PyVCAM Wrapper

Table of Contents

src

Where the source code of the pyvcam module is located. In addition to the code for the module, any additional scripts that are used to help write the module are included as well. The most notable helper script that is not included in the module is constants_generator.py, which generates the constants.py module by parsing the pvcam header file.

pyvcam

The directory that contains the source code to the pyvcam module. These will be the files that will be installed when users will install the module.

camera.py

The camera.py module contains the Camera python class which is used to abstract the need to manually maintain, alter, and remember camera settings through PVCAM.

Create Camera Example

```
from pyvcam import pvc
from pyvcam.camera import Camera

pvc.init_pvcam()                    # Initialize PVCAM
cam = next(Camera.detect_camera()) # Use generator to find first camera.
cam.open()                          # Open the camera.
```

Attributes of Camera:

Attribute	Description
__name	(Instance variable) A private instance variable that contains the camera's name. Note that this should be a **read only** attribute, meaning it should never be changed or set. PVCAM will handle the generation of camera names and will be set to the appropriate name when a Camera is constructed.
__handle	(Instance variable) A private instance variable that contains the camera's handle. Note that this is a **read only** variable, meaning it should never be changed or set. PVCAM will handle the updating of this attribute when the camera is opened or closed.
__is_open	(Instance variable) A private instance variable that is set to True if the camera is currently opened and False otherwise. Note that this is a **read only** variable, meaning it should never changed or set. PVCAM will handle the updating of this attribute whenever a camera is opened or close.
__exposure_bytes	(Instance variable) A private instance variable that is to be used internally for setting up and capturing a live image with a continuous circular buffer. Note that this is a **read only** variable, meaning that it should never be changed or set manually. This should only be modified/ read by the start_live and get_live_frame functions.
__mode	(Instance variable) A private variable that is to be used internally for setting the correct exposure mode and expose out mode for the camera acquisition setups. Note that his is a **read only** variable, meaning that it should never be changed or set manually. This should only be modified by the magic __init__ function and update_mode function. If you want to change the mode, change the corresponding exposure modes with setters bellow.

__exp_time	(Instance variable) A private variable that is to be used internally as the default exposure time to be used for all exposures. Although this variable is **read only**, you can access it and change it with setters and getters below. The basic idea behind this abstraction is to use this variable all the time for all exposures, but if you need a single, quick capture at a specific exposure time, you can pass it in the get_frame, get_sequence, and get_live_frame functions as the optional parameter.
__binning	(Instance variable) A private variable that is to be used internally as the desired binning for acquisitions. Its used for setting up acquisitions with binning and resizing the returned pixel data to 2D numpy array. Although this variable is **read only**, you can access/ modify it below with the binning, bin_x, and bin_y setters/ getters below.
__roi	(Instance variable) A private variable that is to be used internally as the region of interest (roi) for acquisitions. Its used for setting up acquisitions with the specified roi and resizing the returned pixel data to 2D numpy array. Although this variable is **read only**, you can access/ modify it below with the roi setter and getter.
__shape	(Instance variable) A private variable that is to be used internally as the reshape factor for resizing the returned pixel data to 2D numpy array. Note that it is a **read only** variable, meaning that it should never be changed or set manually. Instead, it is calculated and changed automatically internally whenever the binning or roi of the camera has changed by the calculate_reshape function.
__init__	(Magic Method) The Camera's constructor. Note that this method should not be used in the construction of a Camera. Instead, use the detect_camera class method to generate Camera classes of the currently available cameras connected to the current system.
__repr__	(Magic Method) Returns the name of the Camera.
detect_camera	(Class method) Generator that yields the available cameras connected to the system. For an example of how to call detect_camera, refer to the code sample for creating a camera.
open	(Method) Opens a Camera. Will set __handle to the correct value and __is_open to True if a successful call to PVCAM's open camera function is made. A RuntimeError will be raised if the call to PVCAM fails. For more information about how Python interacts with the PVCAM library, refer to the pvcmodule.cpp section of these notes.

