Focus mode. If all frames in the buffer are filled before the application retrieves the data, the oldest frame will be overwritten with new data. By fetching and displaying the most recently stored frame, image data display can be virtually real-time. Briefly, this mode of circular buffer is set up and runs as follows:

- `pl_exp_init_seq ()`: The camera is prepared to acquire and readout data.
- `pl_exp_setup_cont (circ_overwrite)`: The circular buffer mode is selected.
- `pl_exp_start_cont ()`: Continuous data acquisition is started.
- Frames begin arriving in the buffer.

- `pl_exp_check_cont_status ()`: The status of the buffer is checked.
- `pl_exp_get_latest_frame ()`: If there are one or more frames of data, the most recently stored frame is read out.
- Data is processed (for example, the data is displayed).
- The loop is repeated until continuous data acquisition is stopped with `pl_exp_stop_cont ()`, `pl_exp_finish_seq ()`,and `pl_exp_uninit_seq ()`.

When a circular buffer is running in `CIRC_NO_OVERWRITE` mode, the frames in the buffer are filled as data becomes available until all frames are filled. This mode allows for the fastest possible frame rate (with regard to data storage) with no skipping of frames and is equivalent to the Princeton Instruments Nframe mode. If all frames in the buffer are filled before the application retrieves the data, the latest frame will be lost because the oldest frame will not be overwritten. Therefore, the user's routine must be able to read the data out of the buffer faster than the camera can fill the buffer. Briefly, this mode of circular buffer is set up and runs as follows:

- `pl_exp_init_seq ()`: The camera is prepared to acquire and readout data.
- `pl_exp_setup_cont (circ_no_overwrite)`: The circular buffer mode is selected.
- `pl_exp_start_cont ()`: Continuous data acquisition is started.
- Frames begin arriving in the buffer.
- `pl_exp_check_cont_status ()`: The status of the buffer is checked.
- `pl_exp_get_oldest_frame ()`: If there are one or more frames of data, the oldest frame is read out.
- Data is processed (for example, stored elsewhere).
- `pl_exp_unlock_oldest_frame ()`: The oldest frame is unlocked so it becomes available for data storage.
- The loop is repeated until the buffer fills up or continuous data acquisition is stopped with `pl_exp_stop_cont ()`, `pl_exp_finish_seq ()`,and `pl_exp_uninit_seq ()`.

Refer to **Example 3: Circular Buffer** in Chapter 8 for two examples of code for circular buffer operation.

Clear Modes

Clearing removes charge from the CCD 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 CCD. Useful for performing a readout after an exposure has been aborted.
- **CLEAR_PRE_EXPOSURE:** Before each exposure, clears the CCD 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 CCD 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 Control Subsystem (CCS) parallel and serial clock drivers revert to a low power state that saves both power and heat. When CLEAR_..._POST options are used, the continuous 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.

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

Clear Modes decide when to clean the CCD arrays. However, since PI cameras always clean the CCDs at idle times, Clear Modes do not apply to PI cameras and therefore the feature is not available for PI cameras.

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	FLASH_MODE

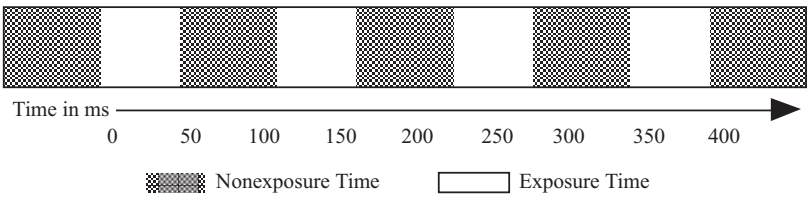
In general, the settings in `pl_exp_setup_seq` apply to each exposure within a sequence. They also apply to every sequence until the *setup* is reset. The only exceptions are in `VARIABLE_TIMED_MODE` and `BULB_MODE`. These two modes ignore the *`exposure_time`* parameter in setup, and rely on a function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

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

Exposure: 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`.

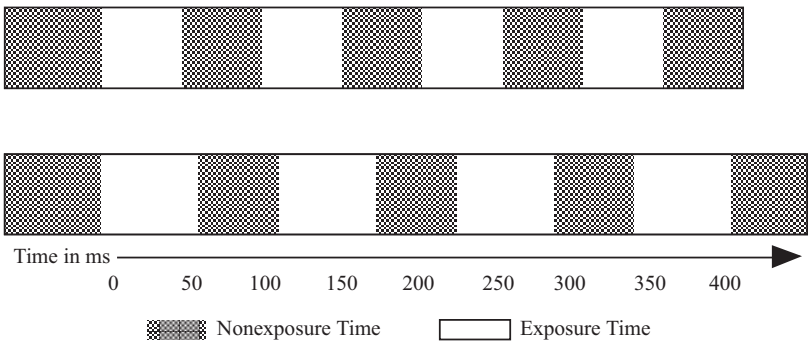


Exposure: VARIABLE_TIMED_MODE

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.

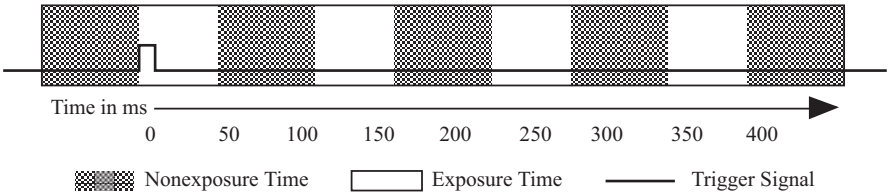
Application example: A filter wheel is used to change the filter color between sequences. The exposure time needed for the darkest filter saturates the pixels when lighter filters are used. The diagram on the next page shows two sample sequences from this example.



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.

Exposure: TRIGGER_FIRST_MODE

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



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.

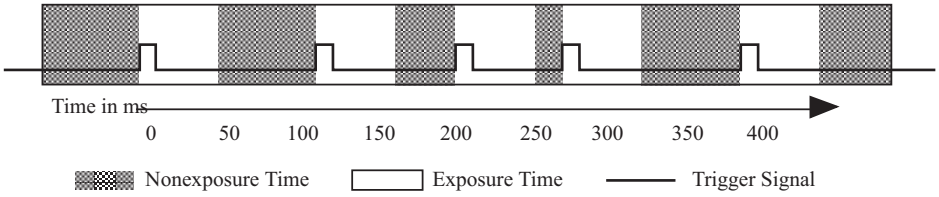
Once the outside event triggers the camera to start exposing, the sequence follows the conditions generated in *`pl_exp_setup_seq`*. Note that all exposure times are equal, and the time intervals between exposures are equal.

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 CCD will begin exposing immediately after *`pl_exp_start_seq`* is called. Once the trigger is received, the CCD 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.

Exposure: STROBED_MODE

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



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.

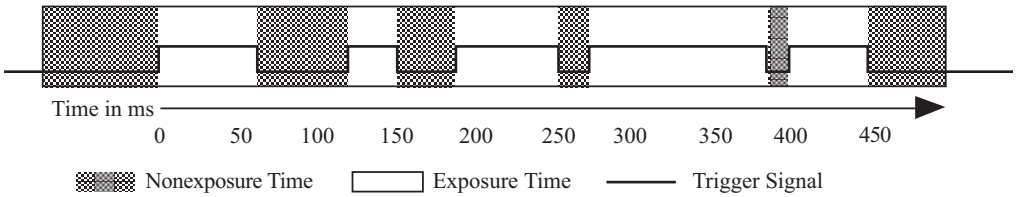
You must have an external trigger signal connected to your camera for this mode to function. If your equipment fails to send a trigger signal, you can stop the sequence by calling *pl_exp_abort*.

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 CCD will begin exposing immediately after *pl_exp_start_seq* is called. Once the trigger is received, the CCD 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.

Exposure: BULB_MODE

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



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 CCD 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 CCD exposes until receiving a false trigger signal, then reads out. After reading out, the CCD exposes again without clearing and waits for the true trigger. Once the external event causes a true trigger, the CCD continues to expose until receiving a false trigger, then reads out. In other words, the CCD will expose from the end of readout until the next false trigger.

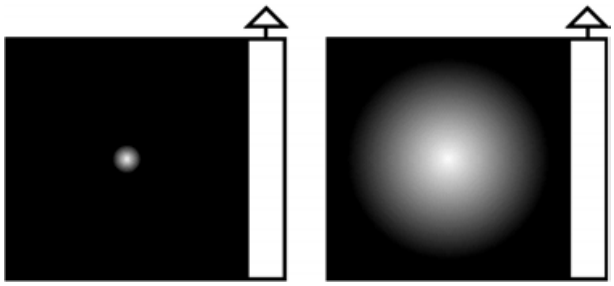
Exposure: FLASH_MODE

Some PVCAM cameras include a flash port—several outside pins with a software-controllable signal. Photometrics uses these pins to drive factory test fixturing. However, the signal can be used to drive other equipment. Aside from the signal on the pins, FLASH_MODE is identical to TIMED_MODE. Consult your camera hardware documentation to see flash port availability and electrical specifications.

Open Delay, Close Delay

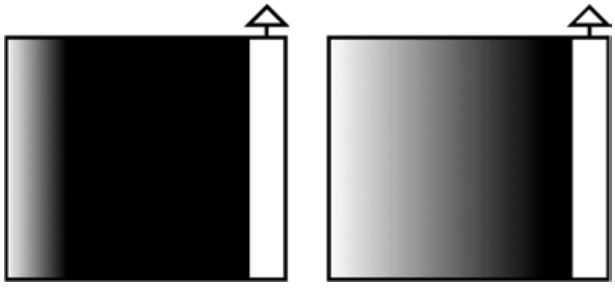
In order to ensure that the entire CCD is exposed for the specified *exposure_time*, the mechanical limitations of the shutter must be considered. Open delay (PARAM_SHTR_OPEN_DELAY) and close delay (PARAM_SHTR_CLOSE_DELAY) account for the time necessary for the shutter to open and close. Remember that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

Iris Shutter



An Iris shutter opens in an expanding circular pattern.

Barn Door Shutter



A Barn Door shutter slides across the exposure area.

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 or Princeton Instruments 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 CCD 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 CCD for `exposure_time`
I->S	Transfer image array to storage array (frame transfer)
Readout	Readout CCD (readout storage array for frame transfer)
WaitT	Wait until trigger
EXP Until notT	Expose CCD 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.	

EXPOSURE: TIMED_MODE			
Clear Mode	**Shutter Mode**	**Command Sequence**	**Notes**
CLEAR_PRE_EXPOSURE	OPEN_PRE_EXPOSURE	*ClearN, OS, EXP, CS, I->S, Readout*	Photometrics only
	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*	Photometrics only
	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*	

EXPOSURE: TRIGGER_FIRST_MODE			
Clear Mode	**Shutter Mode**	**Command Sequence**	**Notes**
CLEAR_PRE_EXPOSURE	OPEN_PRE_EXPOSURE	**EXP+WaitT**, *ClearN, OS, EXP, CS, I->S, Readout*	Photometrics only
	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*	Photometrics only
	OPEN_PRE_SEQUENCE	**OS**, **EXP+WaitT,** *EXP, I->S, Readout,* **CS**	

EXPOSURE: TRIGGER_FIRST_MODE			
Clear Mode	**Shutter Mode**	**Command Sequence**	**Notes**
	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*	

EXPOSURE: STROBED_MODE			
Clear Mode	**Shutter Mode**	**Command Sequence**	**Notes**
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**	Uses Continuous Cleans
	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*	Photometrics only
	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*	

EXPOSURE: BULB_MODE			
Clear Mode	**Shutter Mode**	**Command Sequence**	**Notes**
CLEAR_PRE_EXPOSURE	OPEN_PRE_EXPOSURE	*Clear+WaitT, OS, EXP Until notT, CS, I->S, Readout*	Photometrics only
	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*	Photometrics only
	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*	Photometrics only
	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*	

Source Code Examples

Refer to Chapter 8, pages 107-118, for code examples.

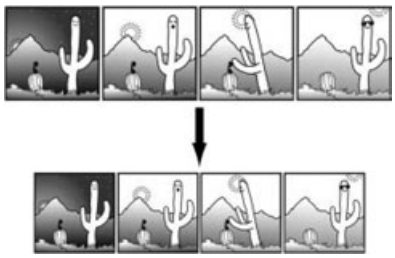
Image Buffers

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.

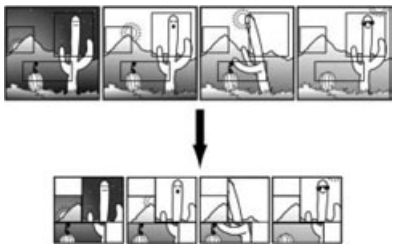
- **Full CCD:** A single exposure where the entire CCD is treated as one region and image data are collected over the full CCD. All the data are stored in a single buffer.

- **Single Exposure, Multiple Images:** A single exposure with multiple regions. The data are stored in several image arrays that are stored inside a single buffer.

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



- **Multiple Exposures, Multiple Images:** A series of exposures with multiple regions. Each exposure must have identical regions. The data are stored in several image arrays that are stored inside a single buffer.



Class 4 places the following constraints on data stored in buffers:

- All exposures in a buffer must have the same set of images (the size, position, and binning must match).
- All data in a buffer must be at the same bit depth (16-bit signed, 16- bit unsigned, 32-bit signed, and so forth.)
- All data in an image are stored in a standard C, two-dimensional array, with the second subscript varying most rapidly.

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 a 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 multiple regions are collected, the stream becomes more complex. If the regions overlap in the serial direction, the data from one region are interleaved with the data from another region.
- Use a buffer. If the data are in multiple regions, *pl_exp_finish_seq* decodes the *pixel_stream* data into the regions. Once decoded, each region can be retrieved as a simple array (see *"Data Array"*). Even though it takes extra time to decode the data and load the buffer, retrieving the data is simple.
- Defer decoding. The original call to *pl_exp_setup_seq* sets up internal structures used to decode pixel_stream into a buffer structure. However, *pl_exp_finish_seq* does not need to be called immediately. As long as the camera (and library) remain open, and *pl_exp_setup_seq* is not called with a new setup, the decoding structures remain valid. This allows a program to collect data quickly, then decode the data when more time is available. Of course, this is impossible if users must be given immediate feedback.

Chapter 3: Camera Communications (Class 0)

Introduction

The functions in this category provide a pipeline for bidirectional communications. The table below lists the current Class 0 functions, and the "Class 0 Functions" section provides detailed descriptions of each. If the Class 0 functions you are interested in are not listed below, check "Appendix B: Obsolete Functions". The Class 0 functions that have been made obsolete now have equivalent `pl_get_param` and `pl_set_param` functions. For more information about the `pl_get_param` and `pl_set_param` parameter ids, refer to "*Chapter 5:Configuration/Setup (Class 2)*", starting on page 43.

List of Available Class 0 Functions

Library

`pl_pvcam_init`

`pl_pvcam_uninit`

`pl_pvcam_get_ver`

Device Driver

`pl_ddi_get_ver`

Camera

`pl_cam_check`

`pl_cam_close`

`pl_cam_get_diags`

`pl_cam_get_name`

`pl_cam_get_total`

`pl_cam_open`

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_INFO_LENGTH`

`PARAM_DD_RETRIES`

`PARAM_DD_TIMEOUT`

`PARAM_DD_VERSION`

Class 0 Functions

PVCAM	**Class 0: Camera Communications**	**pl_cam_check(0)**

NAME

`pl_cam_check` – fails if *hcam* is not the handle of an open camera.

SYNOPSIS

```
rs_bool
   pl_cam_check(int16 hcam)
```

DESCRIPTION

This is a fast check, used internally by many other functions before they access hardware. This function checks whether the input handle, *hcam*, refers to an open camera.

RETURN VALUE

TRUE for a valid handle, FALSE for an invalid handle.

SEE ALSO

`pl_cam_open(0), pl_cam_close(0)`

NOTES

Since this function is a frequent call, it is designed to be highly efficient. It does not access hardware, it checks the internal state tables that are set by `pl_cam_open` and `pl_cam_close`.

PVCAM	**Class 0: Camera Communications**	**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 Communications** **pl_cam_get_diags(0)**

NAME

`pl_cam_get_diags` – fails and returns an error if there are any problems with the camera.

SYNOPSIS

```
rs_bool
   pl_cam_get_diags(int16 hcam)
```

DESCRIPTION

All functions that open or reset the camera perform a short set of checks and diagnostics. The error codes set in these diagnostics are stored in a table. When *hcam* is a valid camera handle, `pl_cam_get_diags` (called immediately after `pl_cam_open`) reads the table and reports any critical error condition by returning FALSE.

Both critical and noncritical subsystem error codes are set, although only critical subsystem failures return a FALSE. Critical subsystems are defined as systems that, if they fail, may prevent the camera from acquiring or reading out an image. Critical and noncritical errors are listed in `pl_error_code` .

RETURN VALUE

FALSE indicates that a critical subsystem is not working, and therefore the camera may not be able to acquire or read out an image. TRUE indicates that no error codes have been set for critical subsystems, but there may be error codes set for noncritical subsystems. Noncritical subsystem errors are considered warnings. Critical and noncritical errors are listed in `pl_error_code` .

SEE ALSO

`pl_cam_open(0)`

NOTES

This function call is designed to be fast, therefore to ensure that camera hardware is attached and functional, `pl_cam_get_diags` can be called before every exposure.

PVCAM **Class 0: Camera Communications** **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_ptr 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 cam_name[CAM_NAME_LEN];
…
pl_cam_get_total(&total_cameras );
for( I=0; I<total_cameras; I++ ) {
pl_cam_get_name(I,cam_name);
printf("Camera%d is called '%s'\n",I,cam_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_ptr 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 won't 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 Communications** **pl_cam_open(0)**

NAME

`pl_cam_open` — reserves and initializes the camera hardware.

SYNOPSIS

```
rs_bool
   pl_cam_open(char_ptr cam_name,int16_ptr 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_camera_open` completes a short set of checks and diagnostics as it attempts to establish communications 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 camera electronics unit. After a successful `pl_cam_open`, call `pl_cam_get_diags`, which reports any error conditions.

RETURN VALUE

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

SEE ALSO

`pl_cam_get_diags(0), 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 Communications** **pl_ddi_get_ver(0)**

NAME

`pl_ddi_get_ver` – returns version number of the current DDI (device driver interface)

SYNOPSIS

```
rs_bool
   pl_ddi_get_ver(uns16_ptr version)
```

DESCRIPTION

This returns a version number for the current device driver interface. The version is a formatted hexadecimal number, of the style:

	low byte	
	---	---
high byte	hi 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 parameter interpretation of any function in the DDI library. A new major release will often require a change in the PVCAM library, but, wherever possible, major releases will be 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 may also be used to flag driver versions constructed for unique customers or situations. 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

Parameter id `PARAM_DD_VERSION(0)`,`pl_pvcam_get_ver(0)`

NOTES

The DDI is the glue layer that lies between PVCAM and the actual device driver. For most users, this function and the DDI itself should be completely ignored. In some rare cases, the DDI library will be shipped separately from the PVCAM library. In those situations, this function will be necessary to ensure that PVCAM and the DDI are compatible versions.

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_ptr version)
```

DESCRIPTION

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

low byte	----------	----------
high byte	hi 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 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.

