thorlabs_tsi_sdk

Rev. D 2020-12-9 ITN003996-D01 0.0.7

Thorlabs Scientific Imaging

Jan 04, 2021

CONTENTS:

Use the following guides to quickly get started programming Thorlabs scientific cameras in Python.

CHAPTER

ONE

CAMERA PROGRAMMING GUIDE

This guide descibes how to program Thorlabs scientific cameras using Python.

- Create an instance of TLCameraSDK by calling the *TLCameraSDK()* constructor. This will initialize and hold open the Camera SDK, so be sure to only create one at a time.
- Use *thorlabs_tsi_sdk.tl_camera.TLCameraSDK.discover_available_cameras()* to get a list of the connected cameras.
- Use *TLCameraSDK.open_camera("serial number")* and pass in the serial number of the camera you want to open. This will return a *TLCamera* object. To open the first camera, you can simply pass in the first item of the camera list obtained in the previous step.
- Set camera properties like *exposure_time_us*, *roi*, and *operation_mode*.
- Adjust *frames_per_trigger_zero_for_unlimited* to set the number of number of frames you want to acquire during each trigger. When *frames_per_trigger_zero_for_unlimited* is equal to 0, the camera will be set to continuous mode. This means on the next trigger the camera will continually acquire frames until *disarmed*. When *frames_per_trigger_zero_for_unlimited* is greater than 0, the camera will acquire that many frames per trigger.
- Prepare the camera to accept hardware or software triggers by calling *TLCamera.arm()*.
- For software triggering, use *TLCamera.issue_software_trigger()* to begin acquisition. For hardware triggering, start sending the hardware trigger signal.
- To poll for an image (see *Frame*), use *thorlabs_tsi_sdk.tl_camera.TLCamera.get_pending_frame_or_null()*.
- If you're using a color camera, you may want to do color processing here. See the *tl_mono_to_color_processor section* for a quick guide to setting up a mono to color processor.
- When finished triggering, call *TLCamera.disarm()*.
- Call *TLCamera.dispose()* to clean up the camera resources.
- Finally, call *TLCameraSDK.dispose()* to clean up and close the SDK resources.

CHAPTER

TWO

MONO TO COLOR PROGRAMMING GUIDE

The following is a quick setup guide for using a mono to color processor to get color images from monochrome color-filtered image data.

- Create an instance of the MonoToColorProcessorSDK by calling the *MonoToColorProcessorSDK()* constructor. This will initialize and hold open the MonoToColorProcessor SDK, so be sure to only create one at a time.
- Create an instance of a MonoToColorProcessor by calling the SDK object's *create_mono_to_color_processor()* method. This method requires several initialization parameters to be passed in, all of which can be queried from a *TLCamera* object:
  - The camera sensor type can be found using *TLCamera.camera_sensor_type*
  - The color filter array phase can be found using *TLCamera.color_filter_array_phase*
  - The color correction matrix can be found using *TLCamera.get_color_correction_matrix()*
  - The default white balance matrix can be found using *TLCamera.get_default_white_balance_matrix()*
  - The bit depth can be found using *TLCamera.bit_depth*
- There are some additional properties that can be set after construction: `color_space` and *output_format*. The color space can be left at the default sRGB to get good-looking images (see *COLOR_SPACE*), however the output format might need to be changed depending on your application. This will determine how the color image data is structured, see *FORMAT* for more details.
- To adjust the color balance of the output image, you can adjust the Red, Blue, and Green gain values using the *red_gain*, *blue_gain*, and *green_gain* properties.
- To transform unprocessed image data to color, the image width and height will also need to be found. The properties *TLCamera.image_width_pixels* and *TLCamera.image_height_pixels* can be used to get this data, however these properties will query that information from the camera, which may take some time. To prevent slowing down color processing, it is recommended to save the value of these properties locally after arming and use the local variables instead. It is safe to record these values after arming since the image width and height cannot be changed while the camera is armed.
- Reminder of how to get image data: When a *TLCamera* instance is *armed* and *triggered*, *poll* for a *Frame* from the camera. The Frame's *image buffer* can be handed off directly to the transform functions in the next step to be color processed.
- When an image needs to be color processed, pass the unprocessed image data to one of the following transform_to_X functions, depending on the desired bit depth of the colored image:
  - *transform_to_48* will return a colored image in 16 bits per channel and 3 channels per pixel, resulting in 48 bits per pixel.

  - *transform_to_32* will return a colored image in 8 bits per channel and 4 channels per pixel, resulting in 32 bits per pixel. The fourth channel is a byte padding that may be used as an Alpha channel.
  - *transform_to_24* will return a colored image in 8 bits per channel and 3 channels per pixel, resulting in 24 bits per pixel.

- When you are finished with a MonoToColorProcessor, cleanly release its resources by calling *dispose()*.
- When you are finished with the MonoToColorProcessorSDK, cleanly release its resources by calling *dispose()*.

CHAPTER

THREE

POLARIZATION PROGRAMMING GUIDE

The following is a quick setup guide for using a polarization processor to get polarized images from monochrome image data.

- Create an instance of the PolarizationProcessorSDK by calling the *PolarizationProcessorSDK()* constructor. This will initialize and hold open the PolarizationProcessor SDK, so be sure to only create one at a time.
- Create an instance of a PolarizationProcessor by calling the SDK object's *create_polarization_processor()* method. There can be multiple instances of this object.
- To transform unprocessed image data to polarized data, the image width and height will also need to be found. The properties *TLCamera.image_width_pixels* and *TLCamera.image_height_pixels* can be used to get this data, however these properties will query that information from the camera, which may take some time. To prevent slowing down color processing, it is recommended to save the value of these properties locally after arming and use the local variables instead. It is safe to record these values after arming since the image width and height cannot be changed while the camera is armed.
- Reminder of how to get image data: When a *TLCamera* instance is *armed* and *triggered*, `poll` for a *Frame* from the camera. The Frame's *image buffer* can be handed off directly to the transform functions in the next step to be color processed.
- When an image needs to be processed, pass the unprocessed image data to one of the following transform_to_X functions, depending on the desired output:
  - *transform_to_intensity* will return an image corresponding to the brightness of the light at each pixel.
  - *transform_to_azimuth* will return an image that shows the angle of polarized light at each pixel.
  - *transform_to_dolp* will return an image that shows the degree of linear polarization at each pixel.
- When you are finished with a PolarizationProcessor, cleanly release its resources by calling *dispose()*.
- When you are finished with the PolarizationProcessorSDK, cleanly release its resources by calling *dispose()*.

CHAPTER

FOUR

WORKING WITH DLLS

Scripts utilizing this package need visibility to the native Camera SDK dlls. Here are a few ways to accomplish this:

- Copy the native DLLs to the working directory of your python script.
- Utilize the ‘os’ module in the standard Python library. This library can be used to append the DLL directory to the PATH variable during runtime (only for your application, doesn’t change system PATH variable), or directly change the current working directory to the DLL folder. See the Python docs for more information.
- Manually modify your system PATH environment variable to include the directory containing the DLLs.