close	(Method) Closes a Camera. Will set __handle to the default value for a closed camera (-1) and will set __is_open to False if a successful call to PVCAM's close camera function is made. A RuntimeError will be raised if the call to PVCAM fails. For more information about how Python interacts with the PVCAM library, refer to the pvcmodule.cpp section of these notes.
get_param	(Method) Gets the current value of a specified parameter. Usually not called directly since the getters/setters (see below) will handle most cases of getting camera attributes. However, not all cases may be covered by the getters/setters and a direct call may need to be made to PVCAM's get_param function. For more information about how to use get_param, refer to the Using get_param and set_param section of the README for the project. **Parameters:** 1. param_id (int): The PVCAM defined value that corresponds to a parameter. Refer to the PVCAM User Manual and constants.py section for list of available parameter ID values. 2. param_attr (int): The PVCAM defined value that corresponds to an attribute of a parameter. Refer to the PVCAM User Manual and constants.py section for list of available attribute ID values.
set_param	(Method) Sets a specified camera parameter to a new value. Usually not called directly since the getters/setters (see below) will handle most cases of setting camera attributes. However, not all cases may be covered by the getters/setters and a direct call may need to be made to PVCAM's set_param function. For more information about how to use set_param, refer to the Using get_param and set_param section of the README for the project. **Parameters:** 1. param_id (int): The PVCAM defined value that corresponds to a parameter. Refer to the PVCAM User Manual and constants.py section for list of available parameter ID values. 2. param_attr (various): The value to set the camera setting to. Make sure that it's type closely matches the attribute's type so it can be properly set. Refer to the PVCAM User Manual for all possible attributes and their types.

check_param	(Method) Checks if a camera parameter is available. This method is useful for checking certain features are available (such as post-processing, expose out mode). Returns true if available, false if not. **Parameters:** 1. param_id (int): The PVCAM defined value that corresponds to a parameter. Refer to the PVCAM User Manual and constants.py section for list of available parameter ID values.
read_enum	(Method) Returns all settings names paired with their values of a specified setting. **Parameters:** 1. param_id (int): The parameter ID.
reset_pp	(Method) If post-processing is available on the camera, the function will call pvc.reset_pp to reset all post-processing features back to their default state. **Parameters:** None
calculate_reshape	(Class Method) This method calculates the new reshape factor for an image whenever a parameter that would change frame dimension (binning, roi) is modified. The reshape factor is used on all methods that return an image to ensure correct image dimensions. Usually you do not need to call this method as it's called automatically when changing binning, roi, etc... **Parameters:** None
update_mode	(Class Method) This method updates the mode of the camera, which is the bit-wise or between exposure mode and expose out mode. It also sets up a temporary sequence to the exposure mode and expose out mode getters will read as expected. This should really only be called internally (and automatically) when exposure mode or expose out mode is modified. **Parameters:** None

get_frame	(Method) Calls the pvcmodule's get_frame function with cameras current settings to get a 2D numpy array of pixel data from a single snap image. This method can either be called with or without a given exposure time. If given, the method will use the given parameter. Otherwise, if left out, will use the internal exp_time attribute. **Parameters:** 1. Optional: exp_time (int): The exposure time to use.

get_sequence	(Method) Calls the pvcmodule's get_frame function with cameras current settings in rapid-succession to get a 3D numpy array of pixel data from a single snap image.
	
	Example:
	
	Getting a sequence
	`# Given that the camera is already opened as openCam`
	
	`stack = openCam.get_sequence(8) # Getting a sequence of 8 frames`
	
	`firstFrame = stack[0] # Accessing 2D frames from 3D stack`
	`lastFrame = stack[7]`
	
	Parameters:
	
	1. num_frames (int): The number of frames to be captured in the sequence.
	2. Optional: exp_time (int): The exposure time to use.
	3. Optional: interval (int): Time to between each sequence frame (in milliseconds).