RETURN VALUE

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

SEE ALSO

`pl_ddi_get_ver(0)`, parameter id `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.

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: **`Camera Dependent`** indicates that this parameter or function is not available to all Roper Scientific cameras. If a parameter or function is marked **`Camera Dependent`**, an ATTR_AVAIL should be called to see if the camera 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 (`PARAM_DD_INFO_LENGTH`). Datatype: **`char_ptr`**
`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 communications link has failed. The camera won't 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 communications 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 communications control. If the driver sends a command to the camera and doesn't 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 communications 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 `hcam`. The version is a formatted hexadecimal number, of the style: high byte low byte ---------- ---------- ---------- hi 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

Virtually 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, you can use the following construction to report errors:

```
char msg[ERROR_MSG_LEN];
if (! pl_pvcam_do_something(. . .) ) {
pl_error_message ( pl_errror_code(),msg ) ;
printf("pvcam_do_thing 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 */
. . . code . . .
}
else {               /* function failed, print msg*/
pl_error_message(pl_error_code(),msg);
printf("pvcam_do_thing 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(pvcam_do_thing()==PV_OK){ /*func succeeded */
. . .
```

or

```
if(pvcam_do_thing()==PV_FAIL){/*func failed, print msg*/
. . .
```

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. Appendix A of this manual lists all error code messages.

List of Available Class 1 Functions

Class 1 Error Code 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. Consult Appendix A in this manual for a complete list of error codes.
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_ptr 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`*), 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.

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 Functions" section, starting on page 46. If the Class 2 functions you are interested in are not listed below, check "Obsolete Functions" in Appendix B (page 139). 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.

Camera Settings

`pl_get_param`

`pl_set_param`

`pl_get_enum_param`

`pl_enum_str_length`

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.

CCD Clearing

`PARAM_ANTI_BLOOMING`

`PARAM_CLEAR_CYCLES`

`PARAM_CLEAR_MODE`

`PARAM_CONT_CLEARS`

`PARAM_MIN_BLOCK`

`PARAM_NUM_MIN_BLOCK`

`PARAM_NUM_OF_STRIPS_PER_CLR`

`PARAM_SKIP_AT_ONCE_BLK`

Temperature Control

`PARAM_COOLING_MODE`

`PARAM_TEMP`

`PARAM_TEMP_SETPOINT`

CCD 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`

Gain

PARAM_GAIN_INDEX
PARAM_GAIN_MULT_ENABLE
PARAM_GAIN_MULT_FACTOR
PARAM_INTENSIFIER_GAIN
PARAM_PREAMP_DELAY
PARAM_PREAMP_OFF_CONTROL

Shutter

PARAM_EXPOSURE_MODE
PARAM_PREFLASH
PARAM_SHTR_CLOSE_DELAY
PARAM_SHTR_GATE_MODE
PARAM_SHTR_OPEN_DELAY
PARAM_SHTR_OPEN_MODE
PARAM_SHTR_STATUS

I/O

PARAM_IO_ADDR
PARAM_IO_BITDEPTH
PARAM_IO_DIRECTION
PARAM_IO_STATE
PARAM_IO_TYPE
PARAM_LOGIC_OUTPUT

CCD Readout

PARAM_CCS_STATUS
PARAM_EDGE_TRIGGER
PARAM_PMODE
PARAM_READOUT_PORT
PARAM_READOUT_TIME

ADC Attributes

PARAM_ADC_OFFSET
PARAM_BIT_DEPTH
PARAM_SPDTAB_INDEX

Capabilities

PARAM_ACCUM_CAPABLE
PARAM_FRAME_CAPABLE
PARAM_MPP_CAPABLE

Other

PARAM_CAM_FW_VERSION
PARAM_CHIP_NAME
PARAM_CONTROLLER_ALIVE
PARAM_HEAD_SER_NUM_ALPHA
PARAM_PCI_FW_VERSION
PARAM_SERIAL_NUM

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_ptr 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_ptr`* 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
```

ATTR_ACCESS

Reports if the `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_ERROR            ACC_EXIST_CHECK_ONLY
ACC_READ_ONLY        ACC_WRITE_ONLY
ACC_READ_WRITE
```

The data type for this attribute is `TYPE_UNS16`.

Note: This is an exception where an enum type is not treated as an unsigned 32-bit integral value

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/or 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 plus a 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_FLT64, TYPE_BOOLEAN.

The data type for ATTR_COUNT is TYPE_UNS32.

ATTR_CURRENT

Current value. The data type for this attribute is defined by ATTR_TYPE.

ATTR_DEFAULT

Default value. 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 data type for this attribute is defined by ATTR_TYPE.

ATTR_MAX

Maximum value. The data type for this attribute is defined by ATTR_TYPE.

ATTR_MIN

Minimum value. The data type for this attribute is defined by ATTR_TYPE.

ATTR_TYPE

Data type of parameter (int16, float 64, enumerated, etc.). The data type for this is TYPE_UNS16. 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.

Data type used by pl_get_param with attribute type (ATTR_TYPE).

TYPE_CHAR_PTR	string
TYPE_INT8	
TYPE_UNS8	
TYPE_INT16	
TYPE_UNS16	
TYPE_INT32	
TYPE_UNS32	
TYPE_FLT64	
TYPE_ENUM	treat as uns32
TYPE_BOOLEAN	
TYPE_VOID_PTR	ptr to void
TYPE_VOID_PTR_PTR	ptr to a void ptr

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

SEE ALSO pl_set_param and pl_get_enum_param

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, with the *ATTR_TYPE* attribute.

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_ptr
                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.

RETURN VALUE

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

SEE ALSO

`pl_get_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_ptr value,char_ptr
                      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, pl_set_param, and pl_enum_str_length`

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. If the readout speed is currently set to 5MHz, a call to *`pl_get_param`* returns a value of 1. A call to *`pl_get_enum_param`* for the readout speed parameter at `index` 1 returns the enumerated type *`5MHz`* (which may or may not be equal to 1). 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_ptr 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`

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).

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: **`Camera Dependent`** indicates that this parameter or function is not available to all Roper Scientific cameras. If a parameter or function is marked **`Camera Dependent`**, an ATTR_AVAIL should be called to see if the camera supports it.

Class 2 Parameter ID	Description
`PARAM_ACCUM_CAPABLE` **`Camera Dependent`**	Returns TRUE 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_ANTI_BLOOMING`
 	
`Camera Dependent`
	Does not apply to all cameras. Enables or disables anti-blooming. Possible values are:
	
	`ANTI_BLOOM_NOTUSED`
	`ANTI_BLOOM_INACTIVE`
	`ANTI_BLOOM_ACTIVE`
	
	Note: The *`ATTR_AVAIL`* attribute can be used to tell the application if this feature is supported.
	
	Datatype: **`enum`**

`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`**

Class 2 Parameter ID	Description
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_CCS_STATUS Camera Dependent	This holds sixteen bits of status data from the Camera Control Subsystem (CCS). Only the lowest 2 bits are currently implemented. These 2 bits give the status of the CCS: Value CCS State 0 idle 1 initializing 2 running 3 continuously clearing A running state occurs any time the CCS is in the process of performing a camera operation (including opening or closing the shutter, exposing, clearing the CCD before a sequence or exposure, parallel or serial shifting, and readout/digitization). After the CCD has finished reading out, the setup determines if the CCS goes to idle or enters continuous clearing mode. Datatype: **int16**
PARAM_CHIP_NAME	The name of the CCD. 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_ptr**
PARAM_CLEAR_CYCLES	This is the number of times the CCD 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 CLEAR_PRE_EXPOSURE CLEAR_PRE_SEQUENCE CLEAR_POST_SEQUENCE CLEAR_PRE_POST_SEQUENCE CLEAR_PRE_EXPOSURE_POST_SEQ CLEAR_NEVER Don't ever clear the CCD. 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**
PARAM_COLOR_MODE **Camera Dependent**	The color mode of the CCD. See enum below for possible values. COLOR_NONE=0 COLOR_RGGB=2 COLOR_NONE = monochrome COLOR_RGGB = RGGB color mask Datatype: **enum**

Class 2 Parameter ID	Description
PARAM_CONTROLLER_ALIVE	This is a general parameter that checks to see if the controller is on and running. Returns a TRUE if the controller is "alive". Datatype: **rs_bool**
PARAM_COOLING_MODE	This is the type of cooling used by the current camera. See enum below for possible values. NORMAL_COOL CRYO_COOL 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**
PARAM_EDGE_TRIGGER Camera Dependent	Does not apply to all cameras. Edge Trigger defines whether the external sync trigger is positive or negative edge active. This is for the ST133 family (1 and 5 MHz) and PentaMAX V5.0. Possible values: EDGE_TRIG_POS=2 EDGE_TRIG_NEG ***Note:*** The *ATTR_AVAIL* attribute can be used to tell the application if this feature is supported. Datatype: **enum**
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 FLASH_MODE VARIABLE_TIMED_MODE ***Note:*** See *"Exposure Mode Constants"* on page 65 for information about these modes. Datatype: **enum**
PARAM_FRAME_CAPABLE Camera Dependent	If true, this camera can run in frame transfer mode (set through PARAM_PMODE). Datatype: **rs_bool**

Class 2 Parameter ID	Description
PARAM_FWELL_CAPACITY **Camera Dependent**	Gets the full-well capacity of this CCD, measured in electrons. Datatype: **uns32**
PARAM_GAIN_INDEX	Gain setting for the current speed choice. The valid range for a gain setting is 1 through *PARAM_GAIN_INDEX* with *ATTR_MAX*, where the max gain may be as high as 16. Values outside this range will be ignored. Note that gain settings 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_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 1 through *PARAM_GAIN_MULT_FACTOR* with *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_ptr**
PARAM_INTENSIFIER_GAIN **Camera Dependent**	Does not apply to all cameras. Intensifier gain has a range of 0-255. ***Note:*** The *ATTR_AVAIL* attribute can be used to tell the application if this feature is supported. Datatype: **int16**
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* attribute with the *PARAM_IO_ADDR* parameter ID. 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**
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, with the *PARAM_IO_ADDR* parameter ID. 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**

Class 2 Parameter ID	Description
PARAM_LOGIC_OUTPUT **Camera Dependent**	Kinds of output are:OUTPUT_NOT_SCAN OUTPUT_SHUTTER OUTPUT_NOT_RDY OUTPUT_LOGIC0 OUTPUT_CLEARING, OUTPUT_NOT_FT_IMAGE_SHIFT OUTPUT_RESERVED OUTPUT_LOGIC1 Datatype: **enum**
PARAM_MIN_BLOCK **Camera Dependent**	This is the CCD skip parameter for the amount to group on the shift register and throw away. Datatype: **int16**
PARAM_MPP_CAPABLE **Camera Dependent**	Indicates whether this CCD runs in MPP mode. The actual value returned is equal to one of four constants: Possible values. MPP_UNKNOWN MPP_ALWAYS_OFF MPP_ALWAYS_ON MPP_SELECTABLE Datatype: **enum**
PARAM_NUM_MIN_BLOCK **Camera Dependent**	This is the CCD skip parameter for the number of minimum block groups to use before valid data. Datatype: **int16**
PARAM_NUM_OF_STRIPS_PER_CLR **Camera Dependent**	This is the CCD skip parameter for the number of strips per clear. Used to define how many clears to use for continuous clears and used with clears to define the clear area at the beginning of an experiment. Datatype: **int16**
PARAM_PAR_SIZE	This is the parallel size of the CCD, 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**

Class 2 Parameter ID	Description
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**
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**
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**

Class 2 Parameter ID	Description
PARAM_PREAMP_DELAY Camera Dependent	This is the number of milliseconds required for the CCD 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_PREFLASH Camera Dependent OBSOLETE	This is the number of milliseconds needed to illuminate the CCD using the flash diode ring before an exposure, dark, or bias. Datatype: **uns16**
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 CCD 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**

PARAM_READOUT_PORT
 	
Camera Dependent
	CCD 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`
	`READOUT_PORT_LOW_NOISE`
	`READOUT_PORT_HIGH_CAP`
	
	Use *PARAM_READOUT_PORT* with *ATTR_COUNT* to read out the number of ports on the system.
	
	Datatype: **enum**

PARAM_READOUT_TIME Camera Dependent	Readout time of current ROI, in ms. Datatype: **flt64**
PARAM_SER_SIZE	Defines the serial-dimension of the active area of the CCD chip. Datatype: **uns16**

Class 2 Parameter ID	Description
PARAM_SERIAL_NUM **Camera Dependent**	This is the serial number of the camera head (not the electronics unit). Datatype: **uns16**
PARAM_SHTR_GATE_MODE **Camera Dependent**	Does not apply to all cameras. INTENSIFIER_SAFE INTENSIFIER_GATING INTENSIFIER_SHUTTER ***Note:*** The *ATTR_AVAIL* attribute can be used to tell the application if this feature is supported. Datatype: **enum**
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. Valid inputs are any number in the range 0 to 65535 milliseconds. 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. Valid inputs are any number in the range 0 to 65535 milliseconds. 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 OPEN_PRE_EXPOSURE OPEN_PRE_SEQUENCE OPEN_PRE_TRIGGER OPEN_NO_CHANGE 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*" in the PVCAM introduction. Datatype: **enum**
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**

Class 2 Parameter ID	Description
PARAM_SKIP_AT_ONCE_BLK **Camera Dependent**	Sets the size of rows skipped at once for PI brand cameras. This is one method to control discard of unwanted areas (outside of ROIs). Datatype: **int32**
PARAM_SPDTAB_INDEX	This selects the CCD readout speed from a table of available choices. Entries are 0-based, so the range of possible values is 0 to max_entries-1; max_entries can be determined using *PARAM_SPDTAB_INDEX* with the *ATTR_MAX* attribute. This setting relates to other speed table values, including *PARAM_BIT_DEPTH*, *PARAM_PIX_TIME*, *PARAM_READOUT_PORT* and *PARAM_GAIN_INDEX*. 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, the summing well exists. Datatype: **rs_bool**
PARAM_TEMP **Camera Dependent**	Returns the current measured temperature of the CCD 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 CCD temperature in hundredths of degrees Celsius (minus 35 °C is represented as -3500). The hardware attempts to heat or cool the CCD 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**

Chapter 6: Data Acquisition (Class 3)

Introduction

Class 3 defines CCD 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. If the Class 3 functions you are interested in are not listed below, check "Appendix B: Obsolete Functions" section on page 149. 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_setup_seq
pl_exp_check_cont_status        pl_exp_start_cont
pl_exp_check_status             pl_exp_start_seq
pl_exp_finish_seq               pl_exp_stop_cont
pl_exp_get_driver_buffer        pl_exp_uninit_seq
pl_exp_get_latest_frame         pl_exp_unlock_oldest_frame
pl_exp_get_oldest_frame         pl_exp_unravel
pl_exp_init_seq                 pl_io_clear_script_control
pl_exp_setup_cont               pl_io_script_control
```

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_CIRC_BUFFER               PARAM_HW_AUTOSTOP
PARAM_EXP_MIN_TIME
```

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`

Decode the multi-region pixel stream into images in a buffer by calling pl_exp_finish_seq (optional)

New Structures

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

```
typedef struct {
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,
*rgn_ptr;
```

Exposure Mode Constants

The six constants below define the exposure mode:

TIMED_MODE	STROBED_MODE
VARIABLE_TIMED_MODE	BULB_MODE
TRIGGER_FIRST_MODE	FLASH_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*.
FLASH_MODE	Activates the flash circuit on the trigger port. Used for factory testing.

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_ptr 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. If the readout has finished, this function decodes the pixel stream pointed to by *`pixel_stream`* and places it into the standard image buffer *`hbuf`*. *`hbuf`* must be able to hold the number of exposures specified. Any errors leave the pixel stream intact, so a further attempt can be made to decode the data if an error can be corrected. Null is an acceptable value for *`hbuf`* .
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 is only necessary when multiple sequences or multiple regions are defined, or when information such as image data, time, and size needs to be stored with the *`pixel_stream`* data. This function and the Class 4 functions are not required for a single region, single exposure; the pixel stream is the raw data for that image. The final format of the image buffer will be the same as that of the readout. Individual exposures may be appended together to create a single, multiple exposure image buffer. See *"Chapter 7: Buffer Manipulation (Class 4)"* for more information on the use of buffers.

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

NAME

`pl_exp_get_driver_buffer -` retrieves a pointer to a preallocated image buffer.

SYNOPSIS

```
rs_bool
   pl_exp_get_driver_buffer(int16 hcam, void_ptr_ptr
                            pixel_stream, uns32_ptr byte_cnt)
```

DESCRIPTION

This function returns a pointer in *`pixel_stream`* to the image buffer that has been previously allocated by a camera device driver. A pointer to the size of the buffer is returned in *`byte_cnt`*.

This function is used to retrieve a pointer to the buffer that may be allocated by the driver. If the driver did not allocate an image buffer, a value of NULL will be returned for *`pixel_stream`*, and a value of zero will be returned for *`byte_cnt`*.

RETURN VALUE

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

SEE ALSO

NOTES

This image buffer is a block of contiguous physical memory that is set aside for data storage when the operating system is started. This preallocation of memory ensures that you will have a contiguous memory block to store data when you are performing continuous data acquisition. A contiguous memory block may be necessary in some situations in which the host computer is heavily loaded with tasks. When the buffer is used for Circular Buffer operation, the number of frames that can be held in the buffer depends on the size of the buffer and the image size.

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_ptr_ptr 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_get_driver_buffer(3)`, `pl_exp_setup_cont(3)`, `pl_exp_start_cont(3)`, `pl_exp_check_cont_status(3)`, and `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_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_ptr_ptr 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_get_driver_buffer(3)`, `pl_exp_setup_cont(3)`, `pl_exp_start_cont(3)`, `pl_exp_check_cont_status(3)`, `pl_exp_unlock_oldest_frame(3)`, and `pl_exp_stop_cont(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_init_seq(3)**

NAME

`pl_exp_init_seq` – initializes the data collection functions.

SYNOPSIS

```
rs_bool
   pl_exp_init_seq(void)
```

DESCRIPTION

This function prepares the portion of the library associated with the exposure control for operation and must be called before any other Class 3 function.

RETURN VALUE

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

SEE ALSO

`pl_pvcam_init(0)`,`pl_pvcam_uninit(0)`,`pl_exp_uninit_seq(3)`

NOTES

You must explicitly call this function after calling `pl_pvcam_init` and before calling any other `pl_exp_` function.