CHAPTER

FIVE

ERROR REPORTING

The Python SDK will throw TLCameraError exceptions whenever the camera reports an issue. These errors are generally caused by invalid camera operation, such as unplugging a camera while it is capturing images or setting an ROI while the camera is armed. The Mono To Color SDK will throw MonoToColorError exceptions whenever the underlying native color processing DLLs encounter an error. One example that would generate a MonoToColorError would be trying to initialize a MonoToColorProcessor using a Monochrome camera. The Polarization SDK will throw a PolarizationError whenever the underlying polarization library returns an error.

These exceptions try to acquire as much information as they can from the low-level DLLs when constructing error messages. These messages will often be helpful to understanding why a specific behavoir might be generating an exception.

For users who use Python's logging module, this SDK utilizes loggers called 'thorlabs_tsi_sdk.tl_camera', 'thorlabs_tsi_sdk.tl_mono_to_color_processor', and 'thorlabs_tsi_sdk.polarization_processor', which correspond to the tl_camera, tl_mono_to_color_processor, and tl_polarization_processor modules respectively. To add custom handlers, use logging.GetLogger() with these names to get access to the internal loggers. By default these loggers only output a message to the console for logging level ERROR or above. Checkout the logging module in the Python docs to learn more about loggers.

CHAPTER

SIX

API

6.1 thorlabs_tsi_sdk package

6.1.1 Submodules

6.1.1.1 thorlabs_tsi_sdk.tl_camera module

tl_camera.py

class thorlabs_tsi_sdk.tl_camera.**Frame**(*image_buffer*, *frame_count*, *time_stamp_relative_ns_or_null*)

Bases: object

Holds the image data and frame count returned by polling the camera for an image.

property image_buffer

Numpy array of pixel data. This array is temporary and may be invalidated after *polling for another image*, *rearming the camera*, or *closing the camera*.

Type np.array(dtype=np.ushort)

property frame_count

Frame number assigned to this image by the camera.

Type int

property time_stamp_relative_ns_or_null

Time stamp in nanoseconds relative to an internal counter mechanism. The timestamp is recorded immediately following the exposure time. It is calculated by taking the pixel clock and dividing it by a model-specific clock base frequency. This value can be used to find the time in nanoseconds between frames (Frame_2.time_stamp_relative_ns_or_null - Frame_1.time_stamp_relative_ns_or_null). If the camera does not support time stamps, then this value will be None.

Type int

class thorlabs_tsi_sdk.tl_camera.**Range**(*min*, *max*)

Bases: tuple

Represents a range of values with a min and max. These objects are derived from NamedTuple and follow tuple semantics.

max: Any

min: Any

class thorlabs_tsi_sdk.tl_camera.**ROI**(*upper_left_x_pixels*, *upper_left_y_pixels*, *lower_right_x_pixels*, *lower_right_y_pixels*)

Bases: tuple

Represents the Region of Interest used by the camera. ROI is represented by two (x, y) coordinates that specify an upper left coordinate and a lower right coordinate. The camera will create a rectangular ROI based on these two points. These objects are derived from NamedTuple and follow tuple semantics.

lower_right_x_pixels: int

lower_right_y_pixels: int

upper_left_x_pixels: int

upper_left_y_pixels: int

class thorlabs_tsi_sdk.tl_camera.**ROIRange**(*upper_left_x_pixels_min*, *upper_left_y_pixels_min*, *lower_right_x_pixels_min*, *lower_right_y_pixels_min*, *upper_left_x_pixels_max*, *upper_left_y_pixels_max*, *lower_right_x_pixels_max*, *lower_right_y_pixels_max*)

Bases: tuple

Represents the range of the Region of Interest used by the camera. ROI is represented by two (x, y) coordinates that specify an upper left coordinate and a lower right coordinate. This class contains 8 fields corresponding to the minimums and maximums of the x and y components for both points. These objects are derived from NamedTuple and follow tuple semantics.

lower_right_x_pixels_max: int

lower_right_x_pixels_min: int

lower_right_y_pixels_max: int

lower_right_y_pixels_min: int

upper_left_x_pixels_max: int

upper_left_x_pixels_min: int

upper_left_y_pixels_max: int

upper_left_y_pixels_min: int

class thorlabs_tsi_sdk.tl_camera.**TLCameraSDK**

Bases: object

The SDK object that is used to create TLCamera objects. There must be only one instance of this class active at a time. Use the *dispose()* method to destroy an SDK instance before creating another instance. *with* statements can also be used with this class to automatically dispose the SDK.

dispose() → None

Cleans up the TLCameraSDK instance - make sure to call this when you are done with the TLCameraSDK instance. If using the *with* statement, dispose is called automatically upon exit.

discover_available_cameras() → List[str]

Returns a list of all open cameras by their serial number string.

open_camera(*camera_serial_number: str*) → *thorlabs_tsi_sdk.tl_camera.TLCamera*

Opens the camera with given serial number and returns it as a TLCamera instance.

Parameters **camera_serial_number** (*str*) – The serial number of the camera to open.

Returns *TLCamera*

set_camera_connect_callback(*handler: Callable[[str, thorlabs_tsi_sdk.tl_camera_enums.USB_PORT_TYPE, Optional[Any], Optional[Any]], None]*, **args: Optional[Any]*, ***kwargs: Optional[Any]*) → None

Sets the callback function for camera connection events. Whenever a USB camera is connected, the provided handler will be called along with any specified arguments and keyword arguments.

Parameters

- **handler** (Callable[[str, *USB_PORT_TYPE*, Optional[Any], Optional[Any]], type(None)]) – Any method with a signature that conforms to this type. It will be called when a USB camera is connected.
- **args** (*Optional[Any]*) – Optional arguments that are forwarded to the handler when it is called.
- **kwargs** (*Optional[Any]*) – Optional keyword arguments that are forwarded to the handler when it is called.

set_camera_disconnect_callback(*handler: Callable[[str, Optional[Any], Optional[Any]], None]*, **args: Optional[Any]*, ***kwargs: Optional[Any]*) → None

Sets the callback function for camera disconnection events. Whenever a USB camera is disconnected, the provided handler will be called along with any specified arguments and keyword arguments

Parameters

- **handler** (*Callable[[str, Optional[Any], Optional[Any]], type(None)]*) – Any method with a signature that conforms to this type. It will be called when a USB camera is disconnected.
- **args** (*Optional[Any]*) – Optional arguments that are forwarded to the handler when it is called.
- **kwargs** (*Optional[Any]*) – Optional keyword arguments that are forwarded to the handler when it is called.

class thorlabs_tsi_sdk.tl_camera.**TLCamera**(*key: type*, *sdk: Any*, *camera: Any*)

Bases: object

Used to interface with a Thorlabs camera. These objects can adjust camera settings and retrieve images. When finished with a camera, call its *dispose* method to clean up any opened resources. These objects can be managed using *with* statements for automatic resource clean up. These objects can only be created by calls to *TLCameraSDK.open_camera*.

dispose() → None

Cleans up the TLCamera instance - make sure to call this when you are done with the camera. If using the *with* statement, dispose is called automatically upon exit.

get_pending_frame_or_null() → Optional[*thorlabs_tsi_sdk.tl_camera.Frame*]

Polls the camera for an image. This method will block for at most *image_poll_timeout* milliseconds. The Frame that is retrieved will have an image_buffer field to access the pixel data. This image_buffer is only valid until the next call to *get_pending_frame_or_null()* or until disarmed. If image data is needed for a longer period of time, use *np.copy(image_buffer)* to create a deep copy of the data.