get_vtm_sequence	(Method) Modified get-sequence to be used for Variable Timed Mode. Before calling it, set the camera's exposure mode to "Variable Timed". The timings will always start at the first given and keep looping around until its captured the number of frames given. **Parameters:** 1. time_list (list of integers): The timings to be used by the camera in Variable Timed Mode 2. exp_res (int): The exposure time resolution. Currently only has milliseconds (0) and microseconds (1). Refer to the PVCAM User Manual class 3 parameter EXP_RES and EXP_RES_INDEX 3. num_frames (int): The number of frames to be captured in the sequence. 4. Optional: interval (int): Time to between each sequence frame (in milliseconds).
start_live	(Method) Calls pvc.start_live to setup a live circular buffer to be used for live mode. This must be called before get_live_frame. This method can either be called with or without a given exposure time. If given, the method will use the given parameter. Otherwise, if left out, will use the internal exp_time attribute. **Parameters:** 1. Optional: exp_time (int): The exposure time to use.
stop_live	(Method) Calls pvc.stop_live to return the camera to it's normal state after acquiring live images. **Parameters:** None
get_live_frame	(Method) Just like get_frame, this function gets a 2D numpy array of pixel data from the live circular buffer. This function waits for an image to be ready from the circular buffer. This function can only return an image after a live circular buffer has been setup with start_live. **Parameters:** None

start_live_cb	(Method) Calls pvc.start_live to setup a live circular buffer to be used for live mode using callbacks to recieve the image. This must be called before get_live_frame_cb. This method can either be called with or without a given exposure time. If given, the method will use the given parameter. Otherwise, if left out, will use the internal exp_time attribute. **Parameters:** 1. Optional: exp_time (int): The exposure time to use.
get_live_frame_cb	(Method) Just like get_frame, this function gets a 2D numpy array of pixel data from the live circular buffer. This function returns the most recently recieved image from the circular buffer. This function can only return an image after a live circular buffer has been setup with start_live_cb. **Parameters:** None

Getters/Setters of Camera:

All getters and setters can be accessed using the example below. There is one large implementation point to make note of:

- All getters/setters are accessed *by attribute.* This means that it will appear that we are accessing instance variables from a camera, but in reality, these getters/setters are making specially formatted calls to the Camera method get_param/set_param. These getters/setters make use of the property decorator that is built into Python. The reasoning behind the usage of the property decorator is that attributes will change dynamically during a Camera's lifetime and in order to abstract PVCAM as far away from the end user as possible, the property decorator allows for users to intuitively view and change camera settings. The downside to this approach is that when a new parameter is required, an associated getter/setter needs to be written and tested. Another downside to this implementation is that attribute lookup time is not instant; instead, a call must be made to the pvcmodule wrapper which will then call PVCAM, which will then return a result to the wrapper, which will finally return the result to the user. The time it takes is currently considered insignificant, but if this were to become an issue, the code could be refactored such that all attributes also have instance variables which are changed only when set_param or their associated setter is called on them.

Using Getters/Setters

```
# Assume cam is an already constructed camera.
curr_gain = cam.gain  # To call getter, simply access it by attribute from
the camera.
```

```
cam.gain = 1          # To call setter, specify which attribute must be set
and assign it a new value.
```

Attribute	Getter/Setter Description
name	(Getter only) Returns the value currently stored inside the Camera's __name instance variable.
handle	(Getter only) Returns the value currently stored inside the Camera's __handle instance variable.
is_open	(Getter only) Returns the value currently stored inside the Camera's __is_open instance variable.
driver_version	(Getter only) Returns a formatted string containing the major, minor, and build version. When get_param is called on the device driver version, it returns a highly formatted 16 bit integer. The first 8 bits correspond to the major version, bits 9-12 are the minor version, and the last nibble is the build number.
cam_fw	(Getter only) Returns the cameras current firmware version as a string.
chip_name	(Getter only) Returns the camera sensor's name as a string.
sensor_size	(Getter only) Returns the sensor size of the current camera in a tuple in the form (serial sensor size, parallel sensor size)
serial_no	(Getter only) Returns the camera's serial number as a string.
bit_depth	(Getter only) Returns the bit depth of pixel data for images collected with this camera. Bit depth cannot be changed directly; instead, users must select a desired speed table index value that has the desired bit depth. Note that a camera may have additional speed table entries for different readout ports. See Port and Speed Choices section inside the PVCAM User Manual for a visual representation of a speed table and to see which settings are controlled by which speed table index is currently selected.
pix_time	(Getter only) Returns the camera's pixel time, which is the inverse of the speed of the camera. Pixel time cannot be changed directly; instead users must select a desired speed table index value that has the desired pixel time. Note that a camera may have additional speed table entries for different readout ports. See Port and Speed Choices section inside the PVCAM User Manual for a visual representation of a speed table and to see which settings are controlled by which speed table index is currently selected.