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,rgn_const_ptr
                     rgn_array,int16 mode,uns32
                     exposure_time,uns32_ptr 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 exposure mode.

`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_get_driver_buffer(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)`, and `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 either allocated by your application or obtained from the driver as a preallocated block of memory, using the `pl_exp_get_driver_buffer` function.

Refer to **Example 3: Circular Buffer** in "Code Examples" for two examples of code for circular buffer operation.

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,rgn_const_ptr rgn_array,int16
                    mode,uns32 exposure_time,uns32_ptr
                    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 exposure mode, *`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 CCD 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_ptr pixel_stream,uns32
                     size)
```

DESCRIPTION

This function will initiate a continuous readout from the camera into a `circ`ular 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_get_driver_buffer(3)`, `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)`, and `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 with a frame size of 250 is OK, but a buffer size of 900 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_ptr 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 CCD 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 (see `pl_get_param` parameter id `PARAM_CCS_STATUS`).

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

Class 3 data collection functions, `pl_get_param` parameter id `PARAM_CCS_STATUS(2)`

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 CCD (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_get_driver_buffer(3), 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), and 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_ptr status,
                       uns32_ptr 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.)
`ACQUISITION_IN_PROGRESS`	Indicates that a Princeton Instruments brand camera is either exposing (`EXPOSURE_IN_PROGRESS`) or reading out the data (`READOUT_IN_PROGRESS`); these individual states are not available with this camera brand.

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. In the last case, `pl_error_code` will be set to one of the following values:

`C0_EXP_FIFO_OVERFLOW`	`C0_EXP_XFER_ERR`
`C0_EXP_NO_ACK`	`C0_EXP_MISSING_DATA`
`C0_EXP_EXTRA_DATA`	`DDI_UNKNOWN_IM_STATUS`

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_ptr
                            status,uns32_ptr byte_cnt,
                            uns32_ptr 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  EXPOSURE_IN_PROGRESS,
READOUT_IN_PROGRESS  ACQUISITION_IN_PROGRESS,
READOUT_COMPLETE  READOUT_FAILED.
```

byte_cnt points to the number of bytes currently stored in the buffer. *buffer_cnt* points to the number of times the buffer has been filled.

`ACQUISITION_IN_PROGRESS` indicates that a Princeton Instruments brand camera is either exposing (`EXPOSURE_IN_PROGRESS`) or reading out the data (`READOUT_IN_PROGRESS`); the two individual states are not available for a Princeton Instruments brand camera.

The total number of bytes transferred can be determined as follows:

```
total_bytes = (buffer_cnt * buffer_size) + byte_cnt
```

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)`, and `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_uninit_seq(3)**

NAME

`pl_exp_uninit_seq` – uninitializes the data collection functions.

SYNOPSIS

```
rs_bool
   pl_exp_uninit_seq(void)
```

DESCRIPTION

This function undoes the preparations done by `pl_exp_init_seq`. After executing this function, acquisition cannot take place.

RETURN VALUE

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

SEE ALSO

`pl_pvcam_init(0)`,`pl_pvcam_uninit(0)`,`pl_exp_init_seq(3)`

NOTES

You must explicitly call this function before calling `pl_pvcam_uninit`.

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_get_driver_buffer(3), 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),` and `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_exp_unravel(3)**

NAME

`pl_exp_unravel -` unravels a single or multiple ROIs from the current data stream.

SYNOPSIS

```
rs_bool
   pl_exp_unravel(int16 hcam, uns16 exposure, void_ptr
                  pixel_stream, uns16 rgn_total,
                  rgn_const_ptr rgn_array, uns16_ptr *
                  array_list)
```

DESCRIPTION

This function will separate a single or multiple Region of Interest from the data stream.

int16 hcam is the handle to open camera

uns16 exposure is the index into the buffer pointing to a specific frame.

void_ptr pixel_stream is the pointer to the buffer containing the frame data.

uns16 rgn_total: is the total number of ROIs in the frame.

rgn_const_ptr: is the pointer to the array of region(s).

uns16_ptr array_list* is the pointer to the array of buffers that the function unravels the data into.

RETURN VALUE

TRUE for success, FALSE for a failure.

SEE ALSO

`pl_exp_setup_cont(3)`, `pl_exp_start_cont(3)`, `pl_exp_check_cont_status(0)`, `pl_exp_get_oldest frame(3)`, and `pl_exp_unlock_oldest_frame(3)`

NOTES

Code example using circular buffer:

```
rgn_type r[]={{0,19,1,0,9,1},{40,59,1,20,24,1}};
uns32 nBytes;
uns16 numFrames = 5;
if(!pl_exp_setup_cont(hCam, 2, r, TIMED_MODE, 500,
   &nBytes,CIRC_NO_OVERWRITE)) return -1;
// Allocating 3x the frame size for a decent circular
buffer
nBytes *= 3;
uns16  *pStream [( nBytes / sizeof(uns16)];
uns16  *pRoi1 [ 200 ]; // size of rgn1{0,19,1,0,9,1}
uns16  *pRoi2 [ 100 ]; // size of
rgn2{40,59,1,20,24,1}
uns16  *pUnraveledData[]={pRoi1,pRoi2};
if(!pl_exp_start_cont(hCam, pStream, nBytes))
return -1;
int16 eStatus;
do
{
```

```
// Passing 'nBytes' twice as filler since the value of
nBytes isn't needed.
     if(!pl_exp_check_cont_status(hCam, &eStatus,
         &nBytes, &nBytes))
  return -1;
 if(eStatus == READOUT_COMPLETE)
 {
  uns16 *pFrame;
  if(!pl_exp_get_oldest_frame(hCam,
              reinterpret_cast<void **>
              (&pFrame)))return -1;
  if(!pl_exp_unravel(hCam, 0, pFrame, 2, r,
              pUnraveledData)) return -1;
  if(!pl_exp_unlock_oldest_frame(hCam)) return -1;
  --numFrames;
 }
}
while( numFrames );
```

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

Class 3 Parameter IDs

Note: `Camera Dependent` indicates that this parameter or function is not available to all Roper Scientific cameras. If a parameter or function is marked `Camera Dependent`, an ATTR_AVAIL should be called to see if the camera 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**

PARAM_CIRC_BUFFER	Tests to see if the hardware/software can perform circular buffer. When a TRUE is returned, the circular buffer function can be used. Datatype: **rs_bool**
PARAM_EXP_MIN_TIME	Gets the minimum effective exposure time that can be set for the camera. For example, the exposure time may be limited by the required overhead for shifting the data through the array. This minimum time will be a floating point value, in seconds. Note that the minimum exposure time returned by this function will be greater than zero; any camera can provide a minimum exposure time of zero. Datatype: **flt64**
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. Possible values are : `EXP_RES_ONE_MILLISEC` `EXP_RES_ONE_MICROSEC`. Datatype: **enum**

Class 3 Parameter ID	Description
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 with the *PARAM_EXP_RES_INDEX* parameter. Datatype: **uns16**
PARAM_EXP_TIME	This is used to examine and change the exposure time in VARIABLE_TIMED_MODE. Datatype: **uns16**
PARAM_HW_AUTOSTOP Camera Dependent	Sets the number of frames to acquire synchronously into a register for PI brand cameras. At the data acquisition, the hardware counts the number of frames transferred, then stops the acquisition when it reaches the count set with PARAM_HW_AUTOSTOP. The maximum number the application can set is 254. If an application needs more than 254, it must set it to ZERO, i.e., a continuous acquisition and issue the STOP command manually to halt the acquisition. For focusing mode, an application should set this parameter to ZERO. Datatype: **int16**

This page intentionally left blank.

Chapter 7: Buffer Manipulation (Class 4)

Introduction

Class 4 places the following constraints on data stored in buffers:

- All exposures in a buffer must have the same set of images (the size, position, and binning must match).
- All data in a buffer must be at the same bit depth (16-bit signed, 16-bit unsigned, 32-bit signed, etc.).
- All data in an image is stored in a standard C two-dimensional array, with the second subscript varying most rapidly.

In addition to the image data itself, a significant amount of auxiliary information is recorded in a buffer. There is no facility for setting the information (besides setting the date), but you can read the information with the *get_* functions in the Buffer Information category below.

List of Available Class 4 Functions

The buffer manipulation functions are divided into three categories: Buffer Information, Allocation and Saving, and Initialization.

Buffer Information

pl_buf_get_bits
pl_buf_get_exp_date
pl_buf_get_exp_time
pl_buf_get_exp_total
pl_buf_get_img_bin
pl_buf_get_img_handle
pl_buf_get_img_ofs
pl_buf_get_img_ptr
pl_buf_get_img_size
pl_buf_get_img_total
pl_buf_get_size
pl_buf_set_exp_date

Allocation and Saving

pl_buf_alloc
pl_buf_free

Initialization

pl_buf_init
pl_buf_uninit

New Constants

Several new constants are used to indicate the bit depth of image data. Since these are constants, not system-dependent types, they are defined in pvcam.h:

`PRECISION_INT8`	This is 8-bit, signed data, in the range -128 to 127.
`PRECISION_UNS8`	This is 8-bit, unsigned data, in the range 0 to 255.
`PRECISION_INT16`	This is 16-bit, signed data, in the range -32768 to 32767.
`PRECISION_UNS16`	This is 16-bit, unsigned data, in the range 0 to 65535.
`PRECISION_INT32`	This is 32-bit, signed data, in the range -2,147,483,648 to +2,147,483,647.
`PRECISION_UNS32`	This is 32-bit, unsigned data, in the range 0 to 4 GB-1.

Image Handles and Pointers

An image handle specifies the image. Like camera handles (*`hcam`*) and buffer handles (*`hbuf`*), an image handle (*`himg`*) is an integer that is an index into a table kept by the PVCAM library. The image handle, usually having the variable name *`himg`*, specifies the source buffer, exposure number, and image number. If that buffer is freed, the handle becomes invalid, causing the table entry to clear and be freed for new assignment. The handle for any image can be obtained through `pl_buf_get_img_handle`.

A slightly different item is an image pointer. Internally, each image is organized as a flat two-dimensional array with the following organization:

```
i0,j0 i0,j1 i0,j2 i0,j3 .... i0,j(j_size-1)i1,j0 i1, j1 i1,
j2 .... i(i_size-1),j(j_size-1)
```

In other words, this is a standard C two-dimensional array, with the second subscript varying most rapidly. Immediately after creation, the j dimension is equivalent to the serial direction of the CCD, while the i dimension is equivalent to the parallel direction. As processing may quickly blur this relationship, the image buffers are presented with the more neutral i, j scheme instead of the concepts serial and parallel.

The `pl_buf_get_img_ptr` function returns the address of element 0 of this array. Since alignment depends on both the current operating system and the current bit depth, a void pointer is returned. The user is responsible for the details of alignment and array organization.

In addition, no information is given concerning the data that follows the last element. This data may be a following image, a following exposure, buffer header information, or operating system memory. In other words, as in normal C memory usage, you are not prevented from writing past the end of affected memory, but this may have unpredictable consequences.

Class 4 Functions

PVCAM **Class 4: Buffer Manipulation** **pl_buf_alloc(4)**

NAME

`pl_buf_alloc` — allocates a buffer based on the current exposure setup.

SYNOPSIS

```
rs_bool
   pl_buf_alloc(int16_ptr hbuf,int16 exp_total,int16
                bit_depth,int16 rgn_total,rgn_const_ptr
                rgn_array)
```

DESCRIPTION

This routine examines the region definition array pointed to by *rgn_array* to determine the memory required to store the images from a single exposure. This routine takes this array as a template for each exposure, and then allows the user to specify the number of exposures in *exp_total* and the amount of storage per pixel in *bit_depth*. *bit_depth* must use one of the following constants:

`PRECISION_INT8, PRECISION_UNS8, PRECISION_INT16, PRECISION_UNS16, PRECISION_INT32,` and `PRECISION_UNS32`.

With this information, enough memory is allocated to hold the data from the set of exposures. A handle to this buffer is passed back in *hbuf*.

RETURN VALUE

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

SEE ALSO

`pl_buf_free(4),pl_buf_get_bits(4)`

NOTES

When using this function, the definitions must match the region definitions in the exposure setup, otherwise memory may be corrupted. If the region definition changes, the buffer must be freed, and another buffer is allocated. Note that *bit_depth* must be equal to one of the `PRECISION_` constants as described at the start of this section.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_free(4)**

NAME

`pl_buf_free` – frees the memory and handle used by a buffer.

SYNOPSIS

```
rs_bool
   pl_buf_free(int16 hbuf)
```

DESCRIPTION

This routine frees the memory associated with *`hbuf`*. The memory is released and the buffer handle becomes invalid.

RETURN VALUE

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

SEE ALSO

`pl_buf_copy(4),pl_buf_load(4)`

NOTES

Although the memory is freed, garbage collection is another issue. Many small buffers may fragment memory, even if most of them are later freed. Garbage collection is a system-dependent operation.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_bits(4)**

NAME

`pl_buf_get_bits` – returns the buffer precision.

SYNOPSIS

```
rs_bool
   pl_buf_get_bits(int16 hbuf,int16_ptr bit_depth)
```

DESCRIPTION

Every exposure and every image in a buffer must be at the same bit depth. This function returns the depth for the images in *hbuf*. The parameter *bit_depth* will be set to one of the following constants (defined in *pvcam.h*):

`PRECISION_INT16`

`PRECISION_UNS16`

`PRECISION_INT32`

Notice that these use the standard PVCAM types (*int16, uns16, int32*) capitalized with the word `PRECISION_` added.

RETURN VALUE

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

SEE ALSO

`pl_buf_change_bits(4)`

NOTES

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_exp_date(4)**

NAME

`pl_buf_get_exp_date` – returns when a picture was taken.

SYNOPSIS

```
rs_bool
   pl_buf_get_exp_date(int16 hbuf,int16 exp_num,int16_ptr
                       year,uns8_ptr month,uns8_ptr
                       day,uns8_ptr hour,uns8_ptr min,
                       uns8_ptr sec,uns16_ptr msec)
```

DESCRIPTION

This returns the time when the specified exposure was decoded. The format is:

Year	current year (i.e. , 2002)
month	1-12 (January through December)
day	1-31 (day number in the current month)
hour	0-23 (24-hour format)
min	0-59
sec	0-59
msec	0-999 milliseconds

To get a time for the entire buffer, it is usually adequate to examine the time for *`exp_num`* `0`, but, depending on the sequence and timing parameters, successive exposures may be taken hours or even days later. To examine the exact exposure time for any successive exposure in the sequence, merely specify a different *`exp_num`*. The exposure end time may be obtained by adding the exposure duration, obtained from the `pl_buf_get_exp_time` function.

RETURN VALUE

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

SEE ALSO

`pl_buf_set_exp_date(4)`,`pl_buf_get_exp_time(4)`, `pl_do_exp(3)`

KNOWN BUGS

If the host computer clock is inaccurate, the time recorded will also be inaccurate. Although most clocks are not accurate to a millisecond, the recorded time should help differentiate between the exposures in a fast sequence. Impossible time values (all 0, for example) usually indicate that the start time was never set.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_exp_time(4)**

NAME

pl_buf_get_exp_time — returns exposure duration.

SYNOPSIS

```
rs_bool
   pl_buf_get_exp_time(int16 hbuf,int16 exp_num,uns32_ptr
                       exp_msec)
```

DESCRIPTION

This returns the exposure duration in milliseconds, in the parameter *exp_msec*. In most cases, the timing for the first exposure is identical for all exposures. In BULB_MODE, however, the exposure time is unknown and can be adjusted for every exposure. This allows the actual time to be read for the individual exposures, by specifying the exposure number in *exp_num* (which is zero-indexed).

RETURN VALUE

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

SEE ALSO

pl_buf_get_exp_date(4)

NOTES

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_exp_total(4)**

NAME

`pl_buf_get_exp_total` – returns number of exposures in the buffer.

SYNOPSIS

```
rs_bool
   pl_buf_get_exp_total(int16 hbuf,int16_ptr total_exps)
```

DESCRIPTION

This returns the number of exposures in the specified buffer, inside the variable *`totl_exps`*. When referring to exposures by number, the first exposure will be exposure number 0 (in typical C fashion). Therefore, the highest allowable exposure number is *`totl_exps`*-1.

RETURN VALUE

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

SEE ALSO

`pl_buf_get_img_total(4),pl_buf_append_exp(4)`

NOTES

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_img_bin(4)**

NAME

`pl_buf_get_img_bin` – returns binning factors for the image.

SYNOPSIS

```
rs_bool
   pl_buf_get_img_bin(int16 himg,int16_ptr ibin,int16_ptr
                      jbin)
```

DESCRIPTION

Default binning is *ibin*=1, *jbin*=1 (no binning, 1 CCD pixel becomes one image pixel). Binning is set when a buffer is created. This function reports on the binning that was used during acquisition, for the image indicated by *himg*.

RETURN VALUE

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

SEE ALSO

`pl_buf_get_img_size(4)`

NOTES

It is assumed that the binning is identical for each exposure. In other words, each image in an exposure has its own binning values, but this information is only entered once; it is not repeated for every exposure in the buffer. The value for exposure 0 will always be identical to the value for every other exposure.

This is usually a safe assumption, but a user might use functions like `pl_buf_get_img_ptr` to insert images that fit, but were taken under radically different conditions, including different binning. In such a case, the value reported for binning will not change, but it will no longer be accurate. It then becomes the user's responsibility to keep track of the binning.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_img_handle(4)**

NAME

`pl_buf_get_img_handle` – obtains handle that refers to a single image in buffer.

SYNOPSIS

```
rs_bool
   pl_buf_get_img_handle(int16 hbuf,int16 exp_num,int16
                         img_num,int16_ptr himg )
```

DESCRIPTION

The image handle, *himg*, is a special handle that is used by the other image functions and many higher analysis functions. The handle is a shorthand method for referring to this image. It specifies the buffer handle, *hbuf*, the exposure number, *exp_num*, and the image number *img_num*. In most cases, this is an extremely fast call. It merely fills in table values, assigns a handle, and returns.

RETURN VALUE

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

SEE ALSO

`pl_buf_get_img_ptr(4)`

NOTES

A pointer to the data in this image is a completely different thing. This address is given by the function `pl_buf_get_img_ptr`, which requires an image handle as input. In general, the handle is useful to other PVCAM functions, while the address is useful to programmers who require direct access to the pixel stream.

Many of the image definition factors: size, offset, and binning, are assumed to be the same across all exposures in the buffer. In other words, the parameters reported for *img_num* in exposure 0 are identical to the parameters reported for *img_num* in every exposure.

Note that both *exp_num* and *img_num* are zero-indexed.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_img_ofs(4)**

NAME

`pl_buf_get_img_ofs` – returns offset position of the image.