Returns *Frame* or None if there is no pending frame

get_measured_frame_rate_fps() → float

Gets the current rate of frames that are delivered to the host computer. The frame rate can be affected by

the performance capabilities of the host computer and the communication interface. This method can be polled for updated values as needed.

Returns float - The current frame rate as measured by the camera SDK.

get_is_data_rate_supported(*data_rate:* thorlabs_tsi_sdk.tl_camera_enums.DATA_RATE) → bool

Scientific-CCD cameras and compact-scientific cameras handle sensor- level data-readout speed differently. Use this method to test whether the connected camera supports a particular data rate. For more details about the data rate options, see the *data_rate* property.

Param data_rate (*DATA_RATE*) - The data rate value to check.

Returns bool - True if the given data rate is supported by the connected camera, false if it is not

get_is_taps_supported(*tap:* thorlabs_tsi_sdk.tl_camera_enums.TAPS) → bool

All CCD cameras support a single tap. Some also support dual tap or quad tap. Use this method to test whether a connected camera supports a particular Taps value. For more information on taps and tap balancing, see is_tap_balance_enabled - *Taps not yet supported by Python SDK*.

Param tap (*TAPS*) - The tap value to check.

Returns bool - True if the connected camera supports the given taps mode, false if not.

get_is_operation_mode_supported(*operation_mode:* thorlabs_tsi_sdk.tl_camera_enums.OPERATION_MODE) → bool

This method can be used to determine if a camera has the ability to perform hardware triggering. Some cameras, such as the zelux, have both triggered and non-triggered models.

Param operation_mode (*OPERATION_MODE*) - The operation mode to check.

Returns bool - True if the connected camera supports the given operation mode, false if not.

get_color_correction_matrix() → numpy.array

Each scientific color camera includes a three-by-three matrix that can be used to achieve consistent color for different camera modelsGet the default color correction matrix for this camera. This can be used with the *MonoToColorProcessor* when color processing an image.

Returns np.array

get_default_white_balance_matrix() → numpy.array

Get the default white balance matrix for this camera. Each scientific color camera includes a three-by-three matrix that corrects white balance for the default color temperature. This can be used with the *MonoToColorProcessor* to provide a default white balance to an image.

Returns np.array

arm(*frames_to_buffer: int*) → None

Before issuing software or hardware triggers to get images from a camera, prepare it for imaging by calling *arm()*. Depending on the *operation_mode*, either call *issue_software_trigger()* or issue a hardware trigger. To start a camera in continuous mode, set the *operation_mode* to SOFTWARE_TRIGGERED, *frames per trigger* to zero, Arm the camera, and then call *issue_software_trigger()* one time. The camera will then self-trigger frames until *disarm()* or *dispose()* is called. To start a camera for hardware triggering, set the *operation_mode* to either HARDWARE_TRIGGERED or BULB, *frames per trigger* to one, *trigger_polarity`* to rising-edge or falling-edge triggered, arm the camera, and then issue a triggering signal on the trigger input. If any images are still in the queue when calling *arm()*, they will be considered stale and cleared from the queue. For more information on the proper procedure for triggering frames and receiving them from the camera, please see the Getting Started section.

issue_software_trigger() → None
If the *operation_mode* is set to SOFTWARE_TRIGGERED and *arm()* is called, then calling this method will generate a trigger through the camera SDK rather than through the hardware trigger input.

The behavior of a software trigger depends on the *frames_per-trigger_zero_for_unlimited* property:

- If *frames_per-trigger_zero_for_unlimited* is set to zero, then a single software trigger will start continuous-video mode.
- If *frames_per-trigger_zero_for_unlimited* is set to one or higher, then one software trigger will generate a corresponding number of frames.

Multiple software triggers can be issued before calling *disarm()*.

IMPORTANT: For scientific-CCD cameras, after issuing a software trigger, it is necessary to wait at least 300ms before adjusting the *exposure_time_us* property.

disarm() → None
When finished issuing software or hardware triggers, call *disarm()*. This allows setting parameters that are not available in armed mode such as *roi* or *operation_mode*. The camera will automatically disarm when *disarm()* is called. Disarming the camera does not clear the image queue – polling can continue until the queue is empty. When calling *disarm()* again, the queue will be automatically cleared.

convert_gain_to_decibels(*gain: int*) → float
Use this method to convert the gain from the *gain* property into units of Decibels (dB).

Returns float

convert_decibels_to_gain(*gain_db: float*) → int
Use this method to convert the gain (in decibels) from the *convert_gain_to_decibels* method back into a gain index. (dB).

Returns int

property exposure_time_us
The time, in microseconds (us), that charge is integrated on the image sensor.

To convert milliseconds to microseconds, multiply the milliseconds by 1,000. To convert microseconds to milliseconds, divide the microseconds by 1,000.

IMPORTANT: After issuing a software trigger, it is recommended to wait at least 300ms before setting exposure.

Type int

property image_poll_timeout_ms
get_pending_frame_or_null() will block up to this many milliseconds to get an image. If the SDK could not get an image within the timeout, None will be returned instead.

Type int

property exposure_time_range_us
Range of possible exposure values in microseconds. This property is Read-Only.

Type *Range*

property firmware_version
String containing the version information for all firmware components. This property is Read-Only.

Type str

property frame_time_us
The time, in microseconds (us), required for a frame to be exposed and read out from the sensor. When

triggering frames, this property may be used to determine when the camera is ready to accept another trigger. Other factors such as the communication speed between the camera and the host computer can affect the maximum trigger rate.

IMPORTANT: Currently, only scientific CCD cameras support this parameter. This property is Read-Only.

Type int

property trigger_polarity
When the *operation_mode* is set to HARDWARE_TRIGGERED or BULB and then *arm()* is called, the camera will respond to a trigger input as a signal to begin exposure. Setting trigger polarity tells the camera to begin exposure on either the rising edge or falling edge of the trigger signal.

Type *TRIGGER_POLARITY*

property binx
The current horizontal binning value.

Type int

property sensor_readout_time_ns
The time, in nanoseconds (ns), that readout data from image sensor. This property is Read-Only.

Type int

property binx_range
The binning ratio in the X direction can be determined with this property. By default, binning is set to one in both X and Y directions. This property is Read-Only. To set binx, see *binx*.

Type *Range*

property is_hot_pixel_correction_enabled
Due to variability in manufacturing, some pixels have inherently higher dark current which manifests as abnormally bright pixels in images, typically visible with longer exposures. Hot-pixel correction identifies hot pixels and then substitutes a calculated value based on the values of neighboring pixels in place of hot pixels. This property enables or disables hot-pixel correction. If the connected camera supports hot-pixel correction, the threshold-range maximum will be greater than zero.

Type bool

property hot_pixel_correction_threshold
Due to variability in manufacturing, some pixels have inherently higher dark current which manifests as abnormally bright pixels in images, typically visible with longer exposures. Hot-pixel correction identifies hot pixels and then substitutes a calculated value based on the values of neighboring pixels in place of hot pixels. This property may be used to get or set the hot-pixel correction threshold within the available range. To determine the available range, query the *hot_pixel_correction_threshold_range* property. If the threshold range maximum is zero, the connected camera does not support hot-pixel correction. To enable hot-pixel correction, use *is_hot_pixel_correction_enabled*.