readout_port	(Getter and Setter) **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Some camera's may have many readout ports, which are output nodes from which a pixel stream can be read from. For more information about readout ports, refer to the Port and Speed Choices section inside the PVCAM User Manual.
speed_table_index	(Getter and Setter) Returns/changes the current numerical index of the speed table of a camera. See the Port and Speed Choices section inside the PVCAM User Manual for a detailed explanation about PVCAM speed tables.
speed_table	(Getter only) Returns a dictionary containing the speed table, which gives information such as bit depth and pixel time for each readout port and speed index.
pp_table	(Getter only) **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns a dictionary containing the current post-processing settings. It is setup to be accessed by the feature, the parameters for each feature, then a tuple for each parameter containing the current, the minimum, and maximum values.
trigger_table	(Getter only) Returns a dictionary containing a table consisting of information of the last acquisition such as exposure time, readout time, clear time, pre-trigger delay, and post-trigger delay. If any of the parameters are unavailable, the dictionary item will be set to 'N/A'.
adc_offset	(Getter only) **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns the camera's current ADC offset value. Only CCD camera's have ADCs (analog-to-digital converters).
gain	(Getter and Setter) Returns/changes the current gain index for a camera. A ValueError will be raised if an invalid gain index is supplied to the setter.

binning	(Getter and Setter) Returns/ changes the current serial and parallel binning values in a tuple. The setter can either a tuple for the binning (x, y) or a single value and will set a square binning with the given number, for example cam.binning = x makes cam.__binning = (x, x). Binning cannot be changed directly on the camera; but is used for setting up acquisitions and returning correctly shaped images returned from get_frame and get_live_frame. The setter has built in checking to see that the given binning it able to be used later.
bin_x	(Getter and Setter) Returns/ changes the current serial binning value. **Deprecated, use binning above**
bin_y	(Getter and Setter) Returns/ changes the current parallel binning value. **Deprecated, use binning above**
roi	(Getter and Setter) Returns/ changes the current region of interest (ROI). This is used for single ROI captures. The setter expects a tuple of integers in the following order: (x_start, x_end, y_start, y_end). The setter also validates the input by checking if the x and y lengths are greater than 0 but less than the sensor's serial and parallel size, respectively.
shape	(Getter only) Returns the reshape factor to be used when acquiring an image. See calculate_reshape. This is equivalent to an acquired images shape.
last_exp_time	(Getter only) Returns the last exposure time the camera used for the last successful non-variable timed mode acquisition in what ever time resolution it was captured at.
exp_res	(Getter and setter) Returns/changes the current exposure resolution of a camera. Note that exposure resolutions have names, but PVCAM interprets them as integer values. When called as a getter, the integer value will be returned to the user. However, when changing the exposure resolution of a camera, either the integer value or the name of the resolution can be specified. Refer to constants.py for the names of the exposure resolutions.
exp_res_index	(Getter only): Returns the current exposure resolution index.
exp_time	(Getter and Setter): Returns/ changes the exposure time the camera will use if not given an exposure time. It is recommended to modify this value to modify your acquisitions for better abstraction.