SYNOPSIS

```
rs_bool
   pl_buf_get_img_ofs(int16 himg,int16_ptr s_ofs,int16_ptr
                      p_ofs)
```

DESCRIPTION

Pixel coordinates in an image begin at 0,0, despite its original position on the CCD. The offset allows that original position to be recreated. The original coordinates are saved in the offset, so that:

`s_ofs = s_offset = s1`(starting serial position)

`p_ofs = p_offset = p1`(starting parallel position)

Each exposure in a sequence shares the same setup, therefore only the image number (specified through *`himg`*) affects the reported offset. The exposure number (also specified through *`himg`*) has no effect.

RETURN VALUE

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

SEE ALSO

NOTES

It is assumed that the offset is identical for each exposure. In other words, each image has its own offset values, but this information is only entered once; it is not repeated for every exposure in the buffer. The value reported for exposure 0 will always be identical to the value reported for every other exposure.

This is usually a safe assumption, but a user might use the image address and direct access to insert images that fit, but were taken under radically different conditions, including different offset. In such a case, the value reported for offset will not change, but it will no longer be accurate. It then becomes the user's responsibility to keep track of the offset.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_img_ptr(4)**

NAME

`pl_buf_get_img_ptr` – returns the address of an image in the data buffer.

SYNOPSIS

```
rs_bool
   pl_buf_get_img_ptr(int16 himg,void_ptr_ptr img_addr)
```

DESCRIPTION

This requires an image handle as input. Given that input, this function returns the address of the first data element inside that image. The user can then directly manipulate or rewrite the data, as desired. It allows optimum efficiency for data manipulation, while still staying inside the PVCAM image buffer structure. The address is returned in *`img_addr`*, which is defined as a pointer to type void. A void pointer must be used, since alignment may vary from buffer to buffer. The user is responsible for knowing the word size and indexing conventions, based on the *`bit_depth, i_size,`* and *`j_size`* of the image.

RETURN VALUE

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

SEE ALSO

`pl_buf_get_img_handle(4)`

NOTES

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_img_size(4)**

NAME

`pl_buf_get_img_size` — returns number of pixels in region.

SYNOPSIS

```
rs_bool
   pl_buf_get_img_size(int16 himg,int16_ptr i_size, int16_ptr
                       j_size)
```

DESCRIPTION

This examines the image specified by the image handle *himg*, and determines the i and j dimensions. The sizes are returned in *I_size* and *j_size* in pixels. Since the pixel addresses begin with 0 (following typical C conventions), the following relationship is true:

```
i_maximum_element_num = i_size - 1
j_maximum_element_num = j_size - 1
```

RETURN VALUE

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

SEE ALSO

`pl_buf_get_img_handle(4)`

NOTES

This size is not necessarily the same as the number of pixels exposed on the CCD. If the region was binned, the CCD area may have had many more pixels than the final data set.

The set of images must be the same for every exposure in the buffer. For example, image 3, exposure 0 must have the same size (and offset and binning) as image 3, exposure 2. The sizes reported for the images in exposure 0 will always be identical to the sizes reported for every other exposure.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_img_total(4)**

NAME

`pl_buf_get_img_total` – returns number of images in each exposure.

SYNOPSIS

```
rs_bool
   pl_buf_get_img_total(int16 hbuf,int16_ptr img_total)
```

DESCRIPTION

This returns the number of images in the first exposure. Every exposure in the same buffer will have exactly this many images, no more, no less. When referring to images by number, counting begins at 0 (in typical C fashion), so the highest allowed image number is actually *`img_total`*-1.

RETURN VALUE

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

SEE ALSO

`pl_buf_get_exp_total(4)`,`pl_buf_get_img_ofs(4)`, `pl_buf_get_img_size(4)`

NOTES

Every exposure in the buffer must have exactly this many images.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_get_size(4)**

NAME

`pl_buf_get_size` – returns size of buffer, in bytes.

SYNOPSIS

```
rs_bool
   pl_buf_get_size(int16 hbuf,uns32_ptr buf_size)
```

DESCRIPTION

This returns the size of a buffer, in bytes, inside the variable *`buf_size`*. This value is useful when memory or disk space is tight, before performing operations such as `pl_buf_copy`.

RETURN VALUE

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

SEE ALSO

NOTES

Buffer size can be estimated if you know the bit depth, number of exposures, and the size of each image. This isn't completely accurate, though, since other information is stored in a buffer: the exposure time and date, exposure duration, size and offset values, etc.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_set_exp_date(4)**

NAME

`pl_buf_set_exp_date` — (re)writes the time that this picture was taken.

SYNOPSIS

```
rs_bool
   pl_buf_set_exp_date(int16 hbuf,int16 exp_num,int16
                       year,uns8 day,uns8 hour,uns8 min,uns8
                       sec,uns16 msec)
```

DESCRIPTION

This allows the time of any exposure to be recorded or rewritten. This should be the time when the exposure started. The format is:

Year	current year (i.e. 1995)
month	1-12 (January through December)
day	1-31 (day number in the current month)
hour	0-23 (24-hour format)
min	0-59
sec	0-59
msec	0-999 milliseconds

To set a single time for the entire buffer, it is usually adequate to set the time for *`exp_num 0`*. (Conversely, this is the time that will be examined when a single reading is desired for an entire sequence.) But, depending on the sequence and timing parameters, successive exposures may be taken hours or even days later. To set the exact exposure date and time for any successive exposure in the sequence, specify a different *`exp_num`*. The exposure end time may be obtained by adding the exposure duration that is obtained from `pl_buf_get_exp_time` function.

RETURN VALUE

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

SEE ALSO

`pl_buf_get_exp_date(4)`,`pl_buf_get_exp_time(4)`, `pl_exp_start_seq(3)`

NOTES

In most cases, the system will be unable to obtain a highly accurate value of the time. (The milliseconds may be particularly inaccurate.) All inputs are checked for proper ranges on input (using the ranges shown above). The inputs will generate appropriate errors if they are out of range. Any value is allowed for the year.

For most exposures, the start of exposure is easy to determine. (Time is measured immediately before a call to `pl_exp_start_seq`.) In some cases (such as triggered exposures), determining the start time may be more difficult. Depending on the communication link to the camera, `pl_exp_check_status` may be a few seconds out of date.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_init(4)**

NAME

`pl_buf_init` — initializes the buffer functions.

SYNOPSIS

```
rs_bool
   pl_buf_init(void)
```

DESCRIPTION

This initializes the pointers and memory needed to use the buffer functions. Since the buffer functions depend on internal tables, these tables must be allocated and initialized before any buffer functions can be used. This function should be called soon after `pl_pvcam_init`.

RETURN VALUE

TRUE for success. FALSE for a failure. Failure sets `pl_error_code`. If the initialization fails, the buffer functions may not be used.

SEE ALSO

`pl_buf_uninit(4)`,`pl_pvcam_init(0)`

NOTES

Currently, buffers are only needed if the exposure includes multiple regions or a complex sequence. In that case, the function `pl_exp_finish_seq` will decode a pixel stream and put the output onto the buffer.

For simple exposures, it may be easier and more efficient to examine the output directly, by using the *`pixel_stream`* array that was passed into `pl_exp_start_seq`. If this is done, the buffer routines will never be needed. It will save space and time if the buffer routines are never referred to and never initialized.

PVCAM **Class 4: Buffer Manipulation** **pl_buf_uninit(4)**

NAME

`pl_buf_uninit` – frees and releases the buffer functions.

SYNOPSIS

```
rs_bool
   pl_buf_uninit(void)
```

DESCRIPTION

This frees and releases all pointers and memory allocated by the buffer initialization. It should be called before calling `pl_pvcam_uninit`. Once the buffers are uninitialized, buffer functions may not be used until the buffer library has been reinitialized.

It is safe to call this function redundantly. If the buffer functions were never initialized, or, if they have already been freed, this does no harm.

RETURN VALUE

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

SEE ALSO

`pl_buf_init(4), pl_pvcam_uninit(0)`

NOTES

Chapter 8: Code Examples

Example 1: pl_get_param & pl_get_enum_param

```
/* This example displays information for currently defined parameter IDs.  */
/* Note: depending on the camera system connected the results will change  */
/* This example is broken into 3 functions  main  calls DisplayParamIdInfo */
/* which calls DisplayEnumInfo to display enumerated data types and        */
/* DisplayIntsFltsInfo to display non-enum data types.                     */

#include <stdio.h>
#include <stdlib.h>
#include "master.h"
#include "pvcam.h"

/* Prototype functions */
static void DisplayIntsFltsInfo (int16 hcam, uns32 param_id);
static void DisplayEnumInfo (int16 hcam, uns32 param_id);
static void DisplayParamIdInfo (int16 hcam, uns32 param_id);

int main(int argc, char **argv)
{
    char cam_name[CAM_NAME_LEN];    /* camera name   */
    int16 hCam;                     /* camera handle */

    /* Initialize the PVCam Library and Open the First Camera */
    pl_pvcam_init();
    pl_cam_get_name( 0, cam_name );
    pl_cam_open(cam_name, &hCam, OPEN_EXCLUSIVE );

    printf( "\nAnti_Blooming\n");
    DisplayParamIdInfo (hCam, PARAM_ANTI_BLOOMING);
    printf( "\nLogic Output\n");
    DisplayParamIdInfo (hCam, PARAM_LOGIC_OUTPUT);
    printf( "\nEdge Trigger\n");
    DisplayParamIdInfo (hCam, PARAM_EDGE_TRIGGER);
    printf( "\nIntensifier Gain\n");
    DisplayParamIdInfo (hCam, PARAM_INTENSIFIER_GAIN);
    printf( "\nGate Mode\n");
    DisplayParamIdInfo (hCam, PARAM_SHTR_GATE_MODE);
    printf( "\nMin Block\n");
    DisplayParamIdInfo (hCam, PARAM_MIN_BLOCK);
    printf( "\nNum Min Block\n");
    DisplayParamIdInfo (hCam, PARAM_NUM_MIN_BLOCK);
    printf( "\nStrips Per Clean\n");
    DisplayParamIdInfo (hCam, PARAM_NUM_OF_STRIPS_PER_CLR);
    printf( "\nReadout Port\n");
    DisplayParamIdInfo (hCam, PARAM_READOUT_PORT);
    printf( "\nController Alive\n");
    DisplayParamIdInfo (hCam, PARAM_CONTROLLER_ALIVE);
    printf( "\nReadout Time\n");
    DisplayParamIdInfo (hCam, PARAM_READOUT_TIME);
    printf( "\nCircular Buffer Support\n");
    DisplayParamIdInfo (hCam, PARAM_CIRC_BUFFER);

    pl_cam_close( hCam );

    pl_pvcam_uninit();

    return 0;
}

/* This will display information we can get from parameter id */
void DisplayParamIdInfo (int16 hcam, uns32 param_id)
```

```
{
    rs_bool status, status2; /* status of pvcam functions                   */
    rs_bool avail_flag;      /* ATTR_AVAIL, param is available              */
    uns16 access;            /* ATTR_ACCESS, param is read, write or exists */
    uns16 type;              /* ATTR_TYPE, param data type                  */

    status = pl_get_param(hcam, param_id, ATTR_AVAIL, (void *)&avail_flag);
    /* check for errors */
    if (status) {
        /* check to see if parameter id is supported by hardware or software */
        if (avail_flag) {
            /* we got a valid parameter, now get access writes and data type
*/
            status = pl_get_param(hcam, param_id, ATTR_ACCESS, (void *)&access);
            status2 = pl_get_param(hcam, param_id, ATTR_TYPE, (void *) &type);
            if (status && status2) {
                if (access == ACC_EXIST_CHECK_ONLY) {
                    printf(" param id %x exists\n", param_id);
                }
                else if ((access == ACC_READ_ONLY) ||
                         (access == ACC_READ_WRITE)) {
                    /* now we can start displaying information           */
                    /* handle enumerated types separate from other data  */
                    if (type == TYPE_ENUM) {
                        DisplayEnumInfo(hcam, param_id);
                    }
                    else {/* take care of the rest of the data types */
                        DisplayIntsFltsInfo(hcam, param_id);
                    }
                }
                else {
                    printf(" error in access check for param_id %x\n",
                           param_id);
                }
            }
            else { /* error occurred calling function */
                printf( "functions failed pl_get_param, with error code %ld\n",
                        pl_error_code());
            }
        }
        else {  /* parameter id is not available with current setup */
            printf( " parameter %x is not available with current hardware"
                    " or software setup\n", param_id);
        }
    }
    else { /* error occurred calling function print out error code */
        printf( "functions failed pl_get_param, with error code %ld\n",
                pl_error_code());
    }

    printf( "Press Enter to Continue..." );
    getchar();
    fflush( stdin );
}               /* end of function DisplayParamIdInfo */

/* This routine assumes the param id is an enumerated type,
   it will print out all the enumerated values that are allowed
   with the param id and display the associated ASCII text. */
static void DisplayEnumInfo (int16 hcam, uns32 param_id)
{
    rs_bool status;         /* status of pvcam functions                */
    uns32 count, index;     /* counters for enumerated types            */
    char enumStr[100];      /* string for enum text                     */
    uns32 enumValue;        /* enum value returned for index & param id */
    /* get number of enumerated values */
    status = pl_get_param(hcam, param_id, ATTR_COUNT, (void *)&count);
    if (status) {
        printf(" enum values for param id %x\n", param_id);
        for (index=0; index < count; index++) {
            /* get enum value and enum string */
            status = pl_get_enum_param(hcam, param_id, index, &enumValue,
                                       enumStr, 100);
```

```
            /* if everything alright print out the results */
            if (status) {
                printf(" index =%ld enum value = %ld, text = %s\n",
                        index, enumValue, enumStr);
            }
            else {
                printf( "functions failed pl_get_enum_param, "
                        "with error code %ld\n", pl_error_code());
            }
        }
    }
    else {
        printf( "functions failed pl_get_param, with error code %ld\n",
                 pl_error_code());
    }
}               /* end of function DisplayEnumInfo */

/* This routine displays all the information associated with the parameter id
   given.  This routine assumes that the data is either uns8, uns16, uns32,
   int8, int16, int32, or flt64 */
static void DisplayIntsFltsInfo (int16 hcam, uns32 param_id)
{
    /* current, min&max, & default values of parameter id */
    union {
        flt64 dval;
        uns32 ulval;
        int32 lval;
        uns16 usval;
        int16 sval;
        uns8  ubval;
        int8  bval;
    } currentVal, minVal, maxVal, defaultVal, incrementVal;
    uns16 type;                         /* data type of parameter id */
    rs_bool status, status2, status3,
            status4, status5;           /* status of pvcam functions */

    /* get the data type of parameter id  */
    status = pl_get_param(hcam, param_id, ATTR_TYPE, (void *)&type);
    /* get the default, current, min and max values for parameter id */
    /* Note : since the data type for these depends on the parameter */
    /* id you have to call pl_get_param with the correct data type   */
    /* passed for param_value.                                       */
    if (status) {
        switch (type) {
            case TYPE_INT8:
                status = pl_get_param(hcam, param_id, ATTR_CURRENT,
                                (void *)&currentVal.bval);
                status2 = pl_get_param(hcam, param_id, ATTR_DEFAULT,
                                (void *)&defaultVal.bval);
                status3 = pl_get_param(hcam, param_id, ATTR_MAX,
                                (void *)&maxVal.bval);
                status4 = pl_get_param(hcam, param_id, ATTR_MIN,
                                (void *)&minVal.bval);
                status5 = pl_get_param(hcam, param_id, ATTR_INCREMENT,
                                (void *)&incrementVal.bval);
                printf(" param id %x\n", param_id);
                printf(" current value = %c\n", currentVal.bval);
                printf(" default value = %c\n", defaultVal.bval);
                printf(" min = %c, max = %c\n", minVal.bval, maxVal.bval);
                printf(" increment = %c\n", incrementVal.bval);
                break;
            case TYPE_UNS8:
                status = pl_get_param(hcam, param_id, ATTR_CURRENT,
                                (void *)&currentVal.ubval);
                status2 = pl_get_param(hcam, param_id, ATTR_DEFAULT,
                                (void *)&defaultVal.ubval);
                status3 = pl_get_param(hcam, param_id, ATTR_MAX,
                                (void *)&maxVal.ubval);
                status4 = pl_get_param(hcam, param_id, ATTR_MIN,
                                (void *)&minVal.ubval);
                status5 = pl_get_param(hcam, param_id, ATTR_INCREMENT,
                                (void *)&incrementVal.ubval);
```

```
    printf(" param id %x\n", param_id);
    printf(" current value = %uc\n", currentVal.ubval);
    printf(" default value = %uc\n", defaultVal.ubval);
    printf(" min = %uc, max = %uc\n", minVal.ubval, maxVal.ubval);
    printf(" increment = %uc\n", incrementVal.ubval);
    break;
case TYPE_INT16:
    status = pl_get_param(hcam, param_id, ATTR_CURRENT,
                          (void *)&currentVal.sval);
    status2 = pl_get_param(hcam, param_id, ATTR_DEFAULT,
                      (void *)&defaultVal.sval);
    status3 = pl_get_param(hcam, param_id, ATTR_MAX,
                          (void *)&maxVal.sval);
    status4 = pl_get_param(hcam, param_id, ATTR_MIN,
                          (void *)&minVal.sval);
    status5 = pl_get_param(hcam, param_id, ATTR_INCREMENT,
                          (void *)&incrementVal.sval);
    printf(" param id %x\n", param_id);
    printf(" current value = %i\n", currentVal.sval);
    printf(" default value = %i\n", defaultVal.sval);
    printf(" min = %i, max = %i\n", minVal.sval, maxVal.sval);
    printf(" increment = %i\n", incrementVal.sval);
    break;
case TYPE_UNS16:
    status = pl_get_param(hcam, param_id, ATTR_CURRENT,
                          (void *)&currentVal.usval);
    status2 = pl_get_param(hcam, param_id, ATTR_DEFAULT,
                          (void *)&defaultVal.usval);
    status3 = pl_get_param(hcam, param_id, ATTR_MAX,
                          (void *)&maxVal.usval);
    status4 = pl_get_param(hcam, param_id, ATTR_MIN,
                          (void *)&minVal.usval);
    status5 = pl_get_param(hcam, param_id, ATTR_INCREMENT,
                          (void *)&incrementVal.usval);
    printf(" param id %x\n", param_id);
    printf(" current value = %u\n", currentVal.usval);
    printf(" default value = %u\n", defaultVal.usval);
    printf(" min = %uh, max = %u\n", minVal.usval, maxVal.usval);
    printf(" increment = %u\n", incrementVal.usval);
    break;
case TYPE_INT32:
    status = pl_get_param(hcam, param_id, ATTR_CURRENT,
                          (void *)&currentVal.lval);
    status2 = pl_get_param(hcam, param_id, ATTR_DEFAULT,
                          (void *)&defaultVal.lval);
    status3 = pl_get_param(hcam, param_id, ATTR_MAX,
                          (void *)&maxVal.lval);
    status4 = pl_get_param(hcam, param_id, ATTR_MIN,
                          (void *)&minVal.lval);
    status5 = pl_get_param(hcam, param_id, ATTR_INCREMENT,
                          (void *)&incrementVal.lval);
    printf(" param id %x\n", param_id);
    printf(" current value = %ld\n", currentVal.lval);
    printf(" default value = %ld\n", defaultVal.lval);
    printf(" min = %ld, max = %ld\n", minVal.lval, maxVal.lval);
    printf(" increment = %ld\n", incrementVal.lval);
    break;
case TYPE_UNS32:
    status = pl_get_param(hcam, param_id, ATTR_CURRENT,
                          (void *)&currentVal.ulval);
    status2 = pl_get_param(hcam, param_id, ATTR_DEFAULT,
                          (void *)&defaultVal.ulval);
    status3 = pl_get_param(hcam, param_id, ATTR_MAX,
                          (void *)&maxVal.ulval);
    status4 = pl_get_param(hcam, param_id, ATTR_MIN,
                          (void *)&minVal.ulval);
    status5 = pl_get_param(hcam, param_id, ATTR_INCREMENT,
                          (void *)&incrementVal.ulval);
    printf(" param id %x\n", param_id);
    printf(" current value = %ld\n", currentVal.ulval);
    printf(" default value = %ld\n", defaultVal.ulval);
    printf(" min = %ld, max = %ld\n", minVal.ulval, maxVal.ulval);
```

```
                printf(" increment = %ld\n", incrementVal.ulval);
                break;
            case TYPE_FLT64:
                status = pl_get_param(hcam, param_id, ATTR_CURRENT,
                                      (void *)&currentVal.dval);
                status2 = pl_get_param(hcam, param_id, ATTR_DEFAULT,
                                      (void *)&defaultVal.dval);
                status3 = pl_get_param(hcam, param_id, ATTR_MAX,
                                      (void *)&maxVal.dval);
                status4 = pl_get_param(hcam, param_id, ATTR_MIN,
                                      (void *)&minVal.dval);
                status5 = pl_get_param(hcam, param_id, ATTR_INCREMENT,
                                      (void *)&incrementVal.dval);
                printf(" param id %x\n", param_id);
                printf(" current value = %g\n", currentVal.dval);
                printf(" default value = %g\n", defaultVal.dval);
                printf(" min = %g, max = %g\n", minVal.dval, maxVal.dval);
                printf(" increment = %g\n", incrementVal.dval);
                break;
            default:
                printf(" data type not supported in this functions\n");
                break;
        }
        if (!status || !status2 || !status3 || !status4 || !status5) {
             printf( "functions failed pl_get_param, with error code %ld\n",
                     pl_error_code());
        }
    }
    else {
         printf( "functions failed pl_get_param, with error code %ld\n",
                  pl_error_code());
    }
}
```