Type int

property hot_pixel_correction_threshold_range
The range of acceptable hot pixel correction threshold values. If the maximum value is zero, that is an indication that hot pixel correction is not supported by the camera. This property is Read-Only.

Type *Range*

property sensor_width_pixels
This property provides the physical width of the camera sensor in pixels. This is equivalent to the ROI-height-range maximum value. This property is Read-Only.

Type int

property gain_range
The range of possible gain values. This property is Read-Only.

Type *Range*

property image_width_range_pixels
The range of possible image width values. This property is Read-Only.

Type *Range*

property sensor_height_pixels
This property provides the physical height of the camera sensor in pixels. It is equivalent to the ROI-width-range-maximum value. This property is Read-Only.

Type int

property image_height_range_pixels
The range of possible image height values. This property is Read-Only.

Type *Range*

property model
Gets the camera model number such as 1501M or 8051C. This property is Read-Only.

Type str

property name
Cameras can always be distinguished from each other by their serial numbers and/or model. A camera can also be named to distinguish between them. For example, if using a two-camera system, cameras may be named "Left" and "Right."

Type str

property name_string_length_range
The minimum and maximum string lengths allowed for setting the camera's name.

Type *Range*

property frames_per_trigger_zero_for_unlimited
The number of frames generated per software or hardware trigger can be unlimited or finite. If set to zero, the camera will self-trigger indefinitely, allowing a continuous video feed. If set to one or higher, a single software or hardware trigger will generate only the prescribed number of frames and then stop.

Type int

property frames_per_trigger_range
The number of frames generated per software or hardware trigger can be unlimited or finite. If set to zero, the camera will self-trigger indefinitely, allowing a continuous video feed. If set to one or higher, a single software or hardware trigger will generate only the prescribed number of frames and then stop. This property returns the valid range for *frames_per_trigger_zero_for_unlimited*. This property is Read-Only.

Type *Range*

property usb_port_type
The *USB_PORT_TYPE* enumeration defines the values the SDK uses for specifying the USB port type. These values are returned by SDK API functions and callbacks based on the type of physical USB port that the device is connected to. This property is Read-Only.

Type *USB_PORT_TYPE*

property communication_interface
This property describes the computer interface type, such as USB, GigE, or CameraLink. This property is Read-Only.

Type *COMMUNICATION_INTERFACE*

property operation_mode
Thorlabs scientific cameras can be software- or hardware-triggered. To run continuous-video mode, set *frames_per_trigger_zero_for_unlimited* to zero and *operation_mode* to SOFTWARE_TRIGGERED. To issue individual software triggers, set *frames_per_trigger_zero_for_unlimited* to a number greater than zero and *operation_mode* to SOFTWARE_TRIGGERED. To trigger frames using the hardware trigger input, set *operation_mode* mode to HARDWARE_TRIGGERED. In this mode, the *exposure_time_us* property is used to determine the length of the exposure. To trigger frames using the hardware trigger input and to determine the exposure length with that signal, set *operation_mode* to BULB.

Type *OPERATION_MODE*

property is_armed
Prior to issuing software or hardware triggers to get images from a camera, call *arm()* to prepare it for imaging. This property indicates whether *arm()* has been called. This property is Read-Only.

Type bool

property is_eep_supported
Equal Exposure Pulse (EEP) mode is an LVTTL-level signal that is active between the time when all rows have been reset during rolling reset, and the end of the exposure time (and the beginning of rolling readout). The signal can be used to control an external light source that will be triggered on only during the equal exposure period, providing the same amount of exposure for all pixels in the ROI.

This property determines whether the connected camera supports EEP mode. This property is Read-Only. To activate EEP mode, see *is_eep_enabled*

Type bool

property is_led_supported
Some scientific cameras include an LED indicator light on the back panel. This property is Read-Only. Use *is_led_supported* to determine whether the connected camera has an LED indicator.

Type bool

property is_cooling_supported
All Thorlabs scientific cameras are designed to efficiently dissipate heat. Some models additionally provide active cooling. Use this property to determine whether the connected camera supports active cooling.

This property is Read-Only.

Type bool

property is_cooling_enabled
All Thorlabs scientific cameras are designed to efficiently dissipate heat. Some models additionally provide active cooling via a Thermoelectric Cooler (TEC). Cameras with TECs have an additional power cable to power the cooler. When the cable is plugged in, the TEC will start cooling. When the cable is unplugged, the TEC will stop and the camera will not have active cooling. Use this property to determine via software whether the TEC cable is plugged in or not.

This property is Read-Only.

Type bool

property is_nir_boost_supported
Some scientific-CCD camera models offer an enhanced near-infrared imaging mode for wavelengths in the 500 to 1000nm range. The Thorlabs website includes a helpful overview of this camera function on the Camera Basics page: https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=8962

This property enables or disables NIR-boost mode. This property is Read-Only.

Type bool

property camera_sensor_type
The camera sensor type. This property is Read-Only.

Type *SENSOR_TYPE*

property color_filter_array_phase
This describes the *color filter array phase* for the camera. This property is Read-Only.

Type *FILTER_ARRAY_PHASE*

property camera_color_correction_matrix_output_color_space
This describes the camera color correction matrix output color space. This property is Read-Only.

Type str

property data_rate
This property sets or gets the sensor-level data rate. Scientific-CCD cameras offer data rates of 20MHz or 40MHz. Compact-scientific cameras offer FPS30 or FPS50, which are frame rates supported by the camera when doing full-frame readout. The actual rate can vary if a region of interest (ROI) or binning is set or if the host computer cannot keep up with the camera. To test whether the connected camera supports a particular data rate, use `get_is_data_rate_supported`.

Type *DATA_RATE*

property sensor_pixel_size_bytes
The pixel size of the camera's sensor in bytes. This represents the amount of space 1 pixel will occupy in the frame buffer. This property is Read-Only.

Type int

property sensor_pixel_width_um
This property provides the physical width of a single light-sensitive photo site on the sensor. This property is Read-Only.

Type float

property sensor_pixel_height_um
This property provides the physical height of a single light-sensitive photo site on the sensor. This property is Read-Only.

Type float

property bit_depth
The number of bits to which a pixel value is digitized on a camera. In the image data that is delivered to the host application, the bit depth indicates how many of the lower bits of each 16-bit ushort value are relevant. While most cameras operate at a fixed bit depth, some are reduced when data bandwidth limitations would otherwise restrict the frame rate. Please consult the camera manual and specification for details about a specific model.

Type int

property roi
By default, the region of interest (ROI) is the same as the sensor resolution. The region of interest can be reduced to a smaller rectangle in order to focus on an area smaller than a full- frame image. In some cases, reducing the ROI can increase the frame rate since less data needs to be transmitted to the host computer. Binning sums adjacent sensor pixels into "super pixels". It trades off spatial resolution for sensitivity and speed. For example, if a sensor is 1920 by 1080 pixels and binning is set to two in the X direction and two in the Y direction, the resulting image will be 960 by 540 pixels. Since smaller images require less data to be transmitted to the host computer, binning may increase the frame rate. By default, binning is set to

one in both horizontal and vertical directions. binning can be changed by setting *binx* or *biny*. It can be different in the X direction than the Y direction, and the available ranges vary by camera model.