exp_mode	(Getter and Setter): Returns/ changes the current exposure mode of the camera. Note that exposure modes have names, but PVCAM interprets them as integer values. When called as a getter, the integer value will be returned to the user. However, when changing the exposure mode of a camera, either the integer value or the name of the expose out mode can be specified. Refer to constants.py for the names of the exposure modes.
exp_out_mode	(Getter and Setter): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns/ changes the current expose out mode of the camera. Note that expose out modes have names, but PVCAM interprets them as integer values. When called as a getter, the integer value will be returned to the user. However, when changing the expose out mode of a camera, either the integer value or the name of the expose out mode can be specified. Refer to constants.py for the names of the expose out modes.
vtm_exp_time	(Getter and Setter): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns/ changes the variable timed exposure time the camera uses for the "Variable Timed" exposure mode.
clear_mode	(Getter and Setter): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns/changes the current clear mode of the camera. Note that clear modes have names, but PVCAM interprets them as integer values. When called as a getter, the integer value will be returned to the user. However, when changing the clear mode of a camera, either the integer value or the name of the clear mode can be specified. Refer to constants.py for the names of the clear modes.
temp	(Getter only): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns the current temperature of a camera in Celsius.

temp_setpoint	(Getter and Setter): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns/changes the camera's temperature setpoint. The temperature setpoint is the temperature that a camera will attempt to keep it's temperature (in Celsius) at.
readout_time	(Getter only): Returns the last acquisition's readout time as reported by the camera in microseconds.
clear_time	(Getter only): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be** raised. Returns the last acquisition's clearing time as reported by the camera in microseconds.
pre_trigger_delay	(Getter only): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns the last acquisition's pre-trigger delay as reported by the camera in microseconds.
post_trigger_delay	(Getter only): **Warning: Camera specific setting. Not all camera's support this attribute. If an unsupported camera attempts to access it's readout_port, an AttributeError will be raised.** Returns the last acquisition's post-trigger delay as reported by the camera in microseconds.

constants.py

constants.py is a large data file that contains various camera settings and internal PVCAM structures used to map meaningful variable names to predefined integer values that camera firmware interprets as settings.

This file is not (and should never be) constructed by hand. Instead, use the most recent version of pvcam.h and run constants_generator.py to build/rebuild this file. A more detailed explanation can be found under the constants_generator.py section, but the basic concept is that pvcam.h is parsed line by line, finding all predefined constants, enums, and structs to grant Python users access to the necessary data to perform basic PVCAM functions that have multiple settings.

There are four main sections to this file that are found following a formatted comment like this ### <SECTION_NAME> ###:

1. Defines
    1. Much of the necessary data from pvcam.h is saved as C preprocessor macros which are easy to strip from the file and construct a Python variable who's name is the macros name and the value is what the macro expands to.
    2. Macros can be thought of as Python variables in this case, as none (of the necessary macros) in pvcam.h expand to more than a literal.
2. Enums
    1. Enums (also known as **enumerated types**) are data types that contain named members used to identify certain integer values. Typically, if no values are specified, the first member will be assigned the value 0, and the successor will be 1+ the value of their predecessor. However, you can specify specific numbers for all members.
    2. Vanilla Python has no enum type (it must be imported; see documentation here), and even still Python's enum class behaves somewhat differently in terms of accessing members. Instead, we will insert a comment above all enumerated types and have their members just be simple Python variables who's values where the integer assigned to them in the pvcam.h file.
3. Structs
    1. While not used (yet), structs are preserved as Python classes. No testing/implementation has been done with these, so results may be unexpected if implemented.
4. Added By Hand
    1. These are quality of life/readability dictionaries that map named settings of various camera modes to their pvcam.h integer value. These allow for fast look-up and intuitive setting changes for end users.

pvcmodule.cpp

pvcmodule.cpp is a set of C++ functions that make use of and extend the Python C-API known as a Python Extension Module. The need for a Python extension module is two-fold: first to allow communication between the *static* PVCAM library and Python scripts, and second for fast acquisition and conversion from native C types (namely C arrays of pixel data) to Python data types.

The extension module needs to be compiled, so it will be necessary to have a C/C++ compiler to successfully install this application. The module will be compiled into a shared-object library, which can then be imported from Python; read more here.