Example 2: pl_set_param

This example assumes data type to set is int16. This routine does do the error checks to make sure you can write to the param and that its param id is an int16.

```
#include <stdio.h>
#include <stdlib.h>
#include "master.h"
#include "pvcam.h"

/* Prototype functions */
static rs_bool SetParamExample (int16 hcam, uns32 param_id, int16 value);

int main(int argc, char **argv)
{
    char cam_name[CAM_NAME_LEN];    /* camera name   */
    int16 hCam;                     /* camera handle */

    /* Initialize the PVCam Library and Open the First Camera */
    pl_pvcam_init();
    pl_cam_get_name( 0, cam_name );
    pl_cam_open(cam_name, &hCam, OPEN_EXCLUSIVE );

    /* Change the min skip block and number of min blocks to 2 and 100 */
    SetParamExample(hCam, PARAM_MIN_BLOCK, 2);
    SetParamExample(hCam, PARAM_NUM_MIN_BLOCK, 100);

    pl_cam_close( hCam );

    pl_pvcam_uninit();

    return 0;
}


rs_bool SetParamExample (int16 hcam, uns32 param_id, int16 value)
```

```
{
    rs_bool status;     /* status of pvcam functions                    */
    rs_bool avail_flag; /* ATTR_AVAIL, param is available               */
    uns16 access;       /* ATTR_ACCESS, param is read, write or exists */
    uns16 type;         /* ATTR_TYPE, param data type                   */

    status = pl_get_param(hcam, param_id, ATTR_AVAIL, (void *)&avail_flag);
    /* check for errors */
    if (status) {
        /* check to see if parameter id is supported by hardware or software */
        if (avail_flag) {
            /* we got a valid parameter, now get access rights and data type */
            status = pl_get_param(hcam, param_id, ATTR_ACCESS, (void *)&access);
            if (status) {
                if (access == ACC_EXIST_CHECK_ONLY) {
                    printf(" error param id %x is an exists check, "
                           "and not writable\n", param_id);
                }
                else if (access == ACC_READ_ONLY) {
                    printf(" error param id %x is a readonly variable, "
                           "and not writeable\n", param_id);
                }
                else if (access == ACC_READ_WRITE) {
                    /* we can set it, let's be safe and check to make sure
                       it is the right data type */
                    status = pl_get_param(hcam, param_id, ATTR_TYPE,
                                          (void *) &type);
                    if (status) {
                        if (type == TYPE_INT16) {
                            /* OK lets write to it */
                            pl_set_param(hcam, param_id, (void *)&value);
                            printf( "param %x set to %i\n", param_id, value );
                        }
                        else {
                            printf( "data type mismatch for param_id "
                                    "%x\n", param_id );
                            status = FALSE;
                        }
                    }
                    else {
                        printf( "functions failed pl_get_param, with "
                                "error code %ld\n", pl_error_code());
                    }
                }
                else {
                    printf(" error in access check for param_id "
                           "%x\n", param_id);
                }
            }
            else { /* error occurred calling function */
                printf( "functions failed pl_get_param, with error code %ld\n",
                        pl_error_code());
            }

        }
        else {  /* parameter id is not available with current setup */
            printf(" parameter %x is not available with current hardware or "
                   "software setup\n", param_id);
        }
    }
    else { /* error occurred calling function; print out error code */
        printf( "functions failed pl_get_param, with error code %ld\n",
                pl_error_code());
    }

    return(status);
}
```

Example 3: Circular Buffer

Latest Frame Mode (FOCUS)

The following is an example of a circular buffer with the latest frame mode set. The example takes the proper steps to set the camera up beforehand. (i.e., pl_cam_open, etc. and that `pl_get_param` with parameter id *`PARAM_CIRC_BUFFER`* was used to verify that the system could perform circular buffer operations) The following code will return the latest frame in the buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include "master.h"
#include "pvcam.h"

static void FocusContinuous( int16 hCam );

int main(int argc, char **argv)
{
    char cam_name[CAM_NAME_LEN];    /* camera name                    */
    int16 hCam;                     /* camera handle                  */
    rs_bool avail_flag;             /* ATTR_AVAIL, param is available */

    /* Initialize the PVCam Library and Open the First Camera */
    pl_pvcam_init();
    pl_cam_get_name( 0, cam_name );
    pl_cam_open(cam_name, &hCam, OPEN_EXCLUSIVE );

    /* check for circular buffer support */
    if( pl_get_param( hCam, PARAM_CIRC_BUFFER, ATTR_AVAIL, &avail_flag ) &&
        avail_flag )
        FocusContinuous( hCam );
    else
        printf( "circular buffers not supported\n" );

    pl_cam_close( hCam );

    pl_pvcam_uninit();

    return 0;
}

void FocusContinuous( int16 hCam )
{
    rgn_type region = { 0, 511, 1, 0, 511, 1 };
    uns32 buffer_size, frame_size;
    uns16 *buffer;
    int16 status;
    uns32 not_needed;
    void_ptr address;
    uns16 numberframes = 5;

    /* Init a sequence set the region, exposure mode and exposure time */
    pl_exp_init_seq();
    pl_exp_setup_cont( hCam, 1, &region, TIMED_MODE, 100, &frame_size,
                       CIRC_OVERWRITE );

    /* set up a circular buffer of 3 frames */
    buffer_size = frame_size * 3;
    buffer = (uns16*)malloc( buffer_size );

    /* Start the acquisition */
    printf( "Collecting %i Frames\n", numberframes );
    pl_exp_start_cont(hCam, buffer, buffer_size );

    /* ACQUISITION LOOP */
    while( numberframes ) {
        /* wait for data or error */
        while( pl_exp_check_cont_status( hCam, &status, &not_needed,
                                         &not_needed ) &&
```

```
                (status != READOUT_COMPLETE && status != READOUT_FAILED) );

        /* Check Error Codes */
        if( status == READOUT_FAILED ) {
            printf( "Data collection error: %i\n", pl_error_code() );
            break;
        }

        if ( pl_exp_get_latest_frame( hCam, &address )) {
            /* address now points to valid data */
            printf( "Center Three Points: %i, %i, %i\n",
                    *((uns16*)address + frame_size/sizeof(uns16)/2 - 1),
                    *((uns16*)address + frame_size/sizeof(uns16)/2),
                    *((uns16*)address + frame_size/sizeof(uns16)/2 + 1) );
            numberframes--;
            printf( "Remaining Frames %i\n", numberframes );
        }
    } /* End while */

    /* Stop the acquisition */
    pl_exp_stop_cont(hCam,CCS_HALT);

    /* Finish the sequence */
    pl_exp_finish_seq( hCam, buffer, 0);

    /*Uninit the sequence */
    pl_exp_uninit_seq();

    free( buffer );
}
```

Oldest Frame Mode (NFRAME)

The following is an example of a circular buffer with the oldest frame mode set. The example takes the proper steps to set the camera up beforehand. (i.e., pl_cam_open, etc. and that `pl_get_param` with parameter id *`PARAM_CIRC_BUFFER`* was used to verify that the system could perform circular buffer operations) This code will return the frames in the order in which they arrived in the buffer, without skipping a frame.

```
#include <stdio.h>
#include <stdlib.h>
#include "master.h"
#include "pvcam.h"

static void AcquireContinuous( int16 hCam );

int main(int argc, char **argv)
{
    char cam_name[CAM_NAME_LEN];    /* camera name                    */
    int16 hCam;                     /* camera handle                  */
    rs_bool avail_flag;             /* ATTR_AVAIL, param is available */

    /* Initialize the PVCam Library and Open the First Camera */
    pl_pvcam_init();
    pl_cam_get_name( 0, cam_name );
    pl_cam_open(cam_name, &hCam, OPEN_EXCLUSIVE );

    /* check for circular buffer support */
    if( pl_get_param( hCam, PARAM_CIRC_BUFFER, ATTR_AVAIL, &avail_flag ) &&
        avail_flag )
        AcquireContinuous( hCam );
    else
        printf( "circular buffers not supported\n" );

    pl_cam_close( hCam );

    pl_pvcam_uninit();

    return 0;
}

void AcquireContinuous( int16 hCam )
{
    rgn_type region = { 0, 511, 1, 0, 511, 1 };
    uns32 buffer_size, frame_size;
    uns16 *buffer;
    int16 status;
    uns32 not_needed;
    void_ptr address;
    uns16 numberframes = 5;

    /* Init a sequence set the region, exposure mode and exposure time */
    pl_exp_init_seq();
    pl_exp_setup_cont( hCam, 1, &region, TIMED_MODE, 100, &frame_size,
                       CIRC_NO_OVERWRITE );

    /* set up a circular buffer of 3 frames */
    buffer_size = frame_size * 3;
    buffer = (uns16*)malloc( buffer_size );

    /* Start the acquisition */
    printf( "Collecting %i Frames\n", numberframes );
    pl_exp_start_cont(hCam, buffer, buffer_size );

    /* ACQUISITION LOOP */
    while( numberframes ) {
        /* wait for data or error */
        while( pl_exp_check_cont_status( hCam, &status, &not_needed,
                                         &not_needed ) &&
               (status != READOUT_COMPLETE && status != READOUT_FAILED) );
```

```
        /* Check Error Codes */
        if( status == READOUT_FAILED ) {
            printf( "Data collection error: %i\n", pl_error_code() );
            break;
        }

        if ( pl_exp_get_oldest_frame( hCam, &address )) {
            /* address now points to valid data */
            printf( "Center Three Points: %i, %i, %i\n",
                    *((uns16*)address + frame_size/sizeof(uns16)/2 - 1),
                    *((uns16*)address + frame_size/sizeof(uns16)/2),
                    *((uns16*)address + frame_size/sizeof(uns16)/2 + 1) );
            numberframes--;
            printf( "Remaining Frames %i\n", numberframes );
            pl_exp_unlock_oldest_frame( hCam );
        }
    } /* End while */

    /* Stop the acquisition */
    pl_exp_stop_cont(hCam,CCS_HALT);

    /* Finish the sequence */
    pl_exp_finish_seq( hCam, buffer, 0);

    /*Uninit the sequence */
    pl_exp_uninit_seq();

    free( buffer );
}
```

Example 4: Standard Mode Acquisition

The following is a simple example of standard mode acquisitions from PVCAM with the minimum set of functions for data acquisition. Note the example is hard-coded for a particular image size of 512 x 512; these normally should be variables.

```
#include <stdio.h>
#include <stdlib.h>
#include "master.h"
#include "pvcam.h"

static void AcquireStandard( int16 hCam );

int main(int argc, char **argv)
{
    char cam_name[CAM_NAME_LEN];    /* camera name                    */
    int16 hCam;                     /* camera handle                  */

    /* Initialize the PVCam Library and Open the First Camera */
    pl_pvcam_init();
    pl_cam_get_name( 0, cam_name );
    pl_cam_open(cam_name, &hCam, OPEN_EXCLUSIVE );

    AcquireStandard( hCam );

    pl_cam_close( hCam );

    pl_pvcam_uninit();

    return 0;
}

void AcquireStandard( int16 hCam )
{
    rgn_type region = { 0, 511, 1, 0, 511, 1 };
    uns32 size;
    uns16 *frame;
    int16 status;
    uns32 not_needed;
    uns16 numberframes = 5;

    /* Init a sequence set the region, exposure mode and exposure time */
    pl_exp_init_seq();
    pl_exp_setup_seq( hCam, 1, 1, &region, TIMED_MODE, 100, &size );

    frame = (uns16*)malloc( size );

    /* Start the acquisition */
    printf( "Collecting %i Frames\n", numberframes );

    /* ACQUISITION LOOP */
    while( numberframes ) {
        pl_exp_start_seq(hCam, frame );

        /* wait for data or error */
        while( pl_exp_check_status( hCam, &status, &not_needed ) &&
               (status != READOUT_COMPLETE && status != READOUT_FAILED) );

        /* Check Error Codes */
        if( status == READOUT_FAILED ) {
            printf( "Data collection error: %i\n", pl_error_code() );
            break;
        }

        /* frame now contains valid data */
        printf( "Center Three Points: %i, %i, %i\n",
                frame[size/sizeof(uns16)/2 - 1],
                frame[size/sizeof(uns16)/2],
                frame[size/sizeof(uns16)/2 + 1] );
```

```
        numberframes--;
        printf( "Remaining Frames %i\n", numberframes );
    } /* End while */

    /* Finish the sequence */
    pl_exp_finish_seq( hCam, frame, 0);

    /*Uninit the sequence */
    pl_exp_uninit_seq();