To determine the available ROI ranges, use the *roi_range*.

Type *ROI*

property roi_range
The rules for rectangular regions of interest (ROIs) vary by camera model. Please consult the camera documentation for more details. The ROI height range indicates the smallest height to which an ROI can be set up to a maximum of the sensor's vertical resolution. The ROI width range indicates the smallest width to which an ROI can be set up to a maximum of the sensor's horizontal resolution.

This property is Read-Only. For setting the ROI, see *roi*.

Type *ROIRange*

property serial_number
This property gets the unique identifier for a camera. This property is Read-Only.

Type str

property serial_number_string_length_range
The minimum and maximum number of characters allowed in the serial number string. This property is Read-Only.

Type *Range*

property is_led_on
Some scientific cameras include an LED indicator light on the back panel. This property can be used to turn it on or off.

Type bool

property eep_status
Equal Exposure Pulse (EEP) mode is an LVTTL-level signal that is active during the time when all rows have been reset during rolling reset, and the end of the exposure time (and the beginning of rolling readout). The signal can be used to control an external light source that will be on only during the equal exposure period, providing the same amount of exposure for all pixels in the ROI.

When EEP mode is disabled, the status will always be EEP_STATUS.OFF. EEP mode can be enabled, but, depending on the exposure value, active or inactive. If EEP is enabled in bulb mode, it will always give a status of BULB.

This property is Read-Only. To activate EEP mode, see *is_eep_enabled*

Type *EEP_STATUS*

property is_eep_enabled
Equal Exposure Pulse (EEP) mode is an LVTTL-level signal that is active between the time when all rows have been reset during rolling reset, and the end of the exposure time (and the beginning of rolling readout). The signal can be used to control an external light source that will be triggered on only during the equal exposure period, providing the same amount of exposure for all pixels in the ROI.

Please see the camera specification for details on EEP mode.

When enabled, EEP mode will be active or inactive depending on the exposure duration. Use *is_eep_supported* to test whether the connected camera supports this mode. Use *eep_status* to see whether the mode is active, inactive, in bulb mode, or disabled.

Type bool

property biny
The current vertical binning value.

Type int

property biny_range
The binning ratio in the Y direction can be determined with this property. By default, binning is set to one in both X and Y directions. This property is Read-Only. To set biny, see *biny*.

Type *Range*

property gain
Gain refers to the scaling of pixel values up or down for a given amount of light. This scaling is applied prior to digitization. To determine the valid range of values, use the *gain_range* property. If the *gain_range* maximum is zero, then Gain is not supported for the connected camera. The units of measure for Gain can vary by camera. Please consult the data sheet for the specific camera model. Query the *gain_range* property to determine the possible values.

Type int

property black_level
Black level adds an offset to pixel values. If the connected camera supports black level, the *black_level_range* will have a maximum greater than zero.

Type int

property black_level_range
Black level adds an offset to pixel values. If black level is supported by a camera model, then this property will have an upper range higher than zero.

black_level_range indicates the available values that can be used for the *black_level* property.

Type *Range*

property image_width_pixels
This property provides the image width in pixels. It is related to ROI width. This property is Read-Only.

Type int

property image_height_pixels
This property provides the image height in pixels. It is related to ROI height. This property is Read-Only.

Type int

property polar_phase
This describes how the polarization filter is aligned over the camera sensor. This property is only supported in polarized cameras. In a polarized camera, each pixel is covered with one of four linear polarizers with orientations of -45°, 0°, 45°, or 90°. The polar phase represents the origin pixel on the sensor. To determine if a camera supports polarization, check the *camera_sensor_type* property. This property is Read-Only.

Type *POLAR_PHASE*

property frame_rate_control_value_range
Frame rate control will set the frames per second of the camera. If frame rate is supported by a camera model, then this property will have an upper range higher than zero.

frame_rate_control_value_range indicates the available values that can be used for the *frame_rate_control_value* property.

Type *Range*

property is_frame_rate_control_enabled
While frame rate control is enabled, the frames per second will be controlled by *frame_rate_control_value*.

The frame rate control adjusts the frame rate of the camera independent of exposure time, within certain constraints. For short exposure times, the maximum frame rate is limited by the readout time of the sensor. For long exposure times, the frame rate is limited by the exposure time.

This property enables or disables the frame rate control feature. If the connected camera supports frame rate control, the threshold-range maximum will be greater than zero.

Type bool

property frame_rate_control_value
The frame rate control adjusts the frame rate of the camera independent of exposure time, within certain constraints. For short exposure times, the maximum frame rate is limited by the readout time of the sensor. For long exposure times, the frame rate is limited by the exposure time.

Type float

exception thorlabs_tsi_sdk.tl_camera.**TLCameraError**(*message*)
Bases: Exception

6.1.1.2 thorlabs_tsi_sdk.tl_camera_enums module

tl_camera_enums.py

class thorlabs_tsi_sdk.tl_camera_enums.**OPERATION_MODE**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

The OPERATION_MODE enumeration defines the available modes for a camera.

SOFTWARE_TRIGGERED = 0
Use software operation mode to generate one or more frames per trigger or to run continuous video mode.

HARDWARE_TRIGGERED = 1
Use hardware triggering to generate one or more frames per trigger by issuing hardware signals.

BULB = 2
Use bulb-mode triggering to generate one or more frames per trigger by issuing hardware signals. Please refer to the camera manual for signaling details.

RESERVED1 = 3

RESERVED2 = 4

class thorlabs_tsi_sdk.tl_camera_enums.**SENSOR_TYPE**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

This describes the physical capabilities of the camera sensor.

MONOCHROME = 0
Each pixel of the sensor indicates an intensity.

BAYER = 1
The sensor has a bayer-patterned filter overlaying it, allowing the camera SDK to distinguish red, green, and blue values.

MONOCHROME_POLARIZED = 2
The sensor has a polarization filter overlaying it allowing the camera to capture polarization information from the incoming light.

class thorlabs_tsi_sdk.tl_camera_enums.**TRIGGER_POLARITY**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

The TRIGGER_POLARITY enumeration defines the options available for specifying the hardware trigger polarity. These values specify which edge of the input trigger pulse that will initiate image acquisition.

ACTIVE_HIGH = 0
Acquire an image on the RISING edge of the trigger pulse.

ACTIVE_LOW = 1
Acquire an image on the FALLING edge of the trigger pulse.

class thorlabs_tsi_sdk.tl_camera_enums.**EEP_STATUS**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

The EEP_STATUS enumeration defines the options available for specifying the device's EEP mode. Equal Exposure Pulse (EEP) mode is an LVTTL-level signal that is active during the time when all rows have been reset during rolling reset, and the end of the exposure time (and the beginning of rolling readout). The signal can be used to control an external light source that will be on only during the equal exposure period, providing the same amount of exposure for all pixels in the ROI. EEP mode can be enabled, but may be active or inactive depending on the current exposure value. If EEP is enabled in bulb mode, it will always give a status of Bulb.

DISABLED = 0
EEP mode is disabled.

ENABLED_ACTIVE = 1
EEP mode is enabled and currently active.