General Structure of a pvcmodule Function

The functions for a pvcmodule function usually follow a three step process:

1. Retrieve data/query from Python script
2. Process acquired data
3. Return data to Python script

Retrieving Data

Functions receive data dynamically through use of parameters, and the pvcmodule's functions are no different. However, the Python API states that all data is of type PyObject, which the C/C++ programming language offer no builtin support for. In addition to, each Python-facing function must only have two arguments: PyObject *self (a pointer to the instance of whichever Python object called this C function) and PyObject *args (a Python tuple object that contains all of the arguments passed into the C function call). However, we can make use of the PyArg_ParseTuple (see example here) function from the Python API to easily coerce the Python objects from the args tuple to their respective C type. In order for the conversion to occur, we must specify which type we want to coerce each Python argument to using a formatted string (see second argument for PyArg_ParseTuple). Each character in the formatted string are known as "format units" and are interpreted in the same order that the variables for the coerced C data are provided. Find below a small list of C data types and their corresponding format units.

C Type	Character Representation
long	l
int	i
double	d
float	f
string (char *)	s
PyObject	O

Arguments of PyArg_ParseTuple

1. args (PyObject *) A Python tuple object that contains the arguments from the Python function call. For example, if a function call from Python is made: `my_c_func(1, "test")`, the *args* tuple would contain two PyObject pointers: one to the Python integer 1 and another to the Python Unicode-String "test".
2. format (char *) A String containing the format units for all of the arguments found in the *args* in the same order in which they appear in the tuple. Going off of the example from the previous argument, the desired formatted string would be `"is"`: 'i' for the integer 1, and 's' for the string "test".

In addition to these two arguments, addresses to the variables in which the coerced C data should be stored must also be passed as arguments to the `PyArg_ParseTuple` call. (See example for more details).

PyArg_ParseTuple Example

```
static PyObject *example(PyObject *self, PyObject *args)
{
    int myNum;
    char *myString;
```

```
    PyArg_ParseTuple(args, "is", &myNum, &myString);
    printf("myNum: %d\n", myNum);         // Prints "myNum: 1"
    printf("myString: %s\n", myString);  // Prints "myString: test"
    Py_RETURN_NONE;
}
```

Processing Acquired Data

Using the data supplied by the Python function call, we can now perform normal camera operations using PVCAM library function calls. The most common form of processing acquired data is to read the camera handle from the arguments provided, then performing a camera operation (changing/reading settings, getting images, etc.) using the acquired handle to identify which camera to perform the action on.

Generally speaking, this part of the function should be very similar to writing normal C/C++ modules that use the PVCAM library. If there is any confusion about how to write C/C++ code to make calls to PVCAM, refer to the *PvcamCodeSamples* found in the *Examples* directory of the PVCAM SDK.

Sometimes, processing data from a Python function call may entail answering a query. If this is the case, we need to specify what to return, and how to convert it into a corresponding Python type.

Return Data to a Python Script

Similar to how issues arose when passing data from the Python function call to the C/C++ module, there is no simple casting solution to convert C/C++ data types to Python data types when returning from a function.

Thankfully, there are some functions that were included in the Python header file included at the top of each module to allow us to cast data to an equivalent Python type.

Cast to Python Type

```
static PyObject *example(PyObject *self, PyObject *args)
{
    char *myString = "ika";
    return PyUnicode_FromString(myString); // Returns a Python string back to
the calling function.
}
```

There is one small catch, however. All Python functions must return an object; there is no such thing as a "void" function. This means that we must always return something in our C/C++ modules as well (which we can tell by looking at the signature!)

If you wish to return None, simply use the Py_RETURN_NONE macro (see the PyArg_ParseTuple example for a visual representation).

Functions of pvcmodule.cpp

Note: All functions will always have the `PyObject *self` and `PyObject *args` parameters. When parameters are listed, they are the Python parameters that are passed into the module.

Function Name	Description
set_g_msg	Helper function that when called, will set the global error message to whatever the error of the last PVCAM error message was. Used before raising a Python Error.
pvc_get_pvcam_version	Returns a Python Unicode String of the current PVCAM version.
pvc_init_pvcam	Initializes the PVCAM library. Returns True upon success, RuntimeError is raised otherwise.
pvc_uninit_pvcam	Uninitializes the PVCAM library. Returns True upon success, RuntimeError is raised otherwise.
pvc_get_cam_total	Returns the total number of cameras currently attached to the system as a Python integer.
pvc_get_cam_name	Given a Python integer corresponding to a camera handle, returns the name of the camera with the associate handle. **Parameters:** 1. Python integer (camera handle).
pvc_open_camera	Given a Python string corresponding to a camera's name, open the camera. Returns True upon success. ValueError is raised if invalid parameter is supplied. RuntimeError raised otherwise. **Parameters:** 1. Python string (camera name).
pvc_close_camera	Given a Python string corresponding to a camera's name, close the camera. Returns True upon success. ValueError is raised if invalid parameter is supplied. RuntimeError raised otherwise. **Parameters:** 1. Python string (camera name).