    free( frame );
}
```

Appendix A: 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 an error message. All of the PVCAM error numbers and their linked error messages are listed in the table that follows. This table will be updated as new error messages are added.

Table 5. Error Codes

Error #	Error Message	Meaning
0	PVCAM_SUCCESS	No error
1	C0_UNKNOWN_ERROR	Unexpected, unanticipated, or undocumented
2	DDI_NOT_PV_DEVICE	This device driver is not a Roper device
3	DDI_BAD_DEV_NAME	No driver found with the specified name
4	DDI_DRIVER_IN_USE	This driver is already in use by another user
5	DDI_ALREADY_OPEN	This driver has already been opened
6	DDI_CANT_OPEN_DRIVER	The driver was found, but could not be opened
7	DDI_CANT_CLOSE_DRIVER	Driver is not currently open; it can't be closed
8	DDI_CLOSE_ERROR	An error occurred while trying to close the driver
9	DDI_ALREADY_ACTIVE	Camera is already taking data; finish or abort
10	DDI_ZERO_SEND_SIZE	Invalid request: transmit zero bytes
11	DDI_ZERO_RECV_SIZE	Invalid request: receive zero bytes
12	DDI_IOPORT_CONFLICT	2 cameras are using the same I/O port
13	DDI_BOARD_NOT_FOUND	Communications board is not at expected location
14	DDI_CABLE_DISCONNECTED	Camera electronics unit cable is not connected
15	DDI_MEM_ALLOC_FAILED	Device driver could not allocate needed memory
16	DDI_IRQID_CONFLICT	2 open cameras are using the same interrupt ID
17	DDI_DRV_CLOS_CLOSE_CAM	Driver not yet opened: pd_cam_close
18	DDI_DRV_CLOS_READ_BYTE	Driver not yet opened: pd_cam_write_read, read
19	DDI_DRV_CLOS_SEND_BYTE	Driver not yet opened: pd_cam_write_read, write
20	DDI_DRV_CLOS_GET_RETRY	Driver not yet opened: pd_driver_get_retries
21	DDI_DRV_CLOS_SET_RETRY	Driver not yet opened: pd_driver_set_retries
22	DDI_DRV_CLOS_GET_TIME	Driver not yet opened: pd_driver_get_timeout
23	DDI_DRV_CLOS_SET_TIME	Driver not yet opened: pd_driver_set_timeout
24	DDI_DRV_CLOS_INFO_LEN	Driver not yet opened: pd_driver_get_info_length
25	DDI_DRV_CLOS_INFO_DUMP	Driver not yet opened: pd_driver_get_info_dump

Error #	Error Message	Meaning
26	DDI_DRV_CLOS_DRV_VER	Driver not yet opened: pd_driver_get_ver
27	DDI_DRV_CLOS_IM_STATUS	Driver not open: pd_driver_get_image_data_status
28	DDI_DRV_CLOS_IM_ABORT	Driver not open: pd_driver_set_image_data_idle
29	DDI_DRV_CLOS_IM_ACTIVE	Driver not open: pd_driver_set_image_data_active
30	DDI_DRV_CLOS_IM_GRAN	Driver not open: pd_driver_get_image_data_gran
31	DDI_BAD_DEVH_CLOSE_CAM	Illegal device handle: pd_cam_close
32	DDI_BAD_DEVH_READ_BYTE	Illegal device handle: pd_cam_write_read, read
33	DDI_BAD_DEVH_SEND_BYTE	Illegal device handle: pd_cam_write_read, write
34	DDI_BAD_DEVH_GET_RETRY	Illegal device handle: pd_driver_get_retries
35	DDI_BAD_DEVH_SET_RETRY	Illegal device handle: pd_driver_set_retries
36	DDI_BAD_DEVH_GET_TIME	Illegal device handle: pd_driver_get_timeout
37	DDI_BAD_DEVH_SET_TIME	Illegal device handle: pd_driver_set_timeout
38	DDI_BAD_DEVH_INFO_LEN	Illegal device handle: pd_driver_get_info_length
39	DDI_BAD_DEVH_INFO_DUMP	Illegal device handle: pd_driver_get_info_dump
40	DDI_BAD_DEVH_DRV_VER	Illegal device handle: pd_driver_get_ver
41	DDI_BAD_DEVH_IM_STATUS	Bad dev handle: pd_driver_get_image_data_status
42	DDI_BAD_DEVH_IM_ABORT	Bad dev handle: pd_driver_set_image_data_idle
43	DDI_BAD_DEVH_IM_ACTIVE	Bad dev handle: pd_driver_set_image_data_active
44	DDI_BAD_DEVH_IM_GRAN	Bad dev handle: pd_driver_get_image_data_gran
45	DDI_SYS_ERR_DEV_DRIVER	System error while accessing the device driver
46	DDI_SYS_ERR_INIT	System error in pd_ddi_init
47	DDI_SYS_ERR_UNINIT	System error in pd_ddi_uninit
48	DDI_SYS_ERR_TOTL_CAMS	System error in pd_ddi_get_total_cams
49	DDI_SYS_ERR_CAM_NAME	System error in pd_ddi_get_all_cam_names
50	DDI_SYS_ERR_OPEN_CAM	System error in pd_cam_open
51	DDI_SYS_ERR_CLOSE_CAM	System error in pd_cam_close
52	DDI_SYS_ERR_READ_BYTE	System error in pd_cam_write_read, read
53	DDI_SYS_ERR_SEND_BYTE	System error in pd_cam_write_read, write
54	DDI_SYS_ERR_GET_RETRY	System error in pd_driver_get_retries
55	DDI_SYS_ERR_SET_RETRY	System error in pd_driver_set_retries
56	DDI_SYS_ERR_GET_TIME	System error in pd_driver_get_timeout
57	DDI_SYS_ERR_SET_TIME	System error in pd_driver_set_timeout
58	DDI_SYS_ERR_INFO_LEN	System error in pd_driver_get_info_length
59	DDI_SYS_ERR_INFO_DUMP	System error in pd_driver_get_info_dump
60	DDI_SYS_ERR_DRV_VER	System error in pd_driver_get_ver

Error #	Error Message	Meaning
61	DDI_SYS_ERR_IM_STATUS	System error in pd_driver_get_image_data_status
62	DDI_SYS_ERR_IM_ABORT	System error in pd_driver_set_image_data_idle
63	DDI_SYS_ERR_IM_ACTIVE	System error in pd_driver_set_image_data_active
64	DDI_SYS_ERR_IM_GRAN	System error in pd_driver_get_image_data_gran
65	DDI_UNKNOWN_DEV_DRIVER	Unknown error while accessing the device driver
66	DDI_UNKNOWN_INIT	Unknown error in pd_ddi_init
67	DDI_UNKNOWN_UNINIT	Unknown error in pd_ddi_uninit
68	DDI_UNKNOWN_TOTL_CAMS	Unknown error in pd_ddi_get_total_cams
69	DDI_UNKNOWN_CAM_NAME	Unknown error in pd_ddi_get_all_cam_names
70	DDI_UNKNOWN_OPEN_CAM	Unknown error in pd_cam_open
71	DDI_UNKNOWN_CLOSE_CAM	Unknown error in pd_cam_close
72	DDI_UNKNOWN_READ_BYTE	Unknown error in pd_cam_write_read, read
73	DDI_UNKNOWN_SEND_BYTE	Unknown error in pd_cam_write_read,write
74	DDI_UNKNOWN_GET_RETRY	Unknown error in pd_driver_get_retries
75	DDI_UNKNOWN_SET_RETRY	Unknown error in pd_driver_set_retries
76	DDI_UNKNOWN_GET_TIME	Unknown error in pd_driver_get_timeout
77	DDI_UNKNOWN_SET_TIME	Unknown error in pd_driver_set_timeout
78	DDI_UNKNOWN_INFO_LEN	Unknown error in pd_driver_get_info_length
79	DDI_UNKNOWN_INFO_DUMP	Unknown error in pd_driver_get_info_dump
80	DDI_UNKNOWN_DRV_VER	Unknown error in pd_driver_get_ver
81	DDI_UNKNOWN_IM_STATUS	Unknown error in pd_driver_get_image_data_status
82	DDI_UNKNOWN_IM_ABORT	Unknown error in pd_driver_set_image_data_idle
83	DDI_UNKNOWN_IM_ACTIVE	Unknown error in pd_driver_set_image_data_active
84	DDI_UNKNOWN_IM_GRAN	Unknown error in pd_driver_get_image_data_gran
85	DDI_SCSI_NOT_PV_CAMERA	This SCSI device is not a Tucson camera
86	DDI_SCSI_NO_PROTOCOL	SCSI protocol breakdown: no device or termination
87	DDI_SCSI_NO_ARBITRATE	SCSI arbitration failure: the bus is busy
88	DDI_SCSI_BAD_XFER	SCSI bad instruction in transfer instruction bloc
89	DDI_SCSI_PHASE_ERROR	SCSI phase error: host & camera disagree on type
90	DDI_SCSI_DATA_ERROR	SCSI data comparison error verifying transfer
91	DDI_SCSI_MGR_BUSY	SCSI manager is busy with another operation
92	DDI_SCSI_SEQUENCE_ERR	SCSI sequencing error
93	DDI_SCSI_BUS_TIMEOUT	SCSI bus timeout waiting for data transfer
94	DDI_SCSI_COMPLETE_ERR	SCSI completion error
95	DDI_SCSI_INTERNAL_ERR	SCSI device indicates an internal error

Error #	Error Message	Meaning
96	DDI_XM_SNDOK	XMODEM
97	DDI_XM_NOSOH	XMODEM
98	DDI_XM_OVERFLOW	XMODEM
99	DDI_XM_RCVOK	XMODEM
100	DDI_XM_RCVCAN	XMODEM
101	DDI_XM_NOACK	XMODEM no ACKnowledge signal received
102	DDI_XM_LASTACK	XMODEM
103	DDI_XM_SNDACK	XMODEM
104	DDI_XM_SNDCAN	XMODEM
105	DDI_XM_MSGEND	XMODEM
106	DDI_XM_BADCKV	XMODEM
107	DDI_XM_BADSOH	XMODEM
108	DDI_XM_NODATA	XMODEM
109	DDI_XM_BADPAK	XMODEM
110	DDI_XM_PAKNUM	XMODEM
111	DDI_XM_PAKSEQ	XMODEM
112	DDI_XM_NOSYNC	XMODEM no SYNC character seen
113	DDI_XM_SYNCTOUT	XMODEM timout while waiting for SYNC character
114	DDI_XM_XMITLOCK	XMODEM transmit ... ?
115	DDI_XM_BADCMD	XMODEM bad command
116	C0_INVALID_HANDLE	This is not the handle of an open camera
117	C0_CAM_ALREADY_OPEN	This user has already opened this camera
118	C0_CAM_NEVER_OPENED	Camera was not opened, so this task can't be done
119	C0_CAM_RESERVED	The camera is in use by another user
120	C0_DRIVER_OUT_OF_MEM	Driver or DDI ran out of (specialized?) memory
121	C0_CANT_READ_TIMEOUT	System couldn't read the timeout for this driver
122	C0_CANT_WRIT_TIMEOUT	System couldn't set the timeout for this driver
123	C0_CANT_READ_RETRIES	System couldn't read the retries for this driver
124	C0_CANT_WRIT_RETRIES	System couldn't set the retries for this driver
125	C0_CAM_TIMEOUT	No response at all from the camera
126	C0_CAM_TIMEOUT_NOISE	Timeout, but some response (noisy line?)
127	C0_RETRIES_EXCEEDED	Not a timeout, but retries didn't work (noisy?)
128	C0_CAM_NAME_OUT_OF_RNG	The number must be in the range 1<=num<=totl_cams
129	C0_CAM_NAME_NOT_FOUND	This is not a valid name for opening the camera

Error #	Error Message	Meaning
130	C0_PACKET_TOO_LARGE	A send or read request used a packet >32768 bytes
131	C0_STATUS_TOO_LARGE	The status info returned contained too many bytes
132	C0_STATUS_TOO_SMALL	The status info returned contained too few bytes
133	C0_NEED_POSITIVE_VAL	The input value must be greater than zero
134	C0_NEED_ZERO_OR_MORE	The input value must be zero or above
135	C0_NULL_POINTER	Input pointer is null, it must be a legal address
136	C0_STSF_EU_CPU	Subsystem fault: electronics unit main CPU
137	C0_STSF_EU_SYS_INTEG	Subsystem fault: EU internal communications
138	C0_STSF_EU_TO_HOST	Subsystem fault: EU-to-host cables
139	C0_STSF_POWER_SUPPLY	Subsystem fault: power supply voltage error
140	C0_STSF_CCS_CHIP	Subsystem fault: CCS chip or memory
141	C0_STSF_CCS_SCRIPT_MEM	Subsystem fault: CCS script memory
142	C0_STSF_CCS_PORTS	Subsystem fault: CCS ports
143	C0_STSF_DISPLAY	Subsystem fault: EU front panel display
144	C0_STSF_SHUTTER_DRIVE	Subsystem fault: shutter driver circuit
145	C0_STSF_TEMP_CONT	Subsystem fault: temperature control circuit
146	C0_STSF_PAR_CLOCK_DRV	Subsystem fault: parallel clock driver
147	C0_STSF_CH_CABLES	Subsystem fault: camera head cables
148	C0_STSF_CH_CPU	Subsystem fault: camera head CPU board
149	C0_STSF_CH_CLOCK_BRD	Subsystem fault: camera head clock board
150	C0_STSF_CH_POWER_BRD	Subsystem fault: camera head power board
151	C0_STSF_CH_VID_BRD_1	Subsystem fault: camera head video board #1
152	C0_STSF_CH_VID_BRD_2	Subsystem fault: camera head video board #2
153	C0_STSF_CH_VID_BRD_3	Subsystem fault: camera head video board #3
154	C0_STSF_CH_VID_BRD_4	Subsystem fault: camera head video board #4
155	C0_STSF_ADC_BOARD_1	Subsystem fault: A/D board #1
156	C0_STSF_ADC_BOARD_2	Subsystem fault: A/D board #2
157	C0_STSF_ADC_BOARD_3	Subsystem fault: A/D board #3
158	C0_STSF_ADC_BOARD_4	Subsystem fault: A/D board #4
159	C0_STSF_OPTION_CARD_1	Subsystem fault: option card #1
160	C0_STSF_OPTION_CARD_2	Subsystem fault: option card #2
161	C0_STSF_OPTION_CARD_3	Subsystem fault: option card #3
162	C0_STSF_OPTION_CARD_4	Subsystem fault: option card #4
163	C0_NO_IMG_DATA	Can't collect data: expected data is zero bytes
164	C0_CCL_SCRIPT_INVALID	Can't collect data: CCS script is invalid

Error #	Error Message	Meaning
165	C0_EXP_FIFO_OVERFLOW	AIA input buffer has overflowed
166	C0_EXP_NO_ACK	Camera didn't acknowledge request for image data
167	C0_EXP_XFER_ERR	Last data transfer from the camera was garbled
168	C0_EXP_EXTRA_DATA	Finished data transfer, but extra data exists
169	C0_EXP_MISSING_DATA	Finished data transfer, some data was missing
170	C0_OPEN_MODE_UNAVAIL	Camera may not be opened in the mode specified
171	C0_WRONG_READ_CLASS	Read operations require the HOST_COMMANDS class
172	C0_WRITE_BYTES_TOO_SML	Command sent to camera must be at least 1 byte
173	C0_WRITE_BYTES_TOO_LRG	Cannot send over 32768 bytes in one transaction
174	C0_READ_BYTES_TOO_SML	A read transaction must transfer at least 1 byte
175	C0_READ_BYTES_TOO_LRG	Cannot read over 32768 bytes in one transaction
176	C0_WRONG_READ_CMD	'read' command is improperly formatted
177	DDI_DRV_CLOS_GET_PIXTIME	Driver not yet opened: pd_driver_get_pixtime
178	DDI_SYS_ERR_GET_PIXTIME	System error in pd_driver_get_pixtime
179	DDI_BAD_DEVH_GET_PIXTIME	Bad dev handle: pd_driver_get_pixtime
180	DDI_UNKNOWN_GET_PIXTIME	Unknown error in pd_driver_get_pixtime
181	DDI_CAM_XOFF	Camera can't communicate after sending an X-OFF
182	C0_BAD_CONTROLLER	Controller for camera not valid
183	C0_CNTRL_CREATE_FAILED	Could not create controller object for camera
184	C0_NO_CONT_STATUS	Status not available for continuous exposure
185	C0_STAT_CNTRL_ERROR	Controller error while requesting status
186	C0_STAT_CMD_ERROR	Command error while requesting status
187	C0_STAT_DMA_OVERRUN	DMA data overrun has occurred
188	C0_STAT_TAXI_VIOLATION	Violation in TAXI communication protocol occurred
189	C0_STAT_MAILBOX_ERROR	Mailbox error while requesting status
190	C0_STAT_CH0_ERROR	Channel 0 transfer not enabled
191	C0_STAT_CH1_ERROR	Channel 1 transfer not enabled
192	C0_CANT_READ_ID	System couldn't read the subsystem part numbers
193	C0_CANT_READ_NAME	System couldn't read the name for this subsystem
194	C0_DEV_HANDLE_UNAVAIL	Camera device handle unavailable
195	C0_PVCAM_NOT_INITED	Camera library not initialized
196	C0_NOT_INITIALIZED	The pg_decode_info structure is not initialized
1000	C01_START_ERROR	unknown error
2000	C2_UNKNOWN_ERROR	Unexpected, unanticipated, undocumented

Error #	Error Message	Meaning
2001	C2_PVCAM_ALREADY_INITED	Init_pvcam has been called twice without closing
2002	C2_PVCAM_NOT_INITED	The PVCAM library was never initialized
2003	C2_FAILED_TO_SET_VALUE	The camera did not accept the new setting
2004	C2_NEED_POSITIVE_VAL	The input value must be greater than zero
2005	C2_NEED_ZERO_OR_MORE	The input value must be zero or above
2006	C2_NULL_POINTER	Input pointer is null, it must be a legal address
2007	C2_FRAME_XFER_ILLEGAL	This CCD does not allow frame transfer operation
2008	C2_FRAME_XFER_REQUIRED	This CCD must be operated in frame transfer mode
2009	C2_MPP_MODE_ILLEGAL	This CCD does not allow mpp-mode clocking
2010	C2_MPP_MODE_REQUIRED	This CCD requires mpp-mode clocking
2011	C2_CLEAR_MODE_INVALID	Requested clear mode is not an allowed choice
2012	C2_SPEED_INVALID	No valid speeds between camera/electronics/host
2013	C2_SPEED_OUT_OF_RANGE	Selected a non-existant speed table entry
2014	C2_CANT_SET_ADC_OFFSET	Camera does not allow offset to be read or set
2015	C2_BAD_CONTROLLER	Controller for camera not valid
2016	C2_NOT_AVAILABLE	Parameter is not available for camera
2017	C2_FAILED_TO_GET_VALUE	The camera did not return the setting
2018	C2_PARAMETER_INVALID	The requested parameter is invalid
2019	C2_ATTRIBUTE_INVALID	The requested attribute is invalid
2020	C2_INDEX_OUT_OF_RANGE	The requested parameter index is out of range
2021	C2_NOT_INPUT	The requested I/O port is not an input port
2022	C2_IO_TYPE_INVALID	The requested I/O port type is not supported
2023	C2_ADDR_OUT_OF_RANGE	The I/O address is out of range
2024	C2_ACCESS_ATTR_INVALID	The I/O port returned access attribute is invalid
2025	C2_CANT_SET_PARAMETER	The requested parameter cannot be set
2026	C2_IO_DIRECTION_INVALID	The returned direction for the I/O port is invalid
2027	C2_NO_ALPHA_SER_NUM	Alphanumeric serial # unavailable for this camera
2028	C2_CANT_OVERSCAN	Camera does not allow overscanning the CCD
2029	C2_CANT_SET_GAIN_MULT	Camera does not allow setting the gain multiplier
3000	C3_INVALID_PIC_TRIGGER_MODE	Invalid PIC trigger mode
3001	C3_NO_COMMUNICATIONS_LINK	Bogus temp
3002	C3_INVALID_SCRIPT	CCL program is not loaded or is invalid
3003	C3_EXP_EXTRA_DATA	Extra data acquired during exposure
3004	C3_EXP_NO_DATA_ACQ	No data acquired during exposure
3005	C3_EXP_FIFO_OVERFLOW	FIFO overflow during exposure

Error #	Error Message	Meaning
3006	C3_EXP_NO_ACKNOWLEDGE	Camera did not acknowledge request during exp
3007	C3_EXP_TRANSFER_ERROR	Transfer error during exposure
3008	C3_EXP_UNKNOWN_STATE	Camera went into unknown state during exp
3009	C3_CANT_DECODE_IN_PROGRESS	Can't decode while readout is in progress
3010	C3_RGN_MAX_EXCEEDED	Trying to exceed the maximum # of regions
3011	C3_RGN_ILLEGAL_DEFN	Dimensions of region to be added is illegal
3012	C3_RGN_ILLEGAL_BINNING	Binning of region to be added is illegal
3013	C3_RGN_OUTSIDE_CCD_DIMENS	Region def extends beyond CCD dimensions
3014	C3_RGN_OVERLAP	Region to be added overlaps a previous region
3015	C3_RGN_INVALID_NUM	Invalid region number
3016	C3_RGN_NOT_FOUND	Region not found
3017	C3_STREAM_PTR_NOT_DEFINED	Pointer to pixel stream is not defined
3018	C3_GROUPS_PTR_NOT_DEFINED	Pointer to decode info structure undefined
3019	C3_NOT_INITIALIZED	pl_init_exp_seq() has not been called
3020	C3_FAILED_TO_SET_VALUE	The value can not be set in the camera
3021	C3_EVENT_NUMBER_INVALID	Frame count for generating event <= 0
3022	C3_EVENT_NOT_SUPPORTED	Specified event is not supported by the O.S.
3023	C3_BAD_CONTROLLER	Controller for camera not valid
3024	C3_EVENT_NOT_SET	Event was not set up
3025	C3_CNTRL_INIT_FAILED	Controller initialization failed
3026	C3_EXP_MODE_NOT_SUPPORTED	Exposure mode not supported by this camera
3027	C3_ILLEGAL_BUFFER_SIZE	Buffer must be integer-multiple of frame size
3028	C3_GET_FRAME_NOT_SUPPORTED	Camera cannot return the specified frame
3029	C3_FRAME_NOT_RETURNED	Specified frame could not be returned
3030	C3_FRAME_BAD_MODE	Frame could not be returned in current mode
3031	C3_NO_DRIVER_BUFFER	Camera does not provide a driver buffer
3032	C3_BUF_NOT_RETURNED	Pointer to buffer could not be returned
3033	C3_BUFFER_OVERRUN	Data Buffer is full no place to xfer data
3034	C3_TAXI_VIOLATION	Communication with device failed, link broken
3035	C3_EXP_RES_OUT_OF_RANGE	Exposure resolution index non-existent
3036	C3_NOT_AVAILABLE	Parameter is not available for camera
3037	C3_IO_PORT_INVALID	Specified I/O port is invalid
3038	C3_FAILED_TO_GET_VALUE	The camera did not return the setting
3039	C3_IO_STATE_OUT_OF_RANGE	Requested I/O state out of range for port
3040	C3_IO_LOCATION_INVALID	Specified script location is invalid

Error #	Error Message	Meaning
3041	C3_IO_NOT_OUTPUT	Specified I/O port is not an output port
3042	C3_EXP_XFER_ERR	Last data transfer from the camera was garbled
3043	C3_EXP_MISSING_DATA	Finished data transfer, some data was missing
3044	C3_STAT_CNTRL_ERROR	Controller error while requesting status
3045	C3_STAT_CMD_ERROR	Command error while requesting status
3046	C3_CAM_NEVER_OPENED	Camera was not opened, so this task can't be done
3047	C3_STAT_DMA_OVERRUN	DMA data overrun has occurred
3048	C3_STAT_TAXI_VIOLATION	Violation in TAXI communication protocol occurred
3049	C3_STAT_MAILBOX_ERROR	Mailbox error while requesting status
3050	C3_STAT_CH0_ERROR	Channel 0 transfer not enabled
3051	C3_STAT_CH1_ERROR	Channel 1 transfer not enabled
3052	C3_UNKNOWN_ERROR	Unexpected, unanticipated, undocumented
4000	C04_HBUF_OUTOFRANGE	HBUF is out of range
4001	C04_HIMG_OUTOFRANGE	HIMG is out of range
4002	C04_NO_FREE_BUFFER_HANDLES	No free buffer handles available
4003	C04_NO_FREE_IMAGE_HANDLES	No free image handles available
4004	C04_BUFFER_ENTRY_ALREADY_SET	Buffer entry is already set
4005	C04_BUFFER_ENTRY_ALREADY _CLEARED	Buffer entry is already cleared
4006	C04_IMAGE_ENTRY_ALREADY_SET	Image entry is already set
4007	C04_IMAGE_ENTRY_ALREADY _CLEARED	Image entry is already cleared
4008	C04_INVALID_IMAGE_HANDLE	Invalid image handle
4009	C04_INVALID_BUFFER_HANDLE	Invalid buffer handle
4010	C04_INVALID_BITDEPTH_VALUE	Bit depth must be enum PRECISION_...
4011	C04_INVALID_IMAGE_NUMBER	Invalid image number
4012	C04_INVALID_EXPOSURE_NUMBER	Invalid exposure number
4013	C04_INVALID_TIME	The time or date is out of range
4014	C04_INVALID_REGION	A region is out of range
14000	C14_UNKNOWN_ERROR	Unexpected, unanticipated, undocumented
14001	C14_CANT_READ_INI_FILE	Unable to read the current INI file. Please run RSConfig.exe
29000	C29_UNKNOWN_ERROR	Unexpected, unanticipated, undocumented
29001	C29_BDEPTH_ILLEGAL	Bit depth must be enum PRECISION_...
29002	C29_BDEPTH_DIFFER	Bit depth source much match destination
29003	C29_BUF_NEEDS_1_EXP	A buffer needs at least 1 exposure

Error #	Error Message	Meaning
29004	C29_BUF_NEEDS_1_IMG	A buffer needs at least 1 image
29005	C29_IMG_DEF_TOO_LARGE	Image definition used too large a value
29006	C29_IMG_DEF_TOO_SMALL	Image size/bin must be larger than zero
29007	C29_IMG_DEF_DIFFER	Image source definition must match dest
29008	C29_IMG_NUM_DIFFER	Source # of images must match dest
30000	C30_UNKNOWN_ERROR	Unexpected, unanticipated, undocumented
30001	C30_CANT_READ_TIME	Unable to read the current system time
30002	C30_END	
31000	C31_INVALID_HEAP	Invalid heap ID: PUBLIC_MEM, PRIVATE_MEM
31001	C31_MEMALLOC_FAILED	Not enough memory to perform alloc
31002	C31_MEMCALLOC_FAILED	Not enough memory to perform calloc
31003	C31_MEMREALLOC_FAILED	Not enough memory to perform realloc
31004	C31_PRIV_MEM_BLOCK_TOO_BIG	Excceeds 64k limit for PRIVATE_MEM
31005	C31_MEMLOCK_FAILED	Memory page locking failed
32000	CCL_TOO_COMPLEX	Too many script entries
32001	CCL_CANT_FRAME_TRANSFER	No frame transfer hardware support
32002	CCL_SCRIPT_IS_NOT_VALID	Invalid script
32003	CCL_REGIONS_OVERLAP	Regions contain some of the same pixels
32004	CCL_INVALID_SERIAL_BINNING	Serial binning == 0 or > region size
32005	CCL_INVALID_PARALLEL_BINNING	Parallel binning == 0 or > region size
32006	CCL_NONMATCHED_PARALLEL _BINNING	Conflicting parallel binning values
32007	CCL_PARALLEL_BINNING_MISALIGNED	Conflicting parallel binning alignment
32008	CCL_INVALID_REGION	Region is not on the CCD
32009	CCL_INVALID_IO_PORT_TYPE	Requested I/O port is not a valid type
32010	C32_NOT_INITIALIZED	The pg_decode_info structure is not initialized

Appendix B: Obsolete Functions

The following list of functions have been made obsolete through the use of `pl_get_param`, `pl_set_param`, `pl_get_enum_param`, and `pl_enum_str_length` functions. They still function correctly and are still supported, but for future programming, the following functions should not be used. For more information about the `pl_get_param` and `pl_set_param` parameter ids, refer to Chapter 5.

Table 6. Obsolete Class 0 Functions and Their pl_set_param/pl_set_param Equivalents

Obsolete Class 0 Function	pl_set_param/pl_get_param Equivalent
pl_dd_get_info	PARAM_DD_INFO
pl_dd_get_info_length	PARAM_DD_INFO_LENGTH
pl_dd_get_retries	PARAM_DD_RETRIES
pl_dd_set_retries	PARAM_DD_RETRIES
pl_dd_get_timeout	PARAM_DD_TIMEOUT
pl_dd_set_timeout	PARAM_DD_TIMEOUT
pl_dd_get_ver	PARAM_DD_VERSION

Table 7. Obsolete Class 2 Functions and Their pl_set_param/pl_set_param Equivalents

Obsolete Class 2 Function	pl_set_param/pl_get_param Equivalent
pl_ccd_get_adc_offset	PARAM_ADC_OFFSET
pl_ccd_get_chip_name	PARAM_CHIP_NAME
pl_ccd_get_clear_cycles	PARAM_CLEAR_CYCLES
pl_ccd_get_clear_mode	PARAM_CLEAR_MODE
pl_ccd_get_color_mode	PARAM_COLOR_MODE
pl_ccd_get_cooling_mode	PARAM_COOLING_MODE
pl_ccd_get_frame_capable	PARAM_FRAME_CAPABLE
pl_ccd_get_fwell_capacity	PARAM_FWELL_CAPACITY
pl_ccd_get_mpp_capable	PARAM_MPP_CAPABLE
pl_ccd_get_par_size	PARAM_PAR_SIZE
pl_ccd_get_pix_par_dist	PARAM_PIX_PAR_DIST
pl_ccd_get_pix_par_size	PARAM_PIX_PAR_SIZE
pl_ccd_get_pix_ser_dist	PARAM_PIX_SER_DIST
pl_ccd_get_pix_ser_size	PARAM_PIX_SER_SIZE

Obsolete Class 2 Function	pl_set_param/pl_get_param Equivalent
pl_ccd_get_pmode	PARAM_PMODE
pl_ccd_get_postmask	PARAM_POSTMASK
pl_ccd_get_postscan	PARAM_POSTSCAN
pl_ccd_get_preamp_dly	PARAM_PREAMP_DELAY
pl_ccd_get_preamp_off_control	PARAM_PREAMP_OFF_CONTROL
pl_ccd_get_preflash	PARAM_PREFLASH
pl_ccd_get_premask	PARAM_PREMASK
pl_ccd_get_prescan	PARAM_PRESCAN
pl_ccd_get_ser_size	PARAM_SER_SIZE
pl_ccd_get_serial_num	PARAM_SERIAL_NUM
pl_ccd_get_summing_well	PARAM_SUMMING_WELL
pl_ccd_get_tmp	PARAM_TEMP (pl_get_param only)
pl_ccd_get_tmp_range	PARAM_TEMP_SETPOINT (from the attributes of the get you can get the min and max allowed temperature settings)
pl_ccd_get_tmp_setpoint	PARAM_TEMP_SETPOINT
pl_ccd_set_adc_offset	PARAM_ADC_OFFSET
pl_ccd_set_clear_cycles	PARAM_CLEAR_CYCLES
pl_ccd_set_clear_mode	PARAM_CLEAR_MODE
pl_ccd_set_pmode	PARAM_PMODE
pl_ccd_set_preamp_off_control	PARAM_PREAMP_OFF_CONTROL
pl_ccd_set_tmp_setpoint	PARAM_TEMP_SETPOINT
pl_ccs_get_status	PARAM_CCS_STATUS
pl_shtr_get_close_dly	PARAM_SHTR_CLOSE_DELAY
pl_shtr_get_open_dly	PARAM_SHTR_OPEN_DELAY
pl_shtr_get_open_mode	PARAM_SHTR_OPEN_MODE
pl_shtr_get_status	PARAM_SHTR_STATUS
pl_shtr_set_close_dly	PARAM_SHTR_CLOSE_DELAY
pl_shtr_set_open_dly	PARAM_SHTR_OPEN_DELAY
pl_shtr_set_open_mode	PARAM_SHTR_OPEN_MODE
pl_spdtab_get_bits	PARAM_BIT_DEPTH
pl_spdtab_get_entries	PARAM_SPDTAB_INDEX with ATTR_MAX
pl_spdtab_get_max_gain	PARAM_GAIN_INDEX with ATTR_MAX
pl_spdtab_get_port	PARAM_READOUT_PORT

Obsolete Class 2 Function	pl_set_param/pl_get_param Equivalent
pl_spdtab_get_port_total	PARAM_READOUT_PORT with ATTR_COUNT
pl_spdtab_get_time	PARAM_PIX_TIME
pl_spdtab_get_gain	PARAM_GAIN_INDEX
pl_spdtab_get_num	PARAM_SPDTAB_INDEX
pl_spdtab_set_gain	PARAM_GAIN_INDEX
pl_spdtab_set_num	PARAM_SPDTAB_INDEX

Table 8. Obsolete Class 3 Functions and Their pl_set_param/pl_set_param Equivalents

Obsolete Class 3 Function	pl_set_param/pl_get_param Equivalent
pl_exp_check_progress	
pl_exp_get_time_seq	PARAM_EXP_TIME
pl_exp_set_time_seq	PARAM_EXP_TIME
pl_exp_set_cont_mode	

Obsolete Class 0 Functions

PVCAM	**Class 0: Camera Communications**	**pl_dd_get_info(0)**

NAME

`pl_dd_get_info` – reads text information about the current device driver.

SYNOPSIS