ENABLED_INACTIVE = 2
EEP mode is enabled, but due to an unsupported exposure value, currently inactive.

ENABLED_BULB = 3
EEP mode is enabled in bulb mode.

class thorlabs_tsi_sdk.tl_camera_enums.**DATA_RATE**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

The DATA_RATE enumeration defines the options for setting the desired image data delivery rate.

RESERVED1 = 0

RESERVED2 = 1

FPS_30 = 2
Sets the device to deliver images at 30 frames per second.

FPS_50 = 3
Sets the device to deliver images at 50 frames per second.

class thorlabs_tsi_sdk.tl_camera_enums.**USB_PORT_TYPE**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

The USB_PORT_TYPE enumeration defines the values the SDK uses for specifying the USB bus speed. These values are returned by SDK API functions and callbacks based on the type of physical USB port that the device is connected to.

USB1_0 = 0
The device is connected to a USB 1.0/1.1 port (1.5 Mbits/sec or 12 Mbits/sec).

USB2_0 = 1
The device is connected to a USB 2.0 port (480 Mbits/sec).

USB3_0 = 2
The device is connected to a USB 3.0 port (5000 Mbits/sec).

class thorlabs_tsi_sdk.tl_camera_enums.**TAPS**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

Scientific CCD cameras support one or more taps. After exposure is complete, a CCD pixel array holds the charges corresponding to the amount of light collected at beach pixel location. The data is then read out through 1, 2, or 4 channels at a time.

SINGLE_TAP = 0
Charges are read out through a single analog-to-digital converter.

DUAL_TAP = 1
Charges are read out through two analog-to-digital converters.

QUAD_TAP = 2
Charges are read out through four analog-to-digital converters.

class thorlabs_tsi_sdk.tl_camera_enums.**COMMUNICATION_INTERFACE**(*value*)
Bases: thorlabs_tsi_sdk.tl_camera_enums._CTypesEnum

Used to identify what interface the camera is currently using. If using USB, the specific USB version can also be identified using USB_PORT_TYPE.

GIG_E = 0
The camera uses the GigE Vision (GigE) interface standard.

LINK = 1
The camera uses the CameraLink serial-communication-protocol standard.

USB = 2
The camera uses a USB interface.

6.1.1.3 thorlabs_tsi_sdk.tl_color_enums module

tl_color_enums.py

class thorlabs_tsi_sdk.tl_color_enums.**FILTER_ARRAY_PHASE**(*value*)
Bases: thorlabs_tsi_sdk.tl_color_enums.__CTypesEnum

The FILTER_ARRAY_PHASE enumeration lists all the possible values that a pixel in a Bayer pattern color arrangement could assume.

The classic Bayer pattern is:

```
-----------------------
|           |           |
|    R      |    GR     |
|           |           |
-----------------------
|           |           |
|    GB     |    B      |
|           |           |
-----------------------
```

where:

- R = a red pixel
- GR = a green pixel next to a red pixel
- B = a blue pixel
- GB = a green pixel next to a blue pixel

The primitive pattern shown above represents the fundamental color pixel arrangement in a Bayer pattern color sensor. The basic pattern would extend in the X and Y directions in a real color sensor containing millions of pixels.

Notice that the color of the origin (0, 0) pixel logically determines the color of every other pixel.

It is for this reason that the color of this origin pixel is termed the color "phase" because it represents the reference point for the color determination of all other pixels.

Every TSI color camera provides the sensor specific color phase of the full frame origin pixel as a discoverable parameter.

BAYER_RED = 0
A red pixel.

BAYER_BLUE = 1
A blue pixel.

GREEN_LEFT_OF_RED = 2
A green pixel next to a red pixel.

GREEN_LEFT_OF_BLUE = 3
A green pixel next to a blue pixel.

class thorlabs_tsi_sdk.tl_color_enums.**FORMAT**(*value*)
Bases: thorlabs_tsi_sdk.tl_color_enums.__CTypesEnum

The FORMAT enumeration lists all the possible options for specifying the order of color pixels in input and/or output buffers.

Depending on the context, it can specify:

- the desired pixel order that a module must use when writing color pixel data into an output buffer
- the pixel order that a module must use to interpret data in an input buffer.

BGR_PLANAR = 0
The color pixels blue, green, and red are grouped in separate planes in the buffer:

BBBBB... GGGGG... RRRRR....

BGR_PIXEL = 1
The color pixels blue, green, and red are clustered and stored consecutively in the following pattern:

BGRBGRBGR...

RGB_PIXEL = 2
The color pixels blue, green, and red are clustered and stored consecutively in the following pattern:

RGBRGBRGB...

class thorlabs_tsi_sdk.tl_color_enums.**FILTER_TYPE**(*value*)
Bases: thorlabs_tsi_sdk.tl_color_enums.__CTypesEnum

The FILTER_TYPE enumeration lists all the possible filter options for color cameras

BAYER = 0
A Bayer pattern color sensor.

6.1.1.4 thorlabs_tsi_sdk.tl_mono_to_color_enums module

tl_mono_to_color_enums.py

class thorlabs_tsi_sdk.tl_mono_to_color_enums.**COLOR_SPACE**(*value*)
Bases: thorlabs_tsi_sdk.tl_mono_to_color_enums._CTypesEnum

A color space describes how the colors in an image are going to be specified. Some commonly used color spaces are those derived from the RGB color model, in which each pixel has a Red, Blue, and Green component. This means the amount of color that can expressed in a single pixel is all the possible combinations of Red, Blue, and Green. If we assume the image data is in bytes, each component can take any value from 0 to 255. The total number of colors that a pixel could express can be calculated as 256 * 256 * 256 = 16777216 different colors.

There are many different color spaces that are used for different purposes. The mono to color processor supports two color spaces that are both derived from the RGB color model: sRGB and Linear sRGB.

SRGB = 0
sRGB or standard RGB is a common color space used for displaying images on computer monitors or for sending images over the internet. In addition to the Red, Blue, and Green components combining to define the color of a pixel, the final RGB values undergo a nonlinear transformation to be put in the sRGB color space. The exact transfer function can be found online by searching for the sRGB specification. The purpose of this transformation is to represent the colors in a way that looks more accurate to humans.

LINEAR_SRGB = 1
Linear sRGB is very similar to sRGB, but does not perform the non linear transformation. The transformation of the data in sRGB changes the RGB intensities, whereas this color space is much more representative of the raw image data coming off the sensor. Without the transformation, however, images in the Linear sRGB color space do not look as accurate as those in sRGB. When deciding between Linear sRGB and sRGB, use Linear sRGB when the actual intensities of the raw image data are important and use sRGB when the image needs to look accurate to the human eye.