pvc_get_param	Given a camera handle, a parameter ID, and the attribute of the parameter in question (AVAIL, CURRENT, etc.) return the value of the parameter at the current attribute. **Note: This setting will only return a Python int or a Python string. Currently no other type is supported, but it is possible to extend the function as needed.** ValueError is raised if invalid parameters are supplied. AttributeError is raised if camera does not support the specified parameter. RuntimeError is raised otherwise. **Parameters:** 1. Python int (camera handle). 2. Python int (parameter ID). 3. Python int (parameter attribute).
pvc_set_param	Given a camera handle, a parameter ID, and a new value for the parameter, set the camera's parameter to the new value. ValueError is raised if invalid parameters are supplied. AttributeError is raised when attempting to set a parameter not supported by a camera. RuntimeError is raised upon failure. **Parameters:** 1. Python int (camera handle). 2. Python int (parameter ID). 3. Generic Python value (any type) (new value for parameter).

pvc_get_frame	Given a camera and a region, return a Python numpy array of the pixel values of the data. Numpy array returned on success. ValueError raised if invalid parameters are supplied. MemoryError raised if unable to allocate memory for the camera frame. RuntimeError raised otherwise. **Parameters:** 1. Python int (camera handle). 2. Python int (first pixel of serial register). 3. Python int (last pixel of serial register). 4. Python int (serial binning). 5. Python int (first pixel of parallel register). 6. Python int (last pixel of parallel register). 7. Python int (parallel binning). 8. Python int (exposure time). 9. Python int (exposure mode).
pvc_set_exp_modes	Given a camera, exposure mode, and an expose out mode, change the camera's exposure mode to be the bitwise-or of the exposure mode and expose out mode parameters. ValueError is raised if invalid parameters are supplied including invalid modes for either exposure mode or expose out mode. RuntimeError is raised upon failure. **Parameters:** 1. Python int (camera handle). 2. Python int (exposure mode). 3. Python int (expose out mode).
valid_enum_param	Helper function that determines if a given value is a valid selection for an enumerated type. Should any PVCAM function calls in this function fail, a "falsy" value wil be returned. "Truthy" is returned if it is a valid enumerated value for a parameter. **Note: This function is not exposed to Python, so no Python parameters are required.** **Parameters:** 1. hcam (int16): The handle of the camera. 2. param_id (uns32): The parameter ID. 3. selected_val (int32): The value to check if it is a valid selection.

pvc_read_enum	Function that when given a camera handle and a enumerated parameter will return a list mapping all valid setting names to their values for the camera. ValueError is raised if invalid parameters are supplied. AttributeError is raised if an invalid setting for the camera is supplied. RuntimeError is raised upon failure. A Python list of dictionaries is returned upon success. **Parameters:** 1. Python int (camera handle). 2. Python int (parameter ID).

The Method Table

All functions that need to be exposed to Python programs need to be included in the method table. The method table is partially responsible for allowing Python programs to call functions from an extension module. It does this by creating a list of PyMethodDef structs with a sentinel struct at the end of the list. The list of method definitions are then passed to the PyModuleDef struct, which contains the necessary information to construct the module.

The method table is a list of PyMethodDef structs, which have the following four fields:

Field Name	C Type	Description
ml_name	char *	Name of the method (when called from Python)
ml_meth	PyCFunction	Pointer to the C implementation of the function in the module.
ml_flags	int	Flag bits indicating how the call to the function should be constructed
ml_doc	char *	Points to the contents of the docstring for the method.

All docstrings for the functions inside of pvcmodule.cpp are statically defined in the pvcmodule.h file.