```
rs_bool
   pl_dd_get_info(int16 hcam,int16 bytes,char_ptr text)
```

DESCRIPTION

This function returns information from the current device driver (specified by *hcam*) including unusual conditions and special information. Since the information may change from system to system, it is presented as unformatted text. The input string text must be allocated to be at least bytes characters long. No more than bytes characters are written into the string text. The size of the complete message can be obtained from the associated parameter id `PARAM_DD_INFO_LENGTH`.

RETURN VALUE

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

SEE ALSO

parameter id `PARAM_DD_INFO_LENGTH`

NOTES

On many systems, there is not a message. If there is not a message, parameter id `PARAM_DD_INFO_LENGTH` returns a length of 0.

PVCAM | **Class 0: Camera Communications** | **pl_dd_get_info_length(0)**

NAME

`pl_dd_get_info_length` – returns length of info message.

SYNOPSIS

```
rs_bool
   pl_dd_get_info_length(int16 hcam,int16_ptr bytes)
```

DESCRIPTION

This is a companion to the `pl_dd_get_info` function, which returns an information message for each device, as specified by *`hcam`*. This function returns the length of that message, in the variable bytes.

RETURN VALUE

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

SEE ALSO

`pl_dd_get_info(0)`

NOTES

Many devices have no message. In other words, they return a value of 0 for bytes.

PVCAM **Class 0: Camera Communications** **pl_dd_get_retries(0)**

NAME

`pl_dd_get_retries` – reads the maximum number of command retransmission attempts that are allowed.

SYNOPSIS

```
rs_bool
   pl_dd_get_retries(int hcam,uns16_ptr max_retries)
```

DESCRIPTION

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 communications link has failed. The camera won't 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 communications link, hardware platform, and operating system. It may also be reset by the user. *hcam* must be a valid camera handle.

RETURN VALUE

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

SEE ALSO

`pl_dd_set_retries(0)`,`pl_dd_get_timeout(0)`,`pl_dd_set_timeout(0)`

NOTES

When the camera is initially opened, the driver uses a default *timeout* and `max_retries`. These numbers, representing reasonable values, were specifically set for that communications link, hardware platform, and operating system. Those values can be examined by calling `pl_dd_get_timeout` and `pl_dd_get_retries`. Both of these values can be changed, but only after the camera has successfully opened. If both numbers are known, the worst-case device driver response may be approximated. See `pl_dd_set_timeout` for a discussion.

The number of retries applies to status communications as well as commands. In other words, if the camera electronics unit sends status data, but the message is garbled, the device driver requests a retransmission. *Max_retries* sets the upper limit to the number of retransmissions that will be requested.

PVCAM **Class 0: Camera Communications** **pl_dd_set_retries(0)**

NAME

`pl_dd_set_retries` — sets the maximum command retry count.

SYNOPSIS

```
rs_bool
   pl_dd_set_retries(int hcam,uns16 max_retries)
```

DESCRIPTION

When a command or status transmission is garbled, the system signals for a retransmission. After a certain number of failed transmissions (the initial transmission plus *max_retries*), the system abandons the attempt and concludes that the communications link has failed. The camera won't close, but the command or status read returns with an error. This command sets the number of allowable retries, before an error is generated. *hcam* must be a valid camera handle.

RETURN VALUE

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

SEE ALSO

`pl_dd_get_retries(0),pl_dd_get_timeout(0),pl_dd_set_timeout(0)`

NOTES

When the camera is initially opened, the driver uses a default *timeout* and *max_retries*. These numbers were specifically set for that communications link, hardware platform, and operating system, and represent reasonable values. Those values may be examined by calling `pl_dd_get_timeout` and `pl_dd_get_retries`, and they can both be changed, but only after the camera has successfully opened. If both numbers are known, the worst-case device driver response may be approximated. See `pl_dd_set_timeout` for a discussion.

Setting `max_retries` to 0 is theoretically reasonable, but in practice, many systems, such as SCSI, require retries.

PVCAM **Class 0: Camera Communications** **pl_dd_get_timeout(0)**

NAME

`pl_dd_get_timeout` – reads the maximum time the driver waits for acknowledgment.

SYNOPSIS

```
rs_bool
   pl_dd_get_timeout(int hcam,uns16_ptr m_sec)
```

DESCRIPTION

When *`hcam`* is a valid camera handle, this function reads the slowest allowable response speed from the camera. This is a crucial factor used in the device driver for communications control. If the driver sends a command to the camera, and doesn't receive acknowledgment within *`m_sec`* milliseconds, the driver times out and returns an error. Unless reset by the user, this time out is a default setting that is contained in the device driver, and is matched to the communications link, hardware platform, and operating system.

RETURN VALUE

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

SEE ALSO

`pl_dd_set_timeout(0),pl_dd_get_retries(0),pl_dd_set_retries(0)`

NOTES

When the camera is initially opened, the driver uses a default *`timeout`* and *`max_retries`*. These numbers, representing reasonable values, were specifically set for that communications link, hardware platform, and operating system. Those values can be examined by calling `pl_dd_get_timeout` and `pl_dd_get_retries`. They can both be changed, but only after the camera has successfully opened.

PVCAM **Class 0: Camera Communications** **pl_dd_set_timeout(0)**

NAME

`pl_dd_set_timeout` – sets the worst-case communications response.

SYNOPSIS

```
rs_bool
   pl_dd_set_timeout(int hcam,uns16 m_sec)
```

DESCRIPTION

When *`hcam`* is a valid camera handle, this function sets the slowest allowable response speed from the camera. This is a crucial factor in device driver communications. If the driver sends a command to the camera, and doesn't receive some sort of acknowledgment within *`m_sec`* milliseconds, the driver times out and returns with an error.

RETURN VALUE

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

SEE ALSO

`pl_dd_get_timeout(0),pl_dd_get_retries(0),pl_dd_set_retries(0)`

NOTES

When the camera is initially opened, the driver uses a default timeout and max_retries. These numbers, representing reasonable values, were specifically set for that communications link, hardware platform, and operating system. Those values may be examined by immediately calling `pl_dd_get_timeout` and `pl_dd_get_retries`. They can both be changed, but only after the camera has successfully opened.

Changing timeout does not mean that each driver call returns within *`m_sec`* milliseconds. Retries and other factors must be considered. The driver then sends the command again. *`Timeout`* only applies to each send-acknowledge cycle. The worst-case driver dead time would be given by

`timeout * (max_retries+1)` + overhead

where overhead may involve minor but unpredictable effects like time slicing, system latency, communications turn around, and driver housekeeping.

When setting *`timeout`*, it is usually wise to set things a little higher than expected. When waiting for a response, a few milliseconds extra is not catastrophic, but terminating prematurely may be.

PVCAM **Class 0: Camera Communications** **pl_dd_get_ver(0)**

NAME

`pl_dd_get_ver` — returns current device driver version number.

SYNOPSIS