6.1.1.5 thorlabs_tsi_sdk.tl_mono_to_color_processor module

tl_mono_to_color_processor.py

class thorlabs_tsi_sdk.tl_mono_to_color_processor.**MonoToColorProcessorSDK**
Bases: object

The SDK object that is used to create MonoToColorProcessor objects. There must be only one instance of this class active at a time. Use the *dispose()* method to destroy an SDK instance before creating another instance. *with* statements can also be used with this class to automatically dispose the SDK.

dispose() → None
Cleans up the MonoToColorProcessorSDK instance - make sure to call this when you are done with the MonoToColorProcessorSDK instance. If using the *with* statement, dispose is called automatically upon exit.

create_mono_to_color_processor(*camera_sensor_type:* thorlabs_tsi_sdk.tl_camera_enums.SENSOR_TYPE, *color_filter_array_phase:* thorlabs_tsi_sdk.tl_color_enums.FILTER_ARRAY_PHASE, *color_correction_matrix:* *numpy.array*, *default_white_balance_matrix:* *numpy.array*, *bit_depth:* *int*) → *thorlabs_tsi_sdk.tl_mono_to_color_processor.MonoToColorProcessor*
Creates a MonoToColorProcessor object using the given parameters.

Parameter

- **camera_sensor_type** (*SENSOR_TYPE*) - The sensor type used by the camera. Use the property *TLCamera.camera_sensor_type* to get this information from a camera.

Parameter

- **color_filter_array_phase** (FILTER_ARRAY_PHASE) - The array phase of the camera's color filter. Use *TLCamera.color_filter_array_phase* to get this information from a camera.

Parameter

- **color_correction_matrix** (np.array) - A 3x3 correction matrix specific to a camera model that is used during color processing to achieve accurate coloration. use *TLCamera.get_color_correction_matrix* to get this information from a camera.

Parameter

- **default_white_balance_matrix** (np.array) - A 3x3 correction matrix specific to a camera model that is used during color processing to white balance images under typical lighting conditions. Use *TLCamera.get_default_white_balance_matrix* to get this information from a camera.

Parameter

- **bit_depth** (int) - The bit depth that will be used during color processing. To get the bit depth of a camera, use *TLCamera.bit_depth*

Returns *MonoToColorProcessor*

class thorlabs_tsi_sdk.tl_mono_to_color_processor.**MonoToColorProcessor**(*key: type*, *sdk: Any*, *mono_to_color_processor_h Any*)

Bases: object

These objects are used to quickly convert monochrome image data to colored image data. When finished with a MonoToColorProcessor, call its *dispose* method to clean up any opened resources. These object can be managed using *with* statements for automatic resource clean up. These objects can only be created by calls to *MonoToColorProcessorSDK.create_mono_to_color_processor()*

dispose() → None

Cleans up the MonoToColorProcessor instance - make sure to call this when you are done with the MonoToColor processor. If using the *with* statement, dispose is called automatically upon exit.

transform_to_48(*input_buffer: numpy.array*, *image_width_pixels: int*, *image_height_pixels: int*) → numpy.array

Convert monochrome image data into a 3-channel colored image with 16 bits per channel, resulting in 48 bits per pixel. The pixel data will be ordered according to the current value of *output_format*.

Parameters

- **input_buffer** (*np.array*) – Single channel monochrome image data. The size of this array should be image_width * image_height. The dtype of the array should be ctypes.c_ushort or a type of equivalent size, the image buffer that comes directly from the camera is compatible (see: *TLCamera.get_pending_frame_or_null()*).
- **image_width_pixels** (*int*) – The width of the image in the image_buffer.
- **image_height_pixels** (*int*) – The height of the image in the image_buffer.

Return np.array 3-channel colored output image, *dtype* = ctypes.c_ushort.

transform_to_32(*input_buffer: numpy.array*, *image_width_pixels: int*, *image_height_pixels: int*) → numpy.array

Convert monochrome image data into a 4-channel colored image with 8 bits per channel, resulting in 32 bits per pixel. The pixel data will be ordered according to the current value of *output_format*.

Parameters

- **input_buffer** (*np.array*) – Single channel monochrome image data. The size of this array should be image_width * image_height. The dtype of the array should be ctypes.c_ushort or a type of equivalent size, the image buffer that comes directly from the camera is compatible (see: *TLCamera.get_pending_frame_or_null()*).
- **image_width_pixels** (*int*) – The width of the image in the image_buffer.
- **image_height_pixels** (*int*) – The height of the image in the image_buffer.

Return np.array 4-channel colored output image, *dtype* = ctypes.c_ubyte.

transform_to_24(*input_buffer: numpy.array*, *image_width_pixels: int*, *image_height_pixels: int*) → numpy.array

Convert monochrome image data into a 3-channel colored image with 8 bits per channel, resulting in 24 bits per pixel. The pixel data will be ordered according to the current value of *output_format*.

Parameters

- **input_buffer** (*np.array*) – Single channel monochrome image data. The size of this array should be image_width * image_height. The dtype of the array should be ctypes.c_ushort or a type of equivalent size, the image buffer that comes directly from the camera is compatible (see: *TLCamera.get_pending_frame_or_null()*).
- **image_width_pixels** (*int*) – The width of the image in the input_buffer.
- **image_height_pixels** (*int*) – The height of the image in the input_buffer.

Return np.array 3-channel colored output image, *dtype* = ctypes.c_ubyte.

property color_space

The color space of the mono to color processor. See *COLOR_SPACE* for what color spaces are available.

Type *COLOR_SPACE*

property output_format

The format of the colored output image. This describes how the data is ordered in the returned buffer from the transform functions. By default it is RGB_PIXEL. See *FORMAT*.

Type FORMAT

property red_gain

The gain factor that will be applied to the red pixel values in the image. The red intensities will be multiplied by this gain value in the final colored image. The default red gain is taken from the *default_white_balance_matrix* that is passed in when constructing a *MonoToColorProcessor*.

Type float

property blue_gain

The gain factor that will be applied to the red pixel values in the image. The blue intensities will be multiplied by this gain value in the final colored image. The default blue gain is taken from the *default_white_balance_matrix* that is passed in when constructing a *MonoToColorProcessor*.

Type float

property green_gain
The gain factor that will be applied to the red pixel values in the image. The green intensities will be multiplied by this gain value in the final colored image. The default green gain is taken from the *default_white_balance_matrix* that is passed in when constructing a *MonoToColorProcessor*.

Type float

property camera_sensor_type
The sensor type of the camera (monochrome, bayer, etc...). This value is passed in during construction and may be read back using this property.

Type *SENSOR_TYPE*

property color_filter_array_phase
The color filter array phase used in this mono to color processor. This value is passed in during construction and may be read back using this property.

Type FILTER_ARRAY_PHASE

property color_correction_matrix
The default color correction matrix associated with the mono to color processor. This value is passed in during construction and may be read back using this property.

Type np.array

property default_white_balance_matrix
The default white balance matrix associated with the mono to color processor. This value is passed in during construction and may be read back using this property.

Type np.array

property bit_depth
The bit depth associated with the mono to color processor. This value is passed in during construction and may be read back using this property.

Type int

exception thorlabs_tsi_sdk.tl_mono_to_color_processor.**MonoToColorError**(*message*)
Bases: Exception

6.1.1.6 thorlabs_tsi_sdk.tl_polarization_enums module

tl_polarization_enums.py

class thorlabs_tsi_sdk.tl_polarization_enums.**POLAR_PHASE**(*value*)
Bases: thorlabs_tsi_sdk.tl_polarization_enums._CTypesEnum

The possible polarization angle values (in degrees) for a pixel in a polarization sensor. The polarization phase pattern of the sensor is:

```
-------------
| + 0 | -45 |
-------------
| +45 | +90 |
-------------
```

The primitive pattern shown above represents the fundamental polarization phase arrangement in a polarization sensor. The basic pattern would extend in the X and Y directions in a real polarization sensor containing millions of pixels. Notice that the phase of the origin (0, 0) pixel logically determines the phase of every other pixel. It

is for this reason that the phase of this origin pixel is termed the polarization "phase" because it represents the reference point for the phase determination of all other pixels.