The Module Definition

The PyModuleDef structure contains all of the information required to create the top-level module object.

Field Name	C Type	Description

m_base	PyModuleDef_Base	Always initialize this member to PyModuleDef_HEAD_INIT
m_name	char *	Name for the new module (must match the name in the Module Init function).
m_doc	char *	Docstring for the module (statically defined in the header file)
m_size	Py_ssize_t	Specifies the additional amount of memory a module requires for it's "state". Only needed if running in sub-interpreters; otherwise set to -1, signifying that the module does not support sub-interpreters because it has global state.
m_methods	PyMethodDef*	A pointer to the method table. Can be NULL if no functions are present.

After creating the module definition structure, it can then be passed into the module creation function.

Module Creation

The module initialization function will create and return the module object directly.

To initialize a module, write the PyInit_{modulename} function, which calls and returns the value of PyModule_Create. See example below:

Creating Extension Module

```
PyMODINIT_FUNC
PyInit_pvc(void)
{
    return PyModule_Create(&pvcmodule);
}
```

constants_generator.py

The purpose of the constants_generator.py file is to easily construct a new constants.py data file should the file become tainted or a new version of PVCAM is released.

The script targets three main parts of the header file: the predefined macros, the enums, and the structs.

Requirements

The constants generator targets the install location of the PVCAM SDK on your machine, meaning that the script will fail to run if you do not have the SDK installed.

Running the Script

In order to run the script, ensure that you are running it from /PyVCAM/pyvcam/src/, or else it will fail to find the correct directory to write the generated constants.py file to.

```
The script can be run using the following command when you are in the correct directory: python constants_generator.py
```

tests

The tests directory contains unit tests to ensure the quality of the code of the module and to also include some basic examples on how to perform basic operations on a camera.

change_settings_test.py (needs camera_settings.py)

change_settings_test.py is used to show one way of keeping camera settings in one file and importing them to update a camera's settings in another file.

This allows the user to quickly change the settings they wish to test on a camera without having to dig through a large testing script and manually changing the settings within it.

Note: camera_settings.py needs to be included in the same directory in order to run this test.

single_image_polling.py

single_image_polling.py is used to demonstrate how to collect single frames from a camera, starting from the detection and opening of an available camera to calling the get_frame function.

Note that this test does not display the frame; only saves it locally to a variable and prints a few pixel points from it. If you want an example of how to quickly display a frame, see single_image_polling_show.py.

single_image_polling_show.py

single_image_polling_show.py is used to demonstrate how to collect a single frame from a camera and use matplotlib's pyplot subpackage in order to display the captured frame.

Note: The test reverses the camera's sensor size when reshaping the array. This is because the camera sensor size tuple is row x column, and the shape of a numpy array is specified by column x row.

test_camera.py

test_camera.py contains the unit tests for this module. It tests the getting, setting, and edge cases of all available settings.

All unit tests can be run from the command line using the command `python -m unittest discover`

setup.py

setup.py is the installer script for this module. Once the package has been downloaded from SVN, navigate the the src directory to run the setup script.

Variables

Variable	Description
packages	List that contains the names of all of the package(s) that will be built with the setup script. In our case, there is only one package.
package_dir	Dictionary that maps the name of the package from the packages list to the directory that contains their source code. Again, this will only have a single member since we only have one package.
pvc_modules	List containing the names of the modules contained within the package. In our case, we only have two Python modules: constants.py and camera.py
include_dirs	List of directories that contain necessary files to include for the python extension module. For example; the pvcam.h file needs to be included, and so does the numpy.h file.
lib_dirs	List of directories that contain necessary libraries needed to compile the python extension module.
libs	List of libraries that can be found from the lib_dirs list that are needed for the compilation of the python extension module.
ext_module	List of Extension objects that model a Python extension module that include: 1. The name of the module (pyvcam.pvc) 2. Where the source code is located 3. All necessary include directories 4. All necessary library directories 5. All necessary libraries

Installing the Package

When you are ready to install the package, navigate to the directory that contains setup.py and run:

setup.py Install Command

```
python setup.py install
```