```
rs_bool
   pl_dd_get_ver (int16 hcam, uns16_ptr version)
```

DESCRIPTION

This returns a version number for the device driver used to access the camera *hcam*. The version is a formatted hexadecimal number, of the style:

	low byte	
	---	---
high byte	hi 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.

RETURN VALUE

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

SEE ALSO

`pl_ddi_get_ver(0)`,`pl_pvcam_get_ver(0)`

NOTES

Open the camera before calling this function. 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.

Obsolete Class 2 Functions

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

NAME

`pl_ro_get_value` – gets a read-only value from the camera hardware.

SYNOPSIS

```
rs_bool
   pl_ccd_get_chip_name(int16 hcam, char_ptr chip_name )
   pl_ccd_get_color_mode (int16 hcam, uns16_ptr color_mode)
   pl_ccd_get_cooling_mode(int16 hcam, int16_ptr
                           cooling_mode)
   pl_ccd_get_frame_capable(int16 hcam, rs_bool_ptr
                            frame_capable)
   pl_ccd_get_fwell_capacity(int16 hcam, uns32_ptr
                             fwell_capacity )
   pl_ccd_get_mpp_capable(int16 hcam, int16_ptr mpp_capable )
   pl_ccd_get_par_size(int16 hcam, uns16_ptr par_size )
   pl_ccd_get_pix_par_dist(int16 hcam, uns16_ptr
                           pix_par_dist)
   pl_ccd_get_pix_par_size(int16 hcam, uns16_ptr
                           pix_par_size)
   pl_ccd_get_pix_ser_dist(int16 hcam, uns16_ptr
                           pix_ser_dist)
   pl_ccd_get_pix_ser_size(int16 hcam, uns16_ptr
                           pix_ser_size)
   pl_ccd_get_postmask(int16 hcam, uns16_ptr postmask )
   pl_ccd_get_postscan(int16 hcam, uns16_ptr postscan )
   pl_ccd_get_preamp_dly(int16 hcam, uns16_ptr preamp_dly )
   pl_ccd_get_preflash(int16 hcam, uns16_ptr preflash )
   pl_ccd_get_premask(int16 hcam, uns16_ptr premask )
   pl_ccd_get_prescan(int16 hcam, uns16_ptr prescan )
   pl_ccd_get_ser_size(int16 hcam, uns16_ptr ser_size )
   pl_ccd_get_serial_num(int16 hcam, uns16_ptr serial_num )
   pl_ccd_get_summing_well(int16 hcam, rs_bool_ptr
                           s_well_exists)
   pl_ccd_get_tmp(int16 hcam, int16_ptr cur_tmp )
   pl_ccd_get_tmp_range(int16 hcam, int16_ptr
                        tmp_hi_val,int16_ptr tmp_lo_val )
   pl_ccs_get_status(int16 hcam, int16_ptr ccs_status )
   pl_shtr_get_status(int16 hcam, int16_ptr shtr_status )
```

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

```
pl_spdtab_get_bits(int16 hcam, int16_ptr spdtab_bits )
pl_spdtab_get_entries(int16 hcam, int16_ptr
                      spdtab_entries)
pl_spdtab_get_max_gain(int16 hcam, int16_ptr
                       spdtab_max_gain)
pl_spdtab_get_port(int16 hcam, int16_ptr spdtab_port )
pl_spdtab_get_port_total(int16 hcam, int16_ptr
                         total_ports )
pl_spdtab_get_time(int16 hcam, uns16_ptr spdtab_time )
```

DESCRIPTION

When the camera is configured at the factory, it is preset with values based on the CCD specifications, characterization tests, and other sources. Some of these functions return information directly from the camera head memory. Some functions return dynamic conditions (such as temperature) while other settings are based on several inputs. In all cases, the *hcam* parameter indicates the piece of hardware from which the information is read. *hcam* must be a valid camera handle obtained from `pl_cam_open`.

All of these variables are read-only – they are informational parameters and cannot be reset. The read/write parameters are documented under `pl_rw_get_values` and `pl_values_set`.

The full list of parameters and their meanings are:

`chip_name` The name of the CCD. 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.

`ccs_status` This variable holds sixteen bits of status data from the Camera Control Subsystem. Only the lowest 2 bits are currently implemented. These 2 bits (*ccs_status* & `0x03`) give the status of the CCS:

Value	CCS State
0	idle
1	initializing
2	running
3	continuously clearing

A running state occurs any time the CCS is in the process of performing a camera operation (including opening or closing the shutter, exposing, clearing the CCD before a sequence or exposure, parallel or serial shifting, and readout/digitization). After the CCD has finished reading out, the setup determines if the CCS goes to idle or enters continuous clearing mode.

`color_mode` The color mode of the CCD. Where 0 = mono and 1 = color mosaic RGGB. This value is stored in the pv_cam_reads structure.

PVCAM	Class 2: Configuration/Setup	pl_ro_get_value(2)
cooling_mode	This is the type of cooling used by the current camera. The value returned will be one of the following constants: NORMAL_COOL – This is an air or water-cooled system. CRYO_COOL – The camera has an attached Dewar.	
cur_tmp	This reads the current temperature of the CCD in C°x 100. For example, a temperature of -35° would be read as -3500. Note that this returns the **measured** temperature, not the **setpoint** (which is reported in pl_ccd_get_tmp_setpoint.)	
frame_capable	If true, this camera can run in frame transfer mode (set through pl_ccd_set_pmode).	
fwell_capacity	The full-well capacity of this CCD, measured in electrons.	
mpp_capable	Indicates whether this CCD runs in MPP mode. The actual value returned is equal to one of the following four constants: MPP_UNKNOWN MPP_ALWAYS_ON MPP_ALWAYS_OFF MPP_SELECTABLE	
par_size	Parallel size of the CCD, in active rows. The full size of the parallel register is actually (par_size + premask + postmask).	
pix_par_size	Size of the active area of a pixel, in the parallel direction, measured in nanometers.	
pix_par_dist	Center-to-center distance between pixels (in the parallel direction) measured in nanometers. This is identical to pix_par_size, if there are no interpixel dead areas.	
pix_ser_size	Size of a single pixel's active area, in the serial direction, measured in nanometers.	
pix_ser_dist	Center-to-center distance between pixels (in the serial direction), in nanometers. This is identical to pix_ser_size, if there are no dead areas.	
postmask	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 which needs to be done after readout to clear the parallel register.	
postscan	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.	
preflash	The number of milliseconds needed to illuminate the CCD using the flash diode ring before an exposure, dark, or bias.	

premask	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 CCD should probably be run in frame transfer mode.
preamp_dly	Number of milliseconds required for the CCD output preamp to stabilize, after it is turned on.
prescan	Number of pixels discarded from the serial register before the first real data pixel.
s_well_exists	If true, this CCD includes a summing well.
serial_num	This is the serial number of the camera head (not the electronics unit).
ser_size	Serial size of the CCD active area, in pixels.
shtr_status	The current state of the camera shutter (actually, the current state of the driver voltage to the shutter). The returned value will be equal to one of the following constants: SHTR_OPENING, SHTR_OPEN, SHTR_CLOSED, SHTR_CLOSING, or SHTR_FAULT. 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 even though the electronics have reset the voltages to open or close the shutter, there is a lag time for the physical mechanism to respond. See pl_shtr_get_open_dly and pl_shtr_get_close_dly in the pl_rw_get_value function list.
spdtab_bits	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.
spdtab_entries	The number of entries in the speed table. Valid entries range from 0 to *spdtab_entries*-1 (inclusive). The current selection may be altered through pl_spdtab_set_num. Zero entries is possible and indicates that there are no valid speeds that span the requirements of the camera head video board, A/D board, communication channel, and host throughput.
spdtab_max_gain	This reports the maximum gain index setting for the current speed selection, not the actual gain. The minimum gain index is always 1. The maximum gain index is usually 16.
spdtab_port	This reports on the CCD readout port being used by the currently selected speed. Different readout ports (used for alternate speeds) flip the image in X, Y, or both.

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

`spdtab_time`

This is the actual speed for the currently selected speed choice. It returns the time for each pixel, in nanoseconds. This can be converted to a camera speed in kiloHertz through the following formula:

$$\text{camera\_speed (kHz)} = \frac{10^6}{\text{pixel\_time (nanoseconds)}}$$

This readout time will change as new speed choices are selected.

`tmp_hi_val`
`tmp_lo_val`

These two values contain the legal range for temperature settings (using the `pl_ccd_set_tmp_setpoint` command) in hundredths of degrees Celsius. Any number inside this range is legal and will be accepted (-3500 = -35°C). Numbers outside the range are ignored. However, just because a temperature is legal does not mean it is possible. The environment and circumstances will dramatically affect which temperatures can be achieved. An air-cooled camera in Antarctica will be able to reach much lower temperatures than a water-cooled camera in the Sahara.

`total_ports`

1, 2, 3, or 4. The number of ports on the system. This affects the CCS program, but most users will probably not care since multi-port operation is transparent at the level of PVCAM.

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

SEE ALSO `pl_rw_value(2),pl_set_value(2),pl_cam_open(2), pl_cam_close(2)`

NOTES PVCAM interfaces to some cameras that do not support the full PVCAM features or variable set. If the user attempts to get a variable that doesn't exist, the system may either synthesize a value (based on available information) or return an error.

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

NAME

`pl_rw_get_value`– returns a read/write value from the camera hardware.

SYNOPSIS

```
rs_bool
   pl_ccd_get_adc_offset (int16 hcam, int16_ptr offset)
   pl_ccd_get_clear_cycles (int16 hcam, uns16_ptr
                            clear_cycles)
   pl_ccd_get_clear_mode (int16 hcam,int16_ptr
                          clear_mode)
   pl_ccd_get_pmode (int16 hcam, int16_ptr pmode)
   pl_ccd_get_preamp_off_control (int16 hcam, uns32_ptr
                                  preamp_off_control)
   pl_ccd_get_tmp_setpoint (int16 hcam,int16_ptr tmp_setpoint)
   pl_shtr_get_close_dly (int16 hcam, uns16_ptr shtr_close_dly)
   pl_shtr_get_open_dly(int16 hcam, uns16_ptr shtr_open_dly)
   pl_shtr_get_open_mode (int16 hcam,int16_ptr shtr_open_mode)
   pl_spdtab_get_gain (int16 hcam,int16_ptr spdtab_gain)
   pl_spdtab_get_num (int16 hcam,int16_ptr spdtab_num)
```

DESCRIPTION

These functions are very similar. Each returns operating conditions and variables from the camera hardware. The *hcam* parameter indicates from which piece of hardware to read the setting, and must be a valid camera handle obtained from `pl_cam_open`.

This set of variables is read/write – all values may be altered and written to the hardware. The write functions are nearly identical, except that they begin with *set_*, and accept non-pointer arguments. A more extensive set of read-only values are documented under the `pl_ro_get_value` heading.

The full list of parameters and their meaning is listed under `pl_set_values`.

RETURN VALUE

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

SEE ALSO

`pl_ro_get_value(2),pl_cam_open(2),pl_cam_close(2)`

NOTES

PVCAM interfaces to some cameras that do not support the full PVCAM features or variable set. If the user attempts to get a variable that doesn't exist, the system may either synthesize a value (based on available information) or return an error.

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

NAME

`pl_set_value` – sets a value in the camera hardware.

SYNOPSIS

```
rs_bool
   pl_ccd_set_adc_offset (int16 hcam, int16 offset)
   pl_ccd_set_clear_cycles (int16 hcam, uns16  clear_cycles)
   pl_ccd_set_clear_mode (int16 hcam, int16 clear_mode)
   pl_ccd_set_pmode(int16 hcam, int16 pmode)
   pl_ccd_set_preamp_off_control (int16 hcam, uns32
                                  preamp_off_control)
   pl_ccd_set_tmp_setpoint (int16 hcam,int16 tmp_setpoint)
   pl_shtr_set_close_dly (int16 hcam, uns16  shtr_close_dly)
   pl_shtr_set_open_dly (int16 hcam, uns16 shtr_open_dly)
   pl_shtr_set_open_mode (int16 hcam, int16  shtr_open_mode)
   pl_spdtab_set_gain (int16 hcam, int16 spdtab_gain)
   pl_spdtab_set_num (int16 hcam, int16 spdtab_num)
```

DESCRIPTION

These functions set operating conditions and variables in the camera hardware. The *hcam* parameter indicates which piece of hardware to apply the setting to, and must be a valid camera handle obtained from `pl_cam_open`. A camera handle of 0 (normally an invalid handle) will simultaneously send the setting to all open cameras (if this is possible).

A complementary set of functions allows all of these values to be read back from the hardware. They are documented under `pl_rw_get_values`. Many of these settings are also dependent on ranges or capabilities documented in the `pl_ro_get_values` functions, such as `pl_ccd_get_frame_capable` and `pl_ccd_get_tmp_range`.

The full list of parameters and their meanings are:

`clear_cycles` This is the number of times the CCD must be cleared to completely remove charge from the parallel register.

`clear_mode`

`clear_mode` defines when clearing takes place:

`CLEAR_NEVER`	Don't ever clear the CCD.
`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.

`offset`

This allows the user to determine the bias offset voltage. Accepts a signed 16-bit argument: the new bias voltage to be set; returns a signed 16-bit value listing the 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. Plainly, before users can alter the offset level, they 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.

`pmode`

This allows the user to select the parallel clocking method. The following list includes all valid constants:

```
PMODE_NORMAL      PMODE_MPP           PMODE_FT
PMODE_FT_MPP      PMODE_ALT_NORMAL    PMODE_ALT_MPP
PMODE_ALT_FT      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.

`preamp_off_control`

This is the exposure time limit in milliseconds above which the preamp is turned off during exposure.

`shtr_close_dly`

The shutter close delay. This is the number of milliseconds required for the shutter to close. The software default values compensate for the standard Photometrics 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. Valid inputs are any number in the range 0 to 65535 milliseconds.

`shtr_open_dly`

The shutter open delay. This is the number of milliseconds required for the shutter to open. The software default values compensate for the standard Photometrics 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. Valid inputs are any number in the range 0 to 65535 milliseconds.

`shtr_open_mode`

Shutter opening conditions, set to one of the following

`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`).

`spdtab_gain`

The new gain setting for the current speed choice. The valid range for a gain setting is 1 through *`spdtab_max_gain`*, where the max gain may be as high as 16. Values outside this range will be ignored. Note that gain settings 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.

`spdtab_num`

This selects the CCD readout speed from a table of available choices. Entries may range from 0 to *`spdtab_entries`* - 1. This setting affects all other `_spdtab_` values including `spdtab_bits, spdtab_gain, spdtab_max_gain ,spdtab_time, and spdtab_port`. After this call, the gain setting always resets to a value that corresponds to 1x. To use a gain other than 1x, `pl_spdtab_set_gain` must be called after `pl_spdtab_set_num`.

PVCAM	Class 2: Configuration/Setup	pl_set_value(2)
`tmp_setpoint`	This sets the desired CCD temperature in hundredths of degrees Celsius (-35 °C is represented as -3500). The hardware attempts to heat or cool the CCD to this temperature. The min/max allowable temperatures are given by *`tmp_hi_val`* and *`tmp_lo_val`*, from the `pl_ccd_get_tmp_range` function. 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's difficult to get much cooling on an air-cooled camera.	
RETURN VALUE	TRUE for success, FALSE for a failure. Failure sets pl_error_code.	
SEE ALSO	`pl_ro_get_value(2),pl_rw_get_value(2),pl_cam_open(0),pl_cam_close(0)`	
NOTES	PVCAM interfaces to some cameras that do not support the full PVCAM features or variable set. If the user attempts to get a variable that doesn't exist, the system may either synthesize a value (based on available information) or return an error.	

Obsolete Class 3 Functions

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

NAME

`pl_exp_check_progress` – checks the progress of the current exposure.

SYNOPSIS

```
rs_bool
   pl_exp_check_progress(int16 hcam, int16_ptr status,
                         uns32_ptr byte_cnt)
```

DESCRIPTION

This function is similar to `pl_exp_check_status` except that it only returns one of the following values:

`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.

In order to detect errors during the acquisition process, you must use `pl_exp_check_status`. *`byte_cnt`* points to the number of bytes of data that 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 progress was checked successfully. FALSE indicates a bad handle or a problem communicating with the camera.

SEE ALSO

`pl_exp_setup_seq(3)`,`pl_exp_start_seq(3)`, `pl_exp_check_status(3)`

NOTES

When using `pl_exp_check_progress` you could call it inside a loop with a timeout. If the timeout expires, then you could call `pl_exp_check_status` to determine if an error occurred (`READOUT_FAILED`).

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

NAME

`pl_exp_get_time_seq` – only used with `VARIABLE_TIMED_MODE`, this function returns the exposure time from the camera.

SYNOPSIS

```
rs_bool
   pl_exp_get_time_seq(init16 hcam,uns16_ptr exposure_time)
```

DESCRIPTION

This is a companion function to pl_exp_set_time_seq. The two functions are used to examine and change the exposure time in `VARIABLE_TIMED_MODE`.

RETURN VALUE

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

SEE ALSO

`pl_exp_set_time_seq(3),pl_exp_setup_seq(3),Exposure Mode Constants(3)`

NOTES

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

NAME

`pl_exp_set_time_seq` – only used with `VARIABLE_TIMED_MODE`, this function sets the exposure time for the next sequence.

SYNOPSIS

```
rs_bool
   pl_exp_set_time_seq(init16 hcam,uns16 exposure_time)
```

DESCRIPTION

This is a companion function to `pl_exp_get_time_seq`. The two functions are used to examine and change the exposure time in `VARIABLE_TIMED_MODE`.

RETURN VALUE

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

SEE ALSO

`pl_exp_get_time_seq(3),pl_exp_setup_seq(3),Exposure Mode Constants(3)`

NOTES

When using `VARIABLE_TIMED_MODE`, this function must be called before the first sequence is run, because `VARIABLE_TIMED_MODE` ignores the exposure time in the `pl_exp_setup_seq`.

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

NAME

`pl_exp_set_cont_mode` - sets circular buffer mode.

SYNOPSIS

```
rs_bool
   pl_exp_set_cont_mode(int16 hcam, int16 mode)
```

DESCRIPTION

This function sets the mode of operation for the circular buffer. *mode* can be set to either `CIRC_OVERWRITE` or `CIRC_NO_OVERWRITE`. This function must be called before calling `pl_exp_start_cont()`.

RETURN VALUE

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

SEE ALSO

`pl_exp_get_driver_buffer(3)`, `pl_exp_start_cont(3)`, `pl_exp_check_cont_status(0)`, `pl_exp_get_oldest_frame(3)`, `pl_exp_get_latest_frame(3)`, `pl_exp_unlock_oldest_frame(3)`, and `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 either allocated by your application or obtained from the driver as a preallocated block of memory, using the `pl_exp_get_driver_buffer` function.

Refer to **Example 3: Circular Buffer** in *"Chapter 8"* for two examples of code for circular buffer operation.

This function has been replaced by `pl_exp_setup_cont`.

Index

P

R

S

T

V

This page intentionally left blank.