PolarPhase0 = 0
0 degrees polarization phase

PolarPhase45 = 1
45 degrees polarization phase

PolarPhase90 = 2
90 degrees polarization phase

PolarPhase135 = 3
135 (-45) degrees polarization phase

6.1.1.7 thorlabs_tsi_sdk.tl_polarization_processor module

tl_mono_to_color_processor.py ***BETA***

class thorlabs_tsi_sdk.tl_polarization_processor.**PolarizationProcessorSDK**
Bases: object

The polarization processor SDK loads DLLs into memory and provides functions to polarization processor instances. Be sure to dispose all PolarizationProcessor objects and then dispose the PolarizationProcessorSDK before exiting the application. *with* statements can also be used with this class to automatically dispose the SDK.

dispose() → None
Cleans up the PolarizationProcessorSDK instance - make sure to call this when you are done with the PolarizationProcessorSDK instance. If using the *with* statement, dispose is called automatically upon exit.

create_polarization_processor() → *thorlabs_tsi_sdk.tl_polarization_processor.PolarizationProcessor*
Creates a Polarization Processor object.

Returns *PolarizationProcessor*

class thorlabs_tsi_sdk.tl_polarization_processor.**PolarizationProcessor**(*key: type*, *sdk: Any*, *polarization_processor_handle: Any*)
Bases: object

These objects are used to convert raw sensor data to polarized data. When finished with a PolarizationProcessor, call its *dispose* method to clean up any opened resources. These object can be managed using *with* statements for automatic resource clean up. These objects can only be created by calls to *PolarizationProcessorSDK.create_polarization_processor()*

The transform functions return an output image with values from 0 to max value for each pixel. To calculate scaled values for each pixel, you may use the following equation:

- polarization_parameter_value = pixel_integer_value * (max_parameter_value / (2^bit_depth - 1)) + min_parameter_value

The suggested min and max for each type are as follows:

- **Azimuth: (range from -90° to 90°)** ◦ min_parameter_value = -90 ◦ max_parameter_value = 180

- **DoLP: (range from 0 to 100%)** ◦ min_parameter_value = 0 ◦ max_parameter_value = 100
- **Intensity: (range from 0 to 100)** ◦ min_parameter_value = 0 ◦ max_parameter_value = 100

dispose() → None
Cleans up the PolarizationProcessor instance - make sure to call this when you are done with the polarization processor. If using the *with* statement, dispose is called automatically upon exit.

transform_to_intensity(*sensor_polar_phase:* thorlabs_tsi_sdk.tl_polarization_enums.POLAR_PHASE, *input_image: numpy.array*, *image_origin_x_pixels: int*, *image_origin_y_pixels: int*, *image_width_pixels: int*, *image_height_pixels: int*, *input_image_bit_depth: int*, *output_max_value: int*) → numpy.array
Transforms raw-image data into the intensity-output buffer, which shows the brightness of the light at each pixel.

Parameters

- **sensor_polar_phase** – The polar phase (in degrees) of the origin (top-left) pixel of the camera sensor.
- **input_image** – Unprocessed input image delivered by the camera.
- **image_origin_x_pixels** – The X position of the origin (top-left) of the input image on the sensor. Warning: Some camera models may nudge input ROI values. Please read values back from camera. See *TLCamera.roi*.
- **image_origin_y_pixels** – The Y position of the origin (top-left) of the input image on the sensor. Warning: Some camera models may nudge input ROI values. Please read values back from camera. See *TLCamera.roi*.
- **image_width_pixels** – Width of input image in pixels
- **image_height_pixels** – Height of input image in pixels
- **input_image_bit_depth** – Bit depth of input image pixels
- **output_max_value** – The maximum possible pixel value in the output images. Must be between 1 and 65535.

Return np.array Array containing the total optical power (intensity) output, *dtype* = ctypes.c_ushort.

transform_to_dolp(*sensor_polar_phase:* thorlabs_tsi_sdk.tl_polarization_enums.POLAR_PHASE, *input_image: numpy.array*, *image_origin_x_pixels: int*, *image_origin_y_pixels: int*, *image_width_pixels: int*, *image_height_pixels: int*, *input_image_bit_depth: int*, *output_max_value: int*) → numpy.array
Transforms raw-image data into a DoLP (degree of linear polarization) output buffer, which is a measure of how polarized the light is from none to totally polarized.

Parameters

- **sensor_polar_phase** – The polar phase (in degrees) of the origin (top-left) pixel of the camera sensor.
- **input_image** – Unprocessed input image delivered by the camera.
- **image_origin_x_pixels** – The X position of the origin (top-left) of the input image on the sensor. Warning: Some camera models may nudge input ROI values. Please read values back from camera. See *TLCamera.roi*.
- **image_origin_y_pixels** – The Y position of the origin (top-left) of the input image on the sensor. Warning: Some camera models may nudge input ROI values. Please read values back from camera. See *TLCamera.roi*.

- **image_width_pixels** – Width of input image in pixels
- **image_height_pixels** – Height of input image in pixels
- **input_image_bit_depth** – Bit depth of input image pixels
- **output_max_value** – The maximum possible pixel value in the output images. Must be between 1 and 65535.

Return np.array Array containing the DoLP (degree of linear polarization) output, *dtype* = ctypes.c_ushort.

transform_to_azimuth(*sensor_polar_phase: thorlabs_tsi_sdk.tl_polarization_enums.POLAR_PHASE*, *input_image: numpy.array*, *image_origin_x_pixels: int*, *image_origin_y_pixels: int*, *image_width_pixels: int*, *image_height_pixels: int*, *input_image_bit_depth: int*, *output_max_value: int*) → numpy.array

Transforms raw-image data into an azimuth-output buffer, which shows the angle of polarized light at each pixel.

Parameters

- **sensor_polar_phase** – The polar phase (in degrees) of the origin (top-left) pixel of the camera sensor.
- **input_image** – Unprocessed input image delivered by the camera.
- **image_origin_x_pixels** – The X position of the origin (top-left) of the input image on the sensor. Warning: Some camera models may nudge input ROI values. Please read values back from camera. See *TLCamera.roi*.
- **image_origin_y_pixels** – The Y position of the origin (top-left) of the input image on the sensor. Warning: Some camera models may nudge input ROI values. Please read values back from camera. See *TLCamera.roi*.
- **image_width_pixels** – Width of input image in pixels
- **image_height_pixels** – Height of input image in pixels
- **input_image_bit_depth** – Bit depth of input image pixels
- **output_max_value** – The maximum possible pixel value in the output images. Must be between 1 and 65535.

Return np.array Array containing the azimuth (polar angle) output, *dtype* = ctypes.c_ushort.

exception thorlabs_tsi_sdk.tl_polarization_processor.**PolarizationError**(*message*)

Bases: Exception

6.1.2 Module contents

PYTHON MODULE INDEX